Paper Name: Engineering Mechanics Paper Code: ME101		
Module No.	**Syllabus**	**Contact Hrs.**
1	Importance of Mechanics in engineering; Introduction to Statics; Concept of Particle and Rigid Body; Types of forces: collinear, concurrent, parallel, concentrated, distributed; Vector and scalar quantities; Force is a vector; Transmissibility of a force (sliding vector). Introduction to Vector Algebra; Parallelogram law; Addition and subtraction of vectors; Lami's theorem; Free vector; Bound vector; Representation of forces in terms of i,j,k; Cross product and Dot product and their applications. Two dimensional force system; Resolution of forces; Moment; Varignon's theorem; Couple; Resolution of a coplanar force by its equivalent force-couple system; Resultant of forces.	**10**
2	Concept and Equilibrium of forces in two dimensions; Free body concept and diagram; Equations of equilibrium. Concept of Friction; Laws of Coulomb friction; Angle of Repose; Coefficient of friction.	**6**
3	Distributed Force: Centroid and Centre of Gravity; Centroids of a triangle, circular sector, quadralateral, composite areas consisting of above figures. Moments of inertia: MI of plane figure with respect to an axis in its plane, MI of plane figure with respect to an axis perpendicular to the plane of the figure; Parallel axis theorem; Mass moment of inertia of symmetrical bodies, e.g. cylinder, sphere, cone.	**7**
4	Introduction to Dynamics: Kinematics and Kinetics; Newton's laws of motion; Law of gravitation & acceleration due to gravity; Rectilinear motion of particles; determination of position, velocity and acceleration under uniform and non-uniformly accelerated rectilinear motion; construction of x-t, v-t and a-t graphs.	**5**

	Plane curvilinear motion of particles: Rectangular components (Projectile motion); Normal and tangential components (circular motion).	
5	Kinetics of particles: Newton's second law; Equation of motion; D.Alembert's principle and free body diagram; Principle of work and energy ; Principle of conservation of energy; Power and efficiency.	**5**

Recommended Books:

1. Engineering Mechanics [Vol-I & II]by Meriam & Kraige, 5th ed. – Wiley India
2. Engineering Mechanics: Statics & Dynamics by I.H.Shames, 4th ed. – PHI
3. Engineering Mechanics by Timoshenko , Young and Rao, Revised 4th ed. – TMH
4. Elements of Strength of Materials by Timoshenko & Young, 5th ed. – E.W.P
5. Fundamentals of Engineering Mechanics by Debabrata Nag & Abhijit Chanda– Chhaya Prakashani
6. Engineering Mechanics by Basudeb Bhattacharyya– Oxford University Press.
7. Engineering Mechanics: Statics & Dynamics by Hibbeler & Gupta, 11th ed. – Pearson

Mathematics-I
M(ME)101

Module I

Matrix: Determinant of a square matrix, Minors and Cofactors, Laplace's method of expansion of a determinant, Product of two determinants, Adjoint of a determinant, Jacobi's theorem on adjoint determinant. Singular and non-singular matrices, Adjoint of a matrix, Inverse of a non-singular matrix and its properties, orthogonal matrix and its properties, Trace of a matrix. Rank of a matrix and its determination using elementary row and column operations, Solution of simultaneous linear equations by matrix inversion method, Consistency and inconsistency of a system of homogeneous and inhomogeneous linear simultaneous equations, Eigen values and eigen vectors of a square matrix (of order 2 or 3), Eigen values of APTP, kA, AP-1P, Caley-Hamilton theorem and its applications.

Module II

Successive differentiation: Higher order derivatives of a function of single variable, Leibnitz's theorem (statement only and its application, problems of the type of recurrence relations in derivatives of different orders and also to find (*n*)0) *y* .
Mean Value Theorems & Expansion of Functions: Rolle's theorem and its application, Mean Value theorems – Lagrange & Cauchy and their application, Taylor's theorem with Lagrange's and Cauchy's form of remainders and its application, Expansions of functions by Taylor's and Maclaurin's theorem, Maclaurin's infinite series expansion of the functions: sin , cos , , log(1), ( ) , *x n x x e* + *x a* + *x n* being an integer or a fraction (assuming that the remainder 0 as *n R* → *n*→∞ in each case).
Reduction formula: Reduction formulae both for indefinite and definite integrals of types
(2 2)
sin , cos , sin cos , cos sin , , , *n n m n m*
n
dx
x x x x x nx m n
x + *a*
∫ ∫ ∫ ∫ ∫ are positive integers.

Module III

Calculus of Functions of Several Variables: Introduction to functions of several variables with examples, Knowledge of limit and continuity, Partial derivatives and related problems, Homogeneous functions and Euler's theorem and related problems up to three variables, Chain rules, Differentiation of implicit functions, Total differentials and their related problems, Jacobians up to three variables and related problems, Maxima, minima and saddle points of functions and related problems, Concept of line integrals, Double and triple integrals.

Module IV

Infinite Series: Preliminary ideas of sequence, Infinite series and their convergence/divergence, Infinite series of positive terms, Tests for convergence: Comparison test, Cauchy's Root test, D' Alembert's Ratio test and Raabe's test (statements and related problems on these tests), Alternating series, Leibnitz's Test (statement, definition) illustrated by simple example, Absolute convergence and Conditional
convergence.

Module-V

Vector Algebra and Vector Calculus: Scalar and vector fields – definition and terminologies, dot and cross products, scalar and vector triple products and related problems, Equation of straight line, plane and sphere, Vector function of a scalar variable, Differentiation of a vector function, Scalar and vector point functions, Gradient of a scalar point function, divergence and curl of a vector point function, Directional derivative. Related problems on these topics. Green's theorem, Gauss Divergence Theorem and Stoke's theorem
(Statements and applications).

[VISCOSITY OF SOLUTION]

AIM :→ Determination of the Unknown percentage compo--sition of Sugar solution by relative Viscosity method using Ostwald's Viscometer.

THEORY :→ Due to internal friction when a fluid passes through one another, it experiences a resistance to its flow, which is known as Viscosity. The co-efficient of viscosity is a measure of this resistance and defined as the tangential force per unit area required to maintain unit difference of viscosity between two layers unit distance apart. Its unit in C.G.S. system is dyne/cm^2.

When a homogeneous fluid of volume 'v' flows through a capillary tube of length 'l', radius 'r', in time 't', Under a driving force 'p', the co-efficient of viscosity is given according to poiseuille's formula by

$$\eta = \frac{\pi P r^4 t}{8lv}$$

The experimental determination of viscocity is rather different. If η_1 and η_2 are the viscocity of two diff--erent fluids of density ρ_1 and ρ_2 respectively which are successively allowed to fall through the same length 'h' of capillary e.g. between the marks of an Ostwald viscometer, the pressures are given by $h\rho_1 g$ and $h\rho_2 g$ and thus η_1 and η_2 are given by

$$\eta_1 = \frac{\pi r^4 t_1 (h\rho_1 g)}{8lv} \quad \text{and} \quad \eta_2 = \frac{\pi r^4 t_2 (h\rho_2 g)}{8lv}$$

DATE:- 28-04-14

$$\frac{\eta_1}{\eta_2} = \frac{\rho_1 t_1}{\rho_2 t_2} \quad \text{or,} \quad \eta_1 = \eta_2 \left(\frac{\rho_1 t_1}{\rho_2 t_2}\right)$$

APPARATUS :→

- Viscometer (ostwald)
- stop watch
- Pipette (5 ml)
- Beaker

REAGENTS :→

- Supplied Sugar solution

PROCEDURE :→

1. 3%, 6%, 9% and Unknown Sugar solutions are supplied.
2. Densities of the above solutions are given.
3. Wash the viscometer and rinse repeatedly with distilled water by sucking it in, releasing and throwing away the washings.
4. Suck in fresh distilled water kept in a beaker, release and start the stop watch as the meniscus touches the upper gradution and stop it when the meniscus touches the lower graduation.
5. Note the time.
6. Repeat the process thrice and take the mean of the three readings.
4. Again repeat the process with 4 sugar solution starting with the least concentrated one.
8. we know, $\eta_1 = \eta_2 \left(\frac{\rho_1 t_1}{\rho_2 t_2}\right)$ Here, η_1 = Viscocity of Sugar, η_2 = Viscocity of water, ρ_1 = density of sugar solution, ρ_2 = density of water, t_1 = time flow of sugar solution, t_2 = time flow of water.

DATE: 08-04-14

RESULTS & CALCULATIONS :→

[Room temperature = 35°c] [Density of water = 0·99406 gm]

TABLE[I] DETERMINATION OF DENSITY :→

Material	Wt. of empty bottle (gm)	Bottle + sugar solution (gm)	Wt. of sugar solution (gm)	Volume of sugar solution (ml)	Density (D=M/V) (gm/ml)
3% sugar solution	16·7	21·6836	4·9836	5	0·99672
6% sugar solution	16·7	21·7502	5·0502	5	1·01004
12% sugar solution	16·7	21·8536	5·1536	5	1·0306
Unknown sugar solution	16·7	21·8292	5·1292	5	1·02584

[VISCOSITY OF WATER = 0·798 C.P]

TABLE[II] DETERMINATION OF TIME OF FLOW :→

Material	Time of flow (sec)	Average time (sec)	Viscosity C.P
0% sugar solution	114	114	0·798
	115		
	113		
3% sugar solution	118	118	0·8282
	119		
	117		
6% sugar solution	125	125	0·8890
	124		
	126		
12% sugar solution	143	143	1·0377
	144		
	142		
Unknown sugar solution	134	134	0·9679
	135		
	135		

ATE:- 28-04-14

CALCULATION :→

Calculation for Viscosity :→

$$\eta = \frac{\rho_1 t_1}{\rho_w t_w} \eta_w$$

$$= \frac{\rho_1 t_1}{0.99406 \times 114} \times 0.798$$

$$\eta(3\%) = \frac{0.99642 \times 118}{0.99406 \times 114} \times 0.798 = 0.8282 \text{ CP}$$

$$\eta(6\%) = \frac{1.09004 \times 125}{0.99406 \times 114} \times 0.798 = 0.8890 \text{ cp}$$

$$\eta(12\%) = \frac{1.0306 \times 143}{0.99406 \times 114} \times 0.798 = 1.0377 \text{ cp}$$

$$\eta(\text{unknown}) = \frac{1.02584 \times 134}{0.99406 \times 114} \times 0.798 = 0.9679 \text{ cp}$$

✓

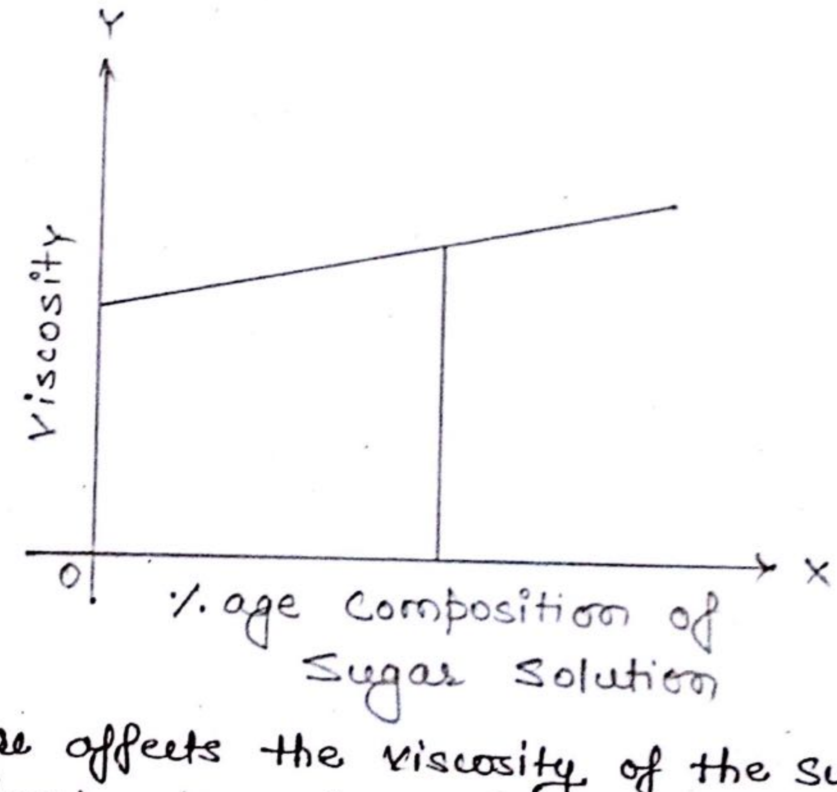
PRECAUTIONS :→

1. The Viscometer must be clean & rinsed.
2. Viscometer should be clampled in a Vertical position & its height must remain constant each time when it is clamped.
3. Exactly Same Volume of the two liquid should be used.
4. The Viscometer should not be disturbed during the measurement of time of flow.

DISCUSSION :→

1. The method used to determine the Unknown percentage of Sugar solution is known as relative Viscosity because the Viscocity of Unknown Sugar solution is found out by comparing it with Viscosity of water.

2. We get the Unknown percentage composition of Sugar solution from the plotted graph (along X axis percentage composition of Sugar Solution and along Y axis Viscosity).



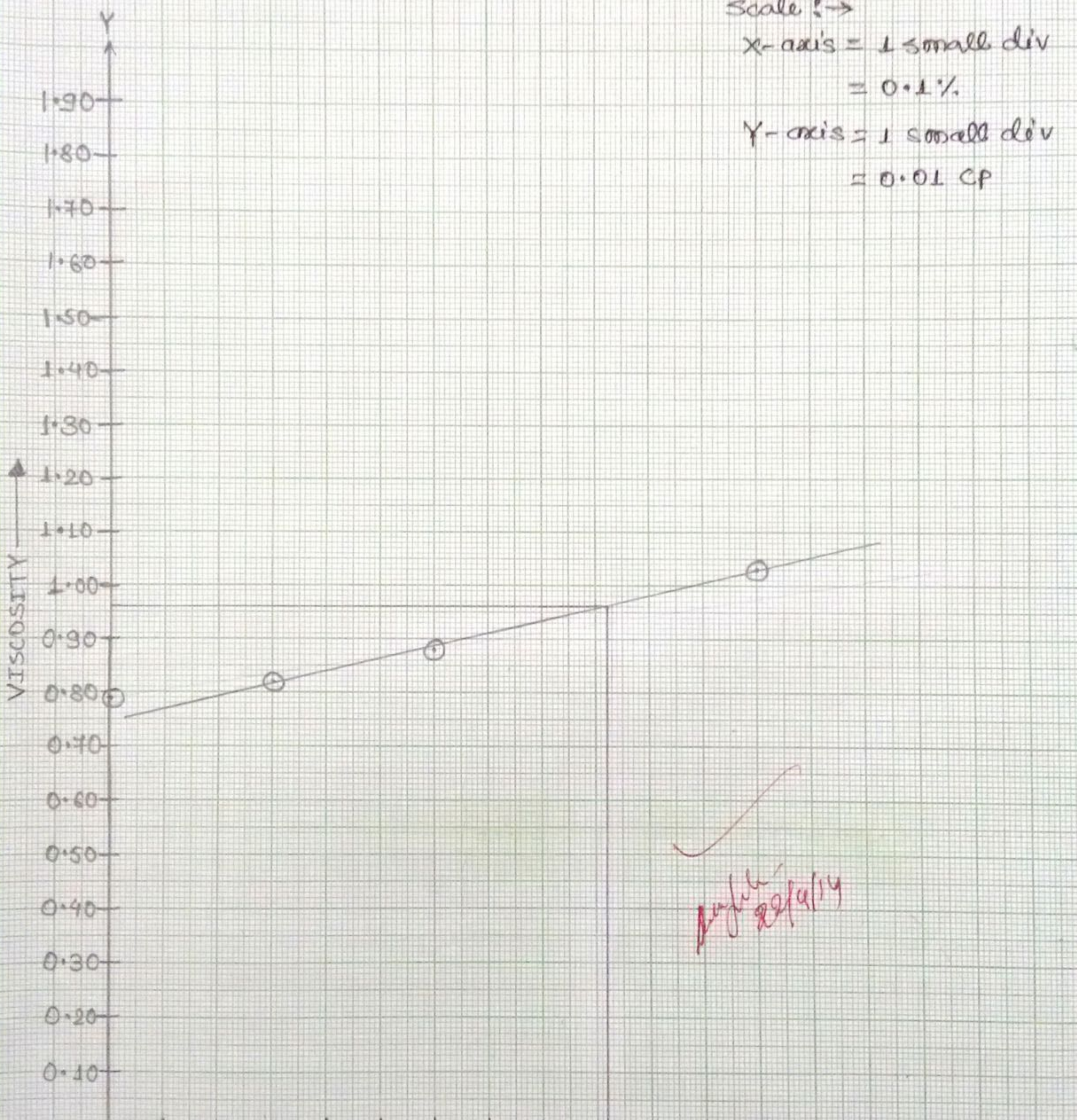
3. Temperature affects the viscosity of the sugar solution because temperature is inversely proportional to density and hence inversly related with Viscosity is directly proportional to density. The temperature is maintained around 25°–30°C.

4. The Viscosity also increases with increase in concentration.

CONCLUSION :→

The Unknown percentage composition of Sugar solution is 9.2.

%age composition Vs Viscosity



Scale :→
X- axis = 1 small div
= 0·1%
Y- axis = 1 small div
= 0·01 CP

% age composition ⟶

[illegible] 22/4/14

Experiment No - 1.

TITLE: VERIFICATION OF THEVENIN'S THEOREM.

OBJECTIVE: To verify the Thevenin's Theorem in the DC circuit.

APPRATUS:

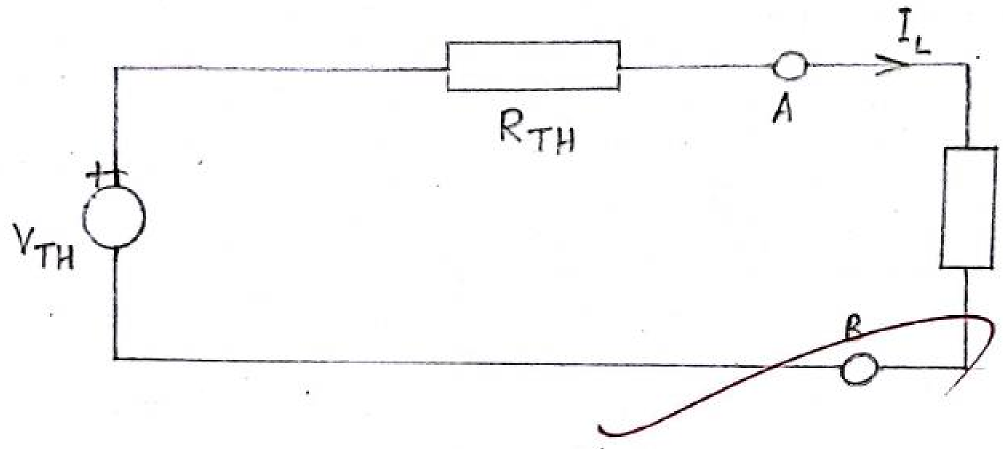
SL. No.	Apparatus Name	Apparatus Type	Range
1.	Trainer kit	DC	—
2.	Voltage Source	DC	12V, ~~5V~~
3.	Resistor 1,2&3	—	(500,100,90)Ω
4.	Ammeter	DC	0-10mA
5.	Voltmeter	DC	0-3V
6.	Multimeter	Digital	0-200Ω

THEORY:

Thevenin's theorem as applied to DC circuit may be Stated as:

Current fowling through a load resistance R_L connected across any two terminal A & B of a linear, bilateral network is given $\frac{V_{TH}}{R_{TH}+R_L}$, where V_{TH} is the open circuit voltage or thevenin's equivalent voltage (i.e. Voltage across terminal AB when R_L is removed) and R_{TH} is the by equivalent resistance of the network as viewed from the [P.T.O

Open circuited load terminals i.e from terminal AB deactivating all independent source.



Mathematically current through the load resistance R_L is given by the equation:-

$$I_L = \frac{V_{TH}}{R_{TH} + R_L}$$

where, I_L = Load Current.

V_{TH} = Open circuit Voltage across the terminals AB.

R_{TH} = Thevenin's Resistance.

R_L = Load Resistance.

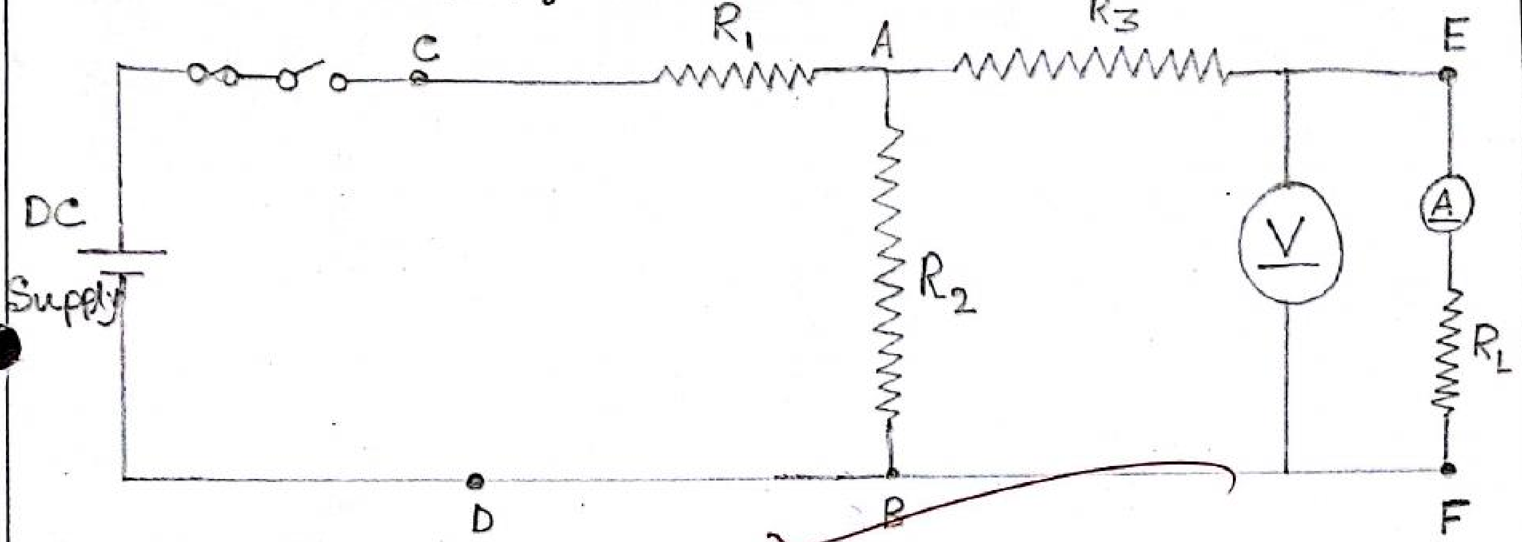
The following are the limitation of this theorem

i) Thevenin's theorem cannot be applicable for non-linear network.

ii) This theorem cannot calculate the Power consumed internally in the circuit or efficiency of the circuit.

P.T.O

CIRCUIT DIAGRAM :



PROCEDURE :

1) Connect the circuit diagram as shown in Fig.

2) Measure the value of R_1, R_2 & R_3.

3) Remove the R_L i.e open the terminal EF.

4) Switch ON the power supply and note down the open circuit voltage $(V_{oc} = V_{TH})$

5) Now remove the voltage source by replacing their internal resistance is assumed to be zero, then short the terminal C&D.

6) Measure the R_{TH} across by opening the terminal EF by multimeter or ammeter-Voltmeter method.

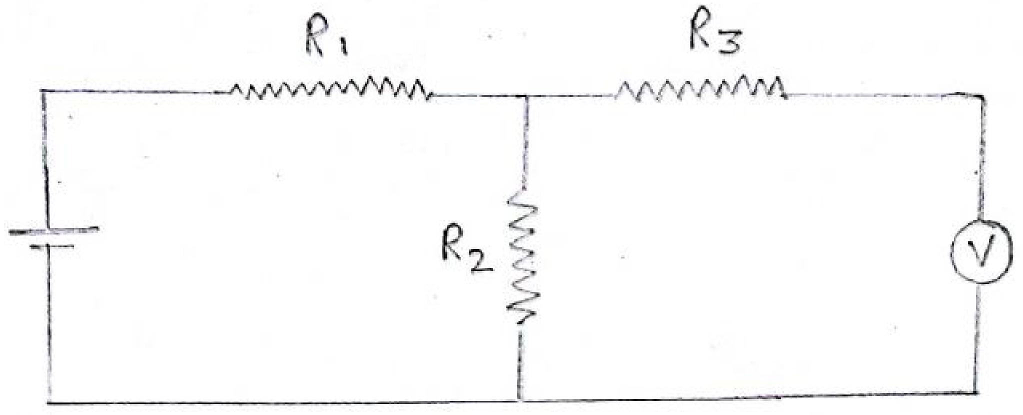
7) Reconnect the Power supply and note ~~down~~ the down the load current I_L. with a load resistance of 25 Ω, 50 Ω and 100 Ω respectively.

8) Swith OFF the Power supply and disconnect the circuit.

P.T.O

Calculation on the basis of theoritical value :

Step I :

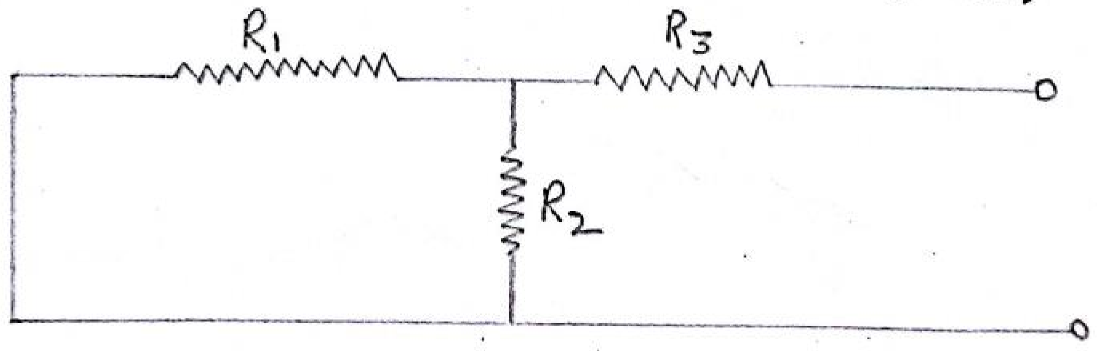


here, $R_1 = 50\Omega$, $R_2 = 100\Omega$, $R_3 = 900\Omega$; $E = 12V$ DC

Now from thevenin's theoram we can say :

$$V_{TH} = \frac{E \times R_2}{R_1 + R_2} = \frac{12 \times 100}{500 + 100} = 2V$$

Step II: Calculation of thevenin's Resistance (R_{TH})



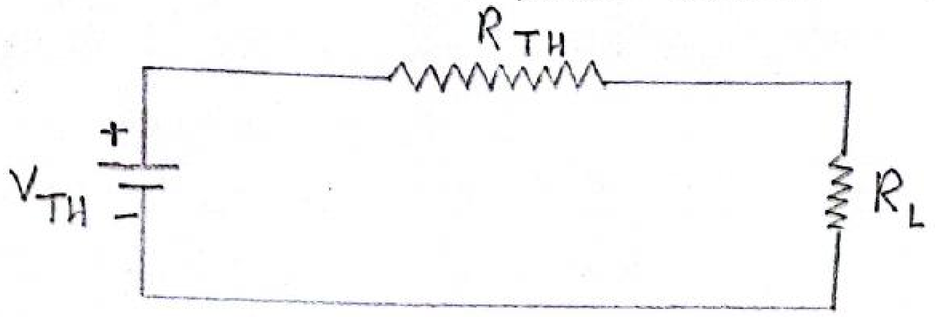
$$R_{TH} = (R_1 || R_2) + R_3$$

$$= \frac{500 \times 100}{500 + 100} + 90$$

$$= 173.33\Omega .$$

P. T. O

Step III: Calcucation of load current (I_L):



$R_1 = 25\Omega$, $R_2 = 50\Omega$, $R_3 = 100\Omega$.

Now, from thevenin's theorm

$$I_L = \frac{V_{TH}}{R_{TH} + R_L}$$

$$\therefore I_{L_1} = \frac{2}{173.33+25} = 0.010\,A = 10\,mA$$

$$I_{L2} = \frac{2}{173.33+50} = 8.95 \times 10^{-3} A = 8.96\,mA$$

$$I_{L3} = \frac{2}{173.33+100} = 7.31 \times 10^{3} A = 7.31\,mA.$$

Calculation:

Thevnin's voltage = 1.9 Volt.

1st reading: Equivalent resistance = 174.2 Ω
Load resistance = 25 Ω
Observed (I_{Lo}) = 10 mA
Calculated (I_{Lc}) = 10.08 mA

$$\therefore \text{Error} = \frac{I_{Lc} - I_{Lo}}{I_{Lc}} \times 100\% = 0.007\%$$

P.T.O

2nd reading :

Equivalent resistance = 174.2 Ω

Load resistance = 50 Ω

Observed (I_{Lo}) = 8.8 mA

Calculated (I_{Lc}) = 8.96 mA

$\therefore$ Error = $\dfrac{I_{Lc} - I_{Lo}}{I_{Lc}} \times 100\%$ = 0.018%

3rd reading :

Equivalent resistance = 174.2 Ω

Load resistance = 100 Ω

Observed (I_{Lo}) = 7.3 mA

Calculated (I_{Lc}) = 7.31 mA

$\therefore$ Error = $\dfrac{I_{Lc} - I_{Lo}}{I_{Lc}} \times 100\%$ = 0.013%

P. T. O

OBSERVATION TABLE :

$R_1 = 500\ \Omega$; $R_2 = 100\ \Omega$; $R_3 = 90\ \Omega$;

Sl No.	Thevenin's Voltage (volt)	Equivalent Resistance (Ω)	Load Resistance (Ω)	Load Current I_{Lo} (mA).
1			25	10
2	1.9	174.2	50	8.8
3			100	7.3

CALCULATION TABLE :

Calculated Thevenin's Voltage = 1.9 Volt.

Calculated Equivalent Resistance = 174.2 Ω ;

Sl. No.	Load Resistance R_L (Ω)	Load Current		Error
		Observed Value I_{Lo} (mA)	Calculated Value I_{Lo} (mA)	$\frac{I_{Le} - I_{Lo}}{I_{Lo}} \times 100\%$
1	25	10	10.08	0.007
2	50	8.80	8.96	0.018
3	100	7.30	7.31	0.013

[illegible] 02/01/15

RESULT :

Thus the Thevenin's theorem is verified.

DISCUSSION :

1). Can we apply the Thevenin's Theorem to AC Circuit ?

Yes, we can apply Thevenin's Theorem to AC Circuit.

2). Can this theorem be applied to network which contains non-linear resistance ?

No, this theorem cannot be applied to circuits containing non-linear resistances.

[illegible] 09/04/15

TITLE: VERIFICATION OF SUPERPOSITION THEOREM.

OBJECTIVE: To verify the Superposition's Theorem in the DC circuit.

APPRATUS:

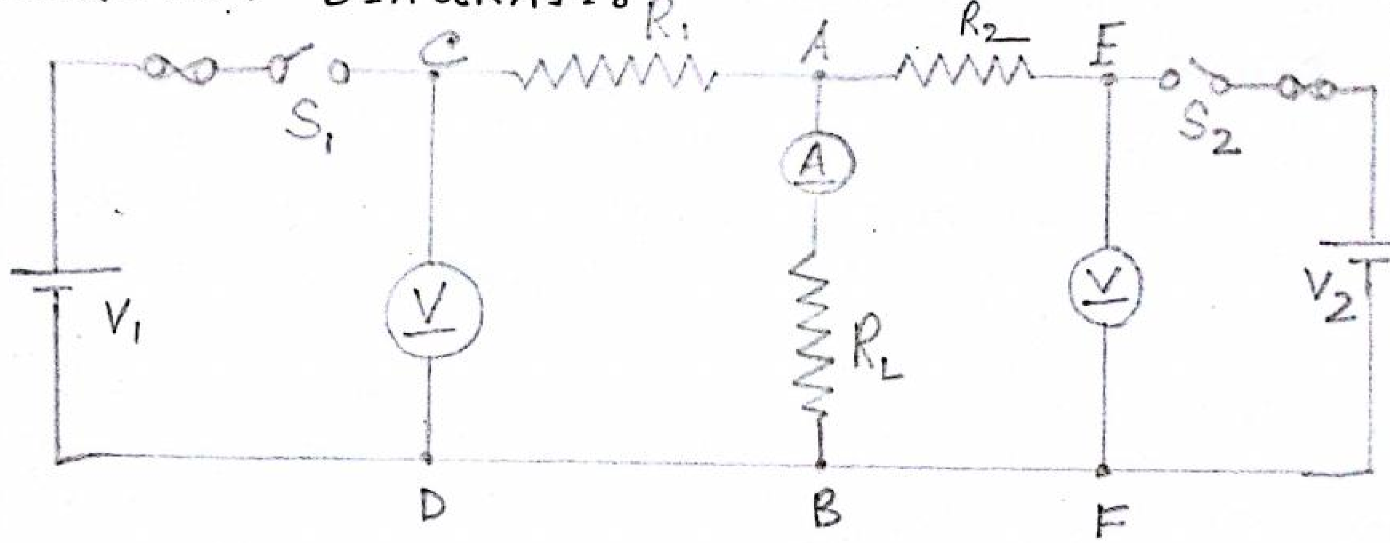
Sl No.	Apparatus Name	Apparatus Type	Range
1.	Trainer Kit	—	—
2.	Voltage Source	DC	12V, 5V
3.	Resistor 1,2 & 3	—	50Ω, 50Ω, 10Ω
4.	Ammeter	DC	0-250mA
5.	Voltmeter 1	DC	0-5 V
6.	Voltmeter 2	DC	0-5 V

THEORY:

Superposition Theorem as applied for DC circuit maybe stated as:

In any linear active bilateral network containing several sources, the current through or voltage across any branch in the network equals the algebraic sum of the current or voltage of each individual source considered separately with all other sources made inoperative, i.e replaced by resistance equal to their internal resistance.

CIRCUIT DIAGRAM:



PROCEDURE:

1. Firstly I connect the circuit.
2. Next I carefully measured the value of R_1, R_2 and R_L
3. Now, I switch ON the power supply by closing switch S_1 & S_2.
4. Next, I Note down the total current (I_L) flowing through resistance R_L due to both the source is measured.
5. Then I replace the source V_1 by its Internal resistance. If internal resistance is zero it is shorted C & D. Switch ON the Power supply by closing switch S. Note down load current I_{L1} through resistance R_L due to the source V_1.

6. Next I reconnect the source V_1 and replace the source V_2 by its internal resistance is zero. it is shorted E & F. Switch ON the power supply by closing switch S_2. Note down the load current I_{L2} through the resistance R_L due to the source V_L.

7. Finally, I switch off the power supply and disconnect the circuit.

[P.T.O]

OBSERVATION TABLE:

$V_1 = 12V$ Volt, $V_2 = 5V$ Volt,

$R_1 = 50\ \Omega$ $R_2 = 50\ \Omega$, $R_L = 10\ \Omega$

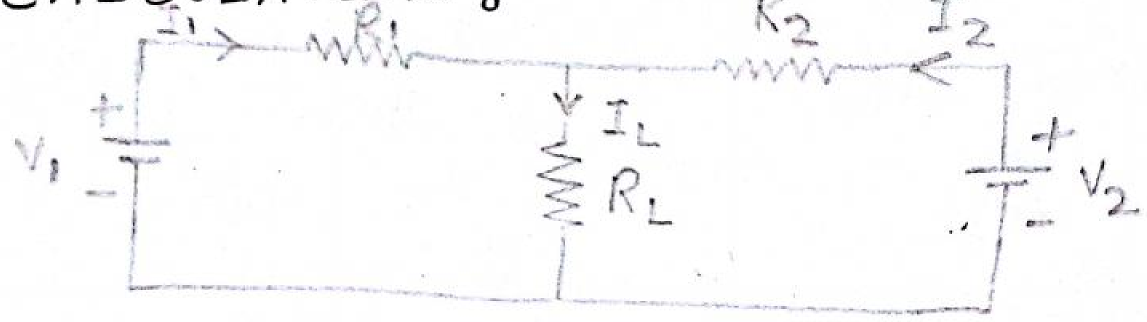
Condition	Measured Value (Me)	
	Load Voltage (Volt)	Load Current (mA)
Both V_1 and V_2 Present	V_L	I_L
	2.4	225
V_1 Present and V_2 Replace by internal resistance	V_{L1}	L_{L1}
	1.7	115
V_2 Present and V_1 Replace by internal resistance	V_{L2}	V_{L2}
	0.7	70
Algebric Sum.	$V_L = V_{L1} + V_{L2}$	$(L_L = L_{L1} + L_{L2})$
	2.4	225

CALCULATION TABLE:

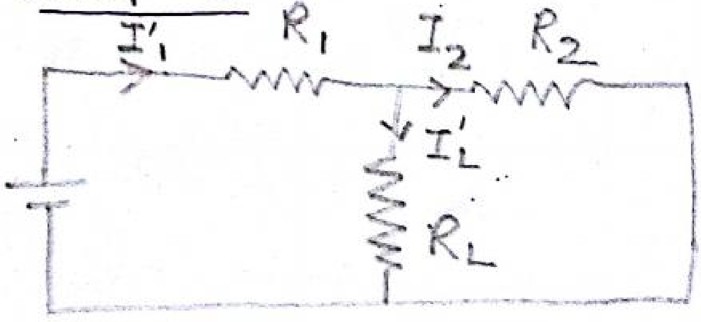
Condition	Load Voltage (Volt)		Error
	Measured Value	Calculated Value	$\frac{Th-me}{Th} \times 100\%$
V_1 Present and V_2 replace by internal resistance	V_{L1m}	V_{L1C}	0.58
	1.7	1.71	
V_2 Present and V_1 replace by internal resistance	V_{L2m}	V_{L2C}	1.12
	0.7	0.708	
Both V_1 and V_2 Present	$V_{Lm} = V_L$	$V_{LC} = V_{L1C} + V_{L2C}$	0.744
	2.4	2.418	

Condition	Load Current (mA)		Error
	measured Value	Calculated value	$\frac{Th - me}{Th} \times 100\%$
V_1 Present and V_2 replace by internal resistance	I_{L1m}	I_{L1c}	9.5
	155	171	
V_2 Present and V_1 replace by internal resistance	I_{L2m}	I_{L2c}	0
	70	70	
Both V_1 & V_2 Present (Algebraic Sum)	$I_{Lm} = I_L$	$I_{Lc} = I_{L1c} + I_{L2c}$	6.63
	225	241	

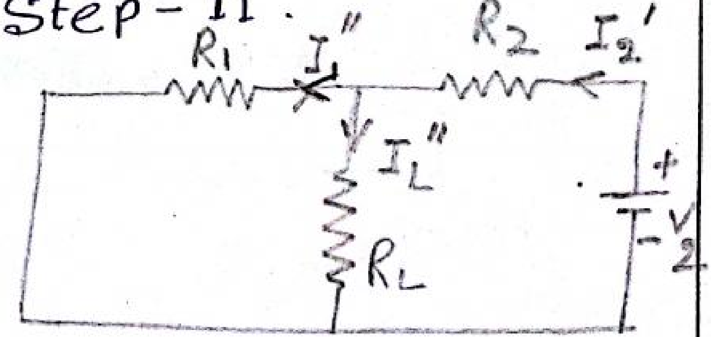
CALCULATION:



Step - I



Step - II.



Req resistance $= R_1 + (R_2 \parallel R_L)$
$= 50 + (10 \parallel 50)$
$= 50 + \left(\frac{10 \times 50}{10+50}\right)$
$= 58.33 \Omega$

Req Resistance $= R_2 + (R_1 \parallel R_L)$
$= 50 + (50 \parallel 10)$
$= 50 + \left(\frac{50 \times 10}{50+10}\right)$
$= 58.33 \Omega$

$I_1' = \frac{V_1}{\text{req resistance}}$
$= \frac{12}{58.33} = 206 \text{ mA}$
$= 0.206 A$

$\therefore I_2' = \frac{V_2}{\text{req resistance}}$
$= \frac{5}{58.33} = 85.71 \text{ mA}$
$= 0.08571 A.$

CALCULATION:

$$I_2' = I_1' \times \frac{R_1}{R_2 + R_L}$$

$$= 206 \times \frac{10}{50+10}$$

$$= 34.33\ mA$$

$$= 0.0343\ A$$

$$\therefore I_L' = I_1' \times \frac{R_L}{R_2 + R_L}$$

$$= 0.206 \times \frac{50}{50+10}$$

$$= 0.171\ A$$

$$= 171\ mA$$

$$\therefore V_L = I_L' \times R_L$$

$$= 0.171 \times 10$$

$$= 1.71\ V$$

$$I_1'' = I_2' \times \frac{R_L}{R_1 + R_L}$$

$$= 85.71 \times \frac{10}{50+10}$$

$$= 14.28\ mA$$

$$= 0.01428\ A.$$

$$\therefore I_L'' = 0.085 \times \frac{50}{50+10}$$

$$= 0.0708\ A$$

$$= 70\ mA.$$

$$V_L = I_L'' \times R_L$$

$$= 0.0708 \times 10$$

$$= 0.708\ V.$$

DISCUSION :-

1. Can we apply the superposition theorem to AC circuit?

⇒ Yes, The only varition in applying the principle of superposition to the ac networks with independent source in that the current ~~Sources and~~ Impedances involving phase instead of just real numbers. (operation with resistor).

2. Does the Non linear system obey the superposition theorem. Explain it ?

Ans: Any system that obeys the superposition theorem is by defination, a linear system, So, a non-linear system is one that does not obey the superposition theorem.

CONCLUSION:

Superposition theorem as applied for DC circuit as well as AC circuit. Then we determine the error by using formula $\frac{Th - Me}{Th} \times 100$ and we get in superposition theorem (6.63-9.35)% error for load current and 0.58%-1.12% error for load voltage. At least we can verify the superposition theorem.

TITLE: VERIFICATION OF NORTON'S THEOREM.

OBJECTIVE: To verify the Norton's Theorem in the DC Circuit.

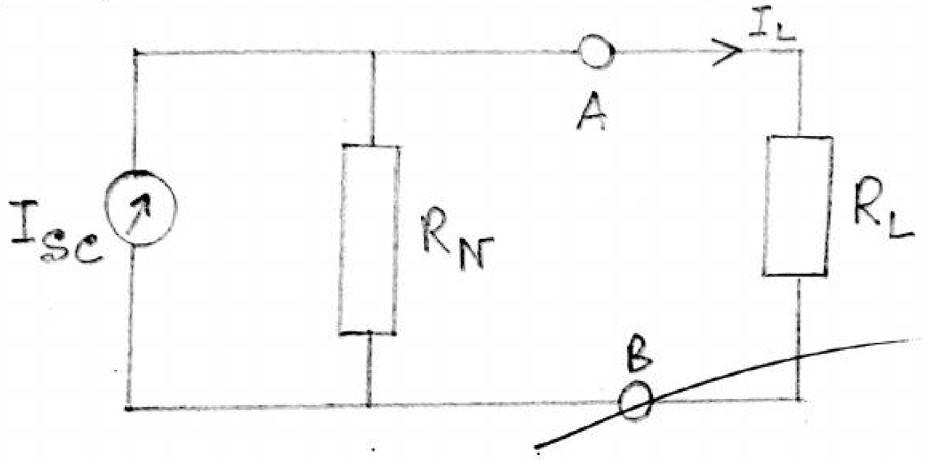
APPRATUS:

Sl. NO	Apparatus Name	Apparatus Type	Range
1.	Trainer kit	—	—
2.	Voltage Source	DC	12V
3.	Resistor 1,2,3&4	—	1000Ω, 2000Ω 500Ω, 2000Ω
4.	Ammeter	DC	0 - 10 mA
5.	Multimeter	Digital	0 - 200Ω

THEORY: Norton's Theorem as applied for DC circuit may be stated as:

Any two terminal linear, active, bilateral networks containing voltage source and resistance when viewed from its output terminals is equivalent to a constant current source and a parallel connected equivalent resistance. The constant current source is of magnitude of the short circuit current at the terminals. The internal resistance is equivalent resistance of the network —

looking back into the terminal with all the sources replaced by their internal resistance.



Mathematically, current through the load resistance R_L is given by the equation

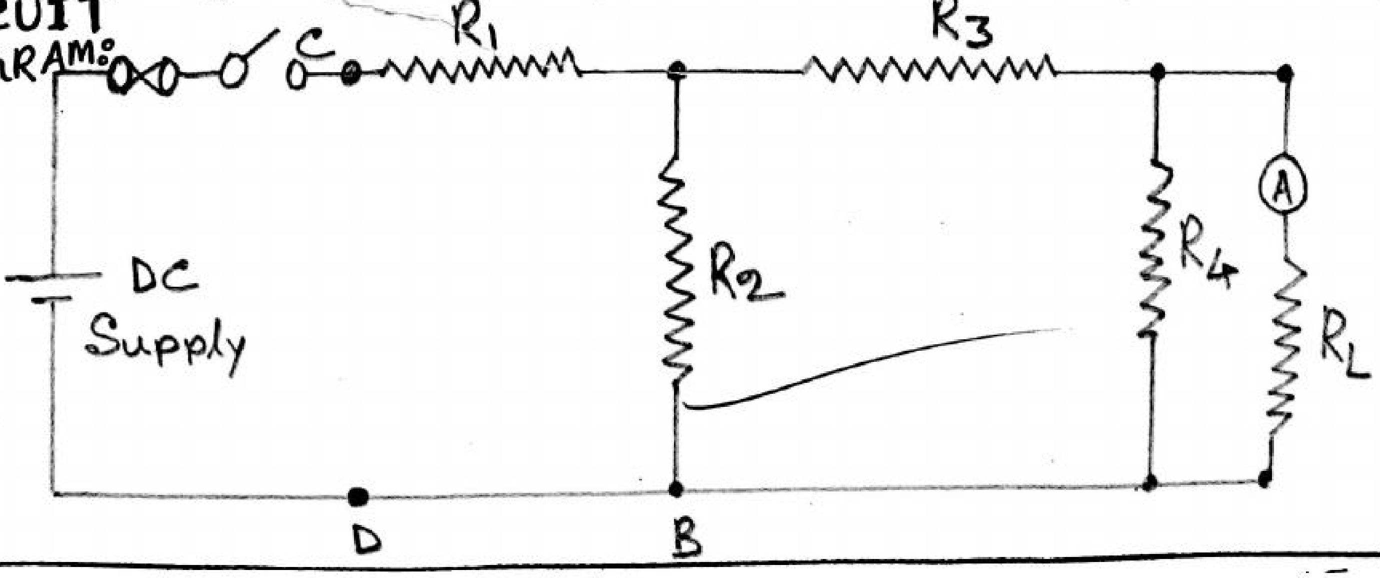
$$I_L = I_{sc} \frac{R_N}{R_N + R_L}$$

where, I_L = Load current R_N = Norton's Resistance

I_{sc} = short circuit across the terminals.

R_L = Load Resistance.

CIRCUIT DIAGRAM:



PROCEDURE :

1) Firstly I connect the circuit.

2) Next I carefully measured the value of R_1, R_2, R_3 & R_4.

3) Now, Correspondingly I remove the R_L and short the line.

4) Afterthat I swite ON the Power supply and note down ammeter reading as short circuit current ($I_{SC} = I_N$).

5) Now, I remove the voltage source by replacing their internal resistance. If the internal resistance is assumed to be zero, then short the terminal C & D.

6) Then I measure the R_N across by opening the terminal EF by multimeter or ammeter-Voltmeter method.

7) Next I note down the load current I_{Lo} with a load resistance of 25Ω, 50Ω and 100Ω respectively and compare with calculated values of I_{Lc}. Also calculated the error ~~of~~ for each load.

8) Finally I switch OFF the Power supply and disconnect the circuit.

P.T.O

OBSERVATION TABLE:

$R_1 = 1K\Omega$, $R_2 = 2K\Omega$, $R_3 = 500\ \Omega$, $R_4 = 2K\Omega$.

Sl No.	Norton's current (mamp)	Equivalent Resistance (Ω)	Load Resistance (Ω)	Load Current I_{Lo} (mA).
1	6.8	740	25	6.4
2			50	6.2
3			100	5.8

CALCULATION TABLE:

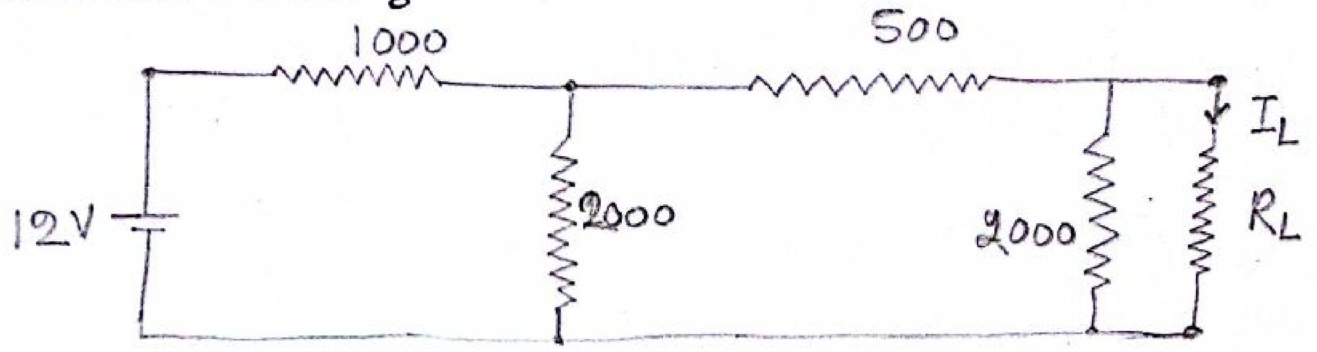
Calculated Norton's Current $= 6.85 \times 10^{-3}$ A

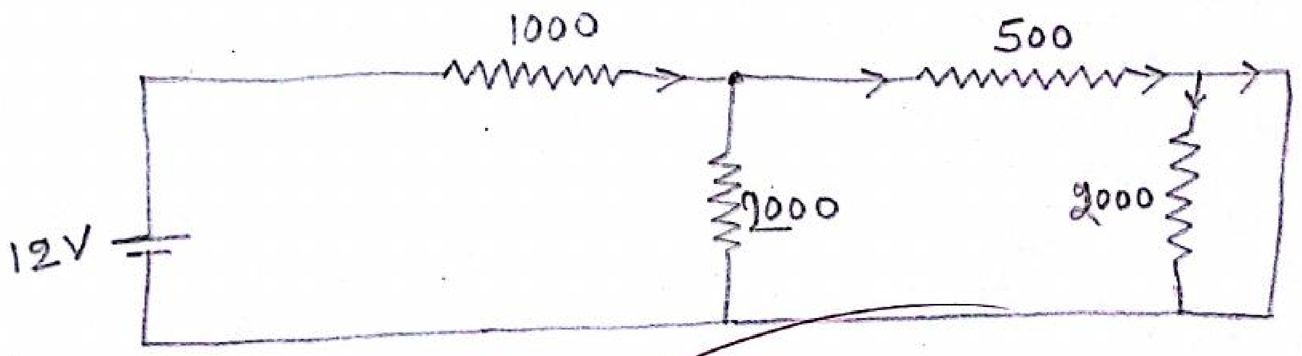
Calculated Equivalent Resistance $= 736.8\ \Omega$.

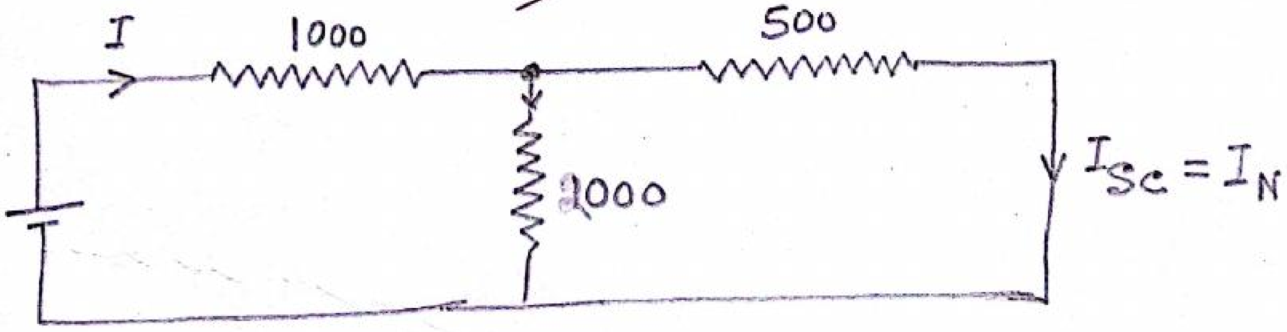
Sl. No.	Load Resistance R_L (Ω)	Load Current		Error
		Observed Value I_{Lo} (mA)	Calculated Value I_{Lc} (mA)	$\frac{I_{Lc} - I_{Lo}}{I_{Lo}} \times 100\%$
1	25	6.4	6.62	3.32
2	50	6.2	6.41	3.27
3.	100	5.8	6.03	3.81

[illegible] 29/01/15

CALCULATION :





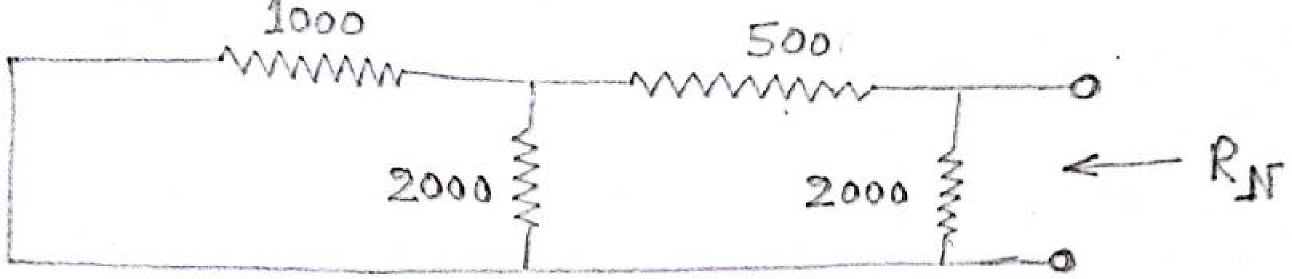


$$R_{eq} = (500 \parallel 2000) + 1000$$
$$= \left(\frac{500 \times 2000}{500 + 2000}\right) + 1000$$
$$= 400 + 1000 = 1400\,\Omega$$

$$I = \frac{12}{1400} = 8.57 \times 10^{-3}\,A.$$

CALCULATION:

$$I_N = I_{sc} = \frac{8.57 \times 10^{-3} \times 2000}{2000 + 500} = 6.85 \times 10^{-3} A$$



$$R_N = \{(1000 \parallel 2000) + 500\} \parallel 2000$$

$$= \{(\frac{1000 \times 2000}{1000 + 2000}) + 500\} \parallel 2000$$

$$= 1166.67 \parallel 2000 = \frac{1166.67 \times 2000}{1166.67 + 2000} = 736.83 \Omega.$$

$$I_L = I_{sc} \frac{R_N}{R_N + R_L}$$

$$I_{L1} = 6.85 \times 10^{-3} \times \frac{736.83}{736.83 + 25} = 6.62 \times 10^{-3} = 6.62 mA$$

$$I_{L2} = 6.85 \times 10^{-3} \times \frac{736.83}{736.83 + 50} = 6.41 \times 10^{-3} = 6.41 mA$$

$$I_{L3} = 6.85 \times 10^{-3} \times \frac{736.83}{736.83 + 100} = 6.03 \times 10^{-3} = 6.03 mA.$$

Error:

Step 1: $\frac{I_{Lc} - I_{Lo}}{I_{Lc}} \times 100\% = \frac{6.62 - 6.4}{6.62} \times 100\% = 3.32$

Step 2:

$$\frac{I_{Le} - I_{Lo}}{I_{Le}} \times 100\% = \frac{6.41 - 6.2}{6.41} \times 100\% = 3.27$$

Step 3:

$$\frac{I_{Le} - I_{Lo}}{I_{Le}} \times 100\% = \frac{6.03 - 5.8}{6.03} \times 100\% = 3.81.$$

RESULT: Thus the Norton's theorem is verified.

DISCUSSION:

1.) Can we apply the Norton's Theorem to AC circuit?

Ans.: Yes, we can apply Norton's Theorem to AC circuits.

2.) Can this theorem be applied to network which contains non-linear resistance?

Ans.: No, this theorem cannot be applied to networks containing non-linear resistance.

CONCLUSION :

From the experiment mathematically current through the load resistance R_L is given by the equation -

$$I_L = I_{sc} \frac{R_N}{R_N + R_L}$$

where, I_L = Load ~~resistance~~ current.

I_{sc} = short circuit current across the terminals.

R_N = Norton's Resistance.

R_L = Load Resistance.

By this way, we can the Norton's voltage and R_N and then we find the current through the load resistance. Then we determine the error by using formula $\frac{I_{Le} - I_{Lo}}{I_{Le}} \times 100\%$ and atleast we can verity the Norton's Theorem.

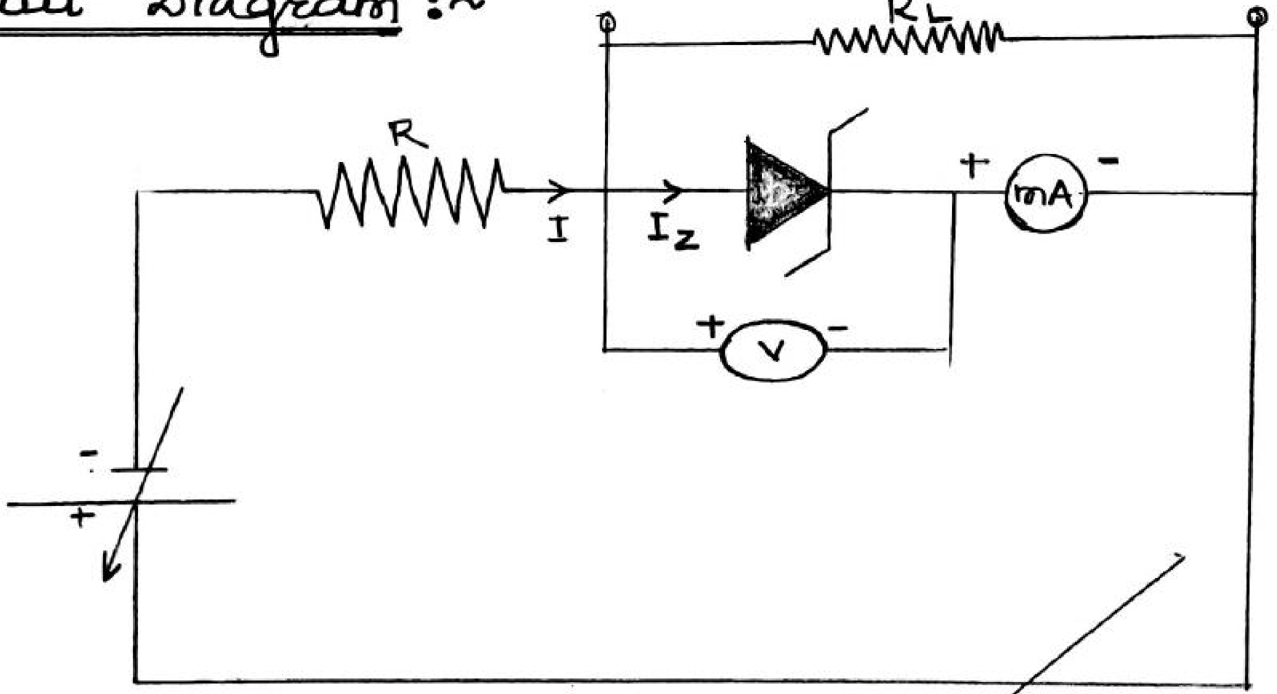
[THE END]

Expt :-4 :~ To study V-I characteristics of Zener Diode.

*Objective :~ a) study V-I characteristics of zener Diode
b) determine the Zener Breakdown voltage.

* Apparatus Required :~ 1. Bread Board
2. Regulated Power Supply (variable).
3. Voltmeter.
4. Milli-ammeter

* Circuit Diagram :~



*. Theory :- Zener diode is heavily doped designed to operate breakdown design. zener diode is generally operated in reverse biased condition. when reverse bias is increased. a value is reached at which current increases greatly from its normal cut off value. This voltage is called zener breakdown voltage.

* **Zener Breakdown :~** Zener breakdown take place when junction is very thin when both side of junction are very heavily doped and conse-quently the depletion layer of junction is narrow when a small reverse biased voltage is app-lied a very strong electric field (about 10^7 V/m) is setup across the thin depletion layer This field is enough to break or rupture the cov-alent bond. Now extremely large number of electrons and holes are produced which cons-titute the reverse saturation current. Zener current is independent of the applied volt-age and depends only on the external res-istance.

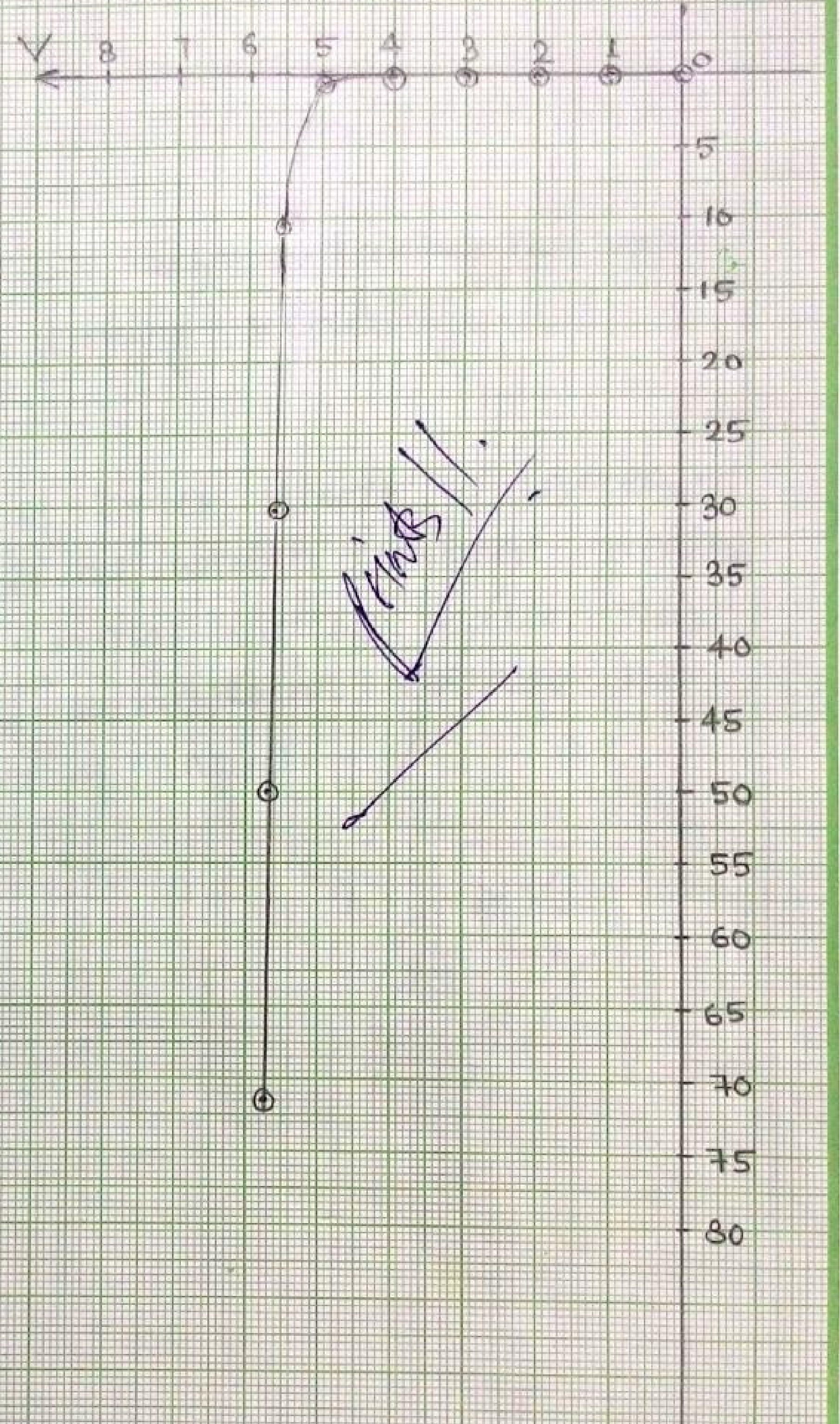
* **Observation Table :~** (For Reverse Bias)

SL.No.	Voltage (V)	Zener Voltage (V_Z)	Zener Current I_Z (mA)
1.	0.00	0	0
2.	1.00	-1.01	0
3.	2.00	-2.01	0
4.	3.00	-3.00	0
5.	4.00	-4.00	0
6.	5.00	-5.00	-0.03
7.	6.00	-5.51	-10.6
8.	7.00	-5.57	-30.5
9.	8.00	-5.61	-50.8
10.	9.00	-5.64	-70.9

* **Calculation :~**

V-I characteristics of zener-diode

Scale
Along X-axis,
10 small div. = 1V
Along Y-axis,
10 small div = 5 mA



$I = I_1 + I_z$

$$\frac{9-5.57}{100} = \frac{5.57}{R_L} + 30.5\times10^{-3}\ \Omega$$

$R_L = 1465.78\ \Omega$

* **Precaution :~**

- check the connection before taking the reading
- Be careful during taking readings.

* **Conclusions :~**

When reverse bias of zener-diode is increased a value is reached at which current greatly increases from its normal cut-off value. This value is the Zener Breakdown voltage i.e 5.57V at 7.0 volt of input.

[illegible]
29/02/15

Expt-3 :- To Study V-I Characteristics of Junction Diode :-

** OBJECTIVE :-

a) Study - V-I characteristics of Junction Diode

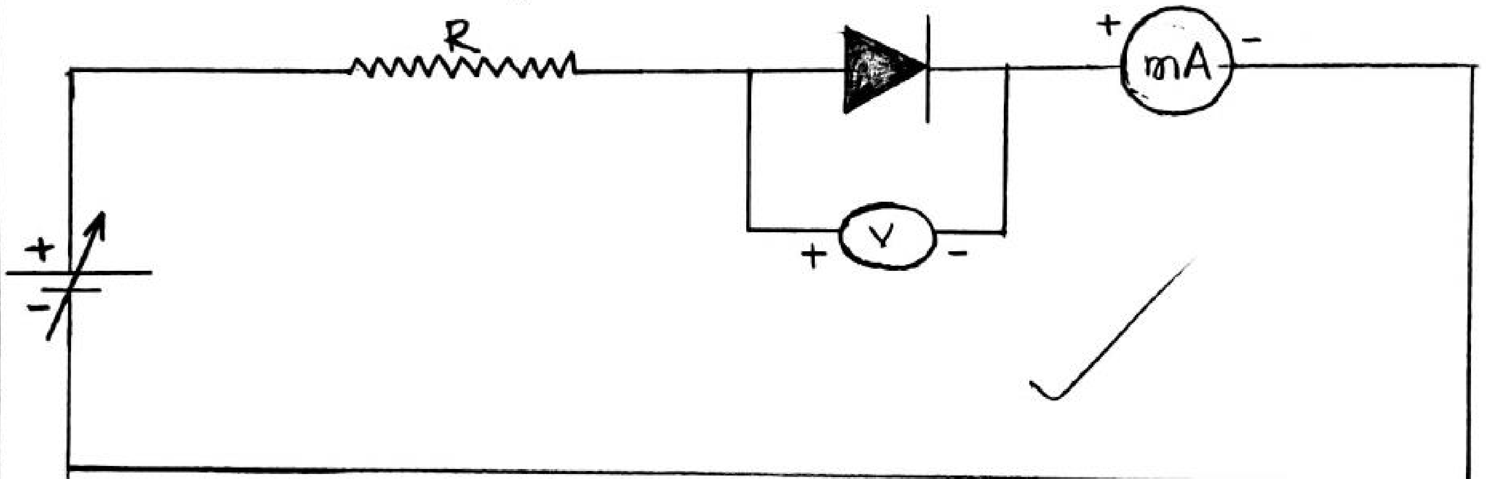
b) Determine the cut-in voltage.

** Apparatus Required :-

1. Bread Board 2. Voltmeter 3. Milli-ammeter

4. Regulated Power supply (variable)

** Circuit Diagram :-



** Theory :-

Diode conducts in forward biased condition. Forward bias means connecting P side or anode of diode with +ve terminal of battery and N side or cathode of diode with negative terminal of battery. The forward biased voltage at which a diode starts conducting is called cut-in-voltage.

But in reverse biased condition as the diode blocks the current flow, only a few microampere current called reverse saturation current will flow through p-n junction diode.

$$R_f = \frac{V_2 - V_1}{I_2 - I_1} = \frac{0.742 - 0.74}{(30-29)\times 10^{-3}} \Omega = 0.002 \times 10^{3} \Omega$$

$R_f = 2\,\Omega$

** Observation Table:-(For Forward Bias)

SL. No	Voltage (V)	Current (mA)
1.	0.1	0
2.	0.2	0
3.	0.3	0
4.	0.4	0
5.	0.5	0.2
6.	0.52	0.3
7.	0.54	0.4
8.	0.56	0.7
9.	0.58	1.1
10.	0.60	1.6
11.	0.62	2.1
12.	0.64	2.9
13.	0.66	5.4
14	0.68	7.6
15.	0.70	12.1
16.	0.72	17.6
17.	0.74	29.0
18.	0.76	46.3
19.	0.78	73.4
20.	0.80	126.8

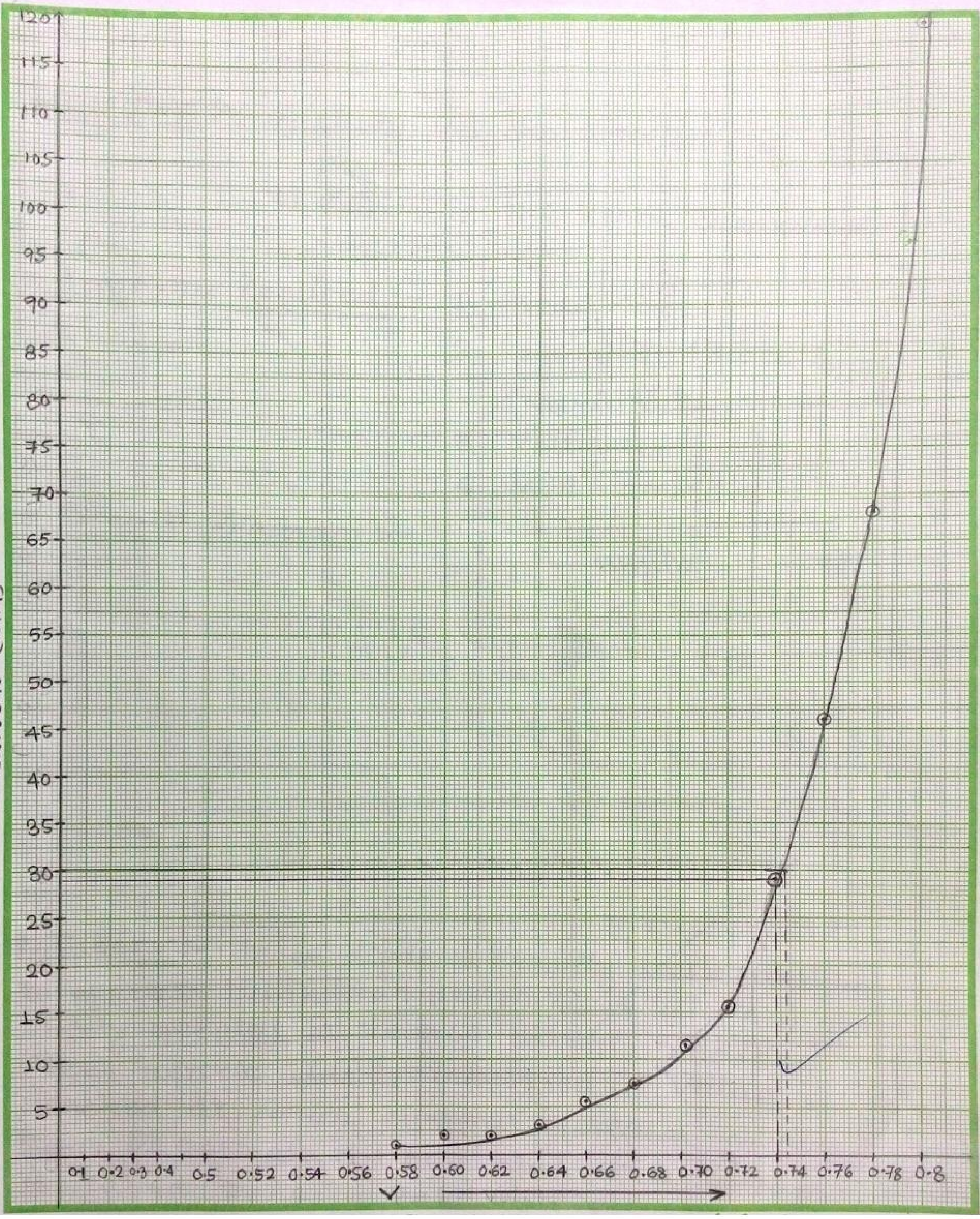
✓

** Precautions:-

1. check the connection before taking the readings.
2. Be careful during taking the readings.

** Conclusion :-

In the V-I characteristics of junction diode shows that from 0.58 V, the current starts to increase. The forward bias resistance is $2\,\Omega$. The forward bias voltage at which a diode starts conducting is 0.58 V.


current (mA) ⟶
120
115
110
105
100
95
90
85
80
75
70
65
60
55
50
45
40
35
30
25
20
15
10
5
0·1
0·2
0·3
0·4
0·5
0·52
0·54
0·56
0·58
0·60
0·62
0·64
0·66
0·68
0·70
0·72
0·74
0·76
0·78
0·8
V ⟶

Expt ÷ 7:~ To study the Input and Output characte-ristics curve for common Emitter and Common Base configuration.

* Objective :~ Study input and output characteristics of CE and CB Mode.

* Apparatus Required :~ 1. Bread Board
2. Regulated Power Supply (Variab
3. Voltmeter
4. Ammeter (mA, μA)

* Circuit Diagram :-

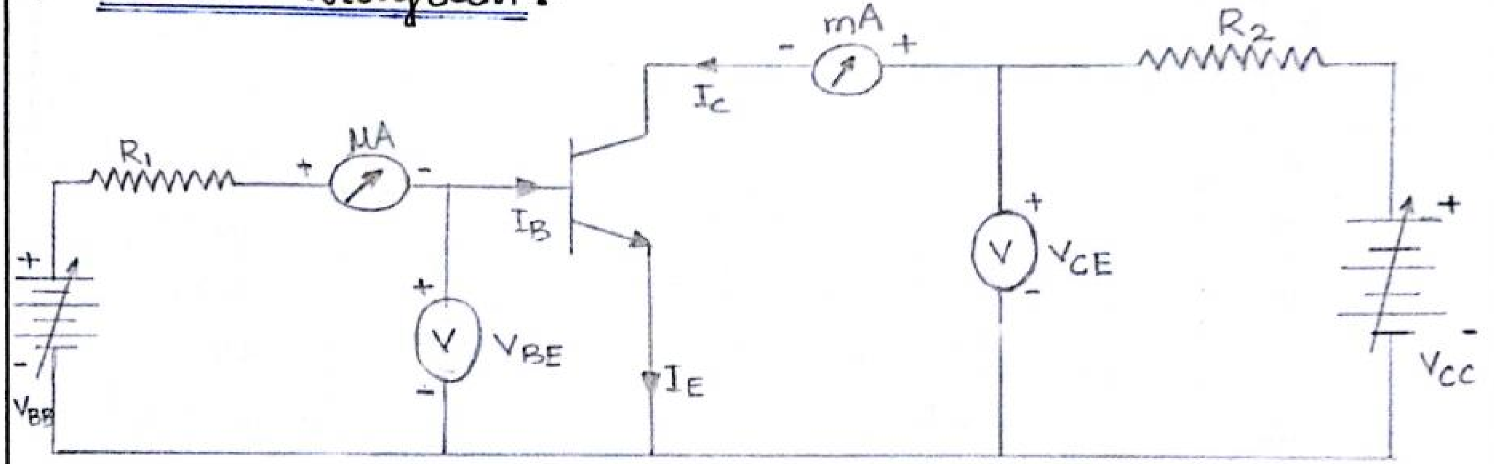


Fig: Common Emitter amplifier circuit

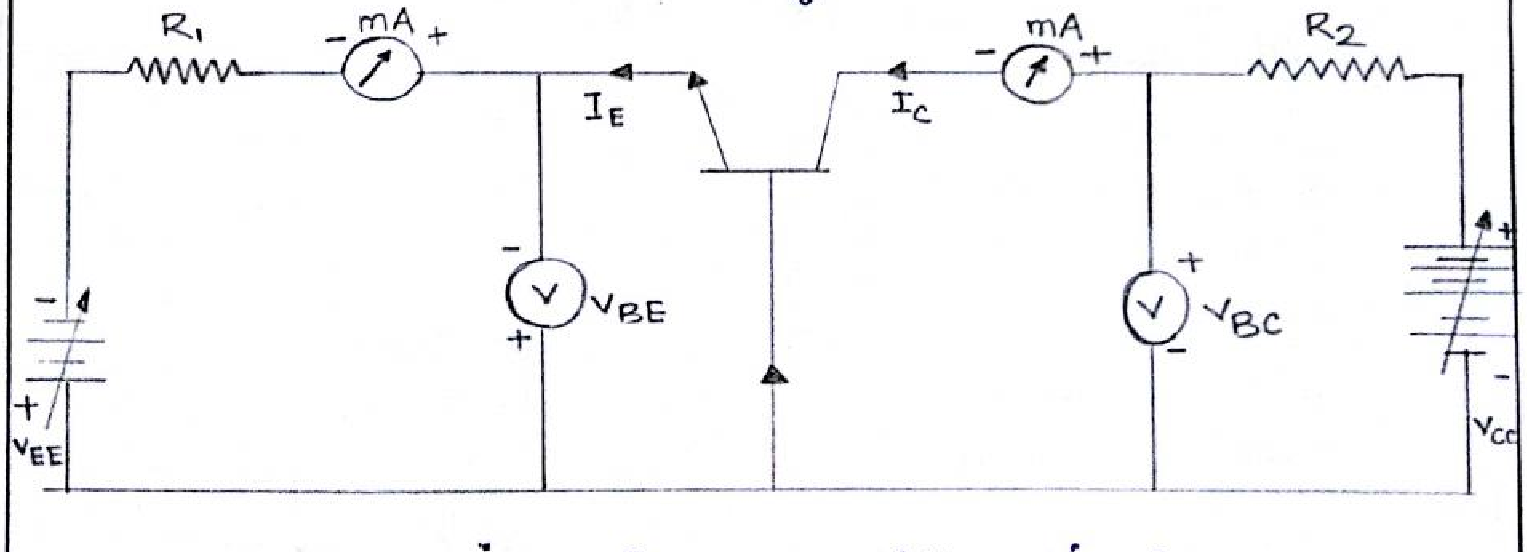


Fig:- common Base amplifier circuit

* Theory :- In CE and CB configuration Input current and output voltage are taken as independent vari-ables, whereas input voltage and output current are taken as dependent variables.

* OBSERVATION TABLE :- (For CE mode)

Input characteristics

$V_{CE} = 5V$		$V_{CE} = 3V$	
Voltage (V_{BE}) V	Current (I_B) μA	Voltage (V_{BE}) V	Current (I_B) μA
0	0	0	0
0.1	0	0.1	0
0.2	0	0.2	0
0.3	0	0.4	0
0.4	0	0.5	0
0.5	0	0.56	1
0.56	1	0.58	3
0.58	3	0.59	5
0.60	7	0.60	6
0.62	15	0.62	11
0.64	28	0.64	33

→ Output characteristics :-

$I_B = 20$ μA		$I_B = 40$ μA	
Voltage (V_{CE}) V	Current (I_c) μA	Voltage (V_{CE}) V	Current (I_c)
0	0	0	0
0.05	0.40	0.10	2.06
0.10	1.39	0.15	4.58
0.12	1.72	0.20	6.23
0.15	2.26	0.24	7.04
0.20	2.65	0.30	7.40
0.30	2.75	0.35	7.44
0.40	2.76	0.40	7.45
		0.45	7.46
		0.50	7.47

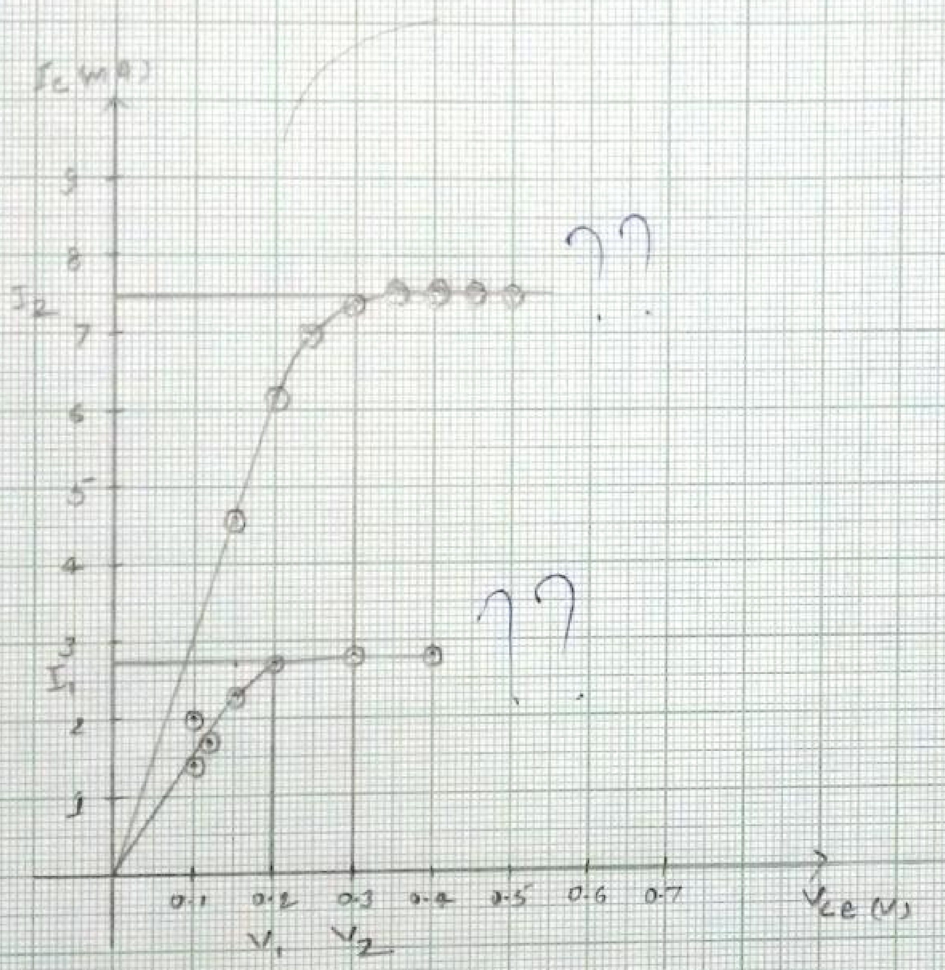
* Calculation :-

** OBSERVATION TABLE :- [For CB mode] Output characteri-stics.

$I_E = 10\,mA$		$I_E = 20\,mA$	
Voltage (V_{CB})	Current (I_C)	Voltage (V_{CB})	Current (I_C)
−0·76	0·9	−0·79	0·8
−0·70	4·6	−0·75	4·4
−0·65	7·4	−0·70	9·1
−0·60	9·1	−0·65	13·4
−0·55	9·8	−0·60	16·6
−0·50	9·8	−0·55	18·6
		−0·50	19·5
		−0·45	19·8
		−0·40	19·8

✓

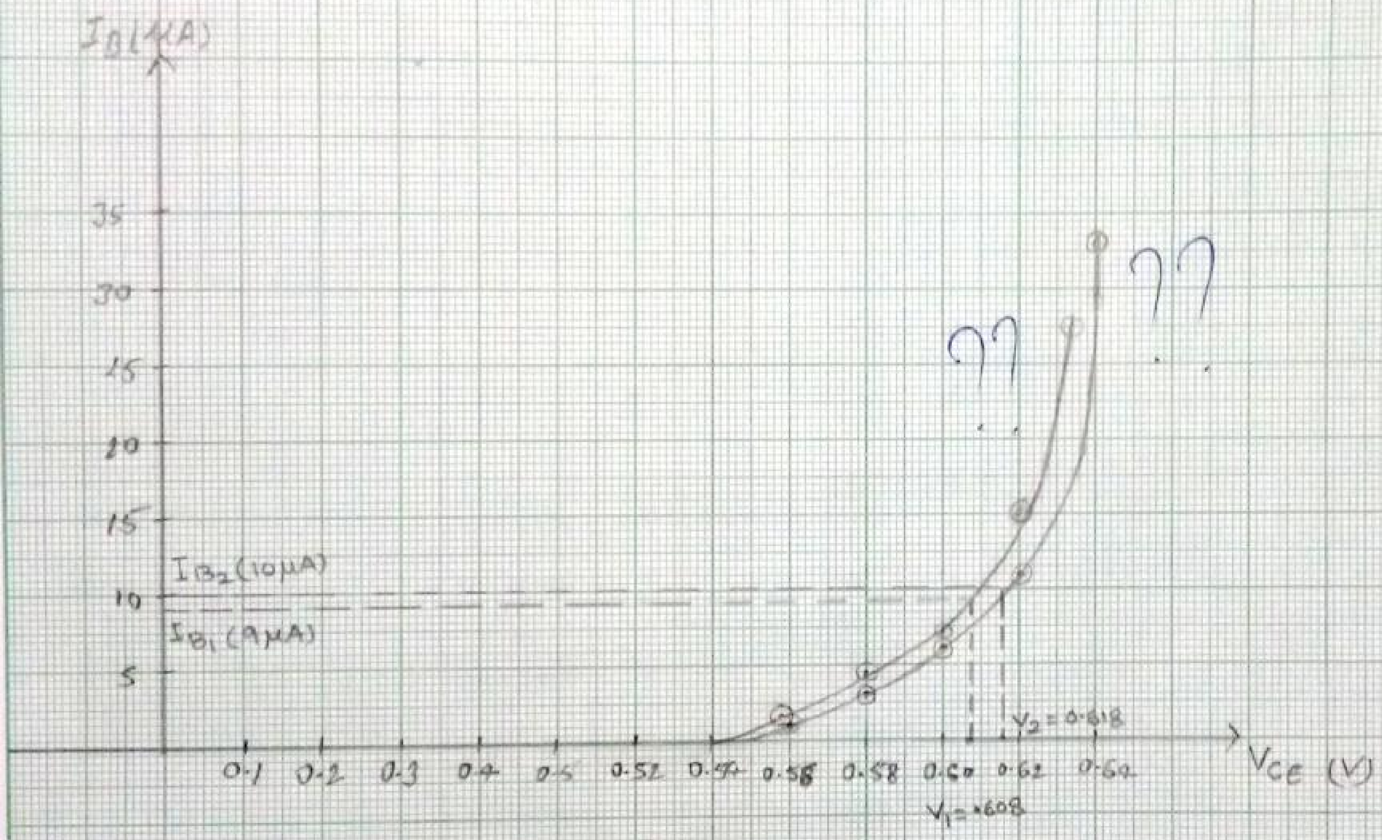
Scale : Along X-axis
1 small div = 0.001 V
Along Y-axis :
1 small div = 0.1 mA

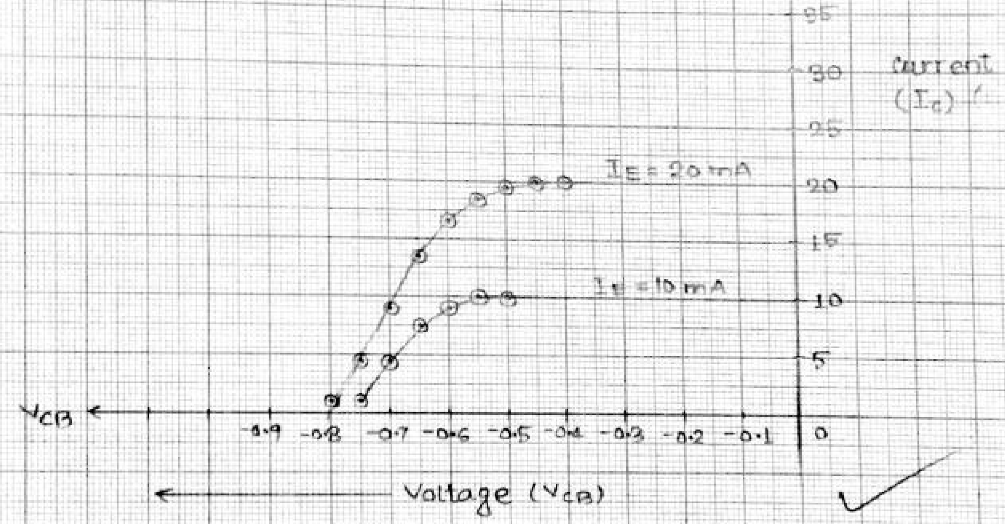


✓

Scale:- Along X axis
1 small div = 0.02 V

Along Y- axis
1 small div = 0.1 μA





Output characteristics of COMMON BASE Mode.

* Calculation :-

$$h_{ie} = \frac{\Delta V_{BE}}{\Delta I_B} = \frac{V_2 - V_1}{I_{B_2} - I_{B_1}} = \frac{0.618 - .608}{(10-9)\times 10^{-6}} = 10\ K\Omega$$

$$h_{fe} = \frac{\Delta I_c}{\Delta I_B} = \frac{I_{c_2} - I_{c_1}}{I_{B_2} - I_{B_1}} = \frac{(7.5-2.8)\times 10^{-3}}{(40-20)\times 10^{-6}} = \frac{4.7\times 10^{3}}{20} = 235$$

$$h_{re} = \frac{\Delta V_{BE}}{\Delta V_{CE}} = \frac{V_2 - V_1}{V_{CE_2} - V_{CE_1}} = \frac{0.618 - 0.608}{5-3} = 0.005$$

$$h_{oe} = \frac{\Delta I_c}{\Delta V_{CE}} = \frac{I_{c_2} - I_{c_1}}{V_2 - V_1} = \frac{7.5-2.8}{0.3-0.2} = 4.7\ m\Omega^{-1}$$

$$= 0.047\ mho$$

* Precautions :~

- check the connection before taking the readings.
- Be careful during taking the reading.

* CONCLUSION :~

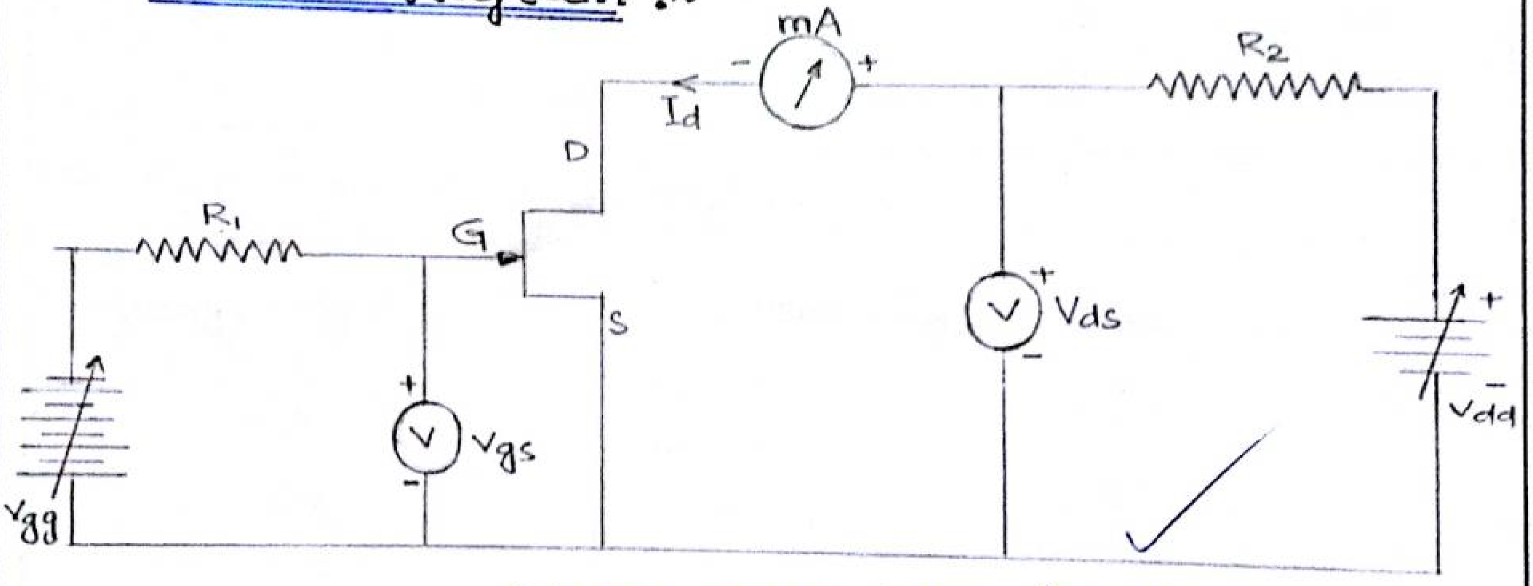
In CE mode the hie and hre for input characteristics are 10 KΩ, 0.05 and hfe and hoe for output characteristics are 235, 0.047.

Expt:~6 :- To study Field Effect Transistor characteristics

** **Objective:~** a) Study static characteristics of JFET
b) Study Transfer characteristics of JFET

** **Apparatus Required :~**
1. Bread Board
2. Regulated Power supply (variable)
3. Voltmeter
4. Ammeter

** **Circuit Diagram :~**



common source JFET circuit

** **Theory :~** Source(S) :- The terminal through which majority carriers enter the bar is known as the source.

* Drain (D) :- The terminal through which majority carriers leaves the bar is reffered as drain.

* Gate (G) :~ The regions on the two sides of the bar heavily doped with impurities opposite to that of the bar are called the gate.

The normal operation of JFET, the gate - source junction is reverse biased and drain is positive potential with respect to source.

* OBSERVATION TABLE :-

[For static or Drain characteristics]

$V_{gs} = 0\,V$

Voltage (V_{ds}) in V	current (I_d) (mA)
1	5.44
2	8.61
3	9.98
4	10.46
5	10.60
6.	10.60
7	10.54
8.	10.44
9.	10.30

$V_{gs} = -1\,V$

Voltage (V_{ds}) (V)	Current (I_d) (mA)
0	0
1	3.70
2	5.57
3	6.27
4	6.49
5	6.60
6	6.64
7	6.64
8	6.64
9	6.60

→ For Transfer characteristics :-

$V_{ds} = 5\,V$

Voltage (V_{gs}) V	Current (I_d) mA
0	10.44
−1	6.32
−2	3.19
−3	0.72
−4	0.00

* Calculation :-

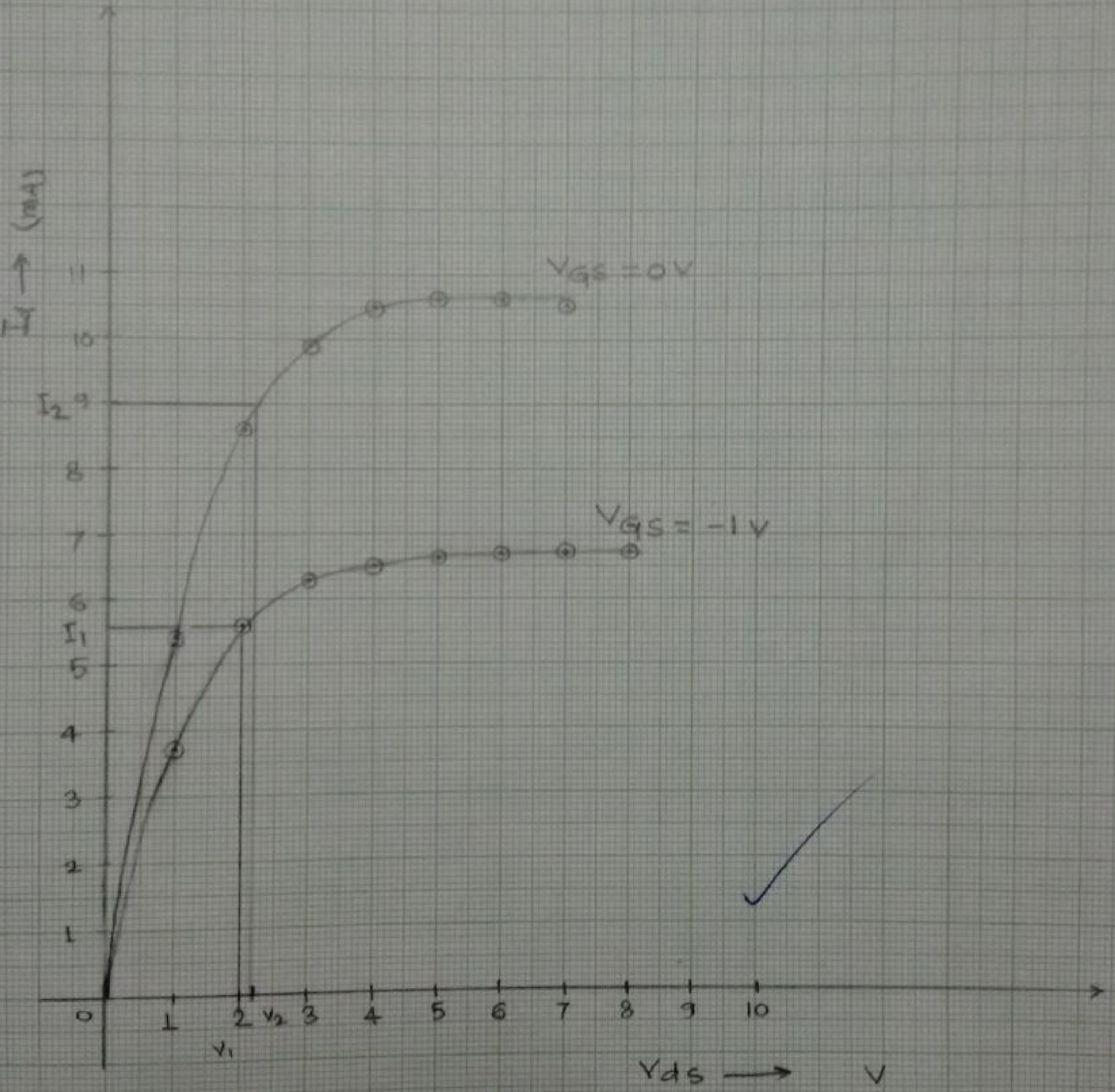
$$r_{ds} = \frac{V_2 - V_1}{I_2 - I_1} = \frac{2.2 - 2}{9 - 5.6}\,\Omega = \frac{0.2}{3.4}\,\Omega = 0.058\,k\Omega = 58\,\Omega$$

$$g_m = \frac{I_2 - I_1}{V_2 - V_1} = \frac{5 - 4.5}{+1.5 - 1.3} = \frac{0.5}{0.2}\,\frac{mA}{V} = 2.5 \times 10^{-3}\ \text{mho}$$

* Precautions :-

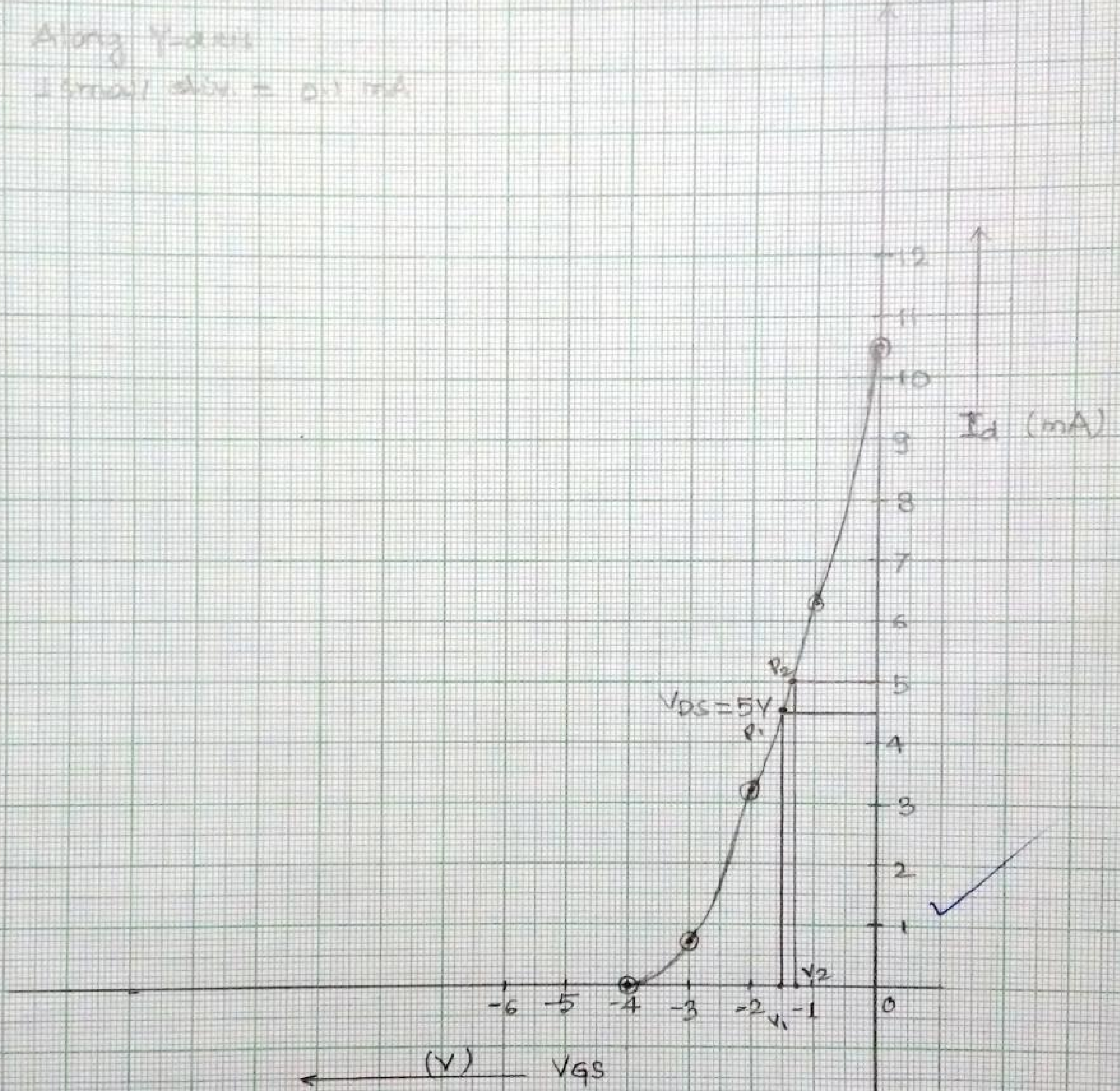
1. check the connection before taking the readings.
2. Be careful during taking the readings.

Scale :- Along X - axis
1 small div = 0.1 V
Along Y - axis
1 small div = 0.1 mA



Static or Drain characteristics of JFET

Scale:-
Along x-axis
1 small div. = 0.1 V
Along Y-axis
1 small div. = 0.1 mA



Transfer characteristics of JFET

* Conclusion :-

The $r_{ds} = 0.058 k\Omega$ or 58Ω and $g_m = 2.5 \times 10^{-3}$ mho for static characteristics and transfer characteri-stics of Field effect Transistor respectively

STUDY THE RLC SERIES CIRCUIT

OBJECTIVE : To study the RLC series circuit and draw the following characteristics :

(i) Frequency vs Resistance.

(ii) Frequency vs Impedence.

(iii) Frequency vs Inductive reactance.

(iv) Frequency vs Capacitive reactance.

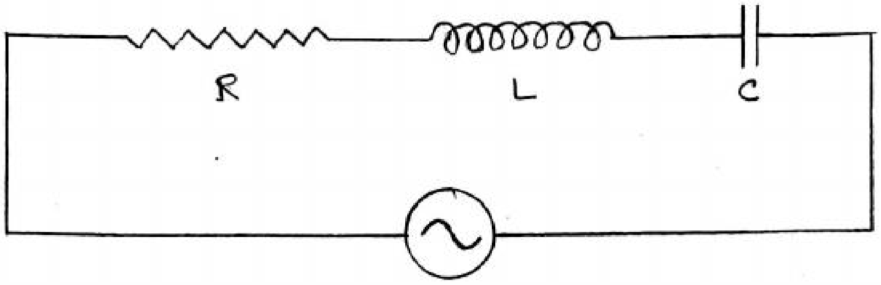
(v) Frequency vs Current.

APPARATUS :

Sl. no.	Apparatus Name	Apparatus Type	Range
1	Resistor	DC	200 Ω
2	Inductor	DC	150 mH
3	Capacitor	DC	1 μC
4	Voltmeter	DC	0 V to 3 V
5	Audio Frequency Generator	AC	100 Hz to 1000 Hz

THEORY :

Consider an AC circuit containing resistance R, inductor L and a capacitor C connected in series as shown in the figure below:



The impedance, $Z = \sqrt{R^2 + X^2}$

$= \sqrt{R^2 + (X_L - X_C)^2}$

Where, $X_L = 2\pi f L$ and $X_C = \frac{1}{2\pi f C}$

At resonance $X_L = X_C$, i.e., $X_L - X_C = 0$.

Therefore, impedance of the circuit is R, i.e., $Z = R$. So, current flowing through ~~the circuit~~ is maximum, given by $I = V/R$. In that condition voltage drop across the inductor and voltage drop across the capacitor is same and the power factor is unity. When these conditions exist, the circuit is said to be in resonance. The frequency at which this occurs is called Resonance frequency, f_r.

At resonance, $X_L = X_C$

$$\text{So, } \omega_r L = \frac{1}{\omega_r C}$$

$$\omega_r^2 = \frac{1}{LC}$$

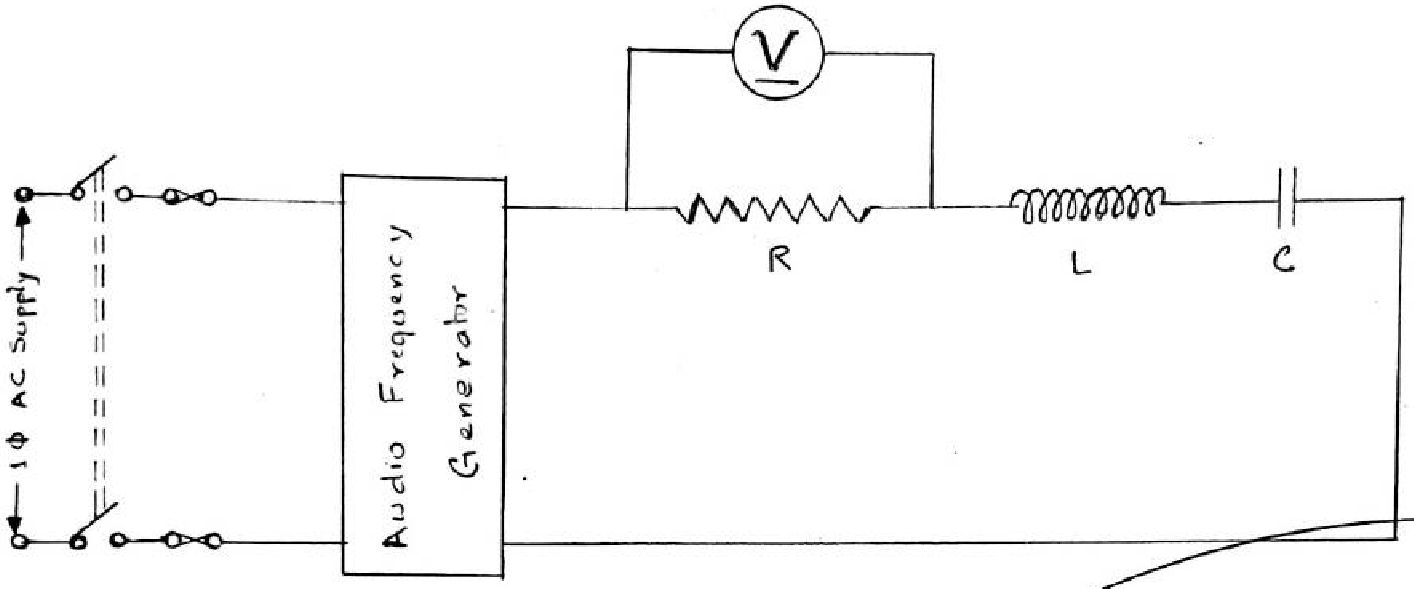
$$\omega_r = \frac{1}{\sqrt{LC}}$$

Therefore, resonance frequency, $f_r = \frac{1}{2\pi\sqrt{LC}}$.

Hence, the value of resonance frequency depends on the parameter of the two energy storing elements.

$$Q \text{ Factor} = \frac{\omega_r L}{R} = \frac{2\pi f_r L}{R}$$

CIRCUIT DIAGRAM :



Sl. No.	Frequency f (Hz)	Resistance R (Ω)	Inductance L (mH)	Inductive Reactance X_L (Ω)	Capacitance C (μF)	Capacitive Reactance X_c (Ω)	Voltage across R V_R (V)	Current $I = V_R/R$ (A)
1	100	200	150	$2\pi \times 100 \times 150 \times 10^{-3}$ $= 94.24$	1	$1 \div (2\pi \times 100 \times 10^{-6})$ $= 1591.5$	0.6	3×10^{-3}
2.	150			141.37		1061.0	0.8	4×10^{-3}
3.	200			188.49		795.7	1.1	5.5×10^{-3}
4.	250			235.61		636.6	1.3	6.5×10^{-3}
5.	300			282.74		530.5	1.6	8×10^{-3}
6.	350			329.86		454.7	1.7	8.5×10^{-3}
7	400			376.99		397.8	1.7	8.5×10^{-3}
8.	450			424.11		353.6	1.65	8.25×10^{-3}
9.	500			471.23		318.3	1.55	7.75×10^{-3}
10.	550			518.36		289.3	1.45	7.25×10^{-3}
11.	600			565.48		265.2	1.3	6.5×10^{-3}
12.	650			612.61		244.8	1.2	6×10^{-3}
13.	700			659.73		227.3	1.15	5.75×10^{-3}
14	750			706.85		212.2	1.05	5.25×10^{-3}
15.	800			753.98		198.9	1.0	5×10^{-3}

[illegible] 5/2/15

CALCULATION :

Resonante Frequency :

Calculated :

$$f_r = \frac{1}{2\pi\sqrt{L \times C}} = \frac{1}{2 \times \pi \times \sqrt{150 \times 10^{-3} \times 1 \times 10^{-6}}} = 410.9 \text{ Hz}$$

Graphical:

$$f_r = 375 \text{ Hz}$$

Bandwidth :

$$BW = \Delta f = f_2 - f_1 = 645 - 225 = 420$$

Q Factor :

Calculated :

$$Q = \frac{2\pi f_r L}{R} = \frac{2 \times \pi \times 410.9 \times 150 \times 10^{-3}}{200} = 1.9$$

Graphical :

$$Q = \frac{f_r}{f_2 - f_1} = \frac{375}{420} = 0.9$$

RESULT: Hence, the characteristics of the RLC series circuit have been studied.

DISCUSSION:

What is resonance? State the resonance condition for series RLC circuit.

Resonance is the condition when voltage drop across the inductor and voltage drop across the capacitor is same and the power factor is unity. In this condition, the impedance of the circuit is equal to the resistor only.

At resonance, $X_L = X_C$

$$\omega_r = \frac{1}{\sqrt{LC}}$$

P.T.O.

Define Band Width and Q-Factor.

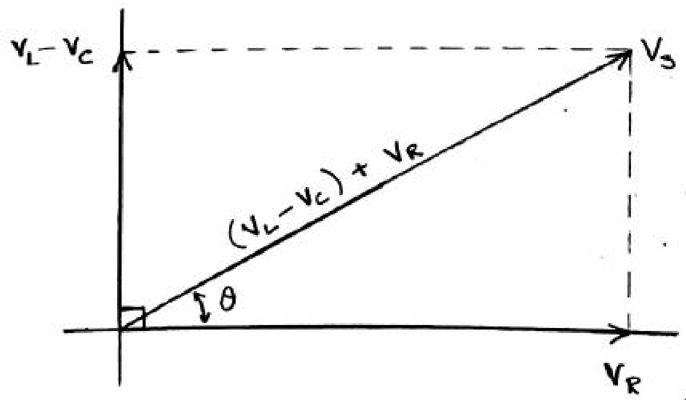
Bandwidth is the range between upper and lower half-power frequencies.

$$\Delta\omega = \omega_2 - \omega_1$$

Q-factor or quality factor is the ratio between resonant frequency and bandwidth.

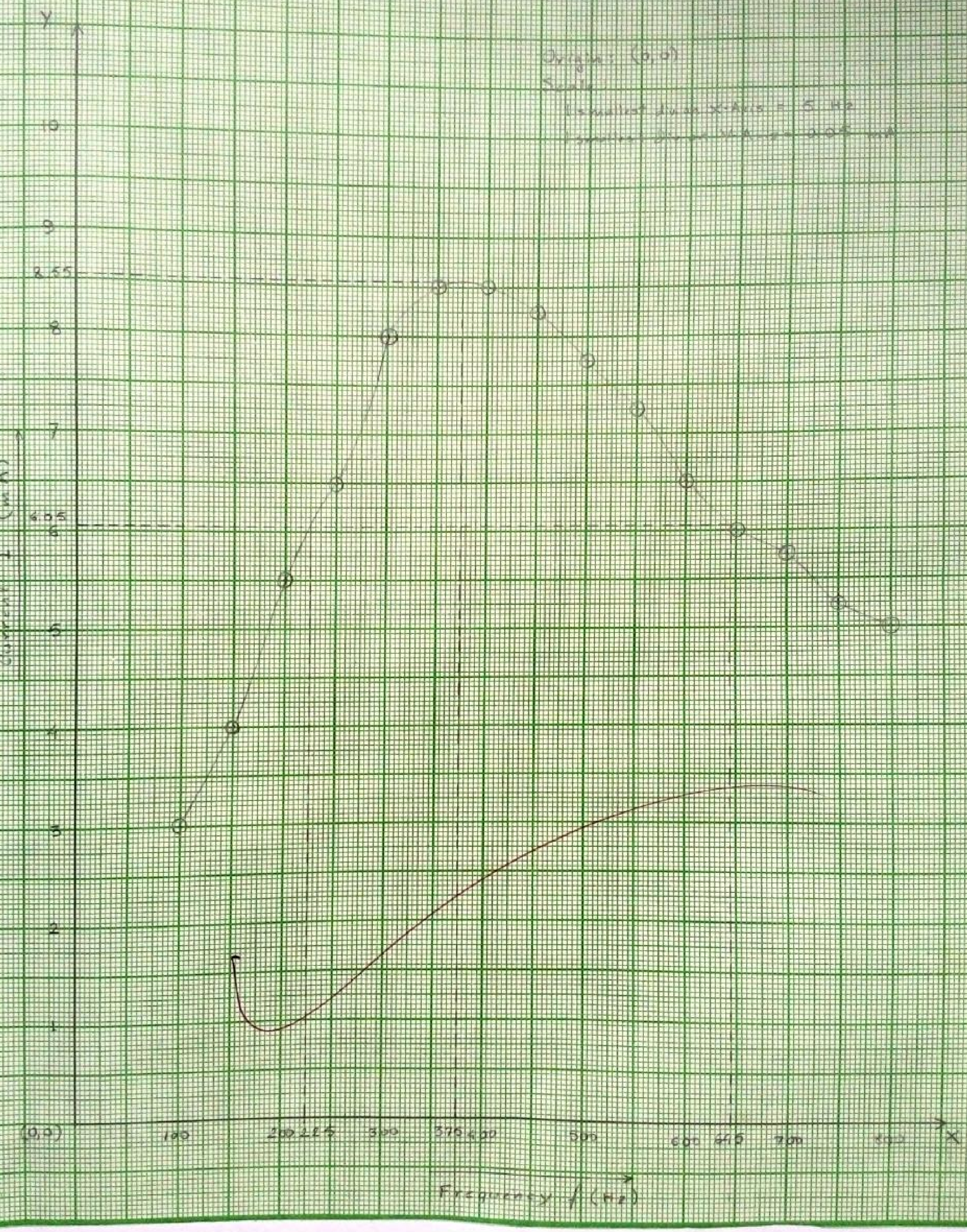
$$Q = \frac{\omega_r}{\Delta\omega}$$

Draw the phasor diagram for series RLC circuit.



[illegible]
12/2

Frequency vs Current Graph.



Expt. No:~ 5 :~ Study on Halfwave and Fullwave Rectifier

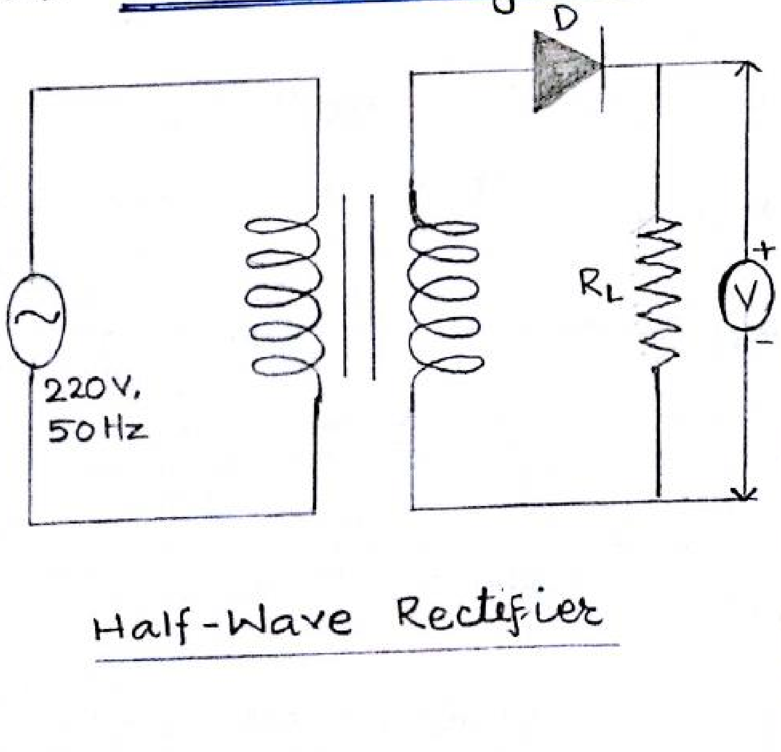
** **Objective** :~ i) To draw the waveshapes of signals at input and output terminals

ii) To measure the AC voltage of input and DC voltage of output.

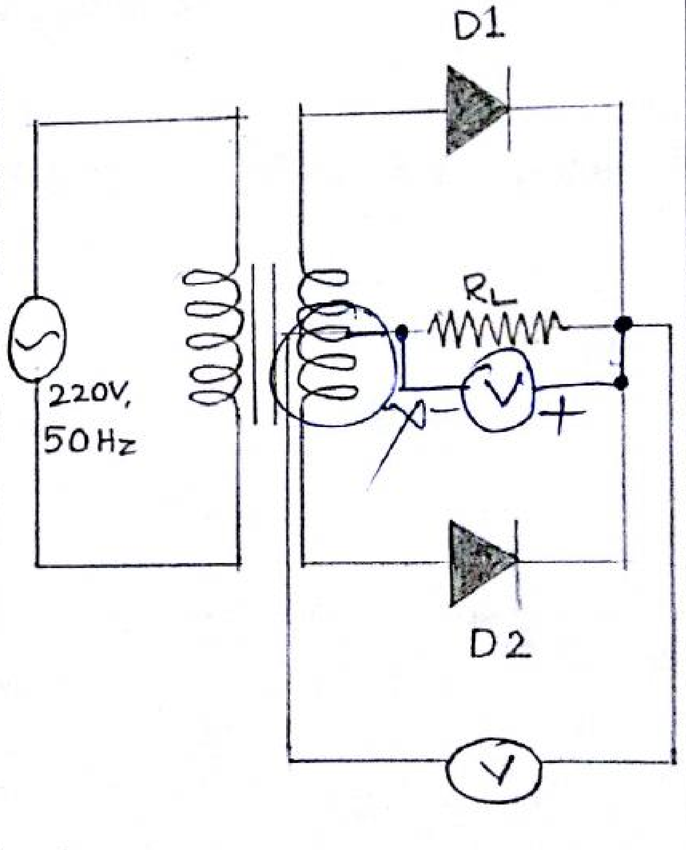
iii) To compare the frequency of input and output: signals.

** **Apparatus Required** :~ 1. Bread Board
2. CRO
3. Multimeter

** **Circuit Diagram** :~



Half-Wave Rectifier



Full-wave Rectifier

** **Theory**:~ For halfwave rectifier circuit, diode D conducts when it is forward biased. current flows through load R_L during positive

half cycle of secondary voltage. during negati-
-ve half cycle diode D is reverse biased &
no current flows. Hence voltage across load
is zero.

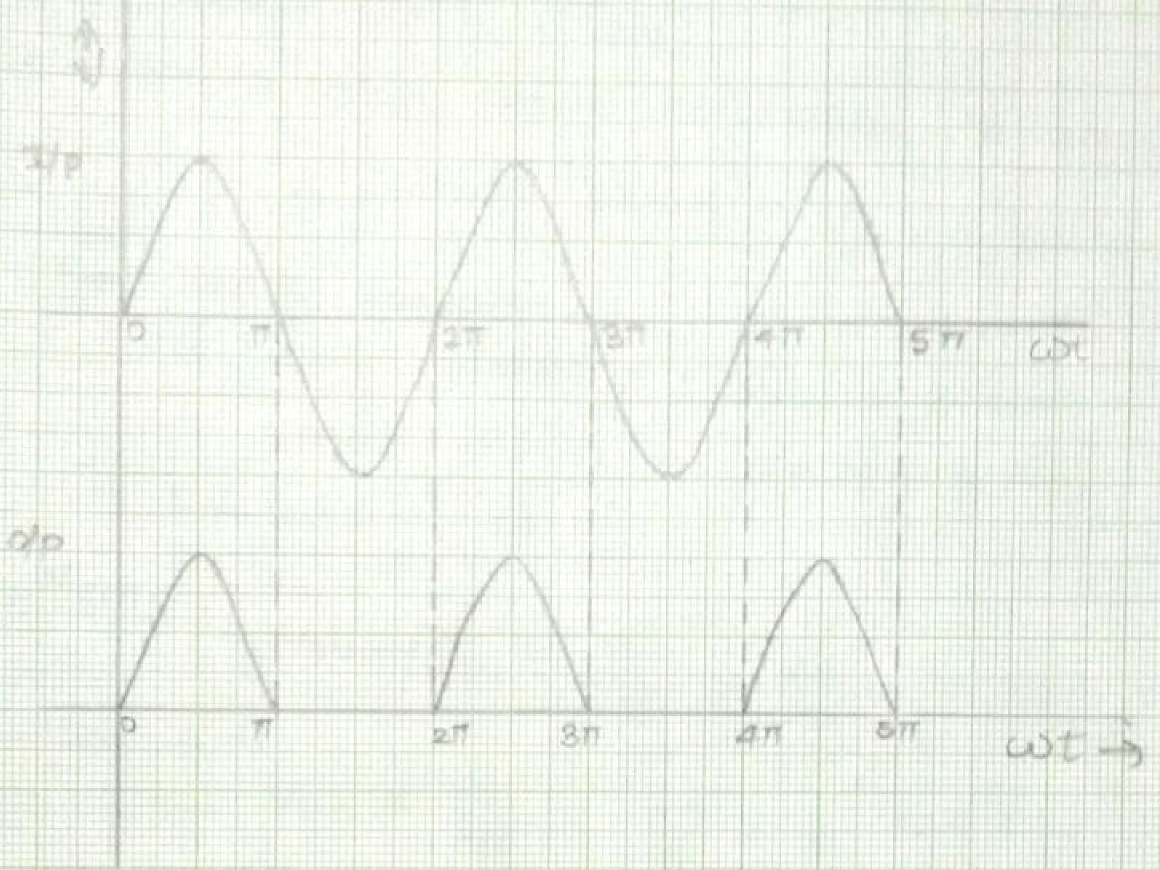
For fullwave rectifier circuit diode D1
conducts when it is forward biased. curr
-ent flows through load R_L during positive
half cycle of secondary voltage and Diode
D2 is off. In negative half cycle diode D1
is off and diode D2 conducts. Hence curre-
-nt flows through load are same direction
for both the half.

The DC voltage $V_{dc} = V_m/\pi$ where V_m is
the peak secondary voltage.

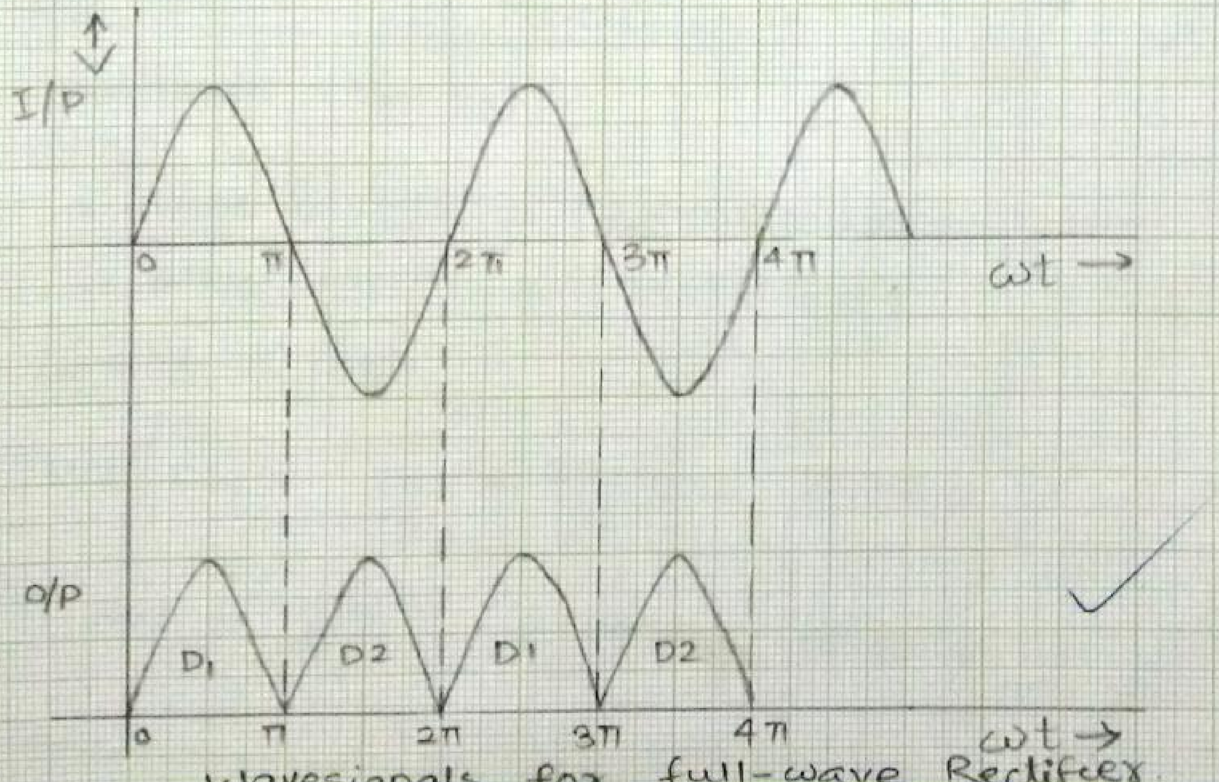
** Observations and Results :~

Parameters	H/W Rectifier	FW Rectifier
Peak Input Voltage V_m	24 V	24 V
Peak Output Voltage V_o	24 V	24 V
DC output Voltage (measured)	7.2 V	14.4 V
DC output Voltage (calculated)	7.64 V	15.27 V
RMS ~~[illegible]~~ Voltage V_{rms}	12 V	16.97 V
Ripple Factor (V_{rms}/V_{dc})	1.21	0.48
Frequency	50 Hz	100 Hz

Study on HW rectifier and FW rectifier



Wavesignals for half-wave rectifier



Wavesignals for full-wave Rectifier

** Calculation :~

→ <u>For half-wave Rectifier</u>

* Amplitude $(V_m = V_o)$

1 large block = 10 V

also, 1 large block contains 5 small block

$\therefore$ 1 small block $= \frac{10}{5}$ V = 2V

~~Vm~~ wave covers

2 large blocks = (2×10) V = 20 V

2 small blocks = (2×2) V = 4 V

$\therefore V_m = V_o = 24$ V

* <u>V_{rms}</u> : $\quad V_{rms}/_{HW} = \frac{V_m}{2} = \frac{24}{2}$ V = 12 V ✓

* <u>V_{DC}</u> : $\quad V_{DC} = \frac{V_m}{\pi} = \frac{24}{\pi}$ V = 7.63 V

* <u>Ripple factor</u> :-

$$\text{Ripple factor} = \sqrt{\left(\frac{V_{rms}}{V_{DC}}\right)^2 - 1} = \sqrt{\left(\frac{12}{7.64}\right)^2 - 1} = 1.21$$ ✓

→ <u>For Full-wave Rectifier</u> :-

* Amplitude $(V_m = V_o)$

wave covers

2 large blocks = (2×10) V = 20V

2 small blocks = (2×2) V = 4 V

$V_m = V_o = 24$ V

* $V_{rms}/_{FW} = \frac{V_m}{\sqrt{2}} = \frac{24}{\sqrt{2}}$ V = 16.97 V

* $V_{dc}/_{FW} = \frac{2V_m}{\pi} = \frac{2\times 24}{\pi}$ V = 15.27 V ✓

* Ripple factor/FW $= \sqrt{\left(\frac{V_{rms}}{V_{dc}}\right)^2 - 1} = \sqrt{\left(\frac{16.97}{15.27}\right)^2 - 1}$

$= 0.48$

* After using capacitor.

Full wave/dc $= 36\ V$

** <u>Conclusion</u> :~

The ripple factor for half-wave rectifier is 1.21 and ripple factor for full-wave rectifier is 0.48. Since ripple factor ~~[illegible]~~ for half-wave rectifier is ~~[illegible]~~ greater than full-wave rectifier, so ~~it~~ ~~[illegible]~~ there is greater pulsation in the output in case of half-wave rectifier and hence it is less effective in converting a.c into d.c.

TITLE : SPEED CONTROL OF DC SHUNT MOTOR.

OBJECTIVE : To study the speed control of a DC shunt motor using A. Field current control.

B. Armature Voltage Control.

APPARATUS :

Sl No.	Apparatus Name	Apparatus Type	Range
1.	DC Motor	DC shunt	28, 220V, 1440 rpm
2.	Ammeter	PMMC	0 - 1.5 - 3 A
3.	Voltmeter	PMMC	0 - 150 - 300 V
4.	Rheostat	Coil	2 A
5.	Tachometer	Digital	0 - 99999 rpm.

THEORY : The equation governing the speed of a dc shunt motor is $N \propto \frac{V - I_a R_a}{\phi}$

where N = Speed of the motor

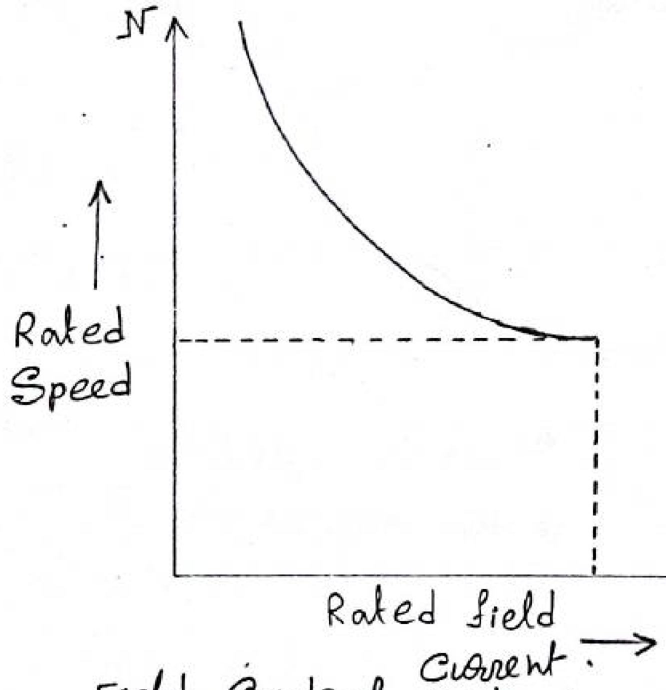
V = applied Voltage

I_a = armature current

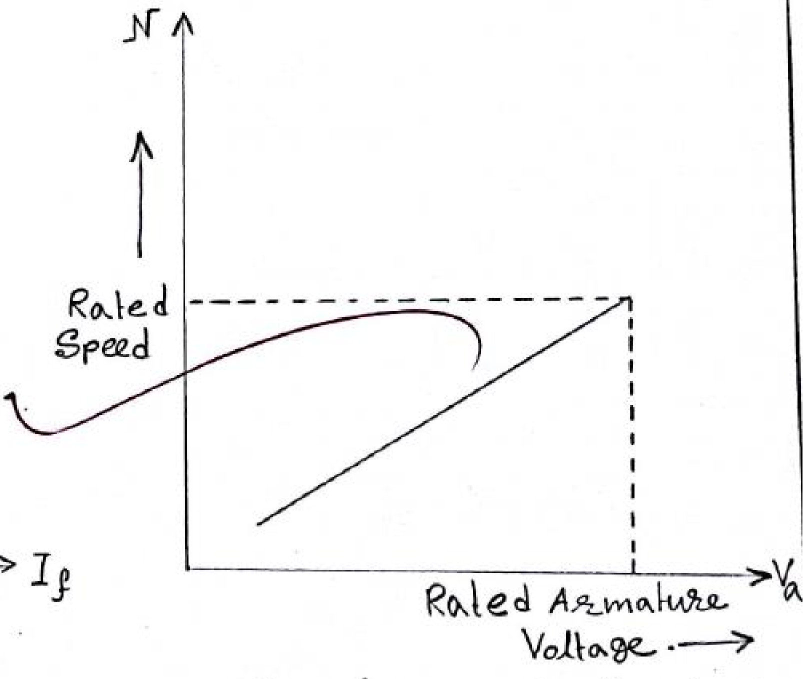
R_a = armature resistance

ϕ = field flux.

In the above equation R_a is constant. So we can control the speed of motor in two ways. Firstly by changing the field flux ϕ and secondly by changing the armature voltage $(V - I_a R_a)$. In the both case we vary the speed of motor by introducing a rhostat in the field circuit and armature circuit respectively.



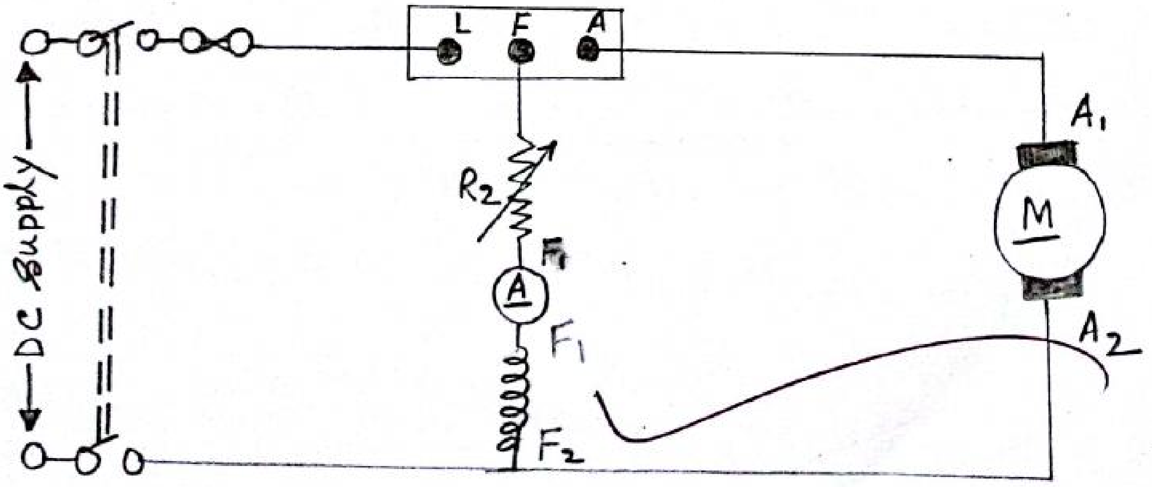
Field Control method



Armature Control Method

CIRCUIT DIAGRAM :

1. FIELD CONTROL METHOD



2. ARMATURE CONTROL METHOD:

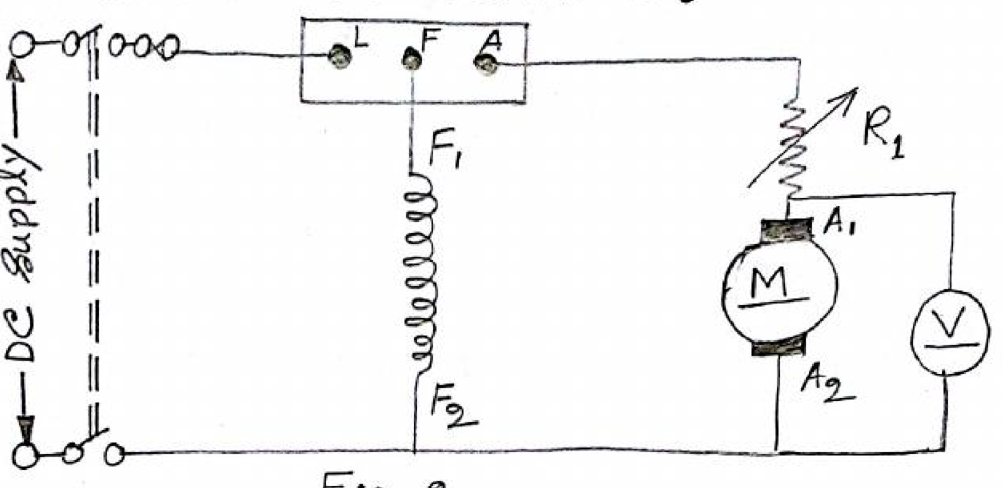


Fig 2.

PROCEDURE:

1) The circuit is connected as shown is the Figure.

2) The d-c supply is switched ON with the minimum resistance in the field circuit.

3) 3 Point starter is switched ON with a switch series rheostat with field coil.

4) Rheostat is varied to change the chage the field current and correspondingly speed is varied.

5) Observation is noted down in the observation table.

6) In next step rheostat is completely connected with the armature coil in series.

7) Again, reostat is connected with the armature coil in series.

8) Observation is noted down in the observation Table.

OBSERVATION TABLE:

Sl No.	Field control Method		Armature Control Method	
	Field Current I_f (amp)	Speed N (rpm)	Armature Voltage Va (Volt)	Speed N (rpm).
1.	0.56	1354	232	1382
2.	0.52	1430	220	1300
3.	0.48	1500	200	1190
4.	0.44	1575	180	1083
5.	0.40	1700	160	885.9

[illegible]
26/02/15

RESULT :

The grafhs for speed control of dc shunt motor by armature control method and field control method as studied in the experiment have been shown.

DISCUSSION :

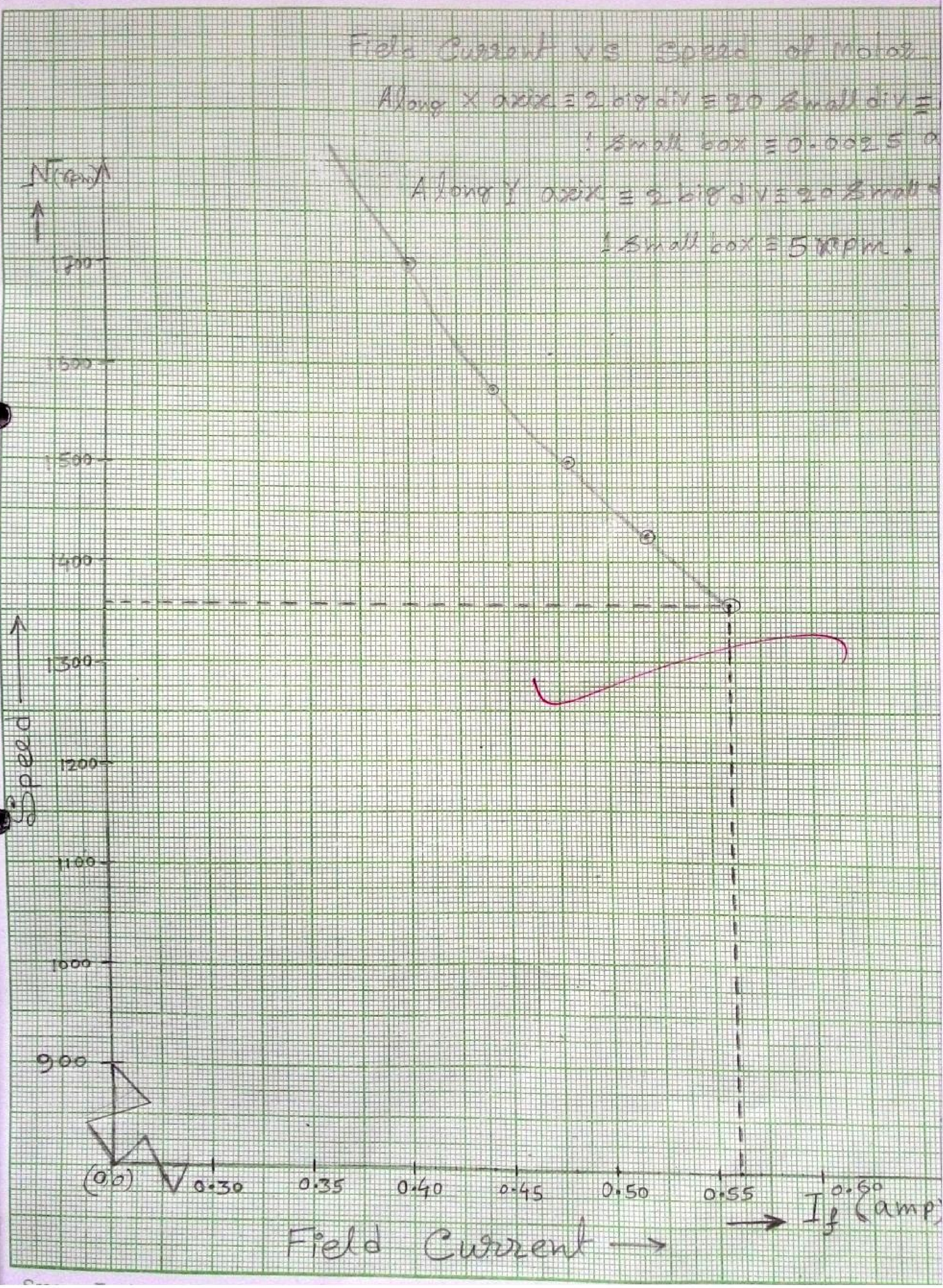
Thus, from the two graphs, we obtain the following conclusions :

(i) Speed of motor decreases with increase in field current.

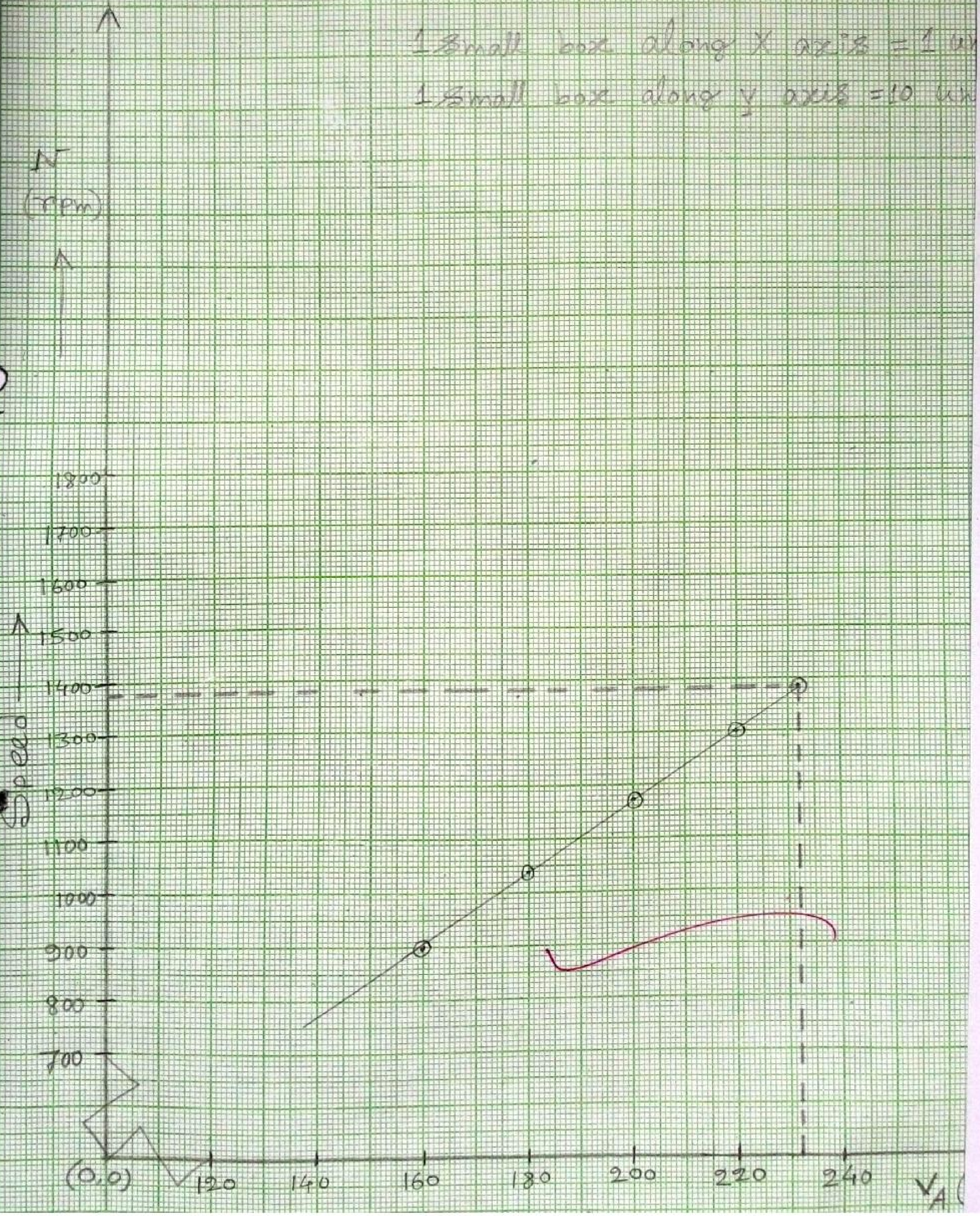
(ii) Speed of motor increases with increase of armature voltage.

[illegible]
19/02/15

Field Control Method :



Armature Control Method



Date ?

RED-OX TITRATION

AIM: Estimation of Iron present in Mohr's salt solution using standardized $KMnO_4$ solution.

THEORY: In a red-ox titration, a Substance is oxidized and the other is reduced. In other words, oxidation & reduction occurs simultaneously. Oxidation is the process of loss of one or more electrons and reduction is the gain of electrons by atoms or ions. The reagent undergoing reduction is called oxidizing agent (oxidant); and the reagent undergoing oxidation is called reducing agent (reductant). In redox titration, a reducing agent is titrated against an oxidizing agent and vice-versa. Normally potassium permaganate and iodine are commonly used as oxidizing agents and malic acid, Ferrous ammonium sulphate (Mohr's salt), sodium thiosulphate, sodium oxalate etc. are commonly used reducing agents. Potassium permanganate is a Powerful and commonly available oxidizing agent and its oxidizing Power depends on the acidity of the solution. In most of these titrations, solution of reducing agents should be made acidic before carrying out the titration. In acidic solution, reduction of potassium permanganate is represented by :-

$$[MnO_4^- + 8H^+ + 5e \longleftrightarrow Mn^{+2} + 4H_2O] \times 2$$

$$[Fe^{+2} \longrightarrow Fe^{+3} + e] \times 5$$

$$2MnO_4^- + 8H^+ + 10Fe^{2+} \longrightarrow 2Mn^{2+} + 8H_2O + 10Fe^{3+}$$

so the complete equation is :→

$$2KMnO_4 + 8H_2SO_4 + 10FeSO_4 \longrightarrow K_2SO_4 + 10MnSO_4 + 5Fe_2(SO_4)_3 + 8H_2O$$

ATE:-

PROCEDURE : Known strength of $KMnO_4$ solution, 2(N) H_2SO_4 solution and unknown strength of Mohr's salt solution are supplied. Now the burette is rinsed with $KMnO_4$ solution & then filled with it. After this, 10 ml of Mohr's salt solution is pipette out in a 250 ml conical flask & 10 ml of distilled water is added to it. Then 10 ml of dilute H_2SO_4 solution is added to it and then it is titrated by standardized $KMnO_4$ solution. At the end point, the colour changes from colourless to pink. Then from the burette reading the amount of iron present in Mohr's salt solution is determined along with the strength of the Mohr's salt solution.

APPARATUS :

- Graduated pipette (10ml)
- Burette
- conical flask (250 ml)
- Measuring Cylinder (10 ml)

REAGENTS :

- Mohr's Salt Solution
- Standardized Potassium permanganate solution
- 2(N) Sulfuric acid

RESULT & CALCULATION:

TABLE: Titration of Mohr's salt solution against standardized $KMnO_4$ solution (Redox - Titration): →

NO. OF OBS.	VOL. OF MOHR'S SALT SOLUTION (ml)	BURETTE READING		VOL.OF $KMnO_4$ SOLUTION CONSUMED (ml)	CONCORDANT READING (ml)	STRENGTH OF $KMnO_4$ SOLUTION (N)	STRENGTH OF MOHR'S SALT SOLUTION (N)	IRON PRESENT (gm/lt)
		INITIAL	FINAL					
{1}	10	30	39·4	9·4				
{2}	10	30	39·4	9·4	9·4	0·1	0·094	5·264
{3}	10	30	39·4	9·4				

From the normality equation, $V_1S_1 = V_2S_2$

where V_1 = Volume of $KMnO_4$ solution (from burette reading)

S_1 = strength of $KMnO_4$ solⁿ (given)

V_2 = volume of Mohr's salt solution (10 ml)

S_2 = strength of Mohr's salt solution

$$S_2 = \frac{V_1S_1}{V_2}$$

$$= \frac{9·4 \times 0·1}{10} (N)$$

$$= 0·094 (N)$$

Preparation of Silver Nanoparticles (Innovative Experiment)

Aim: To observe change in colour with change in dimension of Silver particle

Theory: The definition of a nanoparticle is an aggregate of atoms bonded together with a radius between 1 and 100 nm. There is a sudden shift of all properties of material when they just enter into the nanoscale. As material size reduces from centimeter (bulk) to nanometer scale, properties mostly decreases as much as six order of magnitude to that at macro level. As nanostructures are having reduced dimensions, It leads to increase in surface energy via increase in surface area. If we consider bulk material the surface to volume ratio is low, whereas the surface to volume ratio is increasing enormously in case of nanomaterial. conducting metals may show insulating behaviour in nanodimension. Hence one nanometer is a magical point on the dimensional scale.

Materials needed: Silver nitrate ($AgNO_3$); Sodium Borohydride ($NaBH_4$) Distilled Water.

Procedure: 1) 0.001 M $NaBH_4$ prepared with distilled water in a 100 ml volumetric flask.

2) 0.001 M $AgNO_3$ prepared with distilled water in a 100 ml volumetric flask and 50 ml in taken an Erlenmeyer flask

3) A magnetic stirrer bar added in the flask and stirred it with a constant speed.

4) $NaBH_4$ Solution drop wise added in the flask with stirring

5) After yellow coloration excess $NaBH_4$ Solutio is added.

Observation: The above solution will turn yellow & after addition of 2 to 3 drops of $NaBH_4$ excess $NaBH_4$ solution in that solution it becomes black

Conclusion: At first synthesized silver particles are in nano dimension so exhibiting yellow colour whereas after adding excess $NaBH_4$ solution the solution colour changes gradually from yellow to black due to agglomeration of silver nanoparticles to larger dimension. This agglomeration is due to high surface energy of nanoparticles

pH METRIC TITRATION

AIM :→ Determination of the Unknown strength of HCl solution by standardized NaOH Solution using pH Metric Method.

THEORY :→ Most of the chemical & biochemical processes are profoundly affected by the acidity and / or the alkalinity of the medium of which they take place. All acids dissociate in aqueous solution to yield H^+ ions. Some acids such as HCl, H_2SO_4 and HNO_3 are completely ionized in aquas solution, where as most of Organic acids like HCOOH, CH_3COOH, ionized to a Small extent only. The former are known as 'strong' acids, while the latter are classified as 'weak' acids. If we compare 1(N) HCl and 1(N) CH_3COOH ionizing, then we will notice that, even their molar concentrations are Same. HCl ionized extensively and forms a stronge acid solution whereas the CH_3COOH forms a weak acid sol^n containing lesser amount of H^+ ions. From these observations, we can conduct that the acidity of a solution does not depend upon the molarity of the acid, but upon the concentration of H^+ ions. we can follow from the above discussion that if we know the ions (H^+). concentration i.e $[H^+]$, for a given solution the acidity of that solution can be expressed qualititatively. water is weakly ionized into H^+ ions and OH^- ions accordingly to the following reaction :-

$$H_2O \rightleftharpoons H^+ + OH^-$$

$$H_2O + H_2O \rightleftharpoons H_3O + OH^-$$

The ionic product of water, $K_w = (H^+)(OH^-)$

For pure water $(H^+) = (OH^-) = 10^{-7}$

The H^+ ion concentration can be measured accurately using suitable methods. The acidic and alkaline nature of a solution can be expressed in terms of H^+ ion concentration. Due to some practical difficulties, the acidity or alkalinity of a solution is not expresed in terms of H^+ ions concentration. Sorensen suggested that the used of pH scale.

H^+	10^0	10^{-1}	10^{-2}	10^{-3}	10^{-4}	10^{-5}	10^{-6}	10^{-7}	10^{-8}	10^{-9}	10^{-10}	10^{-11}	10^{-12}	10^{-13}	10^{-14}
pH	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

$\longleftarrow$ acidic $\longrightarrow$ Neutral $\longleftarrow$ alkaline $\longrightarrow$

pH scale is a logarithmic scale. It is defined as the -ve logorithmic of the H^+ ion concentration. The pH has values between 0-14.

On this scale, pH of any solution is defined as :-

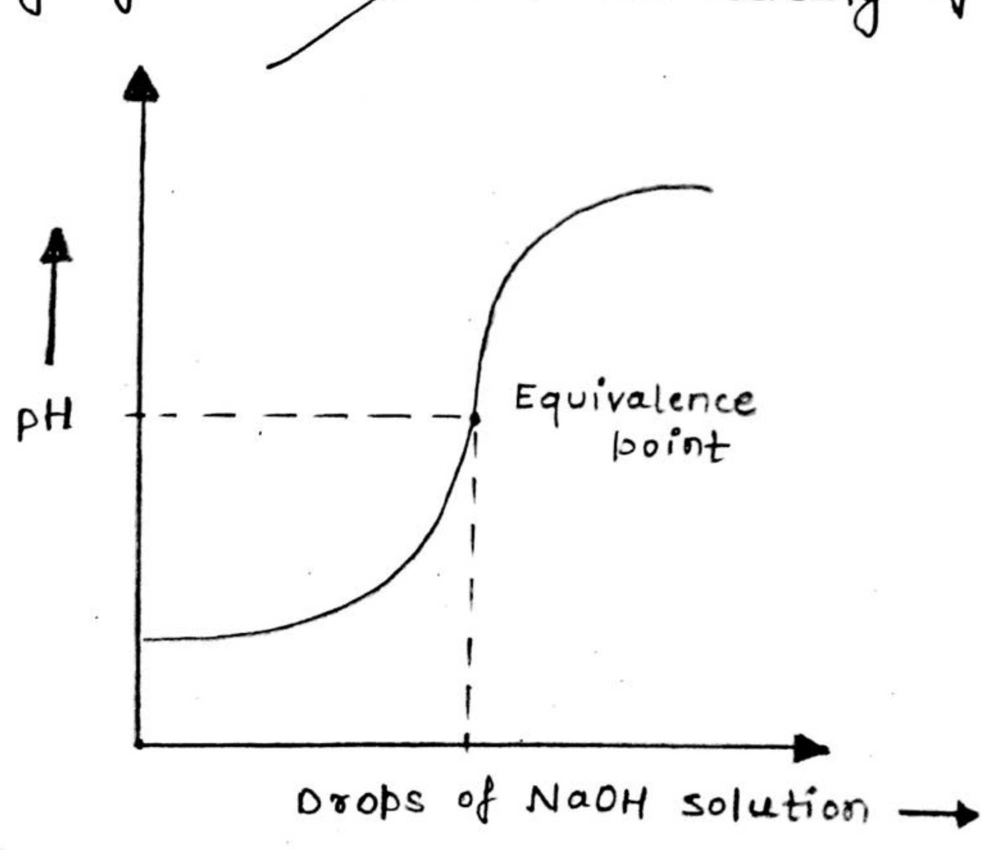
$$pH = -\log_{10} a_{H^+}$$

where a_{H^+} is the activity of the H^+ ions. for dillute solutions, activity can be replaced by concentration i.e.

$$pH = -\log_{10}[H^+]$$

All pH meters have provision for standardizing the glass electrode in a buffer solution of known pH. This is necessary because difference asymmetry potentials. Once adjustment has been made. So that the meter registers correctly the known pH of the buffer solution, the instruments gives the pH of other sol. without any calculation.

Measurement of pH is also employed to monitor the course of acid base titrations. The pH value of the sol. at different stages of acid base neutralization is determined, and is plotted against the volumes of the acid / alkali added. On adding a base to an acid, the pH rises slowly in the initial stages $pH = -\log_{10}[H^+]$; then it changes rapidly at the end point, then it fattens out. The end point of the titration can be detected where the pH changes most rapidly. However, the shape of the inflexion point (i.e. where the pH changes abruptly) and symmmetry of the curve on its two sides depends upon the ionizability of the acid the base used and on the basicity of the acid and the acidity of the base.



APPARATUS :→

- Burette
- Pipette (10 ml)
- Digital pH meter & pH Electrode
- Plastic Beaker (100 ml)

REAGENTS :→

- Unknown HCl Solution
- standardized NaOH (N/10) solution [supplied]

PROCEDURE :→

1. standardized NaOH (N/10) solution is supplied.
2. Unknown HCl solution is Supplied.
3. Rinse the pH electrode with de-ionized water.
4. Pipette out 10 cc. HCl solution into the plastic beaker and add water if necessary, so that both the electrodes are completely immersed with in the Solution. Join the electrode with the Digital pH meter and measure the pH very carefully.
5. Add NaOH solution from burette drop wise (approximately 2-3 drops).
6. Measure the pH value of the Solution after addition of 2-3 drops of NaOH and mildly stir the pH electrode. Repeat the process until you have at least five points beyond the end point.
7. Draw the curve, find the end point.

Results :→

Table : Titration of HCl Using NaOH :→

SERIAL NO.	NO. OF DROPS OF NaOH	pH
{1}	0	2·91
{2}	3	2·99
{3}	6	3·06
{4}	9	3·19
{5}	12	3·39
{6}	15	3·75
{7}	18	8·73
{8}	21	10·19
{9}	24	10·52
{10}	27	10·72
{11}	30	10·85
{12}	33	10·96
{13}	36	11·03
{14}	39	11·11

Calculation :→

V_1 = Volume of NaOH

S_1 = strength of NaOH Solution

V_2 = Volume of HCl = 10 ml

S_2 = strength of HCL

Now, from $V_1 S_1 = V_2 S_2$

$$S_2 = \frac{V_1 S_1}{V_2}$$

$$S_2 = \frac{18/18 \times 0.11}{10} (N)$$

$$= 0.011 (N)$$

DISCUSSION :→

1. NaOH is a secondry standard solution, so it should be standardized with primary standard oxalic acid solution using formula $V_1S_1 = V_2S_2$.

2. No indicator is used during the titration of HCl against NaOH solution.

3. This experiment is more accurate as it is performed by a digital pH meter. Thus we can minimize human error.

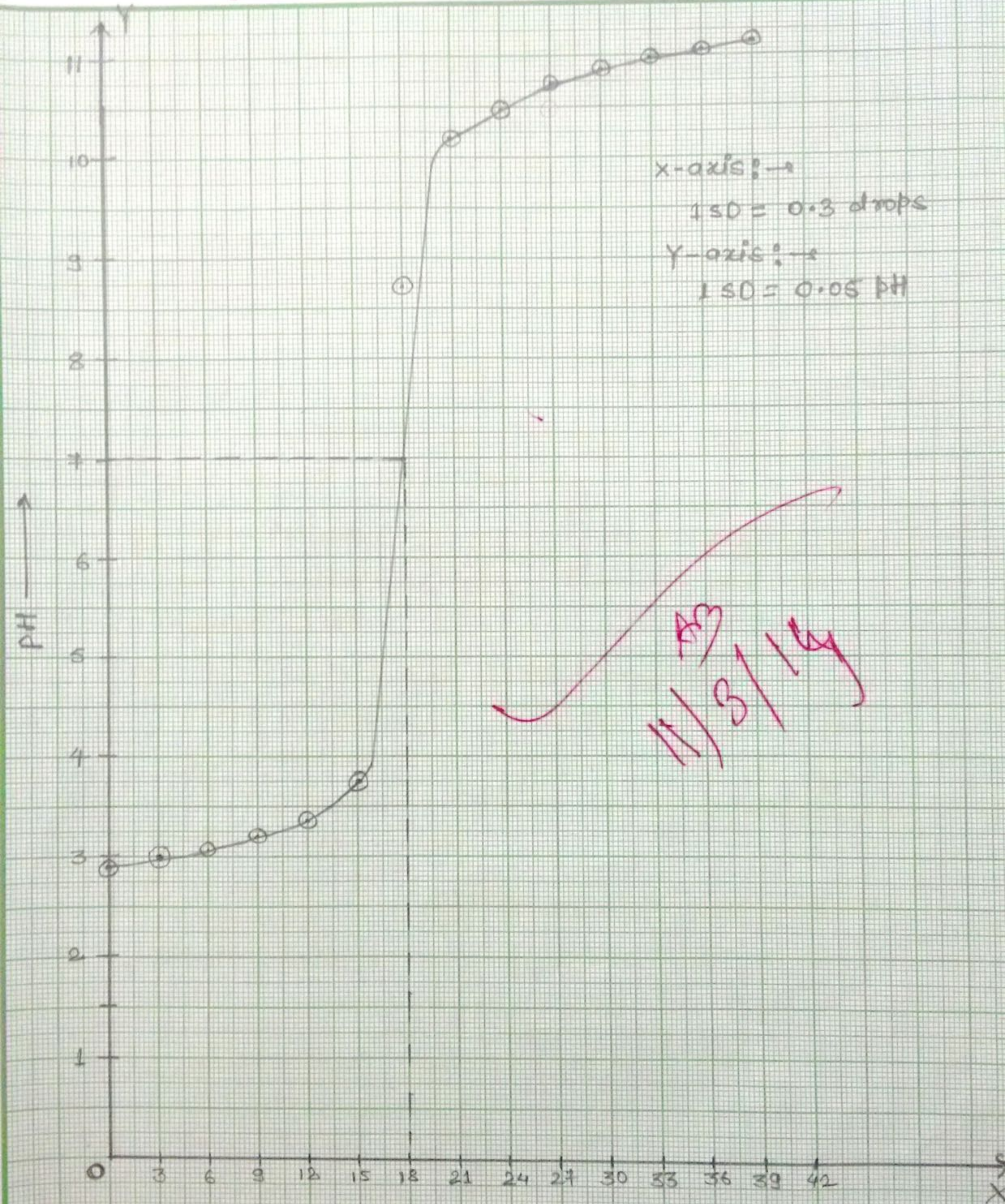
4. After plotting the graph (drops of NaOH sol. along X-axis and pH along Y axis), we get the value of NaOH solution. So the strength of Unknown HCl Solution will be more accurate than other titration.

5. Titration is performed at room temperature (25°-$30^\circ C$).

CONCLUSION :→

The Unknown strength of HCl is determined by standardized NaOH solution with the help of a digital pH meter. Hence the Unknown strength of HCl is 0.011 (N).

PH METRIC TITRATION



Short Circuit and Open Circuit Test of a Single-Phase Transformer

Objective : To determine the parameter of the equivalent circuit of a single phase transformer.

Apparatus :

Sl. No.	Apparatus Name	Apparatus Type	Range
1	Transformer	Core	1 kVA ??
2	Ammeter	Moving Iron (MI)	0 - 2.5 A / 0 - 5 A
3	Voltmeter	MI	0 - 150 V
4	Wattmeter	Dynamo	0 - 750 W

✓

THEORY :

1.) Open Circuit (OC) or No-load Test.

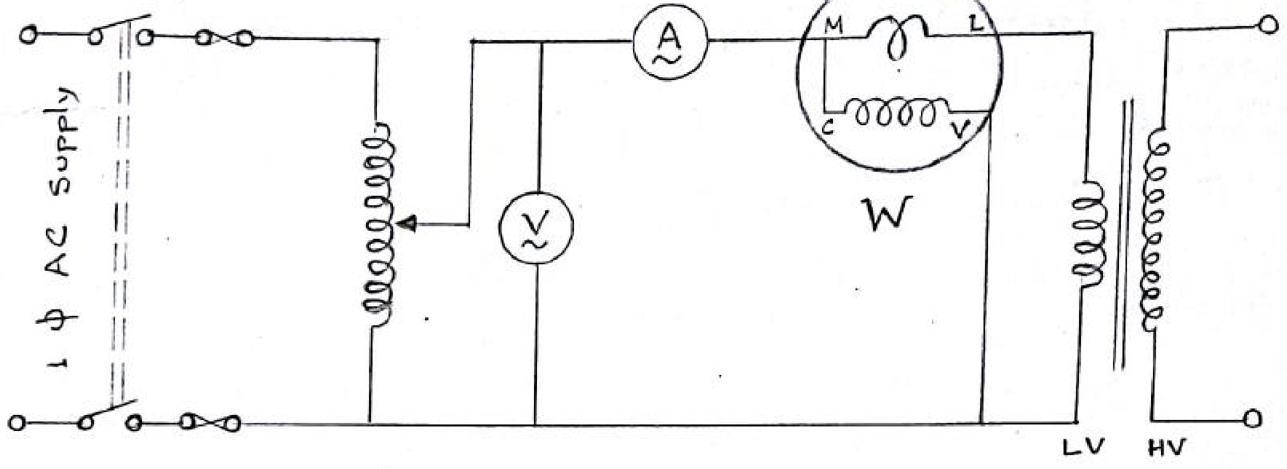
The purpose of this test is to determine the shunt branches parameter of the equivalent circuit of the transformer. This test is performed in LV side which is connected to rated supply voltage at rated frequency and HV side is kept open as shown in figure. The exciting current being about 2 to 6% of full load current and the ohmic loss in the primary, i.e., LV side varies from 0.04% to 0.36% of full load ohmic loss. In view of this ohmic loss during open circuit test is negligible in comparison with the core loss. Hence, the wattmeter reading can be taken as equal to transformer core loss.

2). Short Circuit (SC) Test.

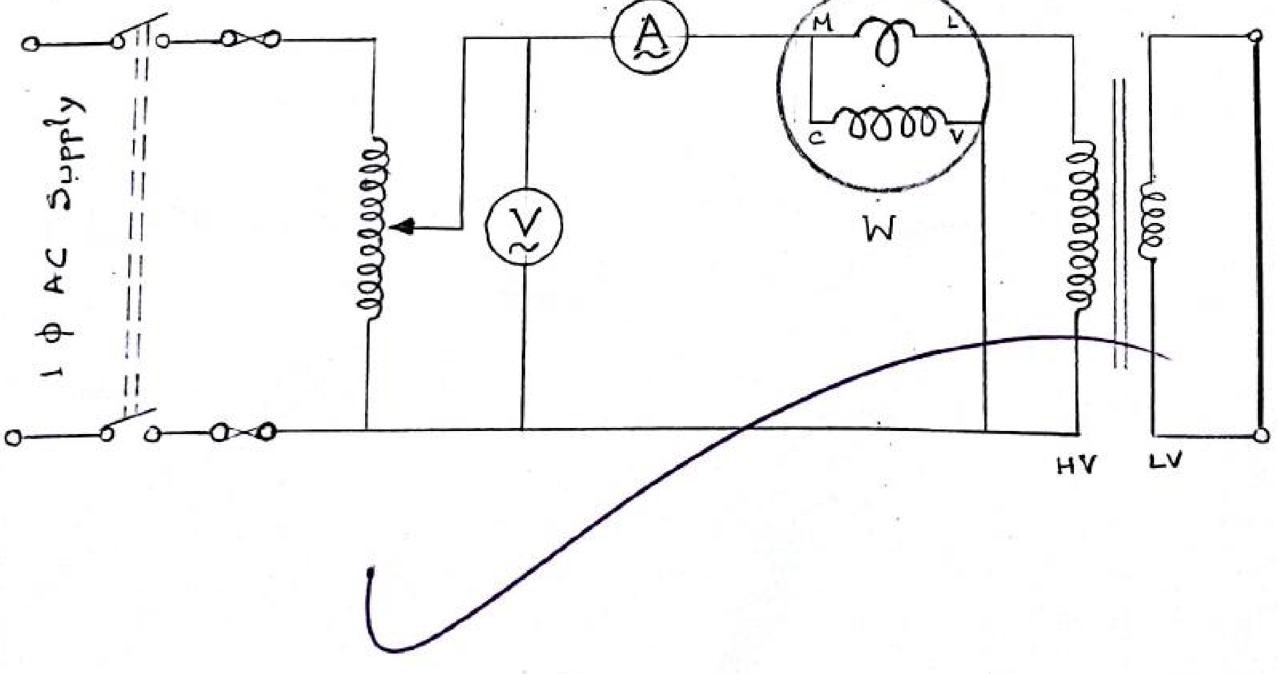
This test is performed to determine the series parameter of equivalent circuit of transformer as well as to obtain the full load copper loss of a single phase transformer. The LV side of the transformer is short circuited and the instruments are placed in HV side. The applied voltage is varied by variac to supply the rated current on HV side. As the primary mmf is almost equal to the secondary mmf in transformer, therefore rated current in high voltage winding cause the flow of rated current in low voltage winding. Since the core loss has been also negligible in comparison with rated voltage core loss, wattmeter, reading can be taken as equal to transformer ohmic loss in both winding.

Circuit Diagram:

1). Open Circuit Test.



2). Short Circuit Test



Observation Table :

Open Circuit Test			Short Circuit Test		
Voltage V_o (volt)	Current I_o (amp)	Power Input W_o (watt)	Voltage V_{sc} (volt)	Current I_{sc} (amp)	Power Input W_{sc} (watt)
110 (fixed)	0.84	$60 \times 0.5 = 30$	12.36	4.55 (fixed)	$52 \times 1 = 52$

Open Circuit :-

$$M.F = \frac{VI\cos\phi}{P} = \frac{150 \times 2.5 \times 1}{750} = \frac{375}{750} = 0.5$$

Short Circuit :-

$$I_{sc} = \frac{1000\ VA}{220\ V} = 4.55\ A$$

$$MF = \frac{150 \times 5 \times 1}{750} = 1$$

[illegible] 19/8/15

CALCULATION :

1). Open Circuit (oc) or No-load test.

Let, V_0 = Applied voltage on low voltage side = 110 V

I_0 = Exciting current or no-load current = 0.84 A

P_0 = Core loss = 30 W

Then, $P_0 = V_0 I_0 \cos\phi_0$

Therefore, no load power factor $\cos\phi_0 = \frac{P_0}{V_0 I_0}$

$$= \frac{30}{110 \times 0.84} = 0.32$$

The energy component of no-load current $I_e = I_0 \cos\phi_0$

$$= 0.84 \times 0.32 = 0.2688$$

The magnetizing ~~f~~ component of no-load current

$$I_m = I_0 \sin\phi_0 = 0.84 \times 0.95 = 0.798$$

Therefore core loss resistance $R_0 = \frac{V_0}{I_e} = \frac{110}{0.2688}$

$$= 409.23\ \Omega$$

And magnetizing reactance $X_0 = \frac{V_0}{I_m} = \frac{110}{0.798} = 137.84\ \Omega$

2). Short Circuit (SC) Test.

Let, V_{sc} = Applied voltage on high voltage side = 12.36 V

I_{sc} = Short circuit current on high voltage side = 4.55 A

P_{sc} = Total ohmic loss = 52 W

Then, the total equivalent resistance referred to high voltage side $R_{eq} = \frac{P_{sc}}{I_{sc}^2} = \frac{52}{(4.55)^2} = 2.5 \ \Omega$

The total equivalent impedence referred to high voltage side $Z_{eq} = \frac{V_{sc}}{I_{sc}} = \frac{12.36}{4.55} = 2.7 \ \Omega$

Therefore, the total equivalent reactance referred to high voltage side $X_{eq} = \sqrt{Z_{eq}^2 - R_{eq}^2}$

$= \sqrt{(2.7)^2 - (2.5)^2}$

$= 1.02 \ \Omega$

RESULT :

Core loss resistance, $R_0 = 409.23\ \Omega$

Magnetizing reactance, $X_0 = 137.84\ \Omega$.

Total equivalent resistance referred to high voltage side, $R_{eq} = 2.5\ \Omega$

Total equivalent reactance referred to high voltage side, $X_{eq} = 1.02\ \Omega$

DISCUSSION:

Open circuit test enables us to determine iron losses and parameter R_0 & X_0 of the transformer.

Short circuit test gives us the full load copper loss. Total resistance R_1 referred to the primary and total leakage reactance referred to the primary of a single load phase transformer.

[The End]

[HETEROGENEOUS EQUILLIBRIUM]

AIM :→ Determination of partition coefficient of acetic acid between n-Butanol and water.

THEORY :→ When a system consists of parts which have different physical properties and are separated by boundry surface, the system is said to be a heterogeneous one. The nernst distribution law states that at constant temperature when different quantities of a solute are allowed to distribute between two immiscible solvents in contacts with each other. Then at equilibrium the ratio of the concentration of the solute in two layers are constant at a particular temperature.

When a solute is shaken in two immiscible liquids, then the solute is found to be distributed between the liquids in a definite manner, if the solute is soluble in each of the solvent. According to the distribution law the distribution coefficient at a particular temperature is given by $K = S_1/S_3$, where S_1 and S_3 represent the concentration of the solute in solvent-1 and solvent-2 respectively. consider a liquid-liquid system, say water and n-Butanol (two immiscible solvents) to which a little quantity of acetic acid is added. Acetic acid will dissolve partly in water and partly in n-Butanol. The two solutions of acetic acid will separate into two layers at equillibrium. The concentration of acetic acid in two layers is different but their ratio is fixed at a constant temperature.

PROCEDURE :→

1. Take two stoppard bottles and marked as bottle I and bottle II. Add the following materials in bottle I and II and stoppard.

BOTTLE - I	BOTTLE - II
n - Butanol : 50 ml	n - Butanol : 75 ml
Water : 50 ml	- - - - - - -
Acetic acid solution 2(N) : 50 ml	Acetic acid solution 2(N) : 75 ml

2. Stoppard bottles are shaken for one hour and allow standing till the two liquid layers are separated.
3. Piptte out 5 ml aqueous layer in to a conical flask. Add 20 ml water and 2-3 drops of phenolphthalein indicator. Shake the mixture and titrate against standard NaOH solution.
4. Piptte out 5 ml organic layer in to a conical flask. Add 20 ml water and 2-3 drops of phenolphthalein indicator. Shake the mixture and titrate against standard NaOH solution.
5. Calculate S_1 and S_3.
6. Find out the ration of the S_1 and S_3 in each case.

REAGENTS :→

- Pure n-Butanol
- Glacial Acetic acid
- (N/2) Sodium Hydroxide solution
- Phenolphthalein indicator

APPARATUS :→

- Stoppard Bottles
- Volumetric flask
- Burette
- Pipette

RESULT AND CALCULATION :→

[TABLE - I] :— RECORD OF TEMPERATURE :→

TEMPERATURE BEFORE EXPERIMENT (°C)	TEMPERATURE AFTER EXPERIMENT (°C)	AVERAGE TEMPERATURE (°C)
28	28	28

[TABLE - II] :— STANDARDIZATION OF AQUEOUS LAYER AND N- BUTANOL LAYER BY NaOH SOLUTION :→

BOTTEL NO.	LAYER TAKEN	VOLUME OF LAYER (ml)	BURETTE READING (ml)	PARTITION CO-EFFICIENT	MEAN VALUE
{1}	Aqueous	5	6.2	1.22	1.21
	Organic	5	7.6		
{2}	Aqueous	5	8.4	1.20	
	Organic	5	10.1		

CALCULATION FOR THE BOTTLE - I :→

[a] Acetic acid in Organic layer

$V_1 S_1 = V_2 S_2$

$\Rightarrow 5 \text{ ml} \times S_1 = 7.6 \times \frac{1}{2}$

$\Rightarrow S_1 = 0.76 \text{ (N)}$

where,
V_1 = volume of acetic acid solution
S_2 = strength of the NaOH solution
S_1 = strength of the acetic acid
V_2 = volume of NaOH solution

[b] Acetic acid in aqueous layer

$V_3 S_3 = V_4 S_4$ Where, V_3 = volume of acetic acid solution

$\Rightarrow S_3 = \frac{V_4 S_4}{V_3}$ S_3 = strength of the acetic acid

$\Rightarrow S_3 = \frac{6.2 \times 1/2}{5}$ (N) V_4 = volume of the NaOH solution

$\Rightarrow S_3 = 0.62$ (N) S_4 = strength of the NaOH solution

The partition co-efficient of acetic acid in bottel-I is given by ;

$$K_1 = \frac{S_1}{S_3} = \frac{0.76}{0.62} = 1.22$$

✓

CALCULATION FOR BOTTEL-II :→

[a] Acetic acid in Organic layer

$V_1 S_1 = V_2 S_2$ where, V_1 = volume of the acetic acid sol^n^

$\Rightarrow S_1 = \frac{V_2 S_2}{V_1}$ S_1 = strength of the acetic acid

$\Rightarrow S_1 = \frac{10.1 \times 1/2}{5}$ (N) S_2 = strength of the NaOH solution

$\Rightarrow S_1 = 1.01$ (N) V_2 = Volume of the NaOH solution

[b] Acetic acid in aqueous layer

$V_3 S_3 = V_4 S_4$ where, V_3 = volume of the acetic acid solution

$\Rightarrow S_3 = \frac{V_4 S_4}{V_3}$ S_3 = strength of the acetic acid

$\Rightarrow S_3 = \frac{8.4 \times 1/2}{5}$ (N) S_4 = strength of the NaOH solution

$\Rightarrow S_3 = 0.84$ (N) V_4 = Volume of the NaOH solution

The partition co-efficient of acetic acid in Bottel-II is given by $K_2 = S_1/S_3 = 1\cdot01/0\cdot84 = 1\cdot20$

Hence the mean partition co-efficient of acetic acid in between n-butanol and water is

$$K = \frac{K_1 + K_2}{2}$$
$$= \frac{1\cdot22 + 1\cdot20}{2}$$
$$= 1\cdot21 \text{ at } 28^\circ c.$$

PRECAUTIONS :→

1. Pipette, Volumetric flask which is used During experiment must be calibrated.
2. During withdrawing aliquots one layer must not be contaminated with other.
3. Temperature should be noted at the beginning as well as at the end of the experiment and mean temperature should be taken as room temperature.

DISCUSSION :→

1. n-Butanol is less polar solvent with respect to water so it will naturally from homogeneous solution with hydrophillic dimer of acetic acid. Hence vigorous shaking, is extremly essential for at least 1 hour, to make the System heterogeneous so that a portion of acetic acid can also be dissolved in to water.
2. Constant temperature should be maintained throughout the experiment. Otherwise a solubility of acetic acid may vary.

3. As density of the water is greater than the n- Butanol, so water occupy the lower layer.
4. Pipetting of individual layer should be done very carefully to avoid the mixing of two liquid layers.
5. During partition coefficent calculation, the concentration ratio is replaced by corresponding Volume of NaOH solution because the concentration parameter will be cancelled out as shown in the Calculation.

CONCLUSION :→

The Partition co-efficient for acetic acid between n- Butanol and water at 28 °c is 1.21.

Expt :- 02 :- **Familiarization with Measuring equipments like Multi-meter, Bread-Board and CRO.**

*** OBJECTIVE :-**

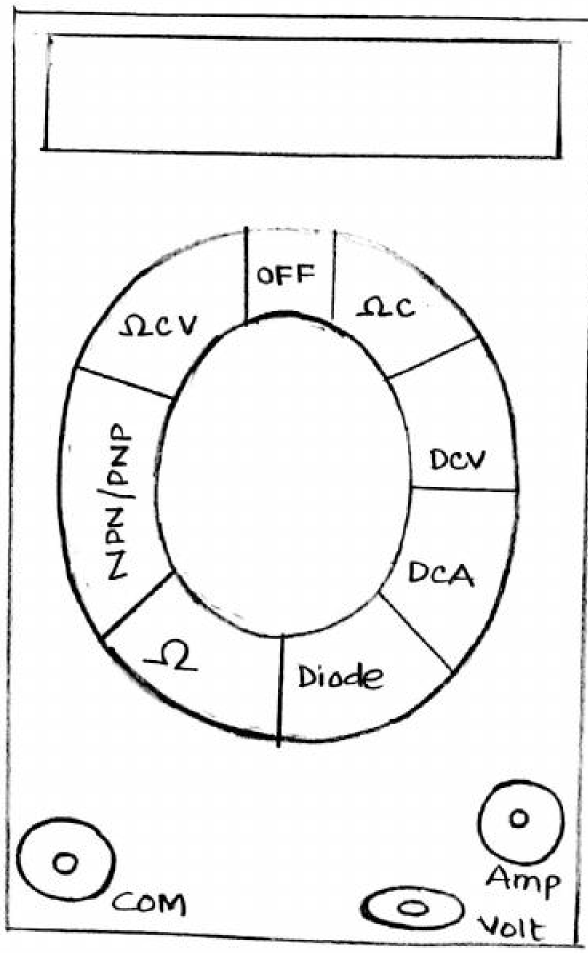
a) To identify the different types of measurements through multi-meter.

b) To understand how to operate a CRO

* **Apparatus Required :-**

1. Bread Board.
2. Digital multimeter (DMM)
3. Cathode Ray oscilloscope (CRO).

Figure:-1 Digital multi-meter (DMM)



A multi-meter (Digital) comes with a few specificati
-on that defines ~~as~~ the range and function it
can measure. For instance, one insight can
measure DC voltage in the range between 400 mV
to 1000 V and resistance can be measured
from 400 Ω to 400 mega ohm. Apart from measu-
-ring the current, voltage and resistance. The
instrument can also test logic, measure diode
characteristics, and test transistor for small
current gain and even measure frequency. To
measure ~~contsi~~ continuity, buzzer is provided
which makes a sound indicating the circuit
is working.

[illegible]
9/02/15

Expt:-01 :- FAMILIARIZATION ~~0~~ WITH ELECTRONIC COMPONENTS SUCH AS RESISTORS CAPACITORS, DIODES, TRANSISTORS etc.

** OBJECTIVES :-

ⓐ Measure resistances and capacitance value using DMM.

ⓑ To identify the anode and cathode terminals of the diodes using, DMM.

ⓒ To identify the transistor whether is npn or pnp

** APPARATUS REQUIRED :-

Digital Multi-meter (DMM)

** THEORY :-

In electronic circuit a number of electronic components are used. These components are resistor, capacitor, Diode, Transistor etc.

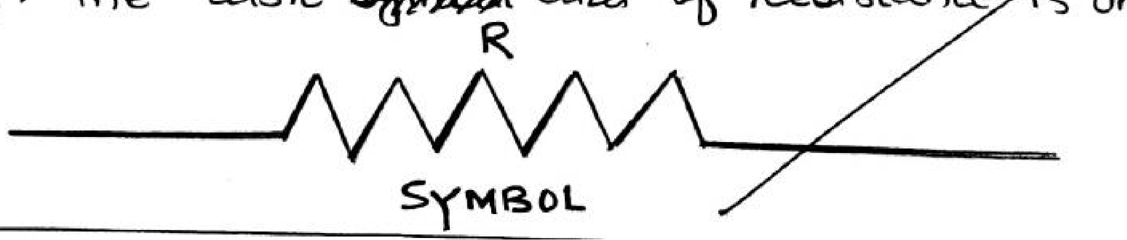
** TYPES OF COMPONENTS :-

1. Active components
2. Passive components

1. **Active components** :- Active components can amplify and require additional voltage source to make it active. Ex. Transistor, FET etc

2. **Passive Components** :- Passive components can't amplify and do not require external voltage source to make it active. Ex. Resistor, capacitor etc.

** **RESISTORS** :- Resistor control the flow of current. relation between voltage, current and resistance is $V=IR$. The basic ~~symbol~~ unit of resistance is ohm(Ω)



** TYPES OF RESISTANCE:-

i) Fixed Resistors
ii) Variable Resistors.

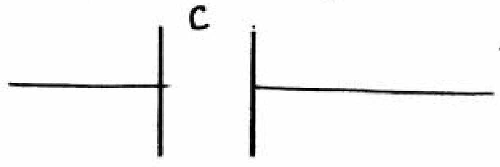
Fixed registers are classified as follows:-

1. carbon compositions resistor
2. Thin film resistor
3. Thick film resistor
4. Metal film resistor

REGISTER COLOR CODE:-

COLOR	VALUE	MULTIPLIER	TOLERANCE (%)
Black	0	0	–
Brown	1	1	±1
Red	2	2	±2
Orange	3	3	±0.5
Yellow	4	4	–
Green	5	5	±0.5
Blue	6	6	±0.25
Violet	7	7	±0.1
Gray	8	8	–
White	9	9	–
Gold	–	-1	±5
Silver	–	-2	±10
None	–	–	±20

CAPACITORS :- It is an electrical device which can store electrical energy. This electrical charge is released in form of current in an electrical circuit. The basic unit of capacitor is Farad.



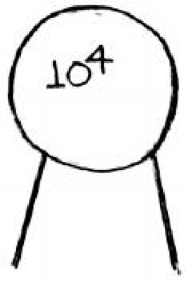
Symbol

TYPES OF CAPACITOR:-

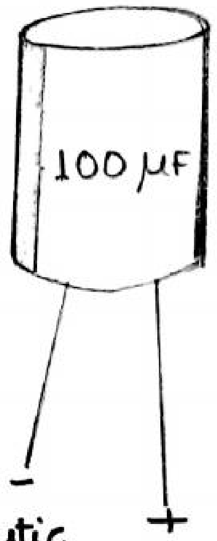
1. Paper capacitor
2. Ceramic

3. Mica capacitor   4. Electrolytic capacitor

In some cases value of capacitor is written as 10^4, it means that $(10,0000/10^6)\ \mu F$.



Ceramic ~~capapic~~ capacitor.

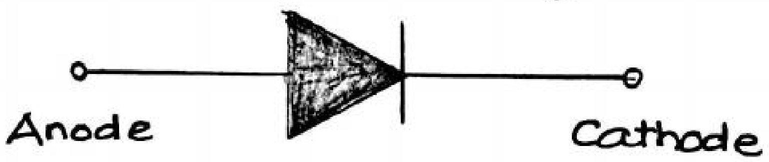


Electrolytic capacitor

DIODES :- Diode is formed when p-type semiconductor is diffused with n-ptype semiconductor. It offers low resistance when it is forward biased and offers high resistance in reverse biased condition.

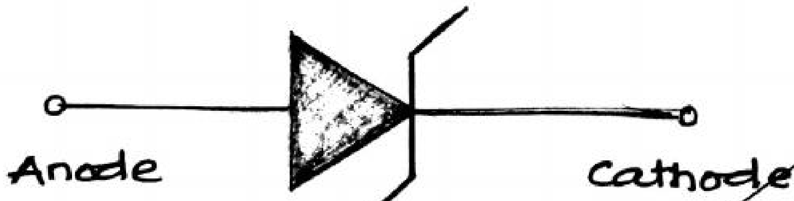
Different types of diodes are available Normal P-N junction Diode, Zener Diode, ~~LED~~ LED etc.

1. **Normal P-N JUNCTION DIODE** :- It is generally used in voltage rectifier circuit.



SYMBOL

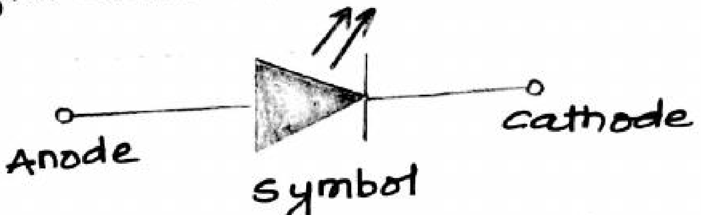
2. **Zener Diode** :- It is generally used in voltage regulator circuit in reverse biased condition.



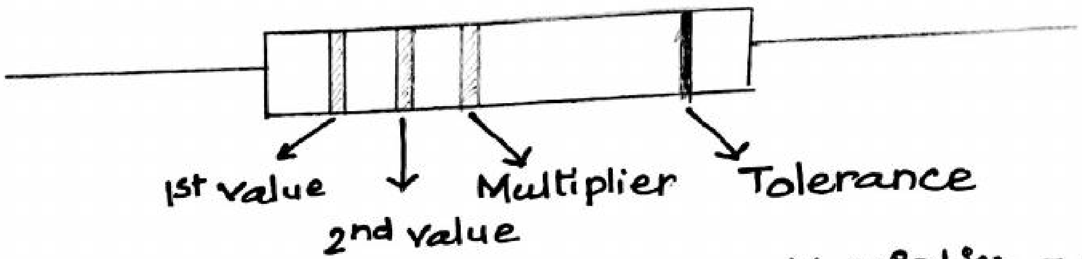
SYMBOL

Example:-

3. LED DIODE (LIGHT EMMITTING DIODE):- These are made of some specific semiconductors, like GaAs, GaP etc.



Resitance on the basis of Resistor color code:-



If 1st value = Brown, 2nd value = Black, Multiplier = orange
Tolerance = Gold.
Then resistance value will be
From resistor color code table. Brown = 1, Black = 0, oran--ge = 3, Gold = ± 5%.
So, $10 \times 10^3 = 10\ K\Omega \pm 5\%$.

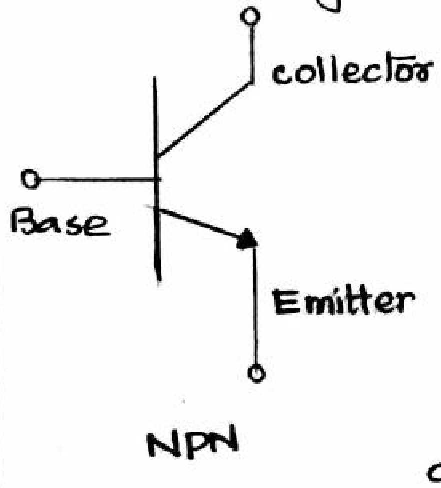
** OBSERVATION TABLE:-

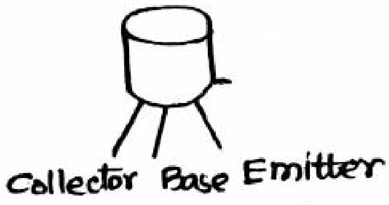
color code				calculat--ed value	Upper limit	Lower Limit	Resista--nce value Measured By DMM
Resistance Value			Toleran--ce				
1st Band	2nd Band	3rd Band	4th Band				
Brown	Black	orange	Silver	$10 \times 10^3 \pm 10\%$	11000	9000	9.83 KΩ
Yellow	Violet	Black	Gold	47 ± 5%	49.35	44.65	46.8 Ω
Brown	Black	orange	None	$10 \times 10^3 \pm 20\%$	12000	8000	9.81 KΩ

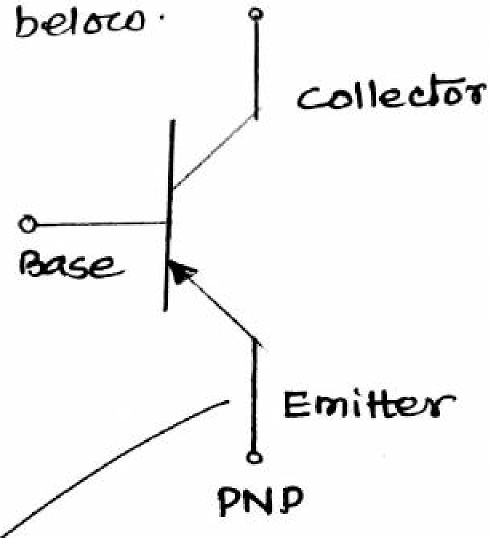
TRANSISTOR :- Transistor means transfer resistor different types of transistors are available.

1. BJT (Bipolar Junction Transistor)
2. FET (Field Effect Transistor).

1* BJT (Bipolar Junction Transistor) :- It is a three layer semiconductors device ie Emitter Base and collector. Two types of BJT's are available NPN and PNP type. Symbols are given below.

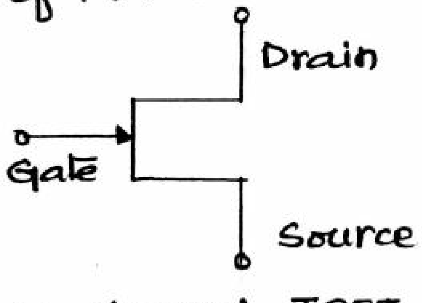




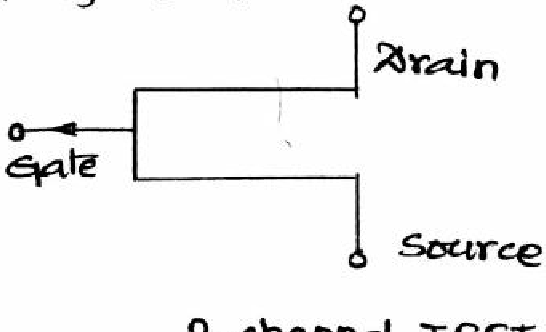


SYMBOL

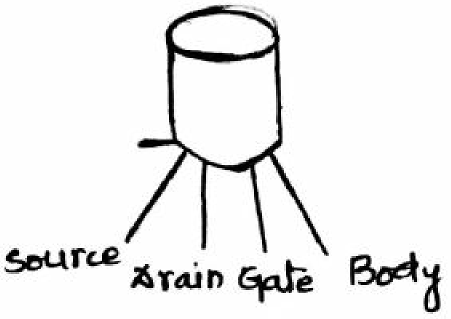
#2. FET :- It is generally a three layer semiconductor device ; i.e Source, Gate and drain. different types of FET's are available i.e JFET, MOSFET



N- channel JFET



P-channel JFET



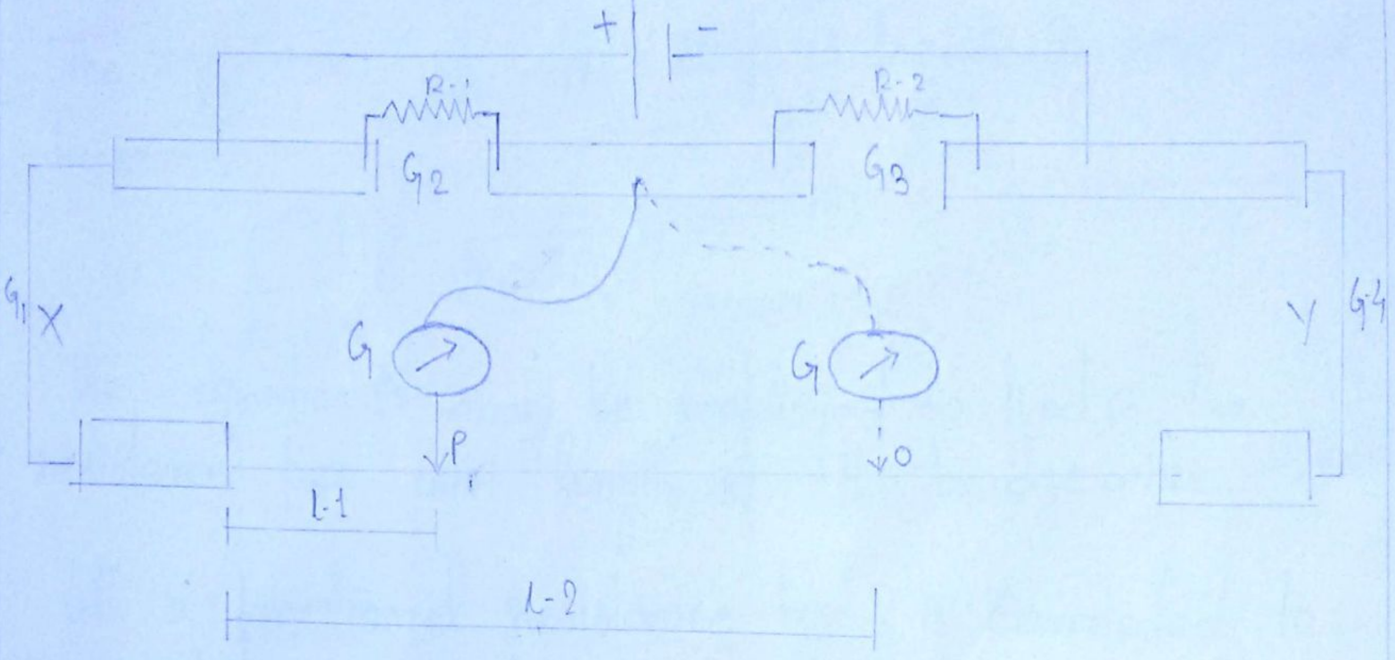
SYMBOL

DETERMINATION OF UNKNOWN RESISTANCE BY CAREY FOSTER METHOD.

AIM To determine the value of unknown resistance by carey foster method.

Objective After performing this experiment ~~Students~~ we will be able to
(i) Design a dc-bridge circuit.
(ii) Construct 'One-ohm' coil.

APPARATUS - Carey foster's bridge, Two equal 1/5 ohm resistances, Power supply, unknown resistance, Table Galvanometer, Resistance box, Plug Commutator.



THEORY -

1. Let the null point be obtained at a distance L_1 from the left end of the wire, when connections are made with a certain resistance X in the extreme left gap G1, Copper. Strip Y in the right gap G-4, R-1 in the Gap G2 and R-2 in the gap G-3 (R-1 = R-2)

2. When X and Y are interchanged, let the null point be obtained at Q at a distance l-2, from left. It can be shown that the resistance ρ per unit length of the bridge wire given by,

$$\rho = \frac{(x-Y)}{(l_2 \sim l_1)}$$

The registance Y of Copper strip is peratically zero and hence,

$$P = \frac{X}{(l_2 \sim l_1)} , \quad \text{—} \; (2)$$

The equation (2) may be employed to find P, the resistance per unit length of the bridge wire.

Let a fractional resistance box S is Connected to the extreme left gap G-1 and an unknown resistance R in the extreme right gap G-4 and a null point is obtained at a distance l_1' from the left end of the wire. Again with the position of S and R interchanged null point is obtained at a distance l_2' from the left end then as before

$$P = \frac{(S-R)}{(l_2' \sim l_1')} ,$$

$$\text{or, } R = S - P(l_2' \sim l_1')$$

Knowing the value of S, P, l_2', l_1' the unknown resistance R Can be found.

Table - 1 Measurment of ρ

obs no	x in Ohm	Position of null point when the copper strip is in		$(l_2 \sim l_1)$ in Cm	$\rho = \frac{x}{(l_2 \sim l_1)}$ in ohm	Mean ρ In ohm/cm
		Extreme right gap (l_1) cm.	Extreme left gap (l_2) cm			
1	0.8	28.3	62.4	32.7	0.024	
2	1	22.9	68.9	46	0.021	
3	1.2	18.2	72.6	54.4	0.022	0.0228
4	1.4	14.2	74.2	60	0.023	
5	1.6	11.5	77.4	65.9	0.024	

Table - 2

Measurment of unknown Resistance R.

obs no	S in ohm	Position of null point when the unknown Resistance is in		$(l_2' \sim l_1')$ in Cm	$R = S - \rho(l_2' \sim l_1')$	Mean R in ohm
		Extreme right gap (l_1') in Cm	Extreme left gap (l_2') Cm			
1	0.8	50.5	44.1	6.4	0.65	
2	1	43.9	50.3	6.4	0.85	
3	1.2	39.5	55.5	16	0.83	0.778
4	1.4	34.4	61.0	26.6	0.79	
5	1.6	27.5	63.8	36.3	0.77	

[illegible] 16.08.16.

Recorded [illegible] 16.08.16.

CALCULATION :-

$$P = \frac{X}{(l_2 \sim l_1)} \quad \text{and} \quad R = S - P\,(l_2' \sim l_1')$$

Discussions :-

During the experiment, due to joule heating, temperature of the wire increases. Hence, the resistance doesnot remain constant, it changes. So, to get more accurate readings, the circuit should be broken and the wire should be given time to cool down.

[illegible]
23.08.16

EXPERIMENT No.: 6

AIM : To determine the specific rotation of a sugar solution of known concentration with the help of a polarimeter.

APPARATUS : The polarimeter set with polarimeter tube; sugar; measuring balance, measuring cylinder; beaker.

THEORY : The rotation θ of the plane of polarization of a polarized light by an active solution of length 'l' cm ($l/10$ dm) containing 'm' gm of active substance per cc. of the solution is given by

$$\theta = slm/10 \quad \text{——— (1)}$$

Here 's' is the specific rotation of the substance, which is the rotation produced by a solution of active substance (sugar) in a non-active substance (water) of decimeter in length containing 1gm of active substance per cc. of the solution (at a given wavelength and temperature).
If strength of the solution be c % by volume then $m = (c/100)$ gm/cc. Then the specific rotation comes out to be the following
From equation (1), $\theta = slm/10$

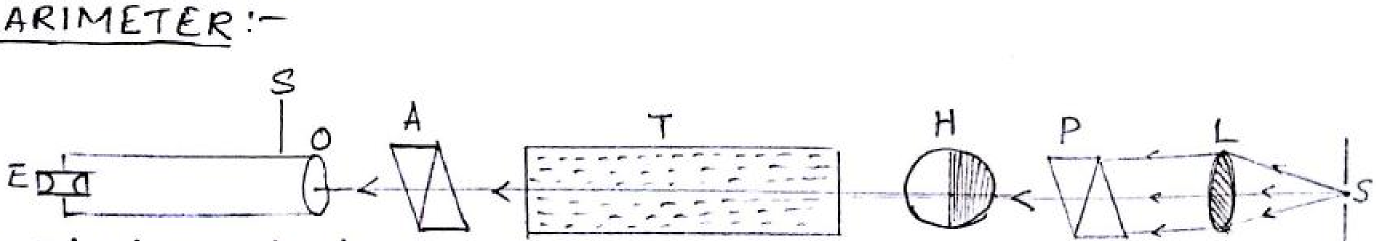
or, $\theta = slc/1000$ $\{\because m = c/100\}$.

or, $s = 1000\,\theta/lc$ degree.cm^3/dmgm

——— (2).

Hence from equation (2) we get the values of specific rotation (s) by knowing rotation (θ), l = Length of the tube and c = concentration.

POLARIMETER :-



P = Polariser, L = lens, S = Source E = Eye piece.
S = Circular Scale A = Analyser, T = Polarimeter tube, H = Half plate

EXPERIMENTAL DATA :

Length of the tube = 21 cm.

Table :- Preparation of solution of C% strength i.e 5% strength by volume.

Volume of Solution to be prepared.	Masses of			Final volume of Sugar Solution i.e 5% solution.
	Flask with approximately 60cc Water	Flask with water + 5gm Sugar.	Sugar in water.	
100 ccm.	m_1 gms = 35+60 =105 gms	m_1 + 5gms = 105 +5 =110gms.	5 gms.	100 cc

Table 2 Circular Scale Reading when Pure-Water fills the tube

V-Constant = 0.1°

No. of observation	Circular Scale Reading	Vernier Reading.	Total.	Mean R_1.
1.	328	4	328.4°	
2.	328	5	328.5°	328.5°
3.	328	6	328.6°	

Table 3 Circular Scale Reading when the solution fills the tube.

V. Constant = 0.1°

No. of observation	Circular Scale Reading	Vernier Reading.	Total	Mean R_2.
1.	331	5	331.5°	
2.	331	6	331.6°	331.6°
3.	331	7	331.7°	

CALCULATIONS : Angle of Rotation = $R_2 \sim R_1$ = θ degree

or, $\theta° = 331.6° - 328.5°$

$= 3.1°$

Recorded

[illegible]

15/10/14

Specific Rotation, $s = 1000\ \theta / lc$

$= 1000 (3.1) / (21)(5)$

$= 3100 / 105 = 29.52$ deg. cm³/dm. gm

Recorded

[illegible]

15/10/14

[illegible]

Discussions :

To find θ for solution of C% strength.
The water in the tube T is thrown away & after washing the tube by little of solution of C% strength, the whole tube is filled with the Solution of C% Strength having no air bubble. Then it is placed in its proper position.

[illegible] 22/10/[illegible]

EXPERIMENT No.: 5

AIM : Verification of Bohr's atomic orbital theory, with the help of Frank-Hertz experiment.

APPARATUS : Frank and Hartz experimental kit with a tetrode filled with Argon vapour.

THEORY : From Bohr's postulation we know that the internal energies of an atom are quantized. We can directly prooved this postulate by this experiment.

In this experimental set-up (Fig. 5.1) there is a tetrode tube filled with Argon vapour. Electrons emitted by heated filament are accelerated by the potential V_{G2K} (applied between cathode and grid 2). At first some electrons reach to plate A provided that their K.E. is sufficient to overcome the retarding potential V_{G1K} (applied between cathode and grid 1). So, initially we see that the plate current increases with increase of V_{G2K}. But as the voltage further increases; the electron energy reaches threshold value to excite the atom in its first excited state, the current abruptly drops. When the V_{G2K} is increased further, again the current starts to increase and when V_{G2K} reaches to a value twice that of first excitation potential again we get the current drops. This observation proves that the energies of the atoms are quantized.

Observations :

Table 1 : $V_{G1K} = 1.5V$, $V_{G2A} = 7.5V$

PLATE CURRENT (I) IN $nA = 10^{-9}A$	V_{G2K} IN VOLT
0.05	10
0.15	11
0.25	12
0.37	13
0.48	14
0.58	15
0.66	16
0.73	17
0.76	18
0.74	19
0.67	20
0.59	21
0.48	22
0.42	23
0.43	24
0.52	25
0.68	26
0.85	27
0.98	28
1.05	29
1.04	30
0.89	31
0.72	32
0.52	33
0.35	34
0.29	35
0.40	36
0.68	37
1.00	38
1.25	39
1.36	40
1.35	41
1.25	42
1.02	43
0.68	44
0.44	45
0.26	46
0.32	47
0.60	48
1.06	49
1.45	50
1.65	51
1.75	52
1.72	53
1.52	54
1.24	55
0.83	56
0.54	57
0.38	58
0.58	59
1.05	60
1.56	61
1.95	62
2.22	63
2.33	64
2.29	65
2.08	66
1.78	67

Recorded [illegible] 3/9/14

GRAPH BETWEEN V_{G2K} ALONG X-AXIS AND 'I' ALONG Y-AXIS

SCALE :

X-AXIS : 1div = 4 Volt

Y-AXIS : 1div = 0.2 nA

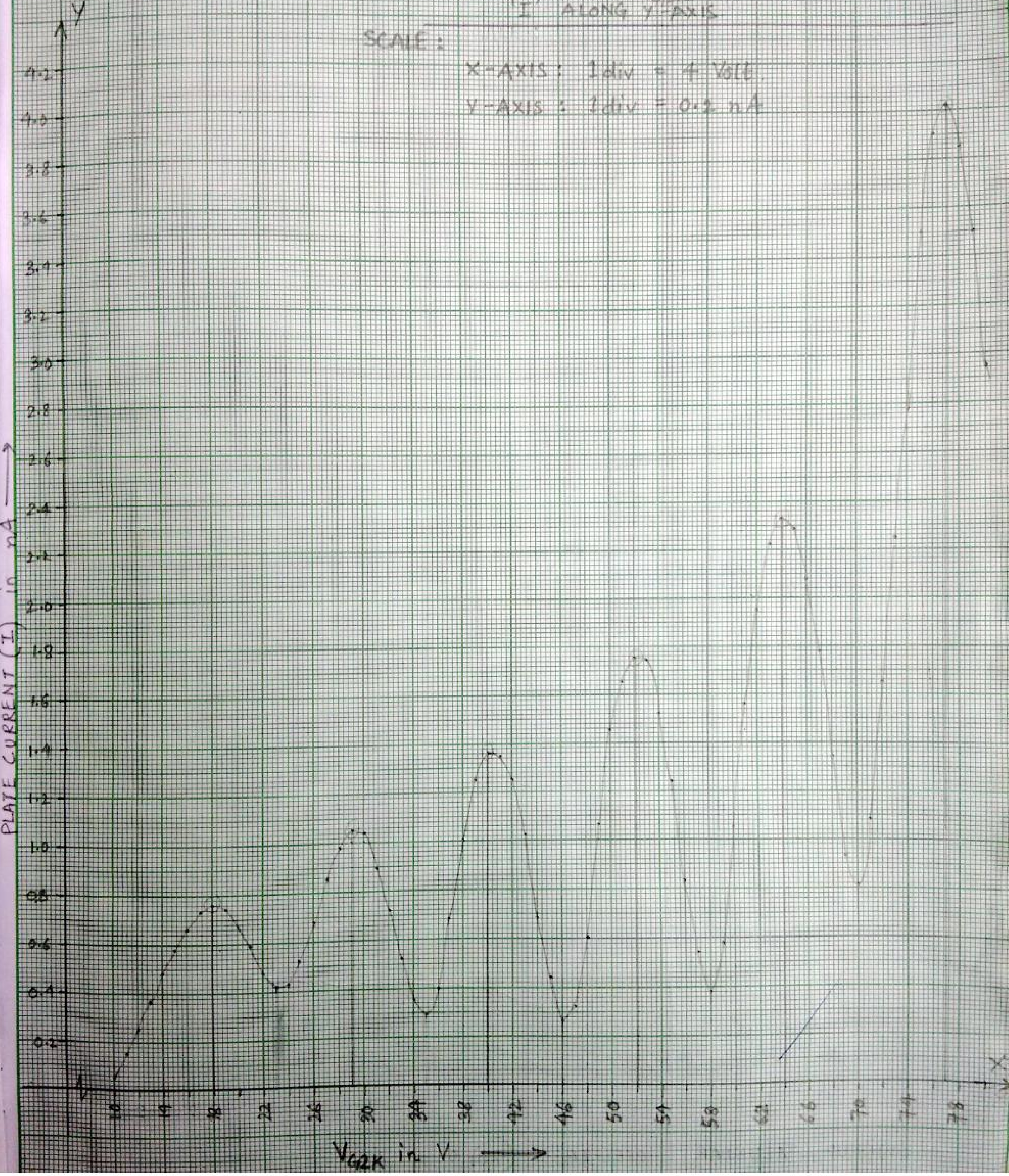


PLATE CURRENT (I) IN nA = 10^{-9} A	V_{G2K} IN VOLT
1.34	68
0.93	69
0.81	70
1.08	71
1.65	72
2.25	73
2.78	74
3.50	75
3.90	76
4.02	77
3.85	78
3.50	79
2.95	80

Table 2 : To measure the excitation potential from graph.

[illegible]

No. of Obs.	Distance between the peaks (V)	Average excitation potential (eV)
1.	11	$11.8 \times 1.6 \times 10^{-19}$
2.	11	$= 18.88 \times 10^{-19}$ eV
3.	12	
4.	12	
5.	13	

DISCUSSIONS : • At the time of experiment turn ON "manual/auto" switch to manual.

• Average horizontal distance between two picks is important. It will give the value of excitation energy.

[illegible]

EXPERIMENT No. 4

AIM : To determine the number of rulings of a plane diffraction grating by laser diffraction method and then to findout the wavelengths of the given unknown radiation.

APPARATUS : Spectrometer fitted with laser source and detector, grating.

THEORY AND THE WORKING FORMULA :
If a parallel beam of light of wavelength λ is coming out from diffraction grating, placed vertically on the optical bench then the diffracted rays from the grating will form at the focal plane, a number of primary maxima or different order numbers (n) on both sides of the central maximum of zero order.

If θ be the angle of deffraction of nth order primary maximum then

$$\sin\theta = nN\lambda \quad \text{———} \ (1).$$

where N = no. of rulings per centimeter of the grating surface. Hence,

$$\lambda = \sin\theta / nN \quad \text{———} \ (2).$$

Eq^n (2) is the working formula of the present experiment. If the value of n is known the wavelength λ of unknown rays can be found out.

Experimental Data :-

Table I. :- Calculation of total no. of rulings of the grating.

No. of obs.	No. of lines per inch (Specified on the grating) N_1	No. of liner per cm is $N = N_1/2.54$ cm.
1.	2500	984.

Table II :- Calculation of sine of ∠ of diffraction and measurement of wavelength.

Order No. (n)	Distance of central maximum from the grating (l) cm	Left of Principal maximum			Right of Principal maximum.			$\lambda = \frac{\sin\theta}{nN}$ Å	Mean λ in Å
		Distance of nth order primary maximum from central maxima (x) cm.	Distance of nth order primary maxima from grating $(S) = \sqrt{x^2+l^2}$ cm.	$\sin\theta = \frac{x}{S}$	Distance of nth order primary maxima from c.m. (x_1) cm.	Distance of n-th order primary maxima from grating $S_1 = \sqrt{x_1^2+l_1^2}$ cm.	$\sin\theta = \frac{x_1}{S_1}$		
1.		4.7	70.15	0.066	4.7	70.15	0.066	6707	
2.	70.	9.5	70.64	0.13	9.4	70.62	0.13	6758	6470.25
3.		14.5	71.48	0.20	14.2	71.42	0.19	6775	
4.		19.5	72.63	0.26	19.3	72.61	0.26	5741	

	Average $\sin\theta$
1.	0.066
2.	0.133
3.	0.20
4.	0.26

Recorded
[illegible]
22/10/14

Recorded
[illegible]
22/10/14

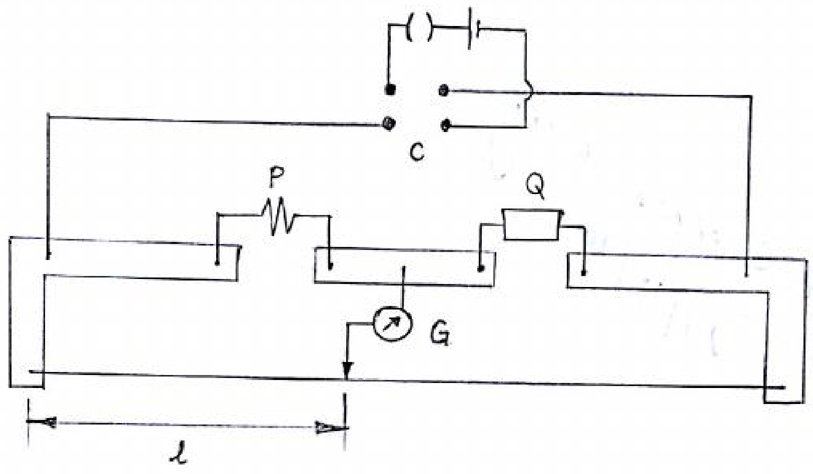
CONCLUSION :—

1. The grating should be placed vertically on the prism table.

2. The intensity of primary maximum on either side aren't becoming equal. This feature may be attributed to the improper placement of the grating on the prism table.

[illegible]
29/10/14

Connection of the circuit :-



EXPERIMENT NO. : 3

OBJECTIVE : Determination of unknown wavelength of a monochromatic light with the help of NEWTON's RINGS.

WORKING PRINCIPLE : When a beam of monochromatic light is incident normally on a combination of a Plano-convex lens L and a glass plate P, a part of each incident ray is reflected from the lower surface of the lens, and a part after refraction through the air film between the lens and the plate, is ~~ft~~ reflected back from the plate surface. These two reflected rays coherent. Hence the reflected rays will interfere and produce a system of alternate dark and bright rings with the point of contact between the lens and the plate as the centre. These rings are known as Newton's Rings.

If D_m is the diameter of the mth bright ring counted from the centre, we have

$$\frac{D_m^2}{4R} = (2m+1)\frac{\lambda}{2} \quad \text{—(i)}.$$

Where R = radius of curvature of the lower surface of the lens L, and λ = wavelength of the light. For (m+n)th bright ring from the centre, we have

$$\frac{D_{m+n}^2}{4R} = (2m+2n+1)\frac{\lambda}{2} \quad \text{—(ii)}$$

Where D_{m+n} = diameter of the (m+n)th ring.

From (i) & (ii), we get.

$$\lambda = \frac{D_{m+n}^2 - D_m^2}{4nR} \quad \text{—— (iii).}$$

Eq^n (iii). is used as the working formula for calculating λ, where diameters measured in cms. λ is obtained in cms.

CALCULATIONS :-

From the graph bet^n D^2 vs m,

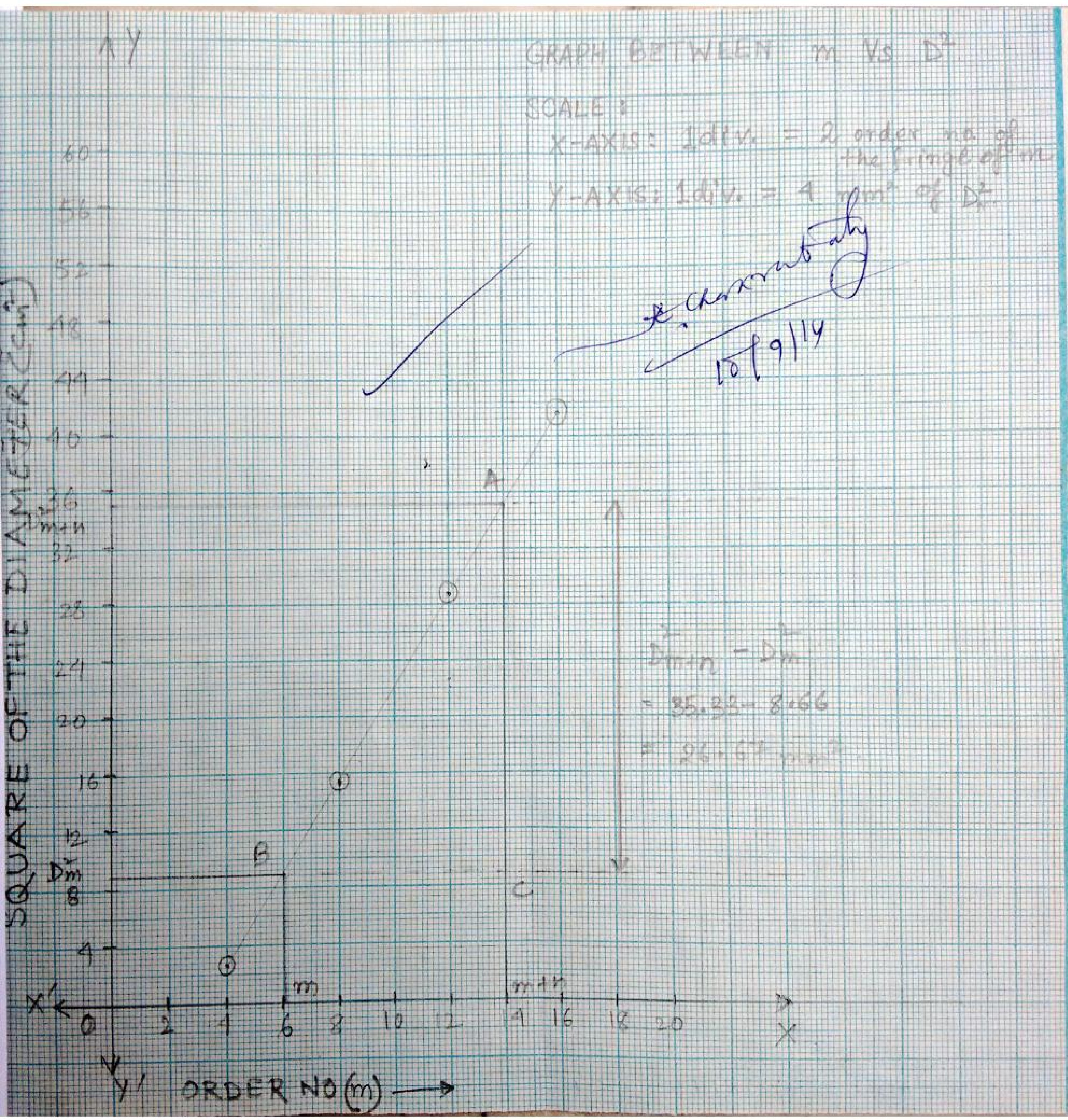
The slope of the line is $(AC/BC) = \left\{ \frac{D_{m+n}^2 - D_m^2}{n} \right\}$.

$$\lambda = \left\{ (D_{m+n}^2 - D_m^2) / 4nR \right\}$$

where n is the no. of rings & R supplied
{ R is the radius of curvature).

$$\lambda = \frac{(36.33 - 8.66)\ mm^2}{4(8)(136)\ mm} = \frac{26.67}{4352}\ mm$$

$$= 6.128 \times 10^{-3}\ mm.$$


GRAPH BETWEEN m Vs D²
SCALE :
X-AXIS : 1div. = 2 order no of the fringe of m
Y-AXIS : 1div. = 4 mm² of D²
SQUARE OF THE DIAMETER (cm²)
ORDER NO (m)
Y
Y'
X
X'
A
B
C
m
m+n
D_{m+n}^2
D_m^2
$D_{m+n}^2 - D_m^2$
= 35.33 − 8.66
= 26.67 mm²
10/9/14
0
2
4
6
8
10
12
14
16
18
20
4
8
12
16
20
24
28
32
36
40
44
48
52
56
60

EXPERIMENTAL RESULTS :

Table 1 : Determination of least Count.

PITCH OF THE SCREW p (mm)	Number of divisions 'n' on the circular scale	Least Count = P/n (mm)
1	100	1/100 = 0.01 mm

[illegible]

Table 2 : Determination of the diameter of ring.

Direction of the eye-piece movement.	Order no. of the fringe	Left (R_1)			Right (R_2)			Diameter $D_m = R_2 \sim R_1$ (mm)	Mean D_m (mm)	D_m^2 (mm^2)
		Main Scale Reading	Vernier Scale Reading	Total (mm)	Main Scale Reading	Vernier Scale Reading	Total (mm)			
L → R	16	56.5	4	56.54	49.5	87	50.37	6.17	6.45	41.6
R → L		55	39	55.39	48	67	48.67	6.72		
L → R	12	55	19	55.19	50	12	50.12	5.07	5.34	28.51
R → L		54.5	2	54.52	48.5	42	48.92	5.60		
L → R	8	54	65	54.65	50.5	7	50.57	4.08	3.95	15.60
R → L		54	61	54.61	50	80	50.80	3.81		
L → R	4	52.5	5	52.55	51	57	51.57	0.98	1.54	2.37
R → L		53	62	53.62	51	53	51.53	2.09		

[illegible] 20.8.17

Table 3 : Determination of the Wavelength of Monochromatic light.

D_m^2 (mm^2)	D_{m+n}^2 (mm^2)	n	R (supplied) mm	Wave length λ mm
8.66	36.33	8	136	6.128×10^{-3}

DISCUSSIONS :

(i). The Newton's ring experiment can be also used to find the wavelength of a monochromatic light. In this case, the radius of curvature of the convex surface of the given lens is supplied or is determined otherwise.

(ii). Since the first ~~few~~ few rings near the centre are deformed, they must be avoided while taking readings for the rings.

(iii). Care must be taken not to disturbed the lens and glass plate combination in any way during the experiment.

[illegible]
10/3/14

Circuit Diagram :

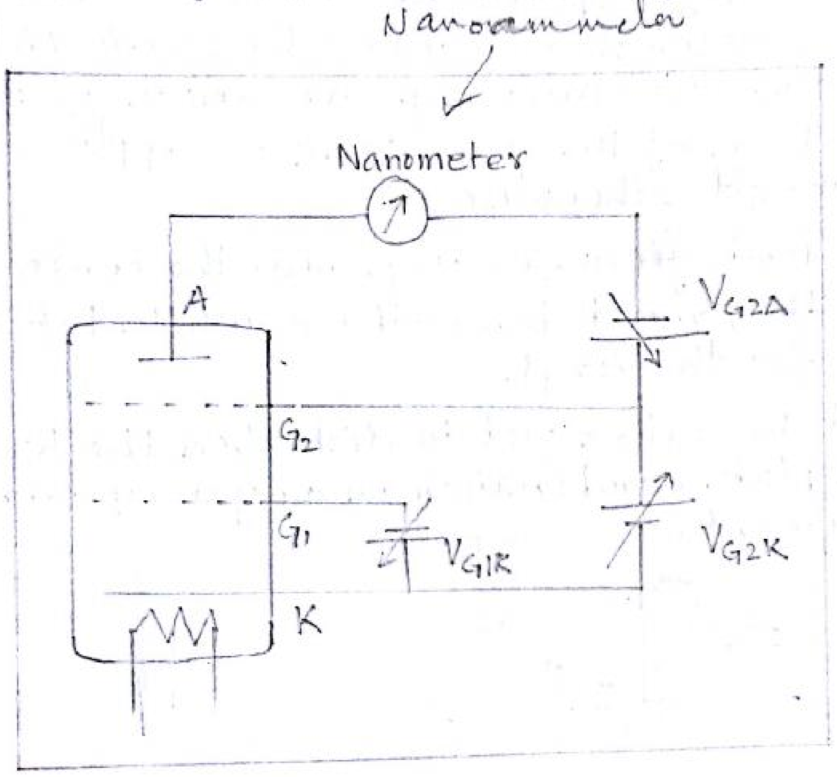


Fig. 5.1. Frank-Hertz Experiment Kit.

EXPERIMENT No. 2

OBJECTIVE : To study the time period of oscillation of a torsional pendulam for different loads and to find the torsional couple per unit twist of the suspension wire.

APPARATUS : Torsional pendulum, stop-watch, balance, slide calipers, etc.

THEORY : The equipment consists of a metal wire fixed rigidly from the upper end and to the lower end of the wire a metal disc is attached, whose moment of inertia can easily be calculated. If the disc is rotated through an $\angle\theta$ and released, the twist in the wire rotates the disk back towards equilibrium. It overshoots and oscillates back and forth like a pendulum, hence the name. The time period of the pendulum can be measured with the help of a stop watch.
When the torsional pendulum disc is twisted away from equilibrium by an $\angle\theta$, the twisted wire exerts a restoring torque proportional to that angle.

$$T = - C\theta \qquad \text{———} \quad (1).$$

Here C is the torsional couple per unit twist.
If the wire is thick and is made of stiff material, C is large. If the wire is long, C is small. If a mass with moment of inertia 'I' is attached to the rod, the torque will give the mass an angular acceleration "α" according to

$$I\alpha = I\frac{d^2\theta}{dt^2}$$

$$I\frac{d^2\theta}{dt^2} = -C\theta \qquad \text{or, } \frac{d^2\theta}{dt^2} = \frac{-C\theta}{I}$$

$$\text{or, } \frac{d^2\theta}{dt^2} + \frac{C\theta}{I} = 0$$

$$\text{or, } \frac{d^2\theta}{dt^2} + \omega^2\theta = 0 \quad \text{where } \omega^2 = \frac{C}{I}.$$

$$\text{or, } T = \frac{2\pi}{\omega} = 2\pi\sqrt{\left(\frac{I}{C}\right)}$$

$$\text{or, } T^2 = 4\pi^2\left(\frac{I}{C}\right) \quad \text{—— (2).}$$

If the mass is placed in the lower end in the form of a disc, relation (2) becomes

$$T^2 = 4\pi^2\left(\frac{I}{C}\right) = \left(\frac{4\pi^2}{C}\right)\left(\frac{MR^2}{2}\right)$$

A plot of T^2 vs I gives a straight line passing through the origin. Form the slope of the curve $4\pi^2/C$, 'C' can be determined.

EXPERIMENTAL DATA :

(1). Mass of Bigger - circular disc = 1000gm

(2). Mass of Smaller - circular disc = 500gm

(3). Mass of Conical Body = 627 gm

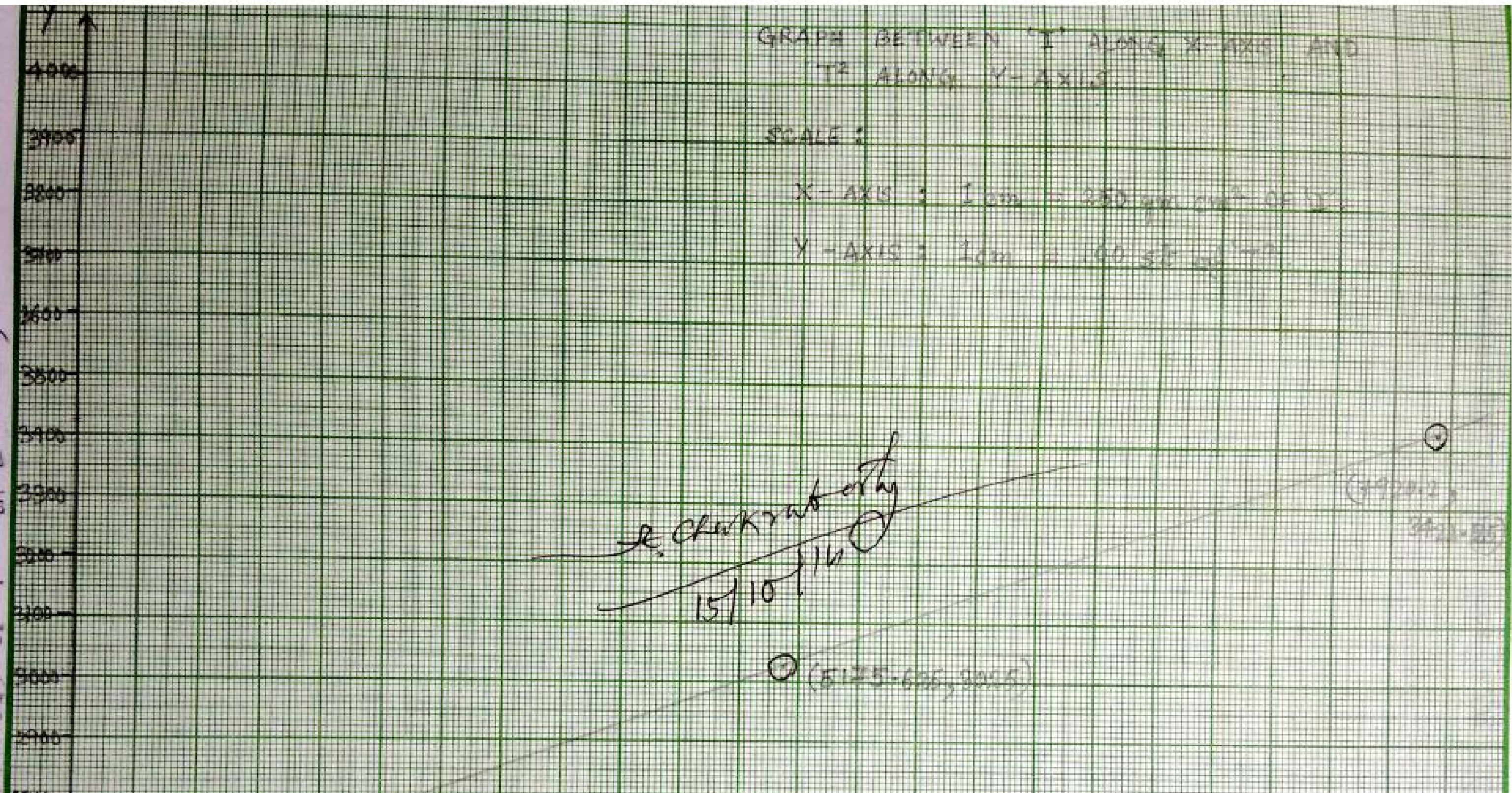

GRAPH BETWEEN 'I' ALONG X-AXIS AND
T² ALONG Y-AXIS
SCALE :
X-AXIS : 1 cm = 250 gm cm² of 'I'
Y-AXIS : 1 cm = 100 s² of T²
(TIME PERIOD)² or T² in s²
MOMENT OF INERTIA (I) in gm cm²
(2744.88, 2669.79)
(5125.696, 3056)
4000
3900
3800
3700
3600
3500
3400
3300
3200
3100
3000
2900
2800
2700
2600
2500
2500
2750
3000
3250
3500
3750
4000
4250
4500
4750
5000
5250
5500
5750
6000
6250
6500
6750
7000
7250
7500
7750
8000
O
X
Y

Table 1 : Determination of Moment of Inertia of the given object.

Type of Object	Mass (M) (in gm)		Dimension of the object (Radius R) (in cm) Obs 1	Obs 2	Obs 3	Mean	Moment of Inertia (I) $gmcm^2$
Bigger Disc	1000	(D)	7.96	7.95	8.03	7.98	$I_1 = \frac{1}{2}MR^2$
		(R)	3.98	3.98	4.01	3.98	= 7920.2
Smaller Disc	500	(D)	9.11	9.02	9.21	9.1	$I_2 = \frac{1}{2}MR^2$
		(R)	4.55	4.51	4.6	4.55	= 5175.625
Conical Body	627	(D)	7.62	7.71	7.62	7.64	$I_3 = \frac{3}{10}MR^2$
		(R)	3.8	3.85	3.8	~~7.~~ 3.82	= 2744.83

[illegible] 20/8/19

[illegible]

Table 2 : Determination of torsional Couple

Type of Object	'I' From Table 1	Time Period (T) (in sec) Obs 1	Obs 2	Obs 3	Mean	C From T^2 vs I Curve
Bigger Disc	~~7880.45~~ 7920.2	58.5	58	59	58.50	$271.24\ gm^{-1}cm^{-2}s^2$
Smaller Disc	5175.625	55.5	55	54.5	55.00	
Conical Body	2744.83	51.5	52.5	51	51.67	

CALCULATION : A graph is drawn between 'I' and T^2 along X-axis and Y-axis respectively. The slope of the curve is

$m = (T_2^2 - T_1^2)/(I_2 - I_1) = \{(58.5)^2 - (51.67)^2\}/(7920.2 - 2744.83)$

$= (3422.25 - 2669.79)/(5175.37)$

$= (752.46)/(5175.37)$

$= 0.1454 \text{ gm cm}^2 \text{ s}^{-2}$

$C = 4\pi^2 / m$

$= 4(3.14)^2 / 0.1454$

$= 4(9.86)/0.1454$

$= 39.43 / 0.1454$

$= 271.24 \text{ gm}^{-1} \text{ cm}^{-2} \text{ s}^2$

C per unit twist $= 10^2 \times C = 27124 \text{ gm}^{-1} \text{ cm}^{-2} \text{ s}^2$.

DISCUSSION : To determine the time period of torsional pendulum using stop watch.

I). All readings should be taken correctly.

II). Metal wire should not vibrate

III), Lab should be well lit.

[illegible] 27.08.17

EXPERIMENT NO.: 1B

AIM : Determination of end correction s of a metre bridge.

APPARATUS : Metre bridge, wires, resistance boxes, Galvanometre, etc.

THEORY : In meter bridge, small resistance exists at each end of the bridge wire due to bad soldering at the two ends of the wire, non-coincidence of the two ends of the wire with the 0 to 100 marks of the meter scale and very small resistance of the copper strips at the two ends. These resistances at the two ends of the bridge wire are known as end errors. Let the end errors at the left and right ends of the bridge wire are respectively equal to the resistances of lengths λ_1 and λ_2 cm of bridge wire. If two unequal resistances P and Q be respectively inserted in the left and right gaps of the metre bridge wire and a null point is obtained at a distance l_1 cm from the left end then by Wheatstone bridge Principle,

$$\frac{P}{Q} = \frac{l_1 + \lambda_1}{(100 - l_1) + \lambda_2} \quad \text{——— (1)}.$$

On interchanging the positions of P and Q if null points are obtained at a distance l_2 cm from the left end then

$$\frac{Q}{P} = \frac{l_2 + \lambda_1}{(100 - l_2) + \lambda_2} \quad \text{——— (2)}.$$

Solving eq^n (1) & (2), we get,

$$\lambda_1 = \frac{l_1 - r l_2}{r - 1}, \quad \lambda_2 = \frac{r l_1 - l_2}{r - 1} - 100$$

where $r = R = \frac{P}{Q}$.

Determination of end corrections of a metre bridge.

No. of obs.	R in the left gap	R in the right gap	Ratio $R = P/Q$	Null points in cm with			λ_1 in cm	λ_2 in cm	Mean λ_1 (cm)	Mean λ_2 (cm)
				Direct Current	Reserved current	Mean				
1	P= 3 Q= 1	Q~~4~~=1 P=3	3	74.7 l_1 23.2 l_2	—	—	2.55	0.45	3.02	0.89
2	P= 5 Q= 1	Q= 1 P= 5	5	83.6 l_1 14.2 l_2	—	—	3.15	0.95		
3.	P= 7 Q= 1	Q=1 P= 7	7	88.2 l_1 9.7 l_2	—	—	3.38	1.98		

Recorded
AC
29/10/14

Recorded
Tanima Das.
29/10/14.

DISCUSSION :

1. The end errors arise due to the unequal resistances of the connectors of the two ends of the meter bridge.
2. Connections must be done properly so that there is no gaps where wires are joined with the binding screws.

Results : Hence, the first end correction

$\lambda_1 = 3.02$ cm.

the second end correction

$\lambda_2 = 0.89$ cm.

A. Chakraborty
05/11/14

EXPERIMENT No.: 1A

AIM : To determine the errors in the different measurements.

ACCURACY : It refers to the closeness of a measured value to a standard or known value. For example, if in lab you obtain a weight measurement of 3.2 kg for a given substance, but the actual or known weight is 10 kg, then your measurement is not accurate. In this case, your measurement is not close to the known value.

PRECISION : It refers to the closeness of two or more measurements to each other. Using the example above, if you weigh a given substance fine times, and get the weights 3.2 kg each time, then your measurement is very precise. Precision is independent of accuracy. You can be very precise but inaccurate, as described above. You can also be accurate but imprecise. For example. if on averge, your measurements for a given substance are close to the known value, but the measurements are far from each other, then you have accuracy without precision.

A. Volume of the cylinder in rigidity modulus experiment.

Measurement of the diamcter using the slide calipers.
Vernier Constant (V.C) = (value of 1 main scale division - value of 1 vernier scale division) x (value of 1 smallest main scale division)
Value of 1 smallest main scale division of the slide calipers ⇒ 'm' vernier scale division = 'n' main scale division. or, $1 \text{ V.S.D.} = \frac{n}{m} \text{ M.S.D.}$

Table 1: (a) Determination of vernier constant of the slide Calipers.

Value of 1 smallest main scale division (A) cm	Value of 1 vernier div. (B), $B = (n/m) A$ (cm)	Vernier constant $= (A - B) = (1 - n/m) A$ cm.
0.1	0.09	0.1 - 0.09 = 0.01

(b) Determination of the diameter of the cylinder by slide callipers

No. of obs.	Main Scale Reading MSR (cm)	Vernier Scale Reading, VSR (cm)	Diameter = MSR + VSR(VC) (cm)	Mean Diameter, d (cm).
1	6.3	0	d_1 = 6.30	
2	6.3	6	d_2 = 6.36	
3	6.3	5	d_3 = 6.35	6.33
4	6.3	4	d_4 = 6.34	
5	6.3	3	d_5 = 6.33	
6	6.3	2	d_6 = 6.32	

Table 2 : Determination of the length of the cylinder by metre scale

No. of Observation	length, h (cm)
1.	10.2
2.	10.3
3.	10.1
4.	10.2
5.	10.2
6.	10.2

Recorded Tanmay Das. 7/11/14

Recorded [illegible] 11/10/14

CALCULATIONS :-

Volume of the cylinder, $V = \frac{\pi d^2}{4} h$ cm³.

$$= \frac{(3.14)(6.33)^2}{4}(10.2) \text{ cm}^3.$$

$$= 320.83 \text{ cm}^3.$$

Analysis :-

(a). Determination of maximum proportional error.

Maximum proportional error in the measurement of volume (V) :-

$$V = \frac{\pi d^2}{4} h \quad \text{or,} \quad \ln V = \ln \pi + 2 \ln d - \ln 4 + \ln h.$$

$$\frac{\delta V}{V} = 2\frac{\delta d}{d} + \frac{\delta h}{h}$$

δd = Value of vernier division = 1 × 0.01cm = 0.01cm.

δh = 0.1cm.

$$\therefore \frac{\delta V}{V} = 2 \cdot \left(\frac{0.01}{d}\right) + \frac{0.1}{h}$$

$$= 2\left(\frac{0.01}{6.3}\right) + \frac{0.1}{10.1} = 0.01307$$

$$\rightarrow \frac{\delta V}{V} \times 100\% = 0.01307 \times 100$$

$$= 1.307\ \%.$$

(b). Determination of standard deviation in measurement of diameter of the cylinder:

No. of Obs. (n)	Diameters (d_i) cm.	Mean diameter $\bar{d} = \frac{\Sigma d_i}{n}$ cm.	Deviation $\bar{d} - d_i = x_i$ cm	standard deviation, $\sigma = \sqrt{\frac{\Sigma x_i^2}{n-1}}$ cm.
1.	d_1 = 6.30	6.33	0.03	2.19×10^{-2}
2.	d_2 = 6.36		-0.03	
3.	d_3 = 6.35		-0.02	
4.	d_4 = 6.34		-0.01	
5.	d_5 = 6.33		0.00	
6.	d_6 = 6.32		0.01	

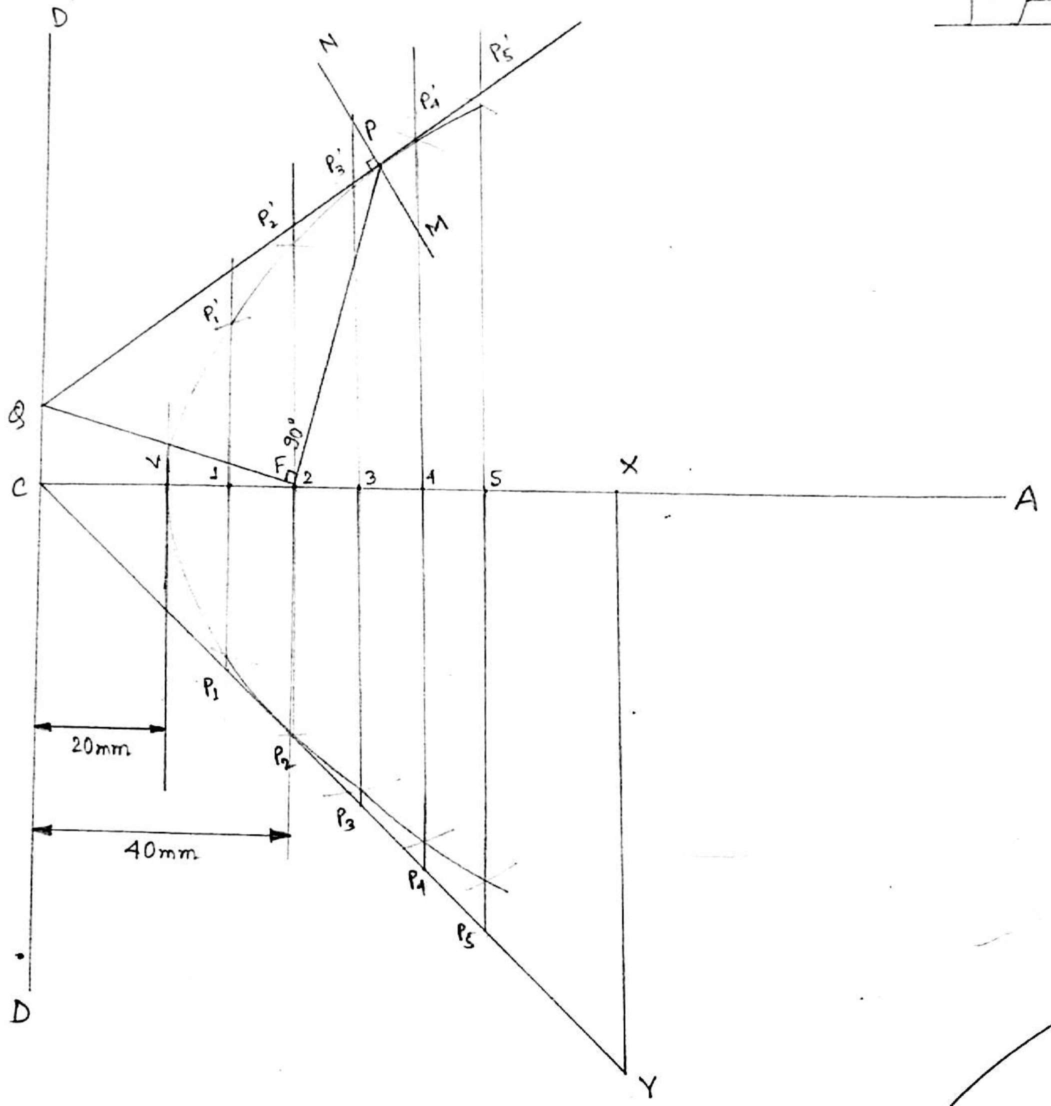
DISCUSSION :

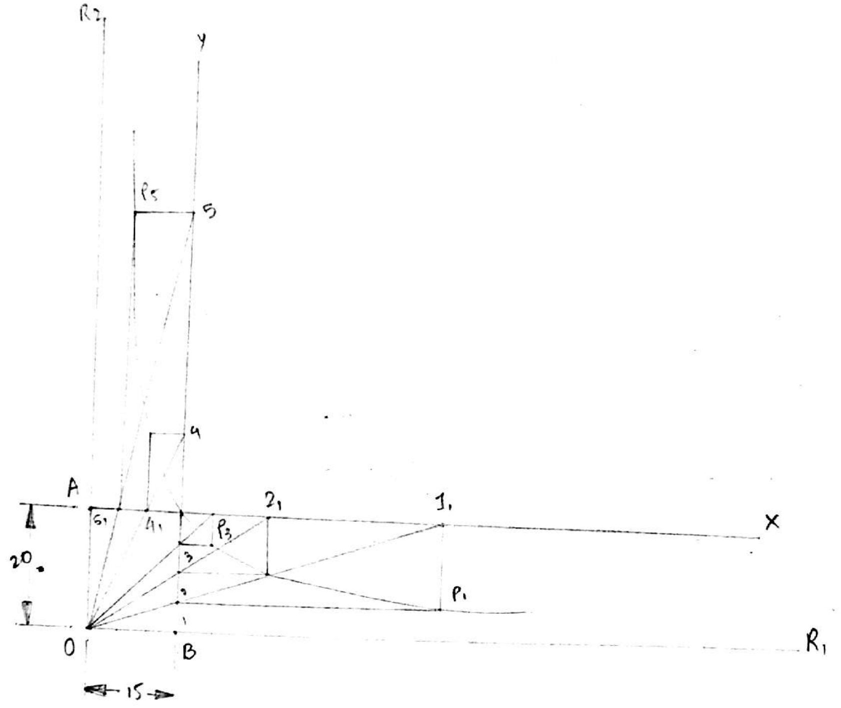
Intrument Error:- When the studs are in touch with each other and the zero of the circular scale has crossed the reference line on the tube, the intrumental error is to be considered to be negative. This error is,, therefore required to be added to the apparent reading. But if the zero on the circular scale fails to reach the reference line, the error is considered to be positive and is required to be subtracted from the apparent reading.

Backlash error :- When a screw moves through a threaded hole there is always some misfit between the two. As a result, when the direction of rotation of the screw is reversed, axial motion of the screw takes place only after screw head is rotated through a certain angle. This lag between the axial and the circular motions of the screw head is termed as the blacklash error.

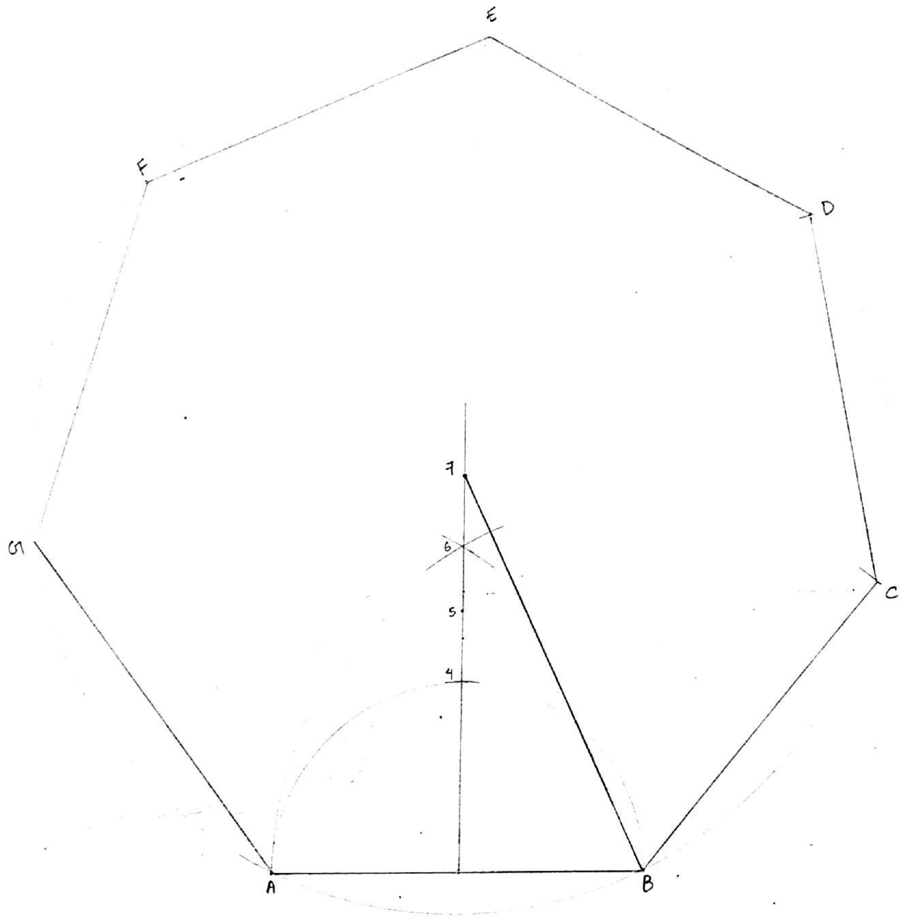
[illegible]
18/11/14

PARABOLA





HYPERBOLA



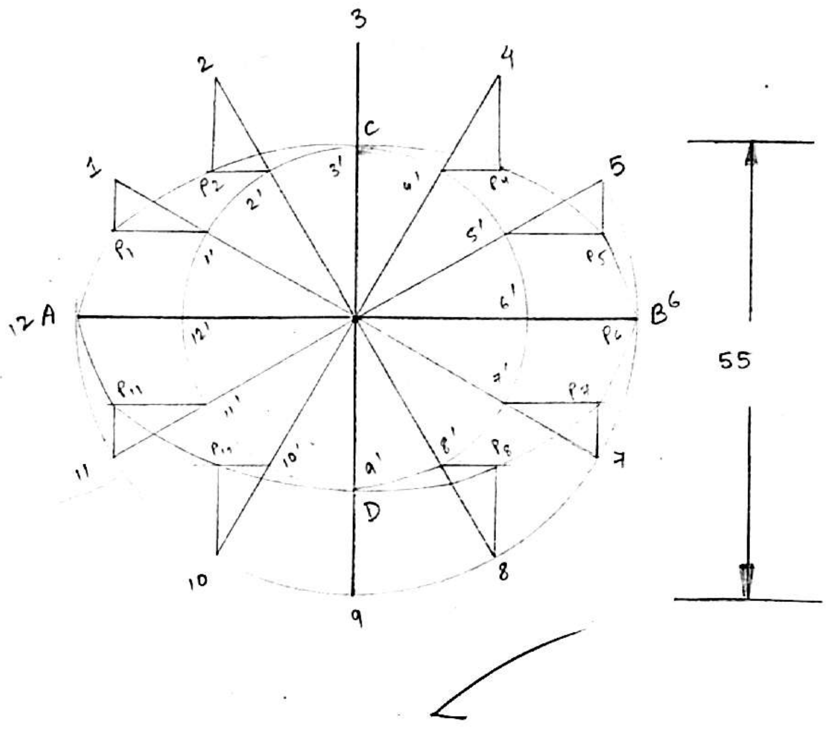
AB = 80 mm

HEPTAGON

9/10

NAME : RAJ DEB			JIS COLLEGE OF ENGINEERING
BTECH , 1st YEAR			
EE - A1	42	EE/16/03	TITLE :
DATE OF COMMENCE			
DATE OF SUBMISSION			SHEET NO. - 52
SCALE : 1:1			CHECKED BY [illegible]

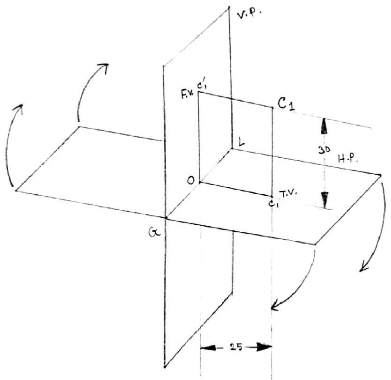
ELLIPSE

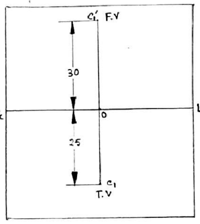


AB = 90 mm

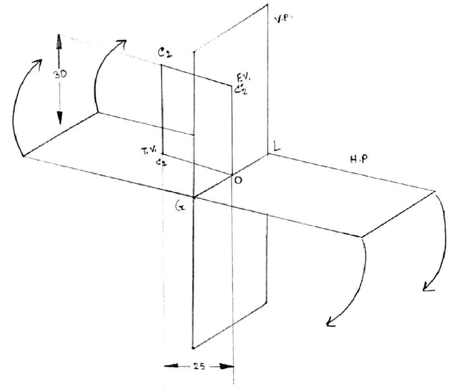
CD = 55 mm

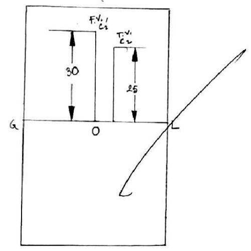
A point C_1 is 30 mm above H.P and 25mm infront of V.P. Draw the projection of the point.



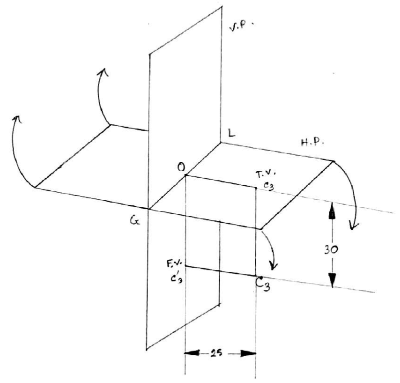


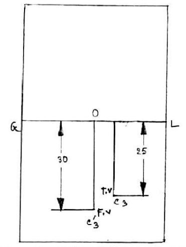
A point C_2 is 30mm above H.P. and 25 mm behind V.P. Draw the projection of the point.



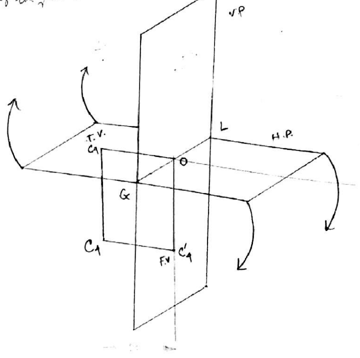


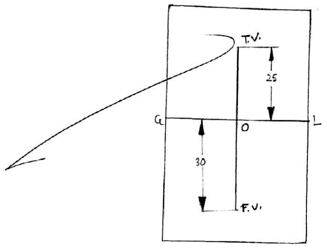
A point C_3 is 30 mm below H.P. and 25 mm in front of V.P. Draw the projection of the point



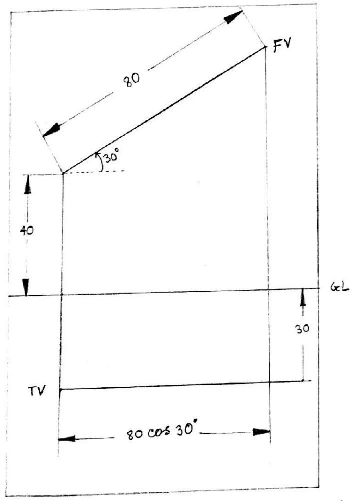


A point C_4 is 30 mm below H.P. and 25 mm behind V.P. Draw the projection of the point.

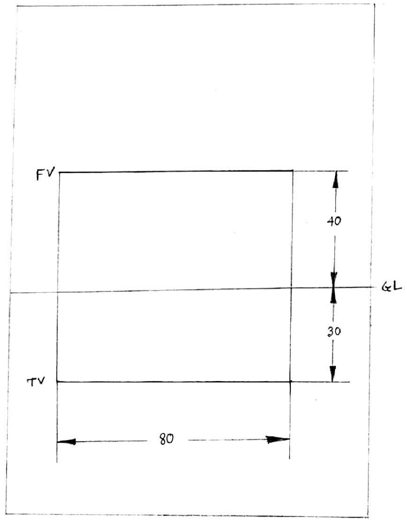




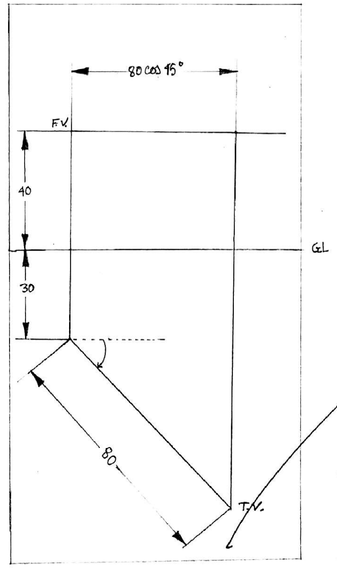
A line AB is 80 mm long and parallel to V.P. and making an angle 30° with HP and it is in front of V.P. by 30 mm and 40 mm above the H.P. Draw projection of this line.



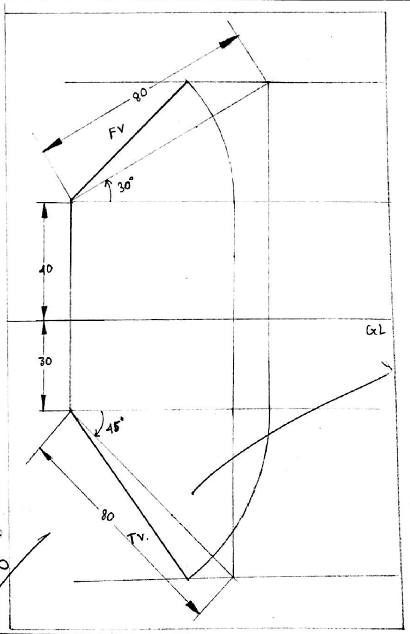
The same line is parallel to both V.P and H.P.



The same line making an angle 15° with V.P.



The same line making an angle 30° with H.P. and 45° with V.P.



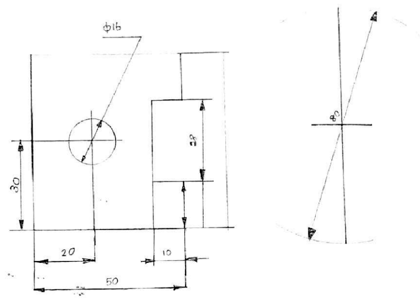
26/10/16

10/10

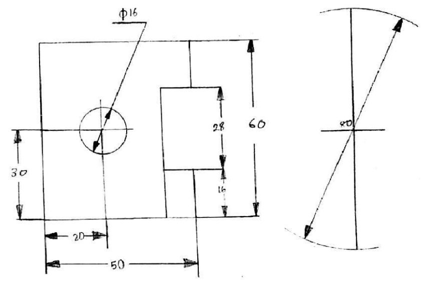
SUBHRADEEP GUPTA			JIS COLLEGE OF ENGINEERING
CLASS-BTECH YEAR-1ST			
B-EE1B	R-71	ID-JIS/16/517	TITLE- PROJECTION OF POINT PROJECTION OF LINE
DATE OF COMMENCE			
DATE OF SUBMISSION 26.10.16			SHEET NO- 4
SCALE-1:1			CHECKED BY-

LINE	GENERAL APPLICATIONS	
A	VISIBLE OUTLINES	
B	IMAGINARY LINES OF INTERSECTION. DIMENSION LINES PROJECTION LINES LEADER LINES HATCHING	OUTLINES OF REVOLVED SECTIONS IN PLACE SHORT CENTRE LINES
C	LIMITS OF PARTIAL OR INTERRUPTED VIEWS AND SECTIONS, IF THE LIMIT IS NOT A CHAIN THIN LINE	
D	LONG - BREAK LINE	
E	HIDDEN OUTLINES HIDDEN EDGES	
F	HIDDEN OUTLINES HIDDEN EDGES	
G	CENTRE LINE LINES OF SYMMETRY TRAJECTIONS	
H	CUTTING PLANES	
J	INDICATION OF LINES OR SURFACES TO WHICH A SPECIAL REQUIREMENT APPLIES	
K	OUTLINES OF ADJACENT PARTS ALTERNATIVE AND EXTREME POSITIONS OF MOVABLE PARTS CENTROIDAL LINES	INITIAL OUTLINES PRIOR TO FORMING. PARTS SITUATED INFRONT OF THE CUTTING PLANE.

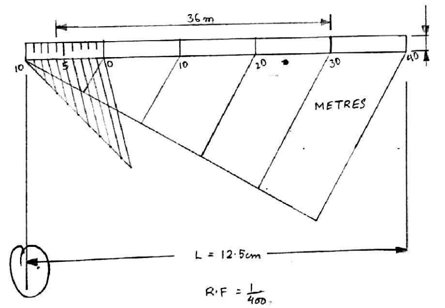
ALIGNED SYSTEM



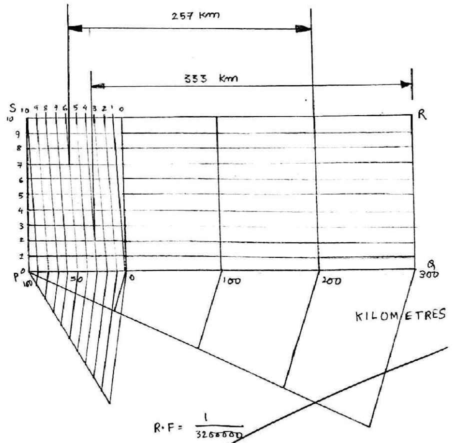
UNIDIRECTIONAL SYSTEM



PLANE SCALE



DIAGONAL SCALE

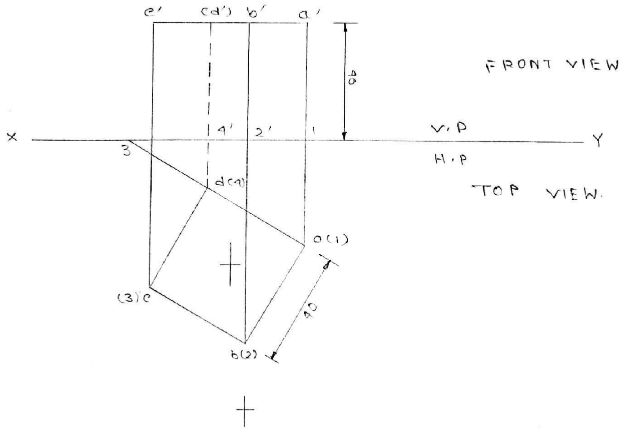


9·5 / 10

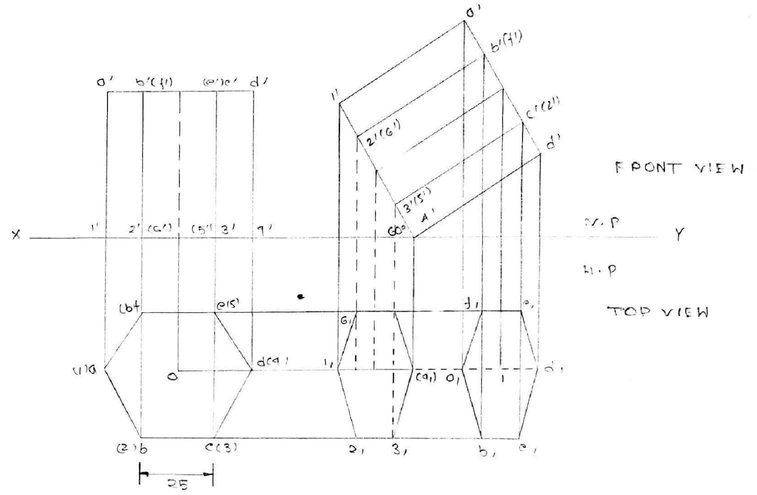
31/08/16

NAME: RAJ DEB	JIS COLLEGE OF ENGINEERING
CLASS: B.TECH, 1st YEAR	
B- R- ID NO:	TITLE:
DATE OF COMMENCE:	
DATE OF SUBMISSION:	SHEET No.-
SCALE:	CHECKED BY-

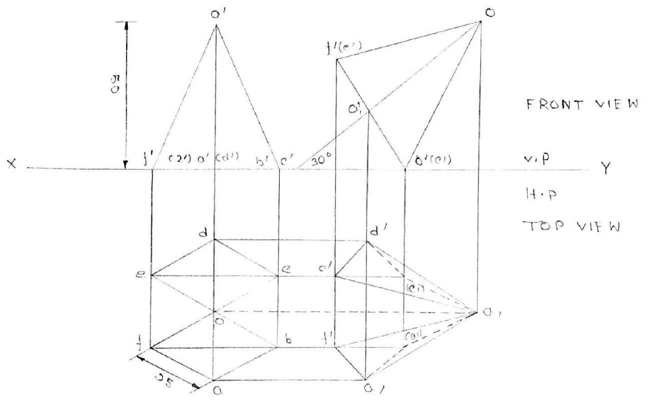
Draw the projection of a cube of 40 mm side resting with a face on H.P such that on it's vertical faces inclined at 30° to V.P



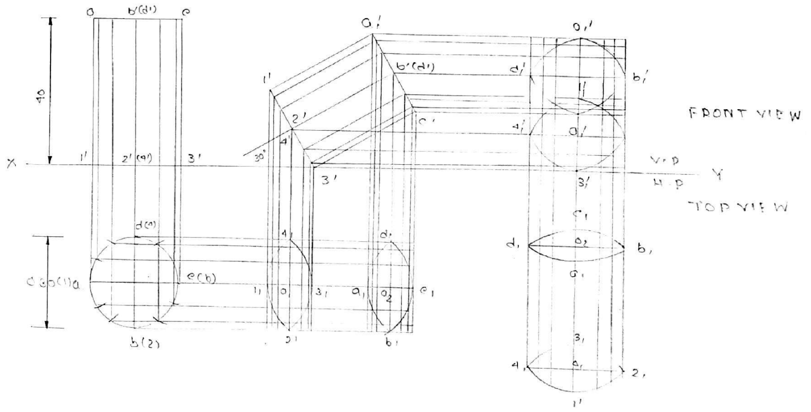
A hexagonal prism, side of base 25 mm long rests with one of it's base corners on H.P such that it's base makes an angle of 60° to H.P and it's axis is parallel to V.P Draw it's projection.



A hexagonal pyramid side of base 25 mm and axis 50 mm long rests with 1 of the five edges of it's base inclined at 30° to HP parallel to HP. Draw it's projection.



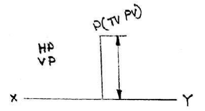
Draw the projection of a cylinder, base 30 mm and axis 40 mm long rests with a point of its base circle on H.P such that the axis is making an angle of 30° with H.P and it's top view.



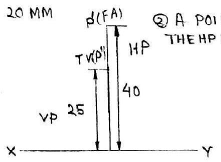
ALL DIMENTION ARE IN MM

NAME- BIDYUT BISWAS			JIS COLLEGE OF ENGINEERING	
CLASS: B TECH(EE) IST Y				
B-1A	R-38	ID-JIS/15/325	TITLE : PROJECTION OF POINTS, LINES AND SURFACE. IST ANGLE PROJECTION OF LINS A SURFACE - HEXAGON	
DATE OF COMMENCE				
DATE OF SUBMISSION				SHEET NO-ME/191/04
SCALE				CHECKED BY-

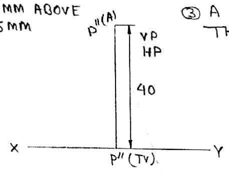
① A POINT IN THE HP AND 20 MM BEHIND THE VP



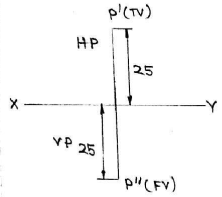
② A POINT 40MM ABOVE THE HP AND 25MM



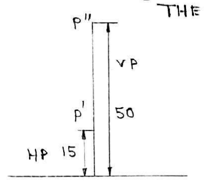
③ A POINT IN THE VP AND 40mm ABOVE THE HP



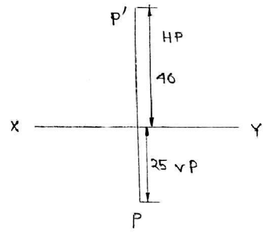
④ A POINT 25MM BELOW THE HP AND 25MM BEHIND THE VP



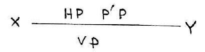
⑤ A POINT 15 MM ABOVE THE HP AND 50MM BEHIND THE VP



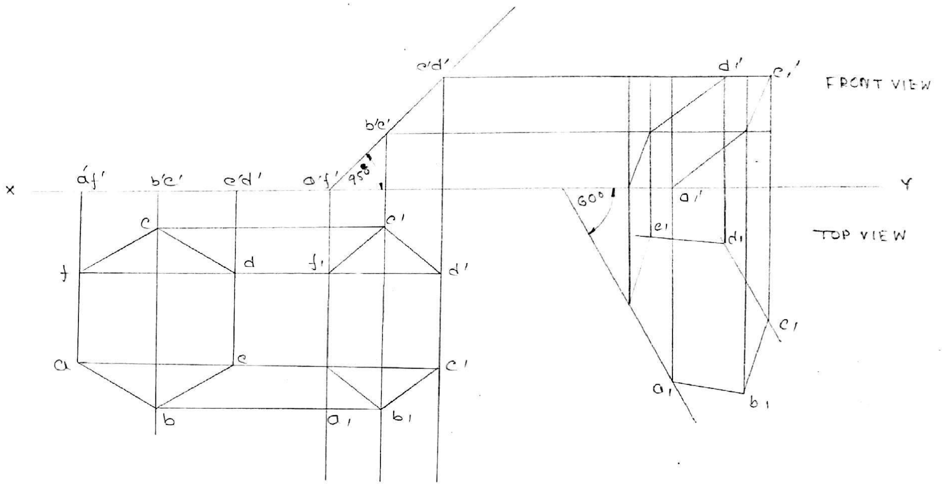
⑥ A POINT 40 mm BELOW THE HP AND 25MM INFRONT OF THE VP



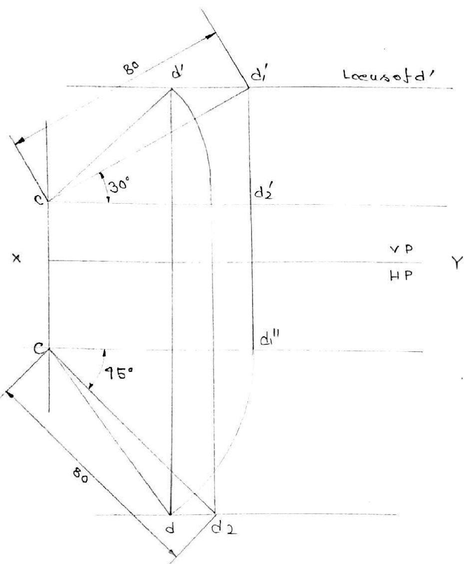
⑦ A POINT IN BOTH THE HP AND VP



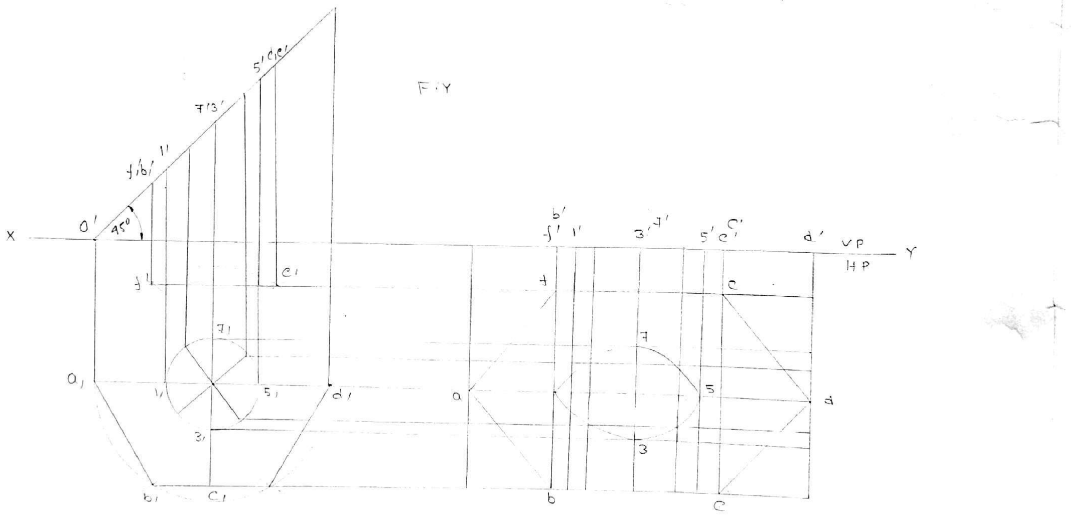
PROBLEM:- DRAW THE PROJECTION OF A REGULAR HEXAGON OF 30MM SIDE HAVING ONE OF IT'S SIDE IS THE HP AND INCLINED AT 60° TO THE VP AND IT'S SURFACE MAKING ANGLE OF 45° WITH THE HP.



PROBLEM 153:- A LINE CD 80MM LONG IS INCLINED AT AN ANGLE OF 30° TO HP AND 45° TO VP THE POINT ~~C 15 20~~ MM ABOVE THE HP AND 30MM INFRONT OF VP DRAW A PROJECTION OF THE STREIGHT LINE.



Cd = TOP VIEW
C'd' = FRONT VIEW



NAME - BIDYUT BISWAS	
CLASS:- B TECH(EE) 1ST YEAR	
B-1A | R-38 | ID- 21S/15/03	TITLE - PROJECTION OF POINT, LINES AND SURFACES, FIRST ANGLE PROJECTION, PROJECTION OF LINES AND SURFACES HEXAGON
DATE OF COMMENCE	
DATE OF SUBMISSION	SHEET NO - ME 191/03
SCALE - 1:1	CHECKED BY

[DETERMINATION OF DISSOLVED OXYGEN]

AIM :→ Determination of dissolved oxygen in a given tap water sample (Winkler method).

THEORY :→ Dissolved oxygen is very important for aquatic life. 5-7 ppm of dissolved oxygen is present in unpolluted water. In presence of good amount of dissolved oxygen (> 8 ppm), aerobic bacteria lead to oxidation of organic compounds present in water. The decrease in dissolved oxygen in turn decreases the population of aquatic life. Dissolved oxygen is usually determind by Winkler's method, It is based on the fact that oxygen oxidizes potassium iodide to iodine. The liberated iodine is titrated against standard sodium thiosulphate (also known as hypo) solution using starch as an indicator. Since dissolved oxygen in water is in molecular state, it as such cannot oxidize KI. Hence manganese hydroxide, obtained by the action of KOH on manganese sulphate ($MnSO_4$) helps to convert molecular oxygen to atomic oxygen by the reaction with H_2SO_4. The atomic oxygen reacts with KI and produce I_2.

$$MnSO_4 + 2KOH \longrightarrow Mn(OH)_2 + K_2SO_4$$

$$2Mn(OH)_2 + O_2 \longrightarrow 2MnO(OH)_2$$

$$MnO(OH)_2 + H_2SO_4 \longrightarrow MnSO_4 + 2H_2O + [O]$$

$$2KI + H_2SO_4 + [O] \longrightarrow K_2SO_4 + H_2O + I_2$$

$$2Na_2S_2O_3 + I_2 \longrightarrow Na_2S_4O_6 + 2NaI$$

Apparatus :→

- Stoppart bottle (250 ml)
- Burette
- Conical flask
- Pipette (5 ml)
- Dropper
- 2 Measuring Cylinders (10 & 250 ml)

Reagent :→

- Manganous Sulphate Solution
- Concentrated H_2SO_4
- Alkaline KI (20%)
- Starch indicator Sol^n
- $KMnO_4$ Solution (N/10)
- Potassium Oxalate Solution
- Hypo Solution

Procedure :→ Take 200 ml of water sample in a 250 ml stoppard bottle avoiding as far as possible contact with air, then add 0.9 ml of conc. H_2SO_4 and 0.2 ml $KMnO_4$ (N/10) solution. Stopper the bottle and shake the content vigorously. If the permanganate colour disappears with in 5 minutes, add additional amount of $KMnO_4$. Add 0.5 ml of potassium oxalate (2% solution), stopper and shake well. Add additional amount of oxalate if permanganate colour is not discharged with in 10 minutes. Add 2 ml of $MnSO_4$ (4.8%) Solution followed 3 ml of alkaline KI solution. Stopper and shake and allow the ppt. to settle. Now add 1 ml of concentrated H_2SO_4 solution and shake untill the ppt. completely dissolved. Take 100 ml of this solution with the help of measuring cylinder (250 ml) and titrate slowly against N/100 hypo solution. When the colour of the solution is very yellowish add 2 ml of freshly prepared starch solution, the solution will turn blue. Continue the titration untill the disappearance of the blue colour and note down the volume of hypo solution.

Result & Calculation :→

Table :– Determination of total amount of dissolved oxygen :→

No. of Obs.	Volume of Sample Taken (ml)	Burette Reading		Volume of Standard Hypo Solution Consumed (ml)	Concordant Volume of Hypo Solution (ml)	Amount of Dissolved Oxygen (ppm)
		Initial	Final			
{1}	100	0	3·9	3·9	3·9	3·12
{2}	100	3·9	7·8	3·9		
{3}	100	7·8	11·8	4·0		

Volume of sample water taken for titration (V_1) = 100 ml

volume of hypo solution required for titration = V_2 = 3·9 ml

Strength of hypo solution (S_2) = $\frac{1}{100}$(N)

Strength of water sample due to dissolved oxygen (S_1) = ?

From the normality equation, we know that

$$V_1 S_1 = V_2 S_2$$

$$\Rightarrow S_1 = \frac{V_2 S_2}{V_1} (N)$$

$$\Rightarrow S_1 = \frac{3 \cdot 9 \times 1}{100 \times 100} (N) = 0 \cdot 00039 (N)$$

DATE: 07-04-14

∵ Equivalent wt. of $O_2 = 8$

Hence the amount of oxygen = (0.00039×8) gm/lit
$= (0.00039 \times 8 \times 1000)$ ppm
$= 3.12$ ppm

The amount of O_2 present in tap water sample = 3.12 ppm

Discussion :→

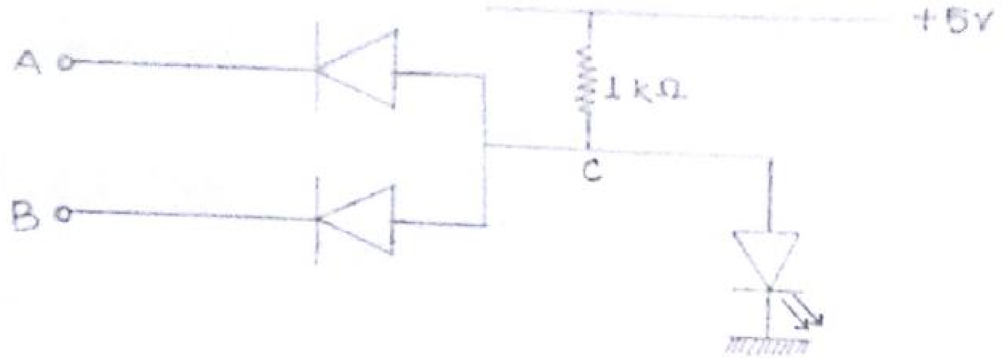
1. Additional amount oxalate solution is required if the permaganate colour is not discharged within 10 minutes.
2. starch combines with iodine to form blue coloured surface complex.
3. starch should be added near the end point. If concentration of iodine is high in solution, the complex formed is highly stable and cannot be broken by Hypo to give the end point.
4. At the end point blue colour will be disappear.
5. The interference due to certain oxidizing agent such as NO_2^- or reducing agent like Fe^{+2} or SO_3^{-2} is removed by excess $KMnO_4$ solution in acidic medium.

Conclusion :→

The calculated amount of Dissolved Oxygen was found to be 3.12 ppm.

Expt. No. :- 08 :- Design of Logic Gates Using Diode - Transistor Logic

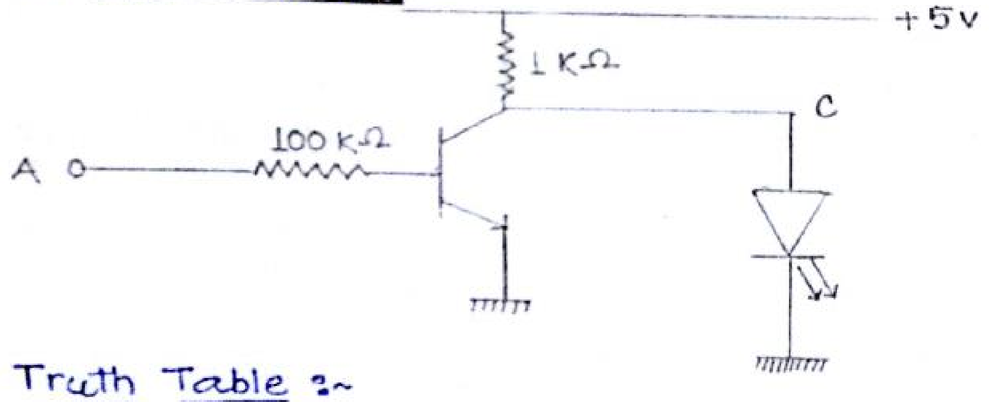
** AND GATE :-



Truth Table :-

A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

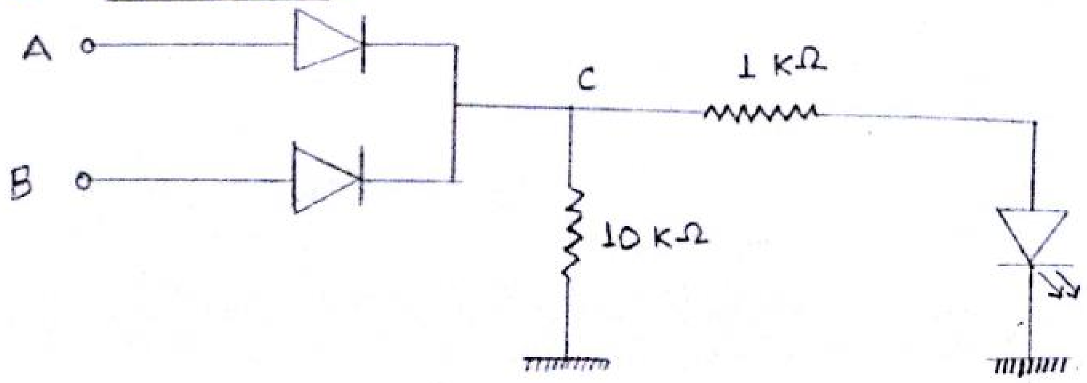
** NOT Gate :-



Truth Table :-

A	C
0	1
1	0

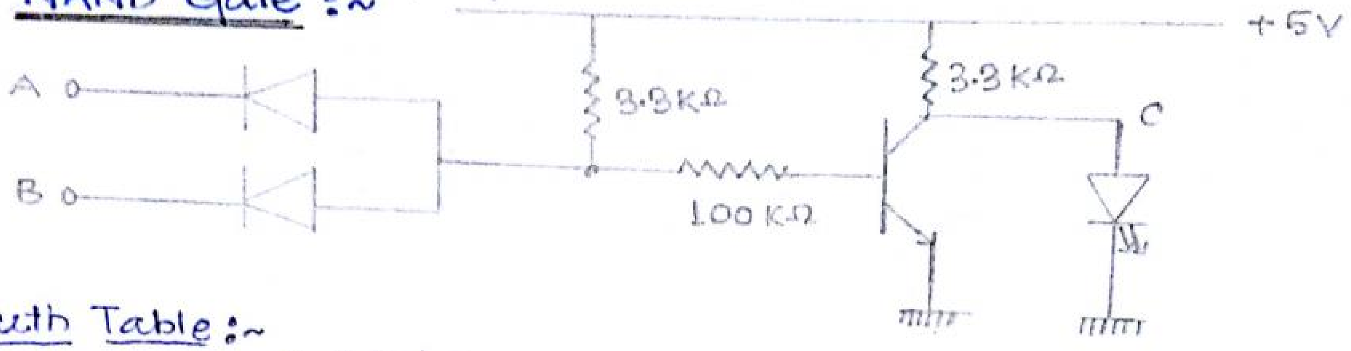
** OR Gate :-



Truth Table :-

A	B	C
0	0	0
0	1	1
1	0	1
1	1	1

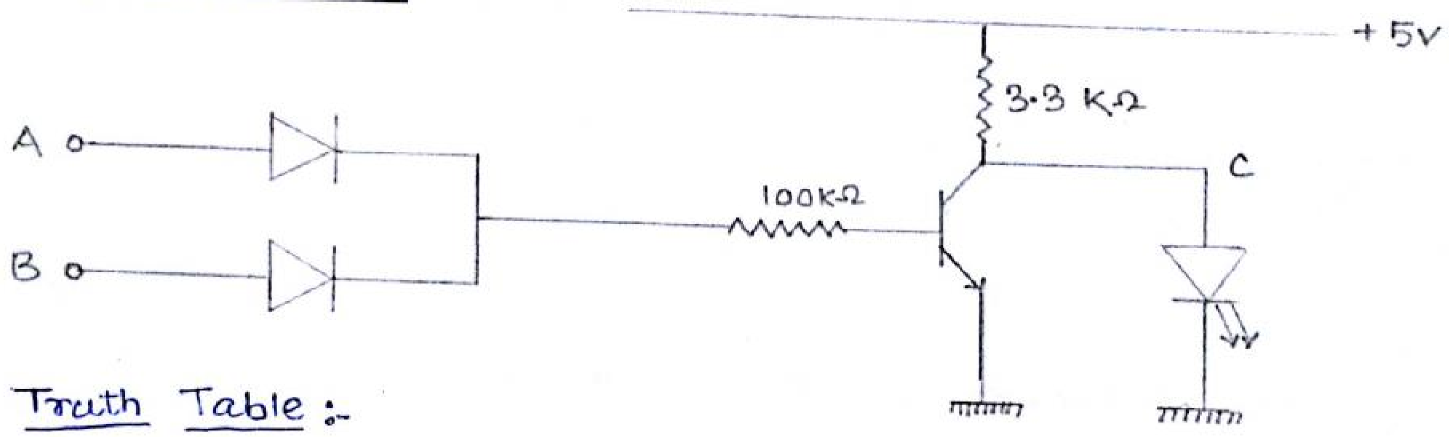
** NAND Gate :~



Truth Table :~

A	B	C
0	0	1
0	1	1
1	0	1
1	1	0

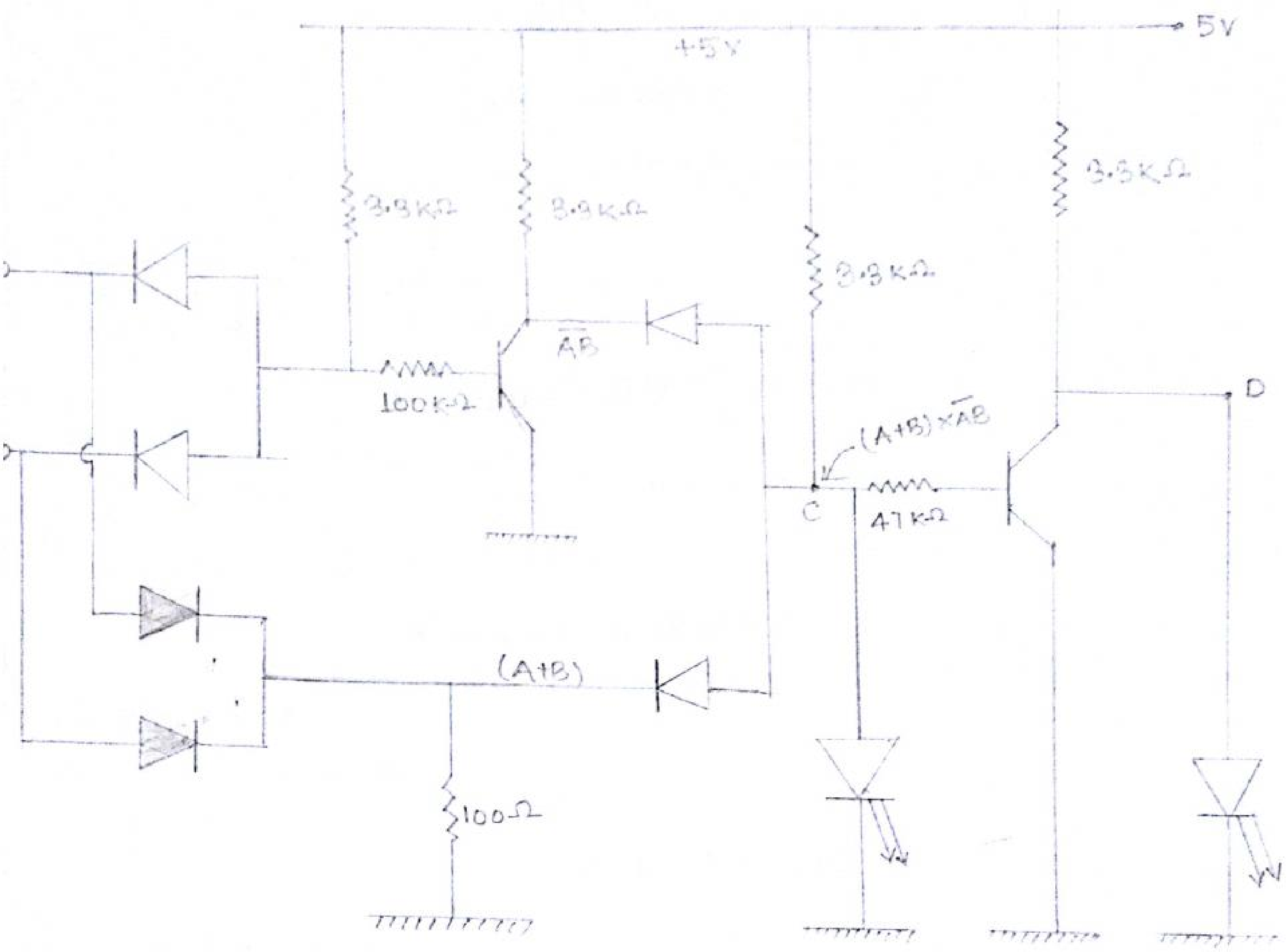
** NOR Gate :~



Truth Table :-

A	B	C
0	0	1
0	1	0
1	0	0
1	1	0

⇒ XOR/XNOR :



Truth Table :-

A	B	C(XOR)	D(XNOR)
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1

[CONDUCTOMETRIC TITRATION]

AIM :→ Determination of the Unknown strength of HCl solution by standardized NaOH Solution Using conductometric Method.

THEORY :→ conductometric titration can be used to detect the end point of a titration (Acid-Base). This method is based upon measurement of conductance during titration. The conductance varies due to the reason that electrical conductance of a solution depends upon the number of ions present and their ionic mobilities.
When conductance values are plotted against volume of titrant added, two straight lines are obtained. The point of intersection of lines gives the end point.
The conductance of acid solution is noted initially as well as after successive addition of acid solution (HCl), the H^+ ions. on adding NaOH solution, the H^+ ions are replaced by slow moving Na^+ ions, decreasing conductance of the solution (HCl is taken in a beaker & NaOH in a burette).

$$[H^+ + Cl^-] + [Na^+ + OH^-] \longrightarrow Na^+ + Cl^- + H_2O$$

When neutralization is complete, further addition of NaOH will cause conductance to increase due to excess of highly mobile $OH^\ominus$ ions. The conductance will thus be minimum at the end point. Thus, if conductance values are plotted against volume of NaOH added, curve of type xyz ('V' shaped) is obtained. The point of intersection (i.e point Y) corresponds to end point.

APPARATUS :→

- Burette
- Pipette (10 ml)
- Digital conductivity meter & conductivity cell
- plastic Beaker (100 ml)

REAGENTS :→

- Unknown HCl Solution
- Standardized NaOH (N/10) solution [Supplied]

PROCEDURE :→

[1] standardized NaOH (N/10) solution is Supplied.

[2] Unknown HCl solution is supplied.

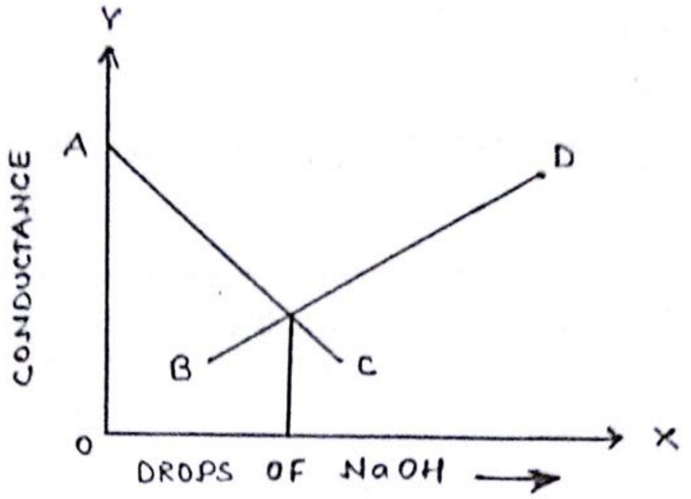
[3] Rinse the conductivity cell with de-ionized water.

[4] Pipette out 10 cc. HCl Solution in to the plastic beaker and add water if necessary, so that both the electrodes are completely immersed within the solution. Join the cell with the conductive bridge and measure the conductivity very carefully.

[5] Add NaOH solution from a burette drop-wise (approximately 2-3 drops).

[6] Measure the conductance of the solution after addition of 2-3 drops of NaOH and mildly stir the conductivity cell. Repeat the process until you have at least five points beyond the end point.

[7] Draw a curve by plotting the conductance against no. of drops of the titrant, find end point & calculate strength of HCl.



[Drops of NaOH solution Vs Conductance Graph]

Electrolytic conductivity is a measure of the ability of a Solution to carry electric current. solution of electrolytes conducts an electric current by migration of ions under influence of an electric Field. Like a metallic conductor, they obey ohm's law. Exception to this law occurs during abnormal conditions, e.g, very high voltage or high frequency current. According to ohm's law, current (I) flowing through a conductor is inversly proportional to its resistance (R) in ohm's.

Mathematically,

$$I = E/R$$

The resistance (R) of a Sample of homogeneous material having length 'l' & cross sectional area 'a' is given by

$$R = \rho(l/a)$$

where ρ is constant & depends upon nature of conductor. It is known as Specific resistance at a given temperature, the conductivity $1/\rho$ depends upon the type of ions present & their concentration.

RESULT :→

TABLE : Titration of Hcl Using NaOH :→

SERIAL NO.	NO. OF DROPS OF NaOH	CONDUCTANCE
{1}	0	0·39
{2}	3	0·36
{3}	6	0·31
{4}	9	0·27
{5}	12	0·22
{6}	15	0·18
{7}	18 ✓	0·14
{8}	21	0·14 ?
{9}	24	0·17
{10}	27	0·21
{11}	30	0·25
{12}	33	0·28
{13}	36	0·33

CALCULATION :→

V_1 = Volume of NaOH solution = $18/17$ ml

V_2 = Volume of HCl solution = 10 ml

S_1 = strength of NaOH solution = 0·12 (N)

S_2 = strength of HCl solution

17 drops of NaOH contains 1 ml of solution

∴ 18 " " " " $18/17$ ml " "

= 1·05 ml of solution

$V_1 = 1.05$ ml

$S_1 = 0.12$ (N)

$V_2 = 10$ ml

We know that,

$$V_1 S_1 = V_2 S_2$$

$$\Rightarrow S_2 = \frac{V_1 S_1}{V_2} = \frac{1.05 \times 0.12}{10} \text{ (N)}$$

$$= 0.126 \text{ (N)}$$

Discussion :→

[1] NaOH is a secondry standard solution, so it should be standardized with primary standard Oxalic acid solution using formula $V_1S_1 = V_2S_2$

[2] No indicator is used during the titration of HCl against NaOH solution.

[3] This experiment is more accurate as it is performed by a digital conductivity meter. Thus we can minimize human error.

[4] After plotting the graph (drops of NaOH soln, along X axis and conductance along Y axis), We get the value of NaOH solution. so the strength of Unknown HCl solution will be more accurate than other titration.

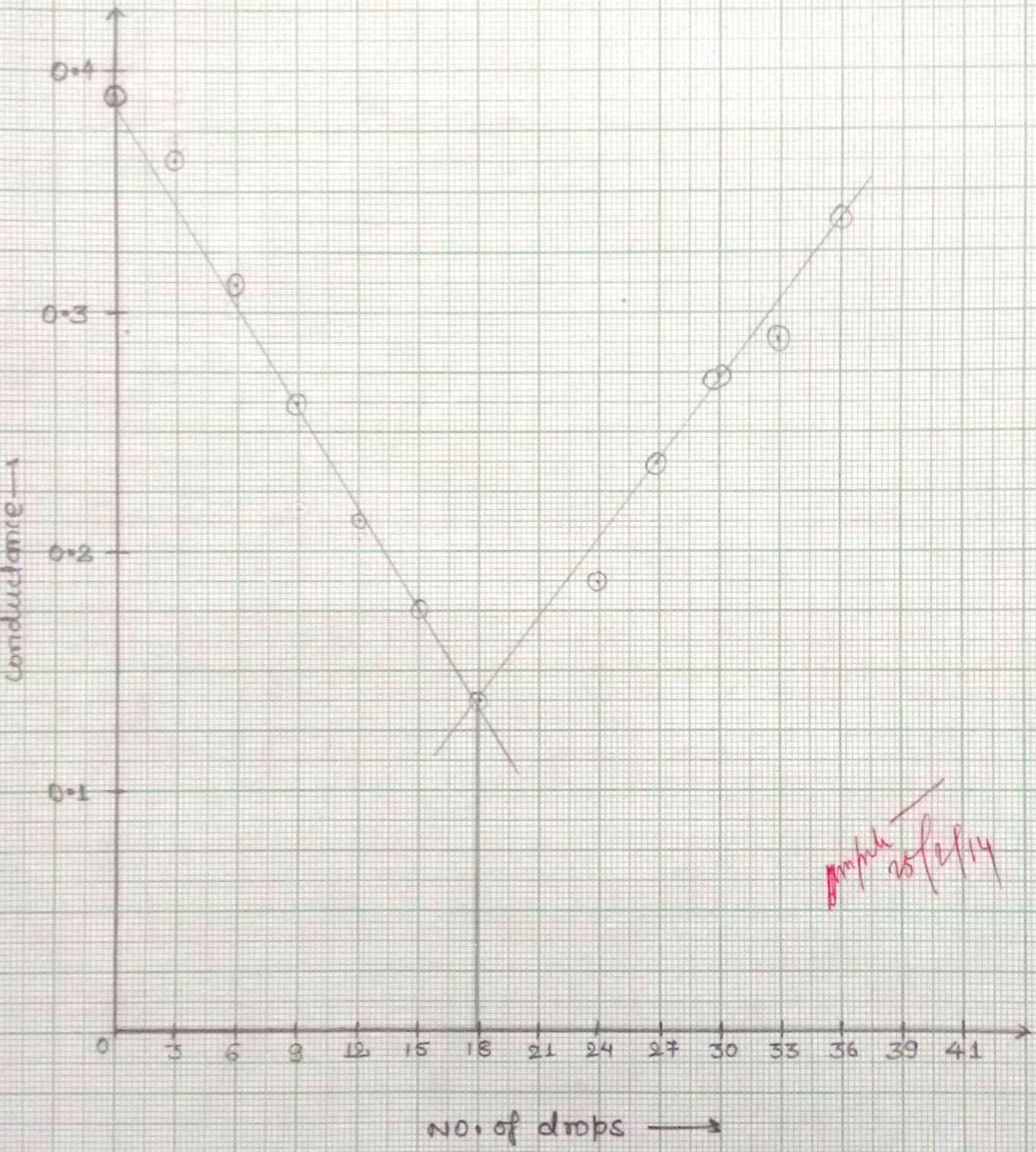
[5] Titration is performed at room temperature (25°c - 30°c).

CONCLUSION :→

The unknown strength of HCl is determined by standardized NaOH solution with help of a digital conductivity meter. Hence the unknown strength of HCl is 0·126 (N).

CONDUCTOMETRIC TITRATION :→

Along X-axis 10 div = 3 drops
Along Y-axis 40 div = 0·1



[illegible] 25/2/14

COMPLEXOMETRIC TITRATION

AIM :→ Estimation of calcium and magnesium hardness separately of a given tap water sample.

THEORY :→ Hardness is defined as soap consuming capacity of water. Bicarbonates of Ca, Mg cause temporary hardness. However chlorides, sulphates and carbonates of Ca & Mg cause permanent hardness. Hard water is not suitable for use as boiler feed water, because it leads to the formation of scales & slugs causing wastage of fuel, chocking of pipes, decreases efficiency & chances of explosin is also present. Hard water may also produce undesirable spots on the fabric due to its reaction with dyes when using in textile finishing. Thus softening of water is very important. In order to use any softening process type & extent of hardness must be known as prerequisite; complexometric titration is one of the best methods for hardness estimation.

Many metals form complexes with such reagents which contain appropriate ligands. EDTA is one such reagent, which forms complexes with metals. In the form of its disodium salt, it is used to estimate Ca^{+2} & Mg^{+2} ions. Dissociation constant values indicate that EDTA behaves like a dicarboxylic acid. Two of its carboxyl groups are strongly acidic the other two hydrogens are released during complex formation.

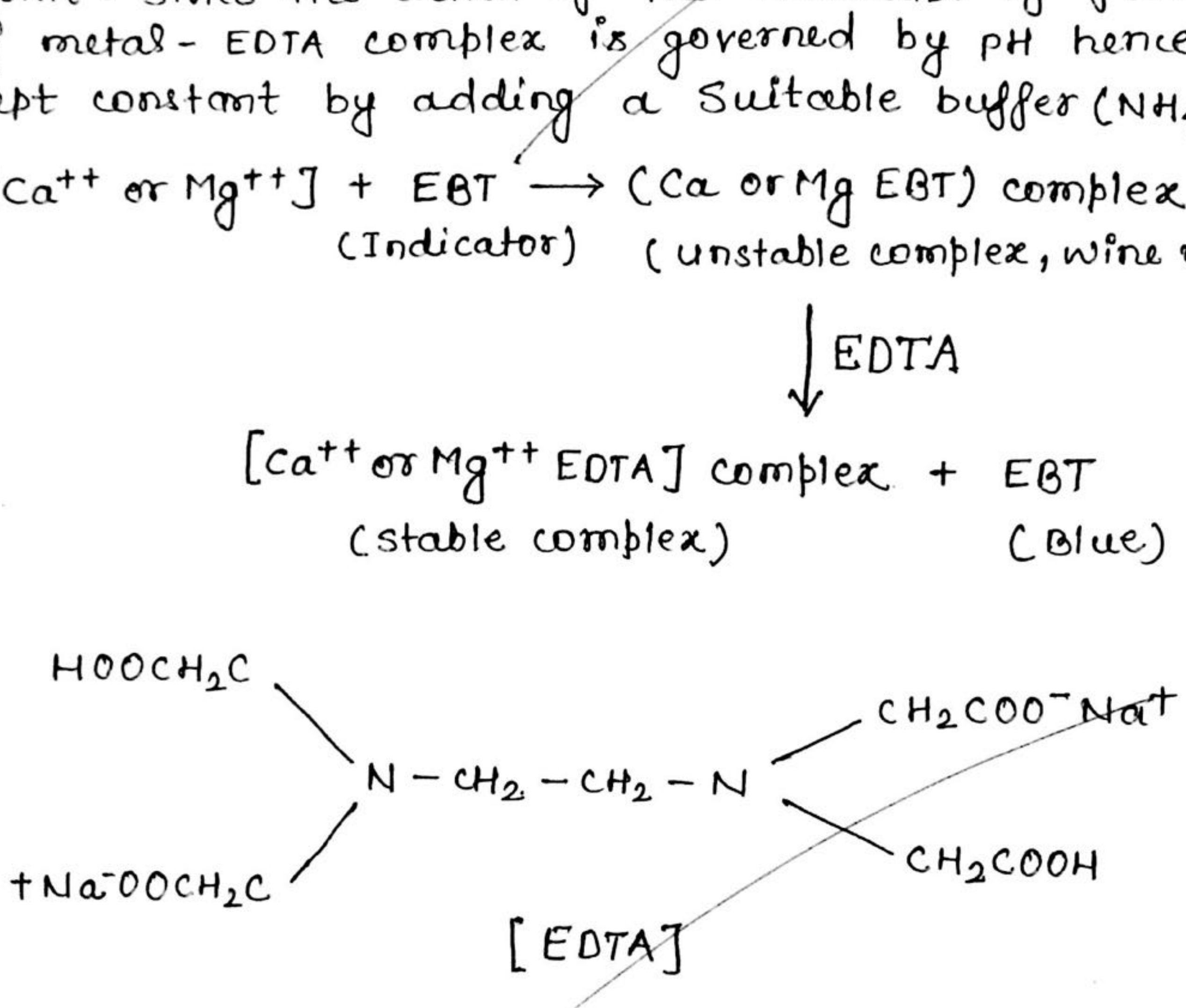
Ionization of this complex depends on the pH of the solution. Hence in this titration pH sensitive indicators are used.

In this estimation of Ca^{+2} and Mg^{+2} with EDTA, an azo-dye called Eriochrome Black-T (EBT) is used as an indicator. This forms a metal indicator complex the stability of which is lower than that of the metal EDTA complex. The solution is initially red. As the titration progresses the metal ions form more stable complex with EDTA, Hence the indicator anions goes to the solution and the colour changes from wine red to blue at the end point. since the action of the indicator of formation of metal-EDTA complex is governed by pH hence it is kept constant by adding a Suitable buffer ($NH_4Cl + NH_4OH$)

$$[Ca^{++} \text{ or } Mg^{++}] + \underset{\text{(Indicator)}}{EBT} \longrightarrow \underset{\text{(unstable complex, wine red colour)}}{(Ca \text{ or } Mg\ EBT)\ \text{complex}}$$

$$\Big\downarrow \text{EDTA}$$

$$\underset{\text{(stable complex)}}{[Ca^{++} \text{ or } Mg^{++}\ EDTA]\ \text{complex}} + \underset{\text{(Blue)}}{EBT}$$



[EDTA]

The quick complex and one step interaction of interaction of polydentate ligands with metals to yield stable complexes, is the principle of these complexometric titration in which metal in solutions are titrated against that of polydentate ligands.

APPARATUS :→

- Pipette (50-25 ml)
- Burette
- Beaker (250 ml)
- Conical flask

REAGENTS :→

- Tap water
- EDTA solution
- Eriochrome black-T indicator
- Calcon indicator
- Buffer solution

PROCEDURE :→

- Determination of total hardness :

50 ml of water sample is taken in 250 ml conical flask. 1 ml of buffer solution is added to it. Now a pinch of EBT indicator is added. The solution becomes wine red. Now the solution is titrated against 0.01(M) EDTA solution. At the end point the colour changes from wine red to blue. The titration is repeated three times and reading is noted.

- Determination of Calcium hardness :

The calcium hardness of the hard water sample was determined by adding 12.5 ml of 3(M) NaOH solution to 25 ml water sample, shaking and keeping for 5 min to precipitate out $Mg(OH)_2$ completely. the resulting solution was finally titrated against standard EDTA solution using small amount of calcon indicator. The end point was observed by the colour change from pink to blue. The process is repeated twice.

RESULT AND CALCULATION :→

TABLE - I : DETERMINATION OF TOTAL HARDNESS →

NO. OF OBS.	VOLUME OF WATER SAMPLE TAKEN (ml)	BURETTE READING		VOL.OF EDTA SOLUTION CONSUMED (ml)	CONCORDANT READING (ml)	HARDNESS IN ppm
		INITIAL	FINAL			
{1}	50	0	13	13	13	260
{2}	50	13	26	13		
{3}	50	26	39·1	13·1		

1000 ml (1M) EDTA solution ≡ 100 gm of $CaCO_3$

13 ml (0·01 M) " " $\equiv \dfrac{100 \times 13 \times .01 \times 1000}{100 \times 100}$

$= 13$ mg of $CaCO_3$

50 ml of Hard water contains 13 mg of $CaCO_3$

1000 ml " " " $\dfrac{13}{50} \times 1000$

$= 260$ mg of $CaCO_3$

Hence, The total Hardness of water sample contain

$= 260$ ppm

1000 ml (1N) Mohr's salt contain 56g of Fe^{+2}

10 ml (1N) ———— $\frac{56 \times 10}{1000}$ g of Fe^{+2}

10 ml (S_2) (N) ———— $\frac{56 \times 10 \times S_2}{1000}$ gm

In 1000 ml S_2(N) ———— $\frac{56 \times 10 \times S_2 \times 1000}{1000 \times 10}$

$= 56 \times S_2$ g of Fe^{+2}

$= 56 \times 0.094$

$= 5.264$ gm of Fe^{+2}

DISCUSSION:

[1] Oxidation reduction takes place Simultaneously in this experiment.

[2] $KMnO_4$ solution is standardized with the help of standard oxalic acid using the formula
$V_1S_1 = V_2S_2$.

[3] $KMnO_4$ acts as a self indicator in this experment.

[4] $KMnO_4$ acts as a stronge Oxidizing agent in Presence of H_2SO_4 thus H_2SO_4 is added in the solution.

[5] strength of Mohr's salt solution is determined with standardized $KMnO_4$ solution using formula
$V_1S_1 = V_2S_2$

[6] When all the reducing agent has been oxidized then the excess drop of $KMnO_4$ resulted in change of the colour.

CONCLUSION :

The amount of Iron present in Mohr's salt solution is 5.264 g.

TABLE-II : DETERMINATION OF CALCIUM HARDNESS :→

NO. OF OBS.	VOLUME OF WATER SAMPLE TAKEN (ml)	BURETTE READING		VOL.OF EDTA SOLUTION CONSUMED (ml)	CONCORDANT READING (ml)	HARDNESS IN ppm
		INITIAL	FINAL			
{1}	25	0	6	6	6	240
{2}	25	6	12	6		
{3}	25	12	18.1	6.1		

1000 ml (1M) EDTA solution ≡ 100gm of $CaCO_3$

6 ml (0.01 M) " " $\equiv \dfrac{100 \times 6 \times .01 \times 1000}{1000 \times 100}$

= 6 mg of $CaCO_3$

25 ml of hard water contains 6 mg of $CaCO_3$

1000 ml " " " $\frac{6}{25} \times 1000$

= 240 mg of $CaCO_3$

Hence, the calcium hardness of water sample is 240ppm.

There the magnesium hardness of water sample is

= (260-240) ppm

= 20ppm.

DISCUSSION :→

[1] NH_4Cl and NH_4OH buffer solution is added to maintain the PH of the reaction medium at 10 which is a prerequisite for formation of metal - EBT Complexes.

[2] After addition of EDTA solution stable metal - EDTA complex is formed. Thus Free EBT is responsible for the blue colouration of the Solution after the titration is over.

[3] Ca^{2+} - EBT complex is very unstable, so that we can't get a sharp colour change at the end point. Therefore in the first step, colur change at the end point of the titration is mainly responsible for the Mg^{+2} - EBT complex. without Mg^{2+} titration is not possible using EBT indicator.

[4] In the second step, Mg^{2+} is removed as $Mg(OH)_2$ so pH become higher than 11, and in this pH range metal EBT indicator is not Suitable. So calcon indicator is used, actually calcon retains its blue colour up to pH 13 in contrast to pH 11 for EBT.

CONCLUSION:

Total hardness of water sample = 260 ppm

Calcium hardness of tap water sample = 240 ppm

Magnesium hardness of tap water = 20 ppm.

TITLE : CHARACTERISTICS OF TUNGSTEN FILAMENT LAMPS.

OBJECTIVE : To study and draw the following characteristics of Tungsten Filament Lamp.

I. Voltage vs. Current

II. Resistance vs. Voltage.

III. Voltage vs. Power.

APPARATUS:

Sl. No	Apparatus Name	Apparatus Type	Range
1.	Tungsten Lamp	A.C	230V, 110W
2.	Ammeter	M.I	0-1A/2A
3.	Voltmeter	M.I	0-150V/300V/600V
4.	wattmeter	Dynamo	0-150W/300W/600W
5.	Variac	A.C	0-270V.

THEORY : There are two types of lamps which are in common use, one is filament lamp and the other is gaseous discharge lamp. The filament lamps are incandescent lamps, e.g. carbon, tungsten etc. The filament of these lamps, when heated due to electric current, emits radiations in visible spectrum.

The filament of incandescent lamp is mostly made
of tungsten wire whose melting point is 3400°c.
At normal working voltage, The filament mate-
rial gets heated to avery high temperature and
emits white light. The filament is made in the
form of a coiled-coil to contain a longer length
of the filament in a shorter space and is enclosed
in an evacuated glass bulb to minimize oxida-
tion of filament material at such a high operating
temperature. Usually the lamps above 15W or
25W are filled with an inert gas, e.g. argon or
nitrogen, to enable the filament to operate at
higher temperature and achive higher lumens/
watt efficiency (in the range of 12-13 watt).

The resistance of filament changes
considerable when switched on. The initial resistance
of the filament in cold condition can be measured by
multimeter or by ammeter-Voltmeter method. The
filament resistance at normal operating temperature
is difficult to measure directly and ~~is~~ therefore,
calculated by using the following relation :

$$R = W/I^2 \ \Omega.$$

where, R = Resistance in ohm when normal Voltage in applied across the lamp

I = Current taken by the lamp in ampere.

W = Power to the lamp in watt.

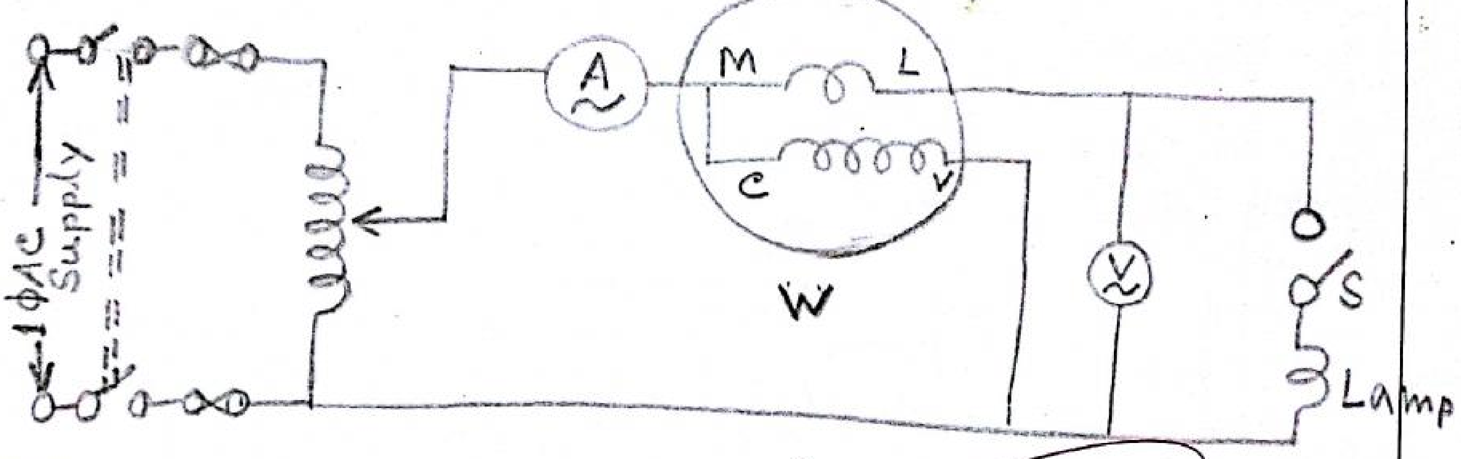
Basic reason of getting all these conductors heated is their resistance. Resistance is the physical property of a substance by virtue of which it opposes the flow of current through it. Conductors offer lower resistance than insulator.

Experiment have shown that the resistivity is affected by the conductor's temperature. The resistivity and, hence, the resistance of most of the conducting materials increases with increase in temperature. The resistance changes with temprature according to the relation: $R_T = R_0[1+\alpha(T-T_0)]$

where R_T & R_0 are the value of resistance of the conductor at T and T_0 respectively and α is a constant called temperature coefficient of resistance. T_0 is often taken to be either room temperature or 0°c. The value of α is always very small for pure metal, so their resistance increase with increasing temperature. The temperature co-efficient of Tungsten Filament and Carbon Filament lamp are 0.0045 and

-0.0005 respectively.

CIRCUIT DIAGRAM :



PROCEDURE :

1) At first, I connect the circuit as shown in Fig

2) After this, I kept the variac in minimum or zero Position.

3) Nextly, I switch ON the power supply and increase the applied voltage gradually in step by step.

4) Now I Noto down the applied voltage, load current, and input power for every step.

5) Finally, I Switch OFF Power Supply and disconnect the circuit from the supply and calculate the resistance at every step.

OBSERVATION TABLE:

Sl No.	Applied Voltage (volt)	Load Current (amp)	Input Power (watt)	Resistance (Ω)
1	40	0.16	6	234.37
2	60	0.21	11	249.43
3	80	0.25	17	272.00
4	100	0.28	23	293.36
5	120	0.32	32	312.50
6	140	0.35	44	359.18
7	160	0.37	56	409.05
8	180	0.40	70	437.50
9	200	0.41	82	487.80
10	220	0.44	96	495.86

[illegible] 09/04/15

CALCULATION :

1) Resistance (R) $= \frac{W}{I^2} = \frac{6}{(0.16)^2} = 234.37\ \Omega$

2) Resistance (R) $= \frac{W}{I^2} = \frac{11}{(0.21)^2} = 249.43\ \Omega$

3) Resistance (R) $= \frac{W}{I^2} = \frac{17}{(0.25)^2} = 272.00\ \Omega$

4) Resistance (R) $= \frac{W}{I^2} = \frac{23}{(0.28)^2} = 293.36\ \Omega$

5) Resistance (R) $= \frac{W}{I^2} = \frac{32}{(0.32)^2} = 312.50\ \Omega$

6) Resistance (R) $= \frac{W}{I^2} = \frac{44}{(0.35)^2} = 409.05\ \Omega$

7) Resistance (R) $= \frac{W}{I^2} = \frac{56}{(0.37)^2} = \cancel{409.05\ \Omega}$

8) Resistance (R) $= \frac{W}{I^2} = \frac{70}{(0.40)^2} = 437.50\ \Omega$

9) Resistance (R) $= \frac{W}{I^2} = \frac{82}{(0.41)^2} = 487.80\ \Omega$

10) Resistance (R) $= \frac{W}{I^2} = \frac{96}{(0.44)^2} = 495.86\ \Omega$

RESULT :

The graph of voltage vs current, resistance vs voltage and power vs voltage is drawn.

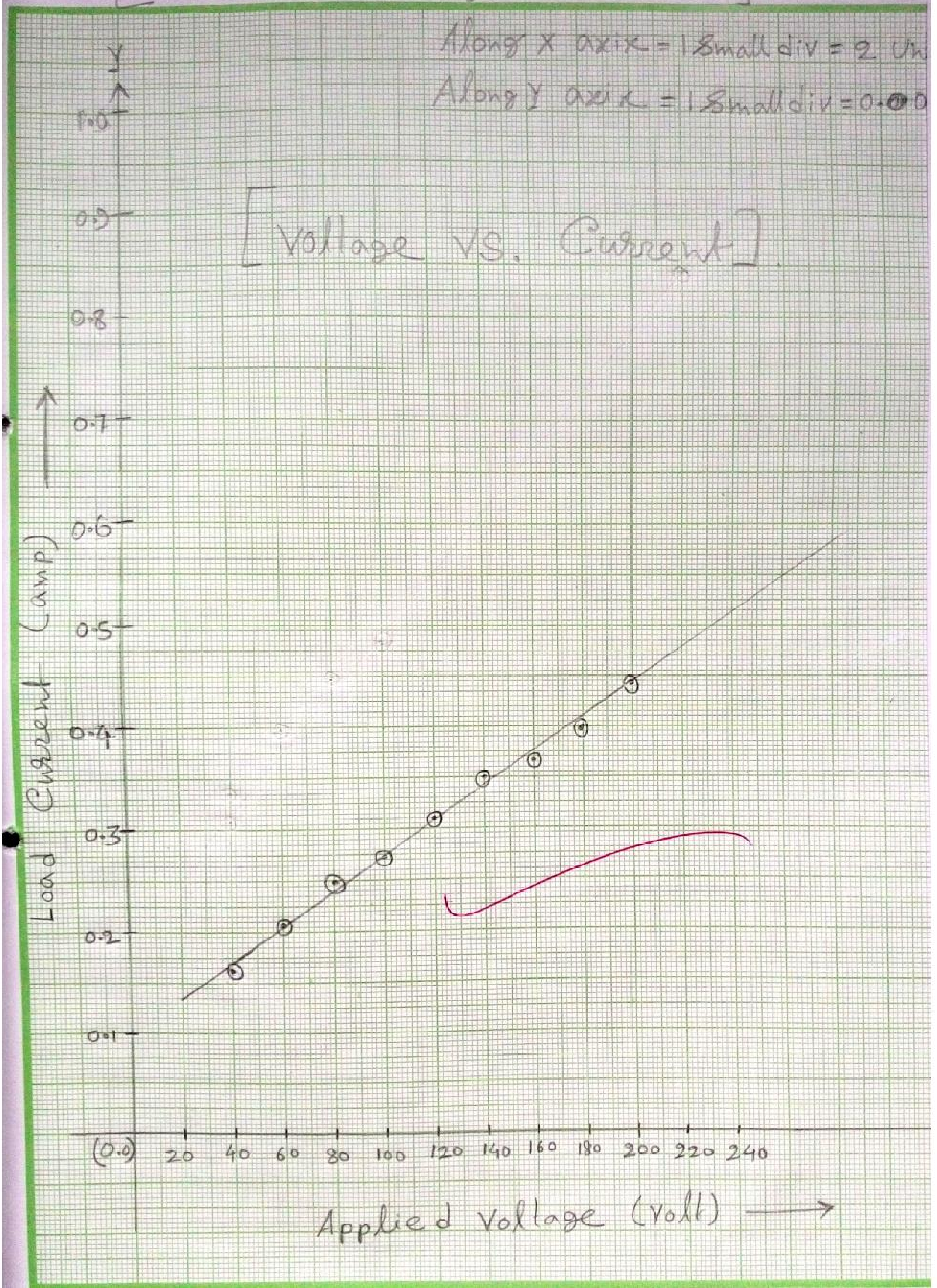
DISCUSSION:

1) what is the nature (i.e. Positive or negative) of the slop of the voltage v.s. Resistance characteristics of Tungsten Filament Lamp? Explain it brifly.

The nature of the slop of the voltage vs Resistance charactoristics of Tungsten Lamp is positive because whenever the temperature increases the resistance also increases. Since resistance and current are imversely proportional to each other. So. whenever the resistance increases, the current decreases and vica-versa.

If temparature is increased and resistance is decreased or vice versa. So, the current would eigther increases or decreases. At a certain point, the temparature increases and the resistance decreases and the current continously raises up thus destroying the fuse of the circuit.

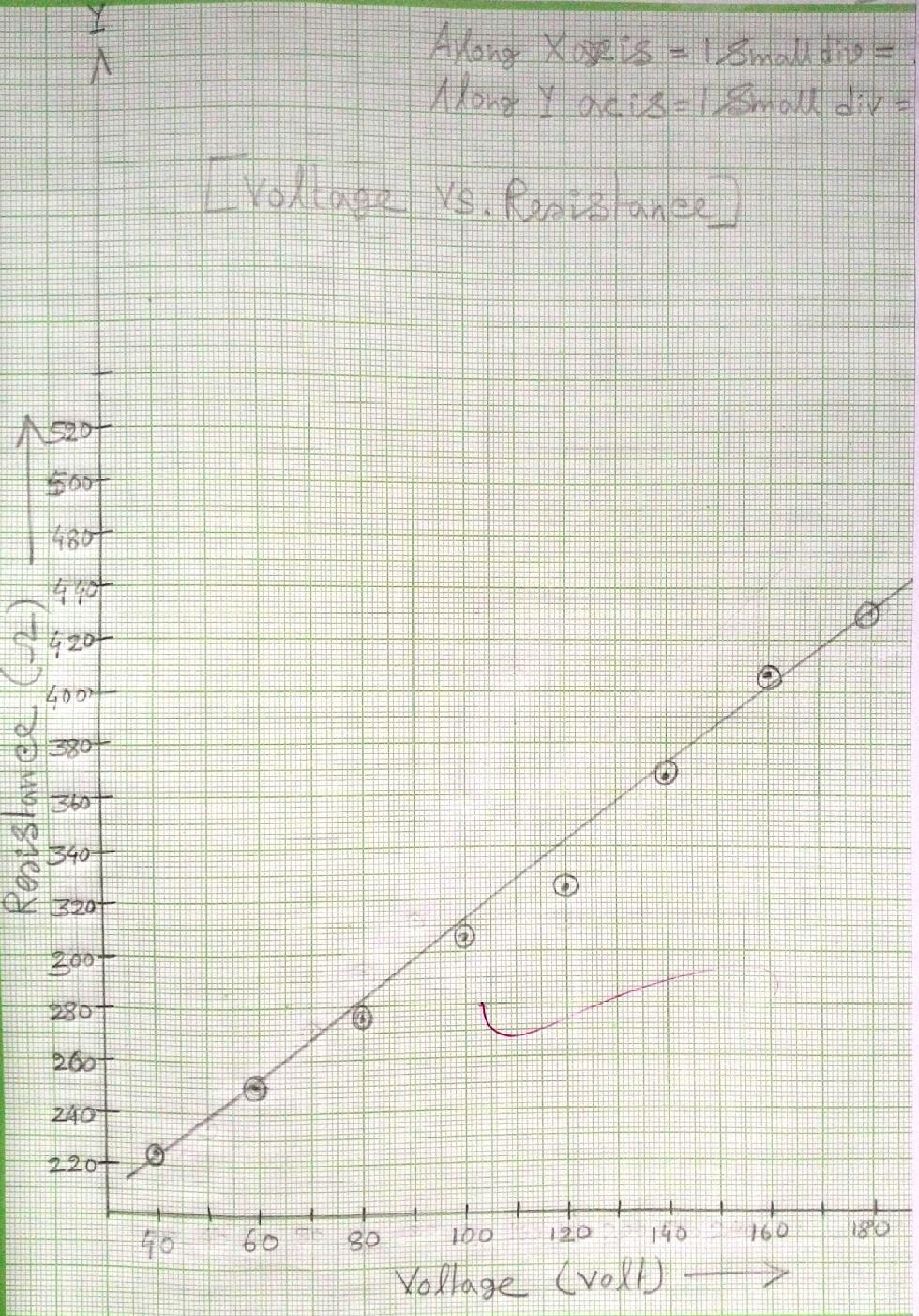
[Graph of Voltage VS Current]



Along X axis = 1 Small div =

Along Y axis = 1 Small div =

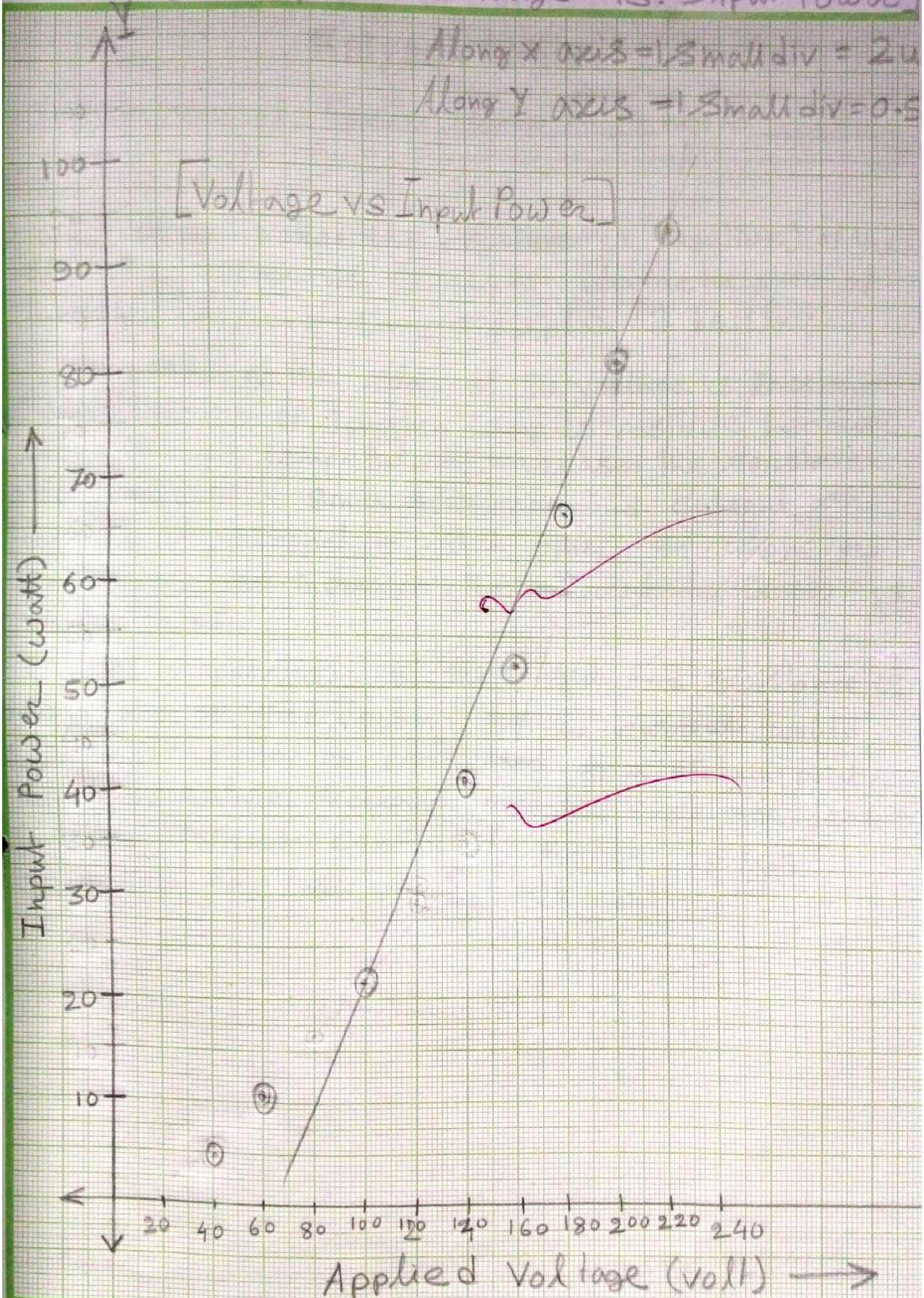
[Voltage vs. Resistance]



[Graph of Voltage vs. Input Power]

Along X axis = 1 Small div = 2

Along Y axis = 1 Small div = 0.5



TITLE : STUDY THE RLC PARALLEL CIRCUIT.

OBJECTIVE: To study the RLC Parallel circuit and draw the following characteristics.

I. Frequency vs. Resistance.

II. Frequency vs. Impedence.

III. Frequency vs. Inductive reactance.

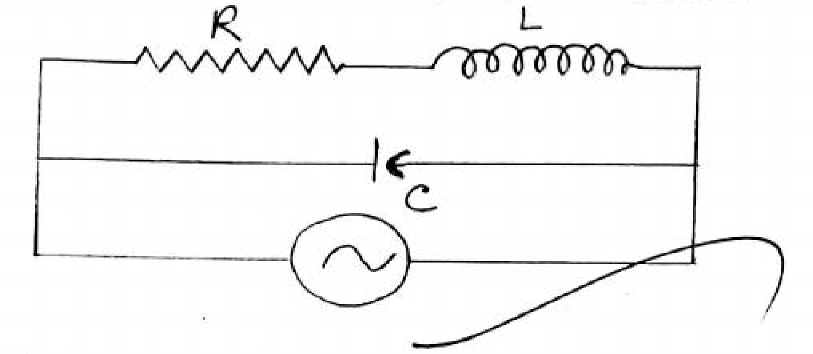
IV. Frequency vs. Capacitive reactance.

V. Frequency vs. Current.

APPARATUS :

Sl. No	Apparatus Name	Apparatus Type	Range	Quantity
1	Resistor	AC	0-200Ω	1
2	Inductor	AC	0-150 mH	1
3	Capacitor	AC	0-1 μF	1
4	Ammeter	AC	—	1
5	Voltmeter	AC	0 - 10 Volt	1
6	Audio Frequency Generator	AC	100 - 1 KHz	1.

THEORY: Conside an AC circuit Containing resistance R, inductor L and a capacitor C connected in parallel as shown in figure below



The current passing through coil is $I_L = \frac{V}{R + jωL} = \frac{V(R - jωL)}{R^2 + ω^2L^2}$

And current passing through capacitor is $I_C = V \times jωC$

So, the total current $I = I_L + I_C$

$$= \frac{V(R - jωL)}{R^2 + ω^2L^2} + V \times jωC$$

$$= V\left(\frac{R}{R^2 + ω^2L^2} - j\left(\frac{ωL}{R^2 + ω^2L^2} - ωC\right)\right)$$

At resonance, the current drawn is at unity p.f. So, in resonance condition imaginary part of the above equation is zero. i.e.

$$\frac{ω_r L}{R^2 + ω_r^2 L^2} = ω_r C \Rightarrow R^2 + ω_r^2 L^2 = \frac{L}{C}$$

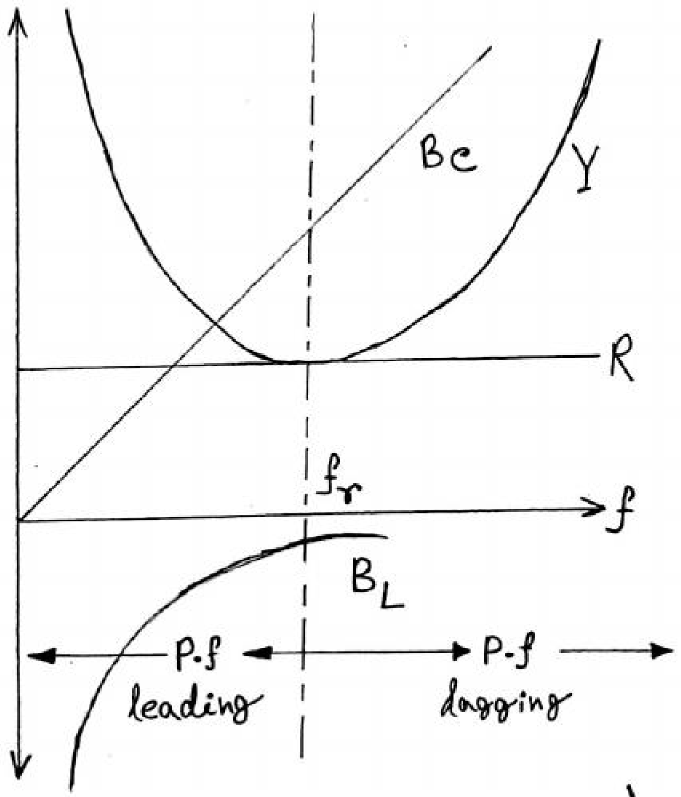
$ω_r = \sqrt{\frac{1}{LC} - \frac{R^2}{L^2}}$; So the resonance frequency

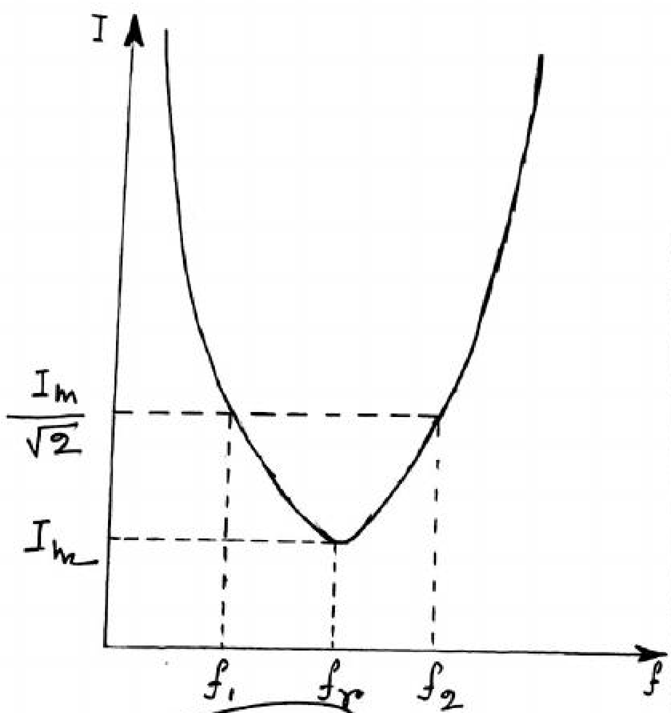
$$f_r = \left(\frac{1}{2\pi}\right)\sqrt{\frac{1}{LC} - \frac{R^2}{L^2}}$$

Since R is very small therefore (R^2/L^2) may be neglected,

$$\text{So } f_r = \frac{1}{2\pi\sqrt{LC}}$$

The variation of resistance, inductive susceptance B_L (reciprocal of X_c) and admitance with respect to frequency are plotted in Fig 1.

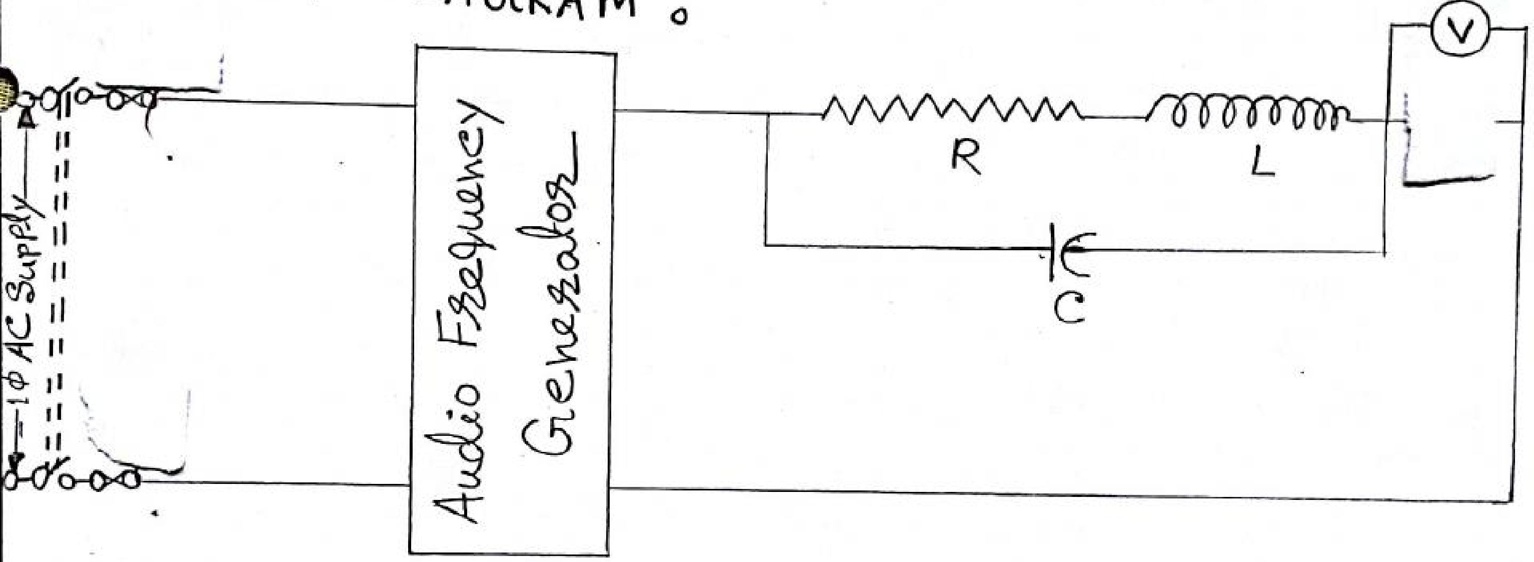




At resonance, the total current drawn by the circuit is $I = V\frac{R}{R^2+\omega_r^2L^2}$. The variation of current I with respect to frequency is also shown in above Fig. From the above fig. bandwidth frequency = $f_2 - f_1$. So Q Factor = $\frac{f_r}{f_2 - f_1}$

The Q Factor can also calculated by the following equation $Q\ \text{Factor} = \frac{\omega_r L}{R} = \frac{2\pi f_r L}{R}$

CIRCUIT DIAGRAM :



PROCEDURE :

1) we will connect the circuit as given in the circuit diagram in the fig.

2) Then we will switch on the power supply.

3) we will very the frequence from 100 Hz to 640 HZ in steps of 60 HZ by adjust frequency variation knob.

4) we will now note down the voltmeter reading indicating voltage across the resistance.

[P.T.O]

OBSERVATION TABLE :

Sl. No.	Frequency f (HZ)	Resistance R (Ω)	Inductance L (mH)	Inductive Resistance X_L (Ω)	capacitance c (μF)	Capacitive Reactance X_c (Ω)	Voltage across R (Volt)	current $I = V_R/R$ (amp)
1.	100	200	150	94.24	1	1591.54	1.2	6
2.	180	200	150	169.64	1	884.19	1.1	5.5
3.	260	200	150	245.044	1	612.13	0.95	4.75
4.	340	200	150	320.442	1	468.10	0.85	4.25
5.	420	200	150	395.84	1	378.94	0.9	4.5
6.	500	200	150	471.23	1	318.3	1.1	5.5
7.	580	200	150	546.63	1	274.4	1.3	6.5
8.	660	200	150	622.035	1	241.14	1.5	7.5
9.	740	200	150	697.43	1	215.07	1.7	8.5
10.	820	200	150	772.83	1	194.09	1.9	9.5
11.	900	200	150	848.23	1	176.83	2	10
12.	980	200	150	923.63	1	162.40	2.1	10.5

[illegible] 18/02/15

5) we will now switch off the power supply and disconnect the circuit.

6) we will now draw the above curve.

CALCULATIONS:

THEORY	GRAPH
1) f_r (Resonating frequency) = $\frac{1}{2\pi\sqrt{LC}} = \frac{1}{2\times\pi\times\sqrt{150\times10^{-3}\times10^{-6}}}$ $= 410.93$ HZ.	1) f_r (Resonating frequency) $= 340$ HZ
2) Quality factor (Q) = $\frac{2\pi f_r L}{R}$ $= \frac{2\times3.14\times410.93\times150\times10^{-3}}{200}$ $= 1.93$	2) Bandwidth $= f_2 - f_1$ $= (540 - 100)$ $= 440$ $I_{min} = 4.25\times10^{-3}$ A $\frac{I_{min}}{\sqrt{2}} = 4.25\times10^{-3}/\sqrt{2}$ $= 6\times10^{-3}$ A
3) $\frac{f_r}{Q} = \frac{410.88}{1.93}$ $= 212.90$ HZ.	Q.F $= \frac{f_r}{B.W} = \frac{340}{440}$ $= 0.77$

DISCUSSION :

1) what is resonance frequency ? State the resonance condition for parallel RLC circuit.

Ans. The condition in a circuit at which inductive reactance becomes equal to capacitive reactance is known as resonance frequency.

1) $\omega L = 1/\omega C$ [This is satisfactory if the resistance are small]

2) At this frequency parallel impedance is maximum.

3) At this frequency the current is in phase with voltage. unity power factor.

2) Define Band width and Q-factor.

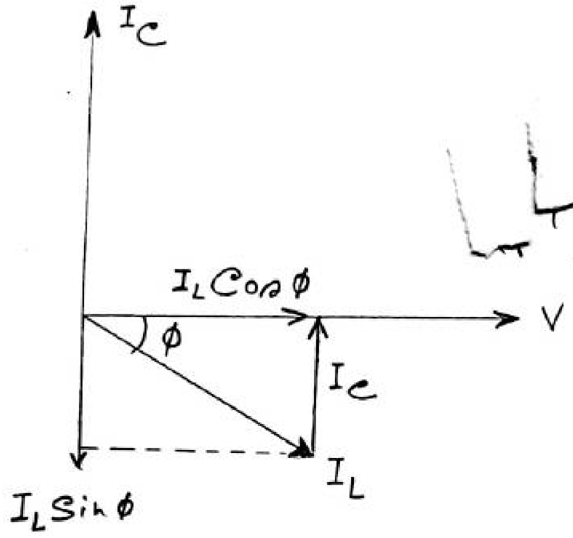
Ans. It is found that the Bandwidth of a given R-L-C circuit at any off-resonance frequencies f_1 and f_2 is given by $B = f_0 \frac{Q}{Q_0} = \sqrt{f_1 f_2}$

$\frac{Q}{Q_0} = f_2 - f_1$, where f_1 and f_2 are frequencies above and below f_0.

Quality Factor or Q Factor $= \frac{F_r}{\text{Band Width}}$, where f_r = resonating frequency.

$$Q\text{factor} = \frac{\omega_r L}{R} = \frac{2\pi f_r L}{R}$$

3). Draw the Phasor diagram for Parallel RLC Circuit.

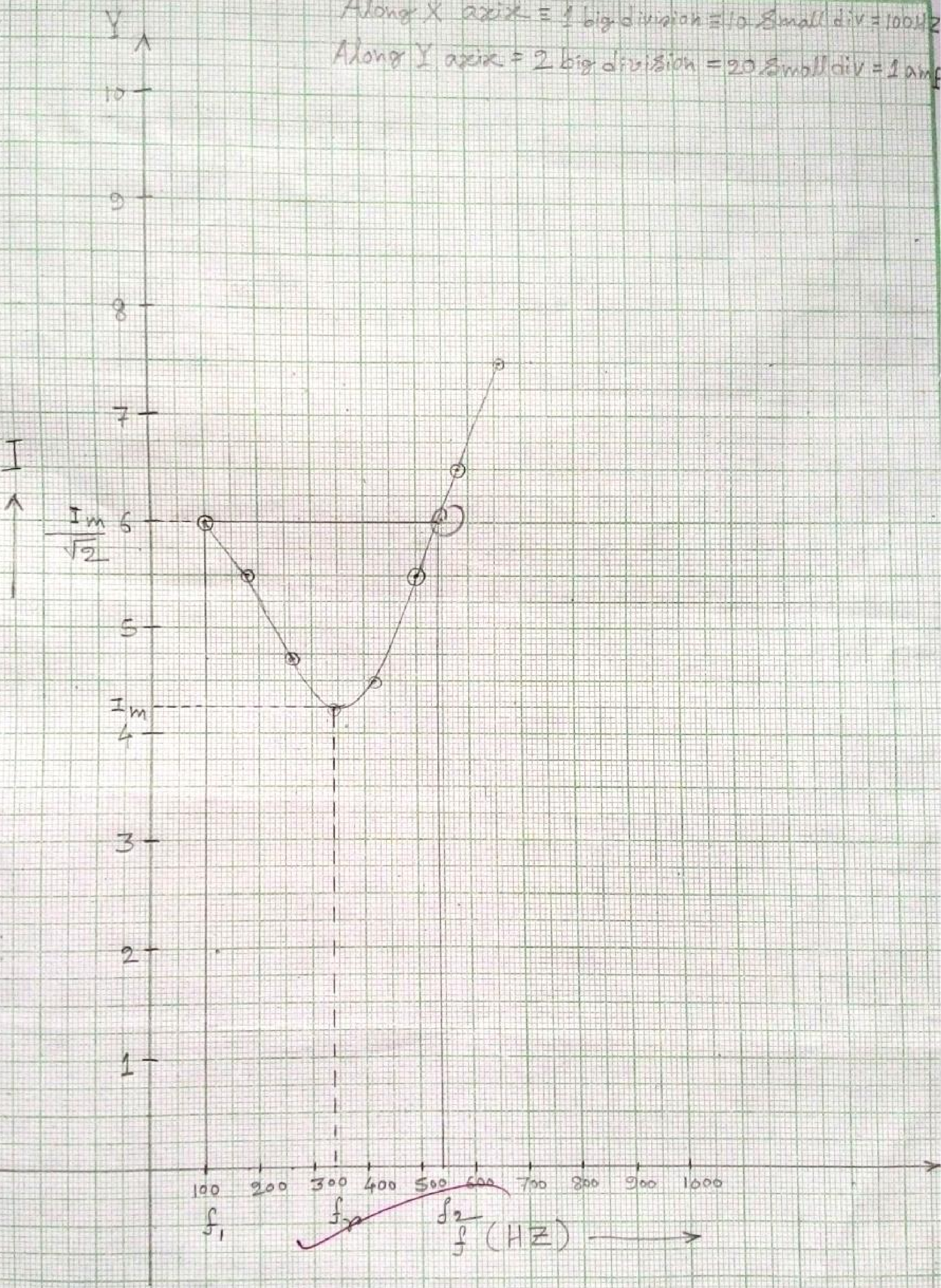


Let, v is the supply voltage, I_C is the current through the capacitor, I_L is the current following through the inductor; ϕ is the Phase angle difference between supply Voltage and current I_L.

Study the RLC Parallel Circuit.

Along X axix = 1 big division = 10 small div = 100 Hz

Along Y axix = 2 big division = 20 small div = 1 amp



TITLE : CHARACTERISTICS OF FLUORESCENT LAMPS.

OBJECTIVE : To study the starting method, minimum striking voltage and effect of varying Voltage or current of a fluresent lamp using A.C supply.

APPARATUS :

Sl No.	Apparatus Name	Apparatus Type	Range
1.	Fluorescent Lamp	AC	40W, 230V
2.	Choke	AC	230V~50HZ
3.	Starter	AC	—
4.	Ammeter	AC	0-500mA
5.	Voltmetter	AC	0-300V, 600V
6.	Wattmeter	AC	0-150/300/600V 2.5/5A
7.	Variac.	AC	0-270V.

THEORY : A fluorescent lamp is a low pressure mercury discharge lamp with internal surface coated with suitable fluoresent material. This lamp consists of a glass tube provided at both ends with caps having two pins and oxide coated tungsten filament. Tube contains argon and krypton gas to facilitate starting with small quantity mercury under low pressure.

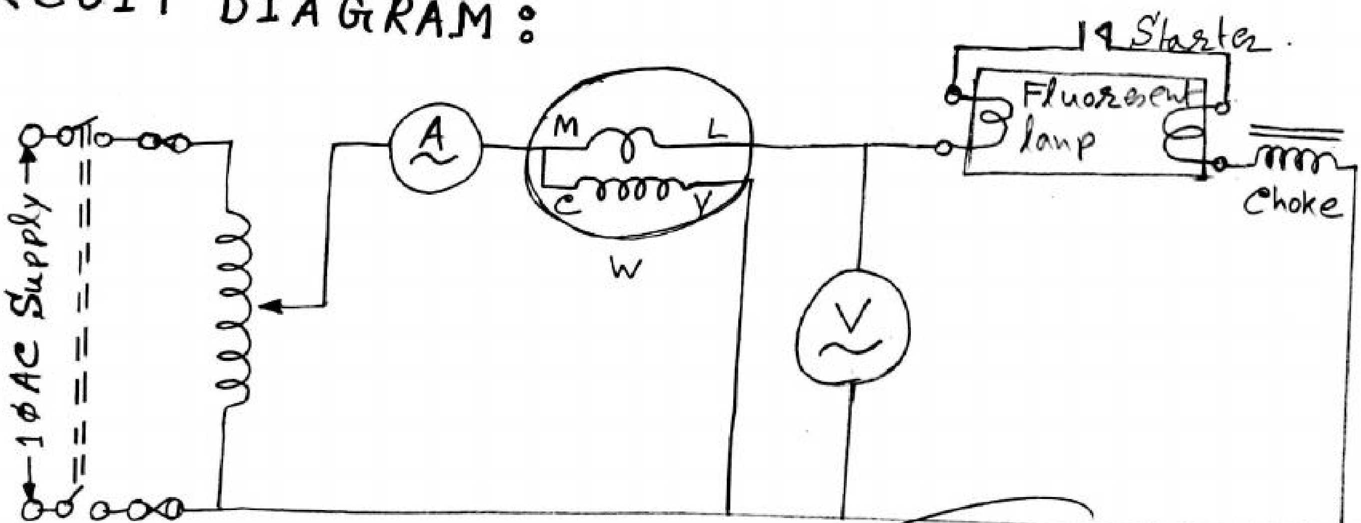
Fluoresent material, when subjected to electro-magnetic radiation of particular wavelength produced by the discharge through mercury vapors, gets excited and in turn gives out radiations at some other wavelength which fall under visible spectrum. Thus the secondary radiations from fluorescent power increase the efficiency of the lamp. Tube lights in India are generally made either 61 cm long 20V rating or 122 cm long 40 watt rating. In order to make a tube light self-starting, a starter and a chock are connected in the circuit.

when switch is on, ~~full~~ supply voltage from the variac appears across the starter electrodes P & Q which are enclosed in a glass bulb filled with argon gas. This voltage causes discharge in the argon gas with consequent heating of the electrodes. Due to this heating, the electrode V which is made of bimetallic strip, bends and cross contact of the starter. At this stage the choke, the filament M_1 & M_2 of the tube T and the starter become connected in series across the supply. A current flows through M_1 and M_2 and heats them. Meanwhile the argon discharge in the starter tube disappears and after a cooling time, the electrodes P & Q cause a sudden break in the circuit. This cause a high value of induced-emf in the chock. The induced emf in the choke is applied across the tube light electrodes M_1 & M_2 & is free electrons in the vicinity of electrodes. Thus the tube light starts giving light output.

Power Factor (P.F) of the lamp is somewhat low is about 0.5 lagging due to the inclusion of the choke. A condenser, if connected across the supply may improve the P.F to about 0.95 lagging. At reduced supply voltage the lamp may click a start but may fail to hold because of non-availability of reduced holding voltage across the tube. Higher normal Voltage reduces the useful life to very great extent.

If applied voltage of a fluorescent lamp is V, line current is I and input Power is $P = VI \cos\phi$; where $\cos\phi$ = power factor of fluorescent lamp.

CIRCUIT DIAGRAM :



PROCEDURE :

1) At first, I connect the circuit as shown in Fig.

2) After this, I kept the variac in minimum or zero position

3) Nextly, I switch on the ac supply and increase gradually till the lamp strikes.

4) Now I note down the reading of striking voltage.

5). Correspondingly; I applied increase the voltage to the rated value step by step and I note down the applied voltage, line current and power input to the lamp.

6). Carefully I applied decrease the voltage step by step till lamp extinguishes and I note down applied voltage, line current and power input to lamp in each step.

7) Then I note down the extinguishing voltage.

8) Finally, I switch OFF the power supply and disconnect the circuit from the supply.

OBSERVATION TABLE :

Sl No.	Applied Voltage Increasing			
	Striking Voltage (Volt)			194
	Applied Voltage (volt)	Line Current ~~(Amp)~~ (mamp)	Power Input (watt)	Power Factor
1.	194	240	26	0.56
2.	198	255	28	0.55
3.	202	265	30	0.55
4.	206	275	31	0.54
5.	210	295	33	0.53
6.	214	305	34	0.52
7.	218	310	36	0.51
8.	220	315	37	0.51
9.				
10				

[illegible]
19/03/15

OBSERVATION TABLE :

Sl No.	Applied Voltage Decreasing			
	Extinguishing Voltage (Volt)			140
	Applied Voltage (Volt)	Line Current (amp)	Power Input (watt)	Power Factor
1.	190	230	24	0.54
2.	180	200	20	0.55
3.	170	160	16	0.56
4.	165	150	13	0.57
5.	160	120	11	0.57
6.	150	75	7	0.58
7.	145	50	4	0.59
8.	140	0	0	0.
9.				
10.				

[illegible]
19/03/15

CALCULATION :

Applied voltage of a fluoresent lamp is V, line current is I and input power $P = VI\cos\phi$ where

$\cos\phi$ = Power factor of fluoresent lamp.

$\therefore P = VI\cos\phi$

or, $\cos\phi = \frac{P}{VI}$

Applied voltage Increasing for striking Voltage :

1. $\cos\phi = \frac{26}{240\times 194} = 5.6\times 10^{-4}\times 10^{3} = 0.56$

2. $\cos\phi = \frac{28}{255\times 198}\times 10^{3} = 0.55$

3. $\cos\phi = \frac{30}{265\times 202}\times 10^{3} = 0.55$

4. $\cos\phi = \frac{31}{275\times 206}\times 10^{3} = 0.54$

5. $\cos\phi = \frac{33}{295\times 210}\times 10^{3} = 0.53$

Applied voltage Decreasing for Extinguishing Voltage(volt):

1) $\cos\phi = \frac{24}{190\times 230}\times 10^{3} = 0.54$

2) $\cos\phi = \frac{20}{180\times 200}\times 10^{3} = 0.55$

3) $\cos\phi = \frac{16}{170\times 160}\times 10^{3} = 0.56$

4) $\cos\phi = \frac{13}{165\times 150}\times 10^{3} = 0.57$.

DISCUSSION :

1. what is the function of starter? what is the function of choke?

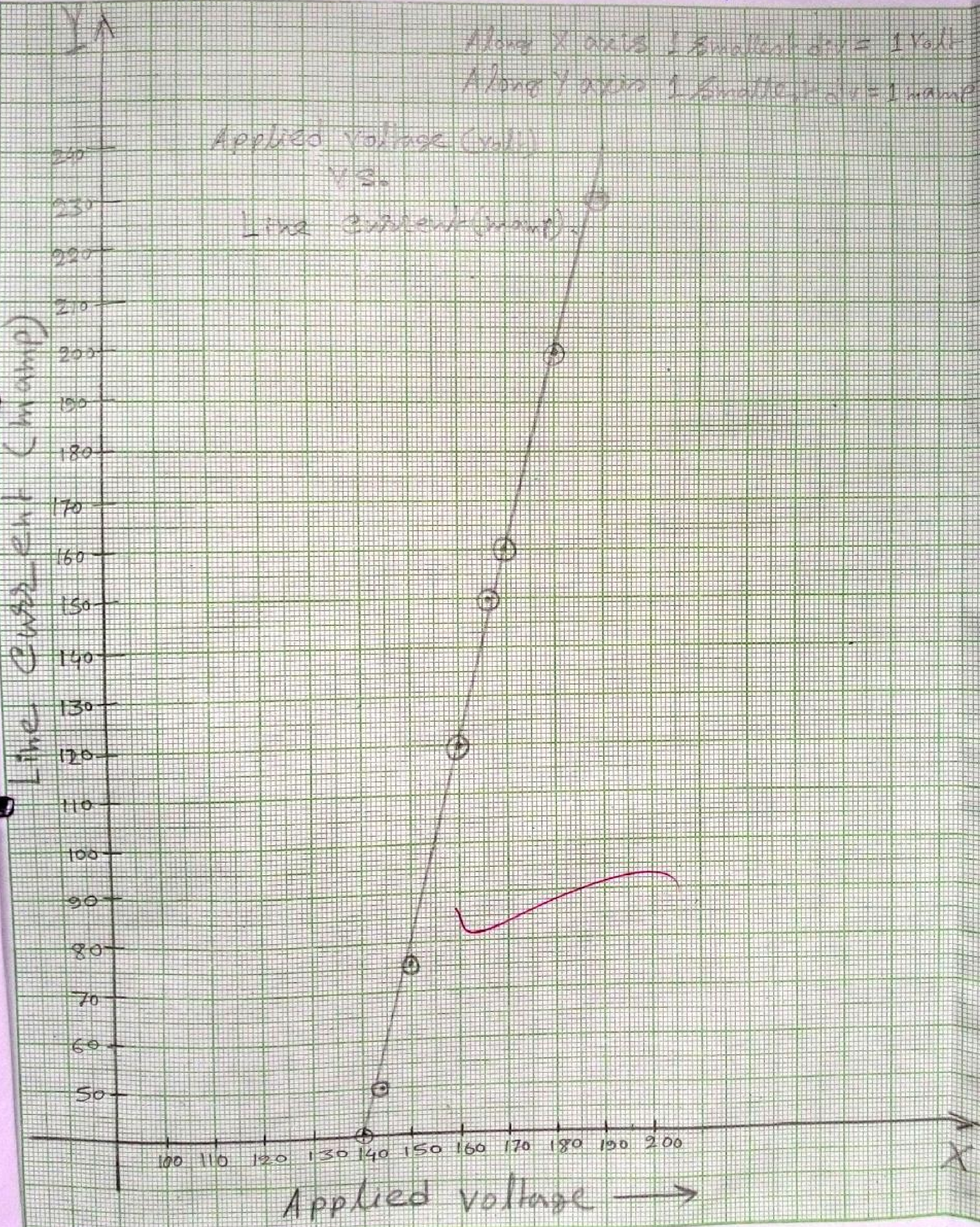
Ans. Function of starter : The gas in the fluoresent tube has a high resistance when not lit, so it takes a high voltage to start it glowing. The 'starter' stores energy and releases it all at once, causing a high voltage. Once the tube is glowing its resistance is lower, so it can continue to run without the starter. In fact the mains voltage ~~is~~ much higher than the tube required to run once the gas inside it has been 'lit' by the starter.

Function of choke : The function of choke it to provide high voltage enough for ionisation to take place in a tube light and often establishment and substance of ionisation limits the voltage across the tube. That is the reason why a tube fuses when the choke shorted.

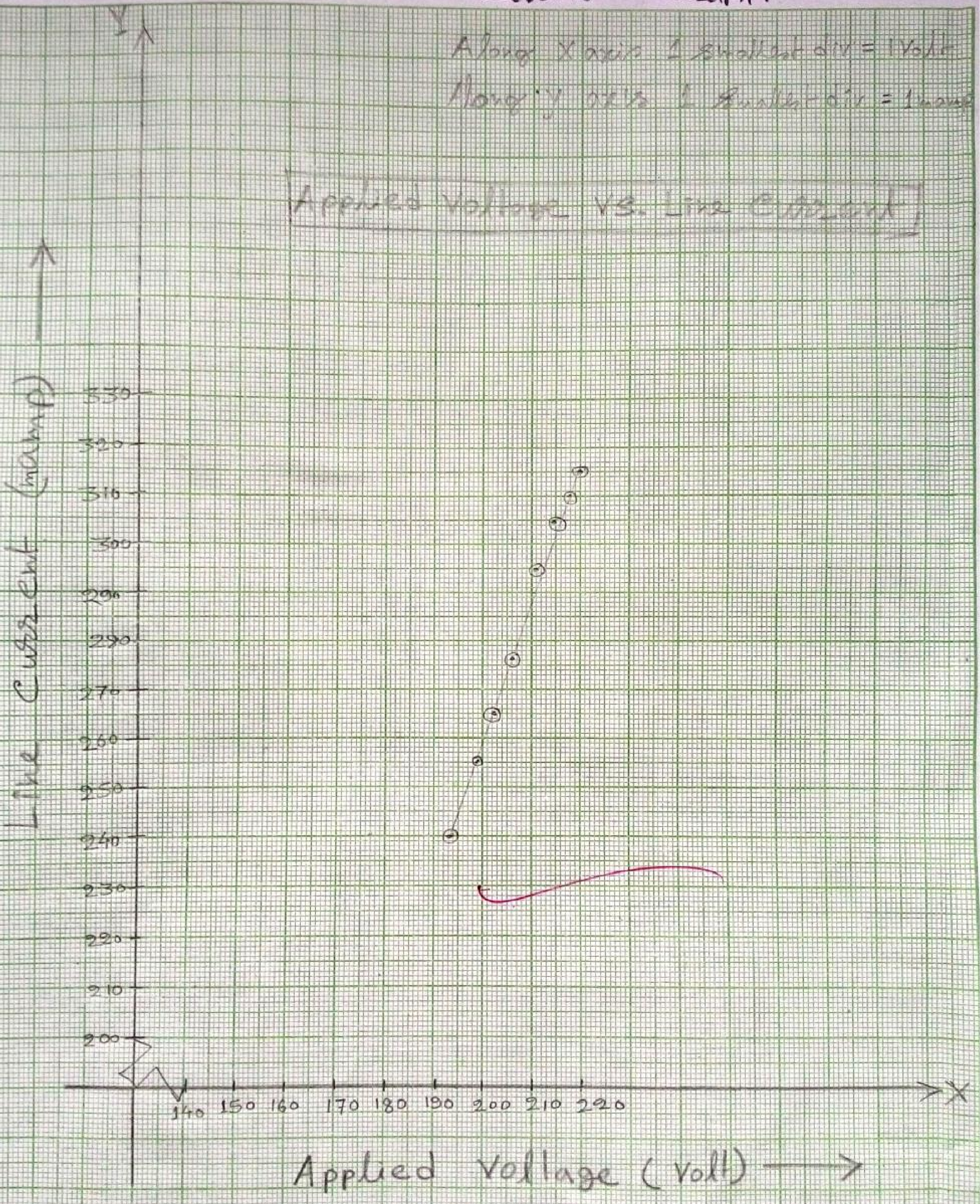
2. Can we use fluoresent lamp in DC ?

Ans) Yes, a fluoresent lamp work on a DC if we give a resistance to choke in series connection.

Applied voltage Decreasing Graph. 7.



Applied Voltage Increasing Graph. 8.



ARGENTOMETRIC METHOD

AIM :→ Determination of chloride ion in a given water sample by Argentometric method (using chromate indicator solution).

THEORY :→ When silver nitrate solution is added to a given solution of chloride containing few drops of indicator (K_2CrO_4), white silver chloride is precipited initially [since $K_{SP}(AgCl) < K_{SP}(Ag_2CrO_4)$]. After finishing all the chloride ions as AgCl, Ag^+ binds CrO_4^{2-} ion and forms a brick red ppt of Ag_2CrO_4. This brick red colour indicates the end point.

$$Ag^+ + Cl^- \longrightarrow \underset{\text{White precipitation}}{AgCl}$$

$$2Ag^+ + CrO_4^{2-} \longrightarrow \underset{\text{Brick red precipitation}}{Ag_2CrO_4}$$

APPARATUS :→

- Burette
- pipette (25 ml)
- Conical flask

REAGENTS :→

- $AgNO_3$ solution (N/50)
- K_2CrO_4 indicator

PROCEDURE :→

[1] Fill the washed and rinsed burette with $AgNO_3$ (N/50) solution.

[2] Wash the pipette with water and then rinse it with chloride solution. pipette out 25ml of chloride solution in a clean conical flask.

[3] Add 4 drops of K_2CrO_4 indicator.

[4] Add $AgNO_3$ from the burette, shaking the flask constantly. A white ppt. AgCl is obtained. After the addition of few ml of $AgNO_3$, a red colour appears in the flask but disappears quickly upon shaking.

[5] Continue the addition drop by drop till a permanent reddish brown colour is obtained. Take three readings.

RESULTS AND CALCULATIONS :→

TABLE : DETERMINATION OF CHLORIDE ION :-

NO. OF OBS.	VOLUME OF SAMPLE TAKEN (ml)	BURETTE READING		VOL.OF $AgNO_3$ CONSUMED (ml)	CONCORDANT VOLUME OF $AgNO_3$ (ml)	AMOUNT OF CHLORIDE ION PRESENT (ppm)
		INITIAL	FINAL			
{1}	25	0	0·5	0·5	✓	
{2}	25	0·5	1·0	0·5	0·5	14·2
{3}	25	1·0	1·6	0·6		

We know that, $V_1 S_1 = V_2 S_2$

Where, V_1 = volume of $AgNO_3$

S_1 = strength of $AgNO_3 = \frac{1}{50}(N)$

S_2 = strength of water due to present of $Cl^{\ominus}$ ion

V_2 = Volume of water

$$\therefore \; S_2 = \frac{V_1 S_1}{V_2}$$

DATE: 24-02-2014

$$S_2 = \frac{0.5 \times 1/50}{25} (N)$$

$$= 0.0004 (N)$$

Amount of chloride ion present $= 0.0004 \times 35.5$ gms/lit

$= 0.0004 \times 35.5 \times 10^3$ ppm

$= 14.2$ ppm

DISCUSSION :→

[1] After addition of silver nitrate from burette in to the solution, silver chloride is precipitated.

[2] solubility product of silver chromate is greater than silver chloride, thus silver chloride is precipatated untill all the chloride ions are attached to the silver and forming silver chloride.

[3] The brick red colouration of the solution is due to formation of silver chromate.

CONCLUSION :→

The amount of chloride ion (determined by Argentometric method) Present in given water sample is 14.2 ppm.

ALKALINITY OF WATER SAMPLE

AIM :→ Determination of alkalinity of a given water sample.

THEORY :→ Alkalinity establishes the buffering capacity of water and affects how much acid is required to change the pH. The alkalinity of water is due to presence of hydroxide ion (OH^-), carbonate ion (CO_3^{2-}) and bicarbonate ion (HCO_3^-). Therefore, we have to find out the concentration in ppm of these ions.

In the first step, the water sample is titrated with acid using phenolphthalein as indicator. The colour change (pink to colorless) indicates neutralization of hydroxide ion (OH^-) to water and / or half neutralization of carbonate ion (CO_3^{2-}) to bicarbonate ion (HCO_3^-).

In the second step, further titration of the above water sample using methyl orange as indicator is carried out. The colour change (yellowish to pink) indicates complete neutralization of bicarbonate ion (HCO_3^-) which is formed from (CO_3^{2-}) and /or present in the water sample originally. The concentration of both ions may be calculated from the sets of titration.

Ist Step :→

$$OH^- + H^+ = H_2O$$
$$CO_3^{2-} + H^+ = HCO_3^-$$

For the above process phenolphthalein is used as indicator.

DATE:- 16-02-2014

IInd step :→

$$HCO_3^- + H^+ = H_2O + CO_2$$

For the above process methyl orange is used as indicator. The possibility combinations of ions causing alkalinity in water are :

{i} OH^- only, CO_3^{2-} only, HCO_3^- only or

{ii} OH^- and CO_3^{2-}, CO_3^{2-} and HCO_3^- together

{iii} The possibility of OH^- and HCO_3^- ions together is very remote since they combine together to form CO_3^{2-} ions.

$$OH^- + HCO_3^- = CO_3^{2-} + H_2O$$

In our lab addition of phenolphthalein to water sample does not lead to any pink colouration which indicates that OH^- and CO_3^{2-} are absent. But in addition of methyl orange the colour become yellow that is titrated till the yellow colour is changed to orange. This indicates HCO_3^- alkalinity.

PROCEDURE :→

{1} After washing, rinse and fill the burette with N/50 H_2SO_4 (supplied).

{2} Wash the conical flask with distilled water and add 10 ml. of the sample solution (to be analyzed for alkalinity).

{3} Add 1-2 drops of methyl orange indicator. The solution will acquire yellow colour.

{4} Note the initial reading of burette and start adding the acid till addition of last drop changes the colour from yellow to pink.

{5} Record the final reading in the observation table.

{6} Repeat to get at least three concordant reading.

DATE:- 16.02.2014

RESULTS :→

TABLE: DETERMINATION OF ALKALINITY OF WATER

NO. OF OBS.	VOLUME OF WATER SAMPLE TAKEN (ml)	BURETTE READING		VOL.OF H2SO4 CONSUMED (ml)	CONCORDANT READING (ml)	STRENGTH OF H_2SO_4 (N)	$CaCO_3$ ALKALINITY OF WATER SAMPLE (ppm)	HCO_3^- CONCENTRATION IN WATER SAMLE (mg/l)
		INITIAL	FINAL					
{1}	10	0	2.2	2.2				
{2}	10	2.2	4.4	2.2	2.2	1/50	220	268.4
{3}	10	4.7	2.3	2.3				

CALCULATION :→

1000 ml of 1 (N) H_2SO_4 sol^n^ ≡ 1000 ml of 1 (N) $CaCO_3$ sol^n^

= 50 gm of $CaCO_3$

Hence,

1 ml of 1(N) H_2SO_4 solution $= \frac{50}{1000}$ gm of $CaCO_3$

$= \frac{50}{1000} \times 1000$ mg of $CaCO_3$

= 50 mg of $CaCO_3$

DATE:- 16-02-2014

2.2 ml of N/50 H_2SO_4 solution $= 2.2 \times \frac{50}{50}$ mg $CaCO_3$

$= 2.2$ mg of $CaCO_3$

Hence,

10 ml of water sample contains 2.2 mg of $CaCO_3$

1 ml " " " " $\frac{2.2}{10}$ mg of $CaCO_3$

1000 ml " " " " $\frac{2.2}{10} \times 1000$ mg of $CaCO_3$

Thus, alkalinity (in terms of $CaCO_3$) $= 100 \times 2.2$ ppm

$= 220$ ppm

50 mg of $CaCO_3 \equiv 61$ mg of HCO_3^-

HCO_3^- ion concentration $= \frac{61}{50} \times 100 \times 2.2$

$= 268.4$ mg/lit

DISCUSSION :→

Neutralization of CO_3^{2-} Proceeds in two steps. In the first step conversion of CO_3^{2-} to HCO_3^- makes the solution alkaline and phenolphthalein (whose pKin is 8.6 showing colour changes in the pH region of 9.6 to 10.6) is the suitable indicator. In the second step there is total neutralization of HCO_3^- leading to formation of H_2CO_3 PKin 3.7 shows color change in the pH region of 2.7 to 4.7. Hence in our experiment only methyl orange is used as the indicator.

DATE:- 16-02-2014

PRECAUTION :→

For estimation of HCO_3^- one must not use excess methyl orange then the readings will not be very correct and estimation will not be accurate. The apparatus should be well cleaned with distilled water prior to the experiment. If it is not done then it may so happen that impurities present in the un-cleaned containes will alter the pH and characteristics of indicator.

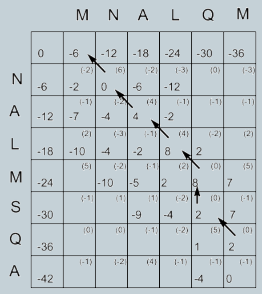
Temperature sholld be maintained between 25°C - 30°C.

CONCLUSION :→

The alkalinity of water sample in terms of $CaCO_3$ is 220 ppm and in terms of HCO_3^- ion concentration is 268.4 mg/lit measured at temperature of 25°C (room temperature).

Introduction to Graph Theory

HANDBOOK OF GRAPH THEORY FOR FRESHER'S



PREM SANKAR C

M TECH TECHNOLOGY MANAGEMENT

DEPT OF FUTURES STUDIES ,KERALA UNIVERSITY

Outline

1. History of Graph Theory
2. Basic Concepts of Graph Theory
3. Graph Representations
4. Graph Terminologies
5. Different Type of Graphs

Why Graph Theory ?

- Graphs used to model pair wise relations between objects
- Generally a network can be represented by a graph
- Many practical problems can be easily represented in terms of graph theory

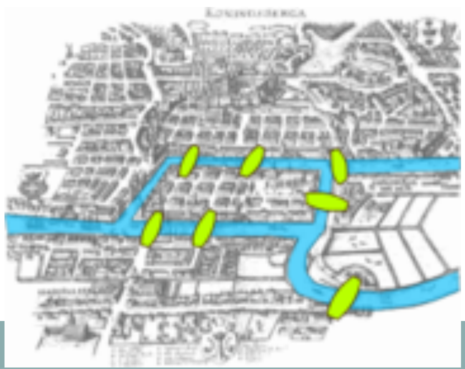
Graph Theory - History

The origin of graph theory can be traced back to Euler's work on the Konigsberg bridges problem (1735), which led to the concept of an Eulerian graph. The study of cycles on polyhedra by the Thomas P. Kirkman (1806 - 95) and William R. Hamilton (1805-65) led to the concept of a Hamiltonian graph.

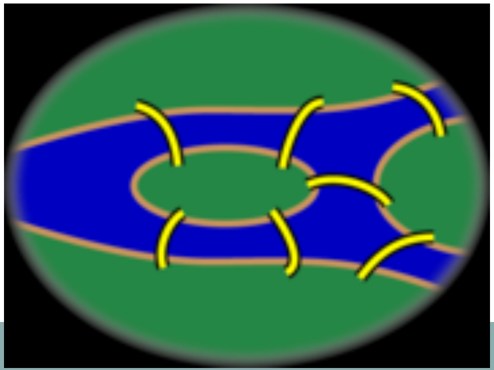
Graph Theory - History

- Begun in 1735
- Mentioned in Leonhard Euler's paper on "*Seven Bridges of* Konigsberg ".

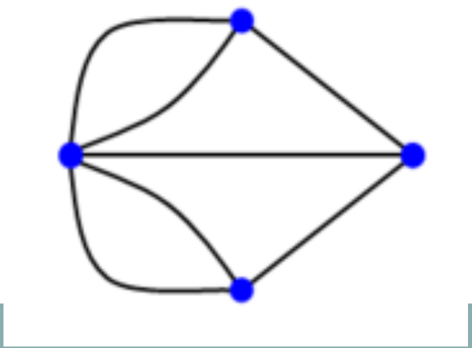
Problem : Walk all 7 bridges without crossing a bridge twice



$\longrightarrow$



$\longrightarrow$

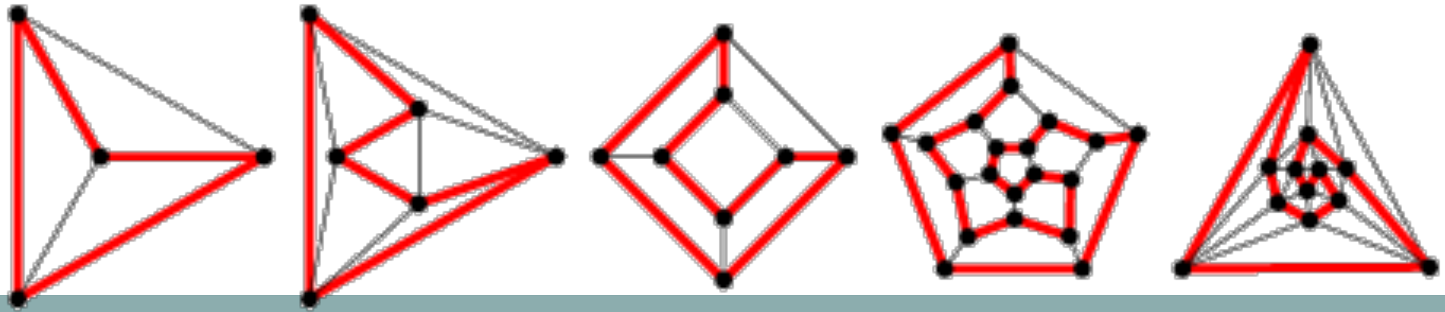


Graph Theory – History…….

Cycles in Polyhedra - polyhedron with no Hamiltonian cycle

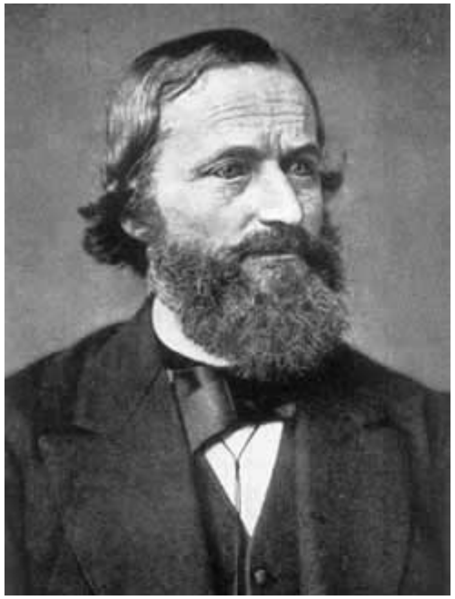
Thomas P. Kirkman

William R. Hamilton

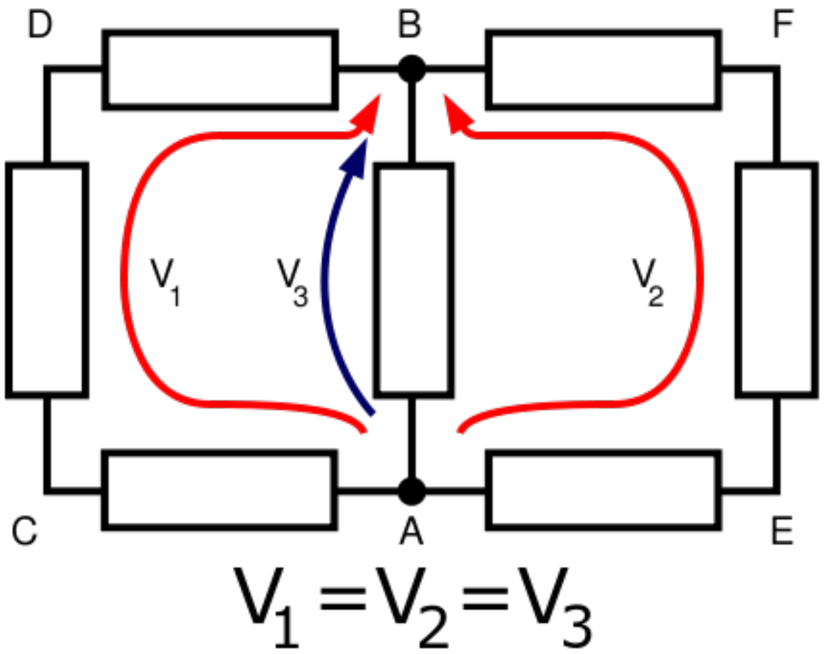

Hamiltonian cycles in Platonic graphs

Graph Theory – History.....

Trees in Electric Circuits



Gustav Kirchhoff



Basic Concepts of Graph Theory

Definition: Graph

- A graph is a collection of nodes and edges
- Denoted by $G = (V, E)$.

  V = **nodes** (vertices, points).

  E = **edges** (links, arcs) between pairs of nodes.

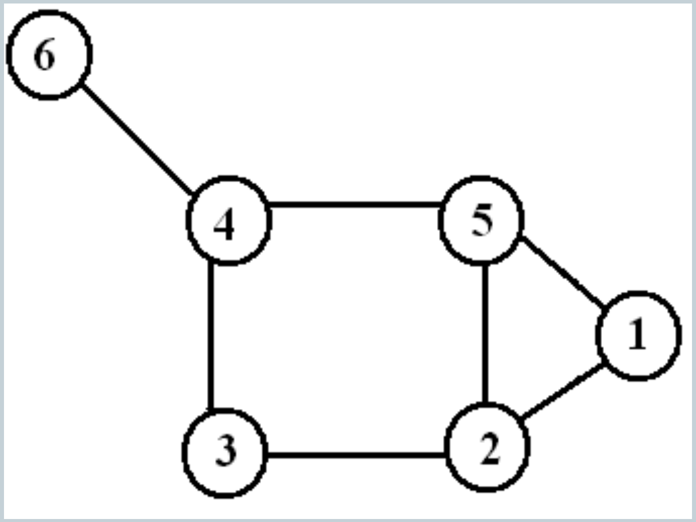
  Graph size parameters: $n = |V|$, $m = |E|$.

-

Vertex & Edge

- Vertex /Node
  - Basic Element
  - Drawn as a node or a dot.
  - V**ertex set** of G is usually denoted by V(G), or V or V_G
- Edge /Arcs
  - A set of two elements
  - Drawn as a line connecting two vertices, called end vertices, or endpoints.
  - The edge set of G is usually denoted by E(G), or E or E_G
- Neighborhood
  - For any node v, the set of nodes it is connected to via an edge is called its neighborhood and is represented as N(v)

Graph :Example



- n:= 6 , m:=7
- Vertices (V) :={1,2,3,4,5,6}
- Edge (E) := {1,2},{1,5},{2,3},{2,5},{3,4},{4,5},{4,6}}
- *N(4) := Neighborhood (4) ={6,5,3}*

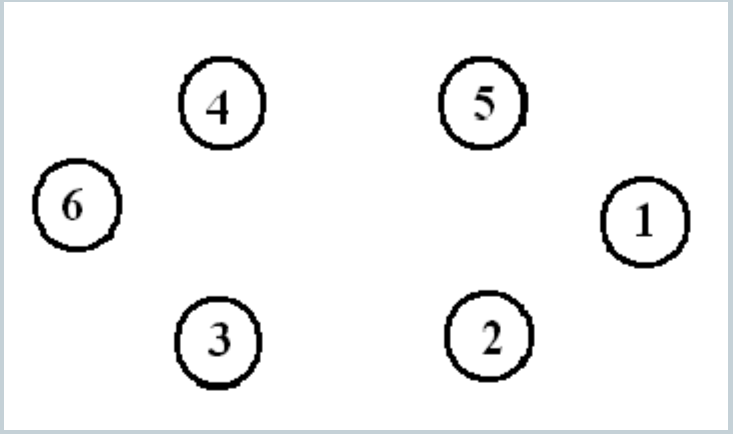
Edge types:

- **Undirected**;
  - E.g., distance between two cities, friendships…
- **Directed**; ordered pairs of nodes.
  - E.g ,…
  - Directed edges have a **source** (head, origin) and **target** (tail, destination) vertices

- **Weighted ;** usually weight is associated .

Empty Graph / Edgeless graph

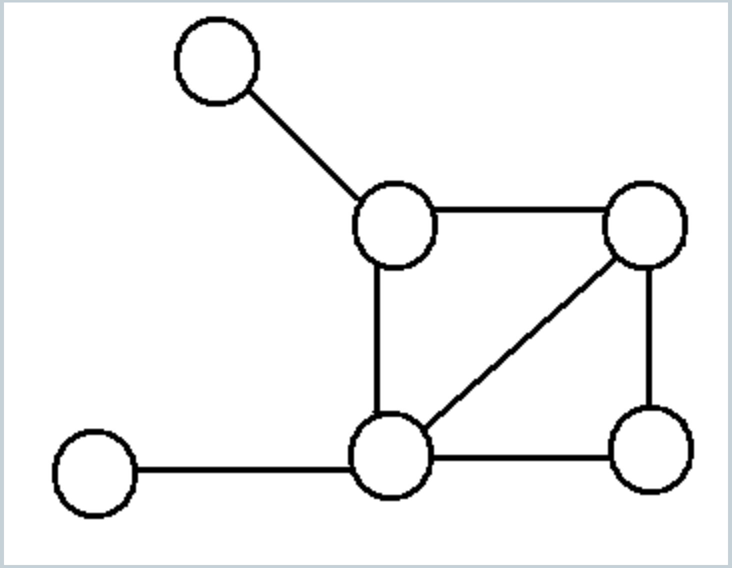
- No edge



- Null graph
  - No nodes
  - Obviously no edge

Simple Graph (Undirected)

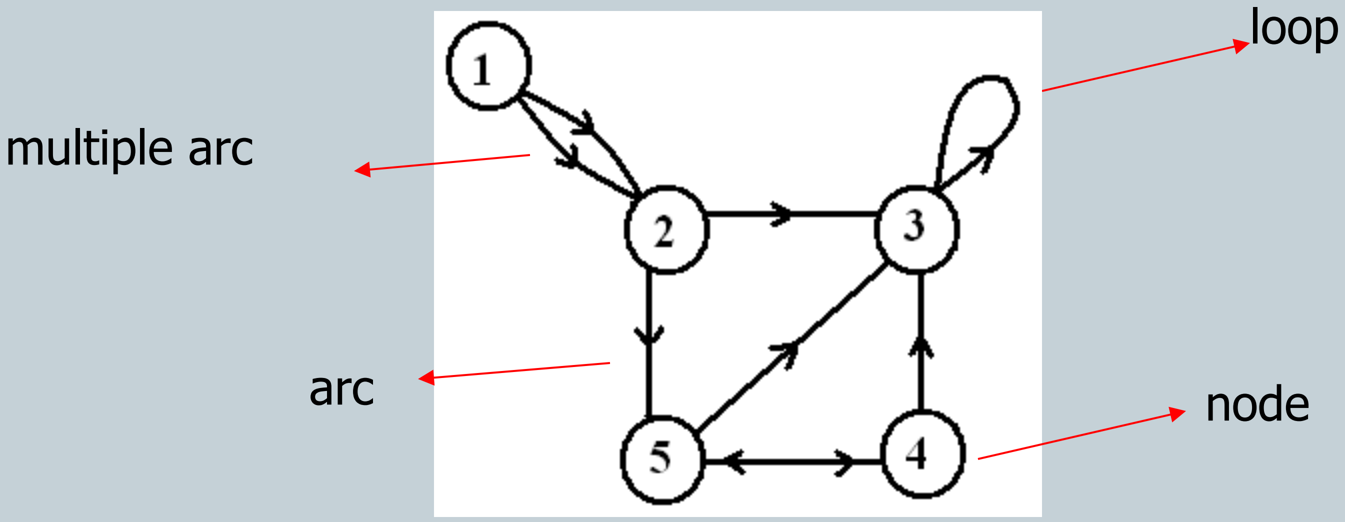
- Simple Graph are undirected graphs without loop or multiple edges
- A = AT



For simple graphs, $\sum_{v_i \in V} \deg(v_i) = 2|E|$

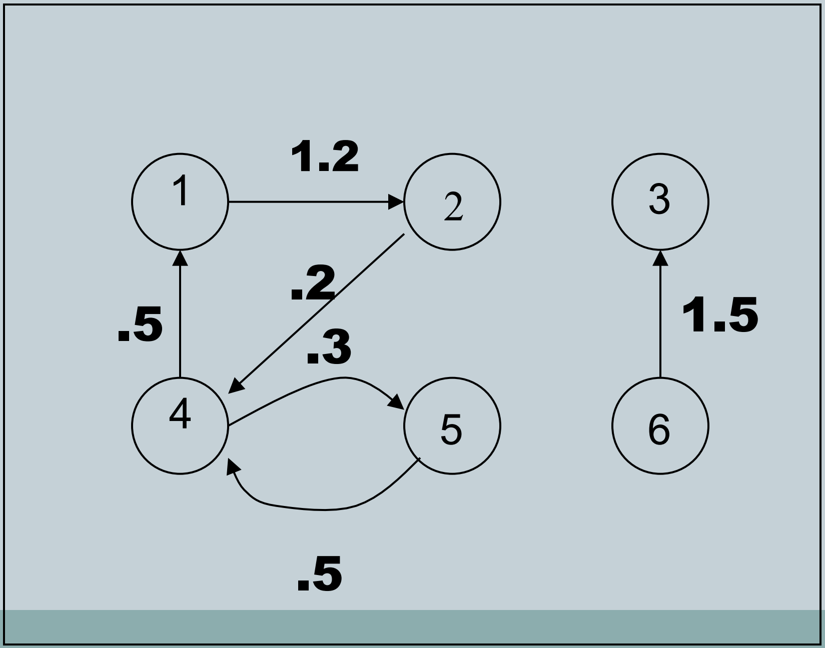
Directed graph : (digraph)

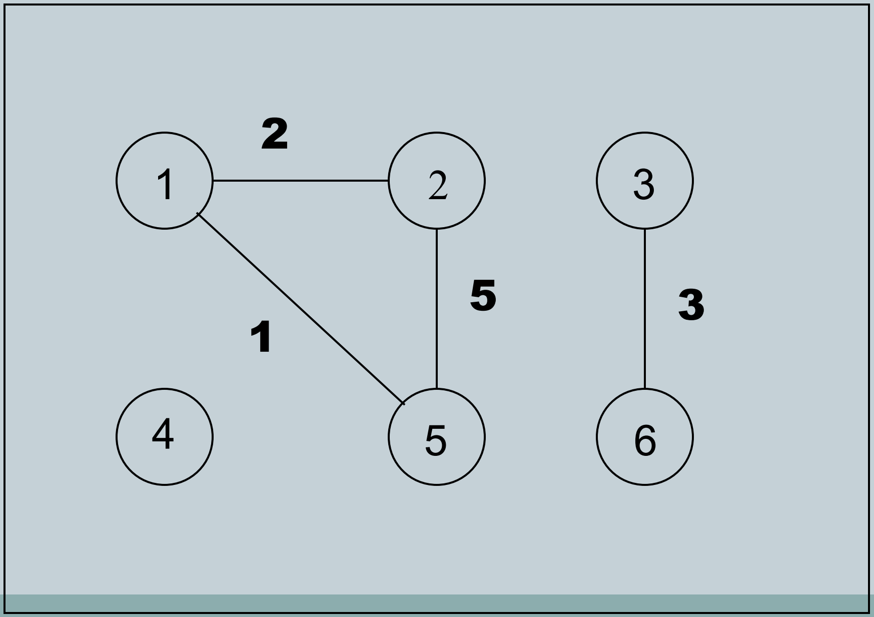
- Edges have directions
- A !=AT



Weighted graph

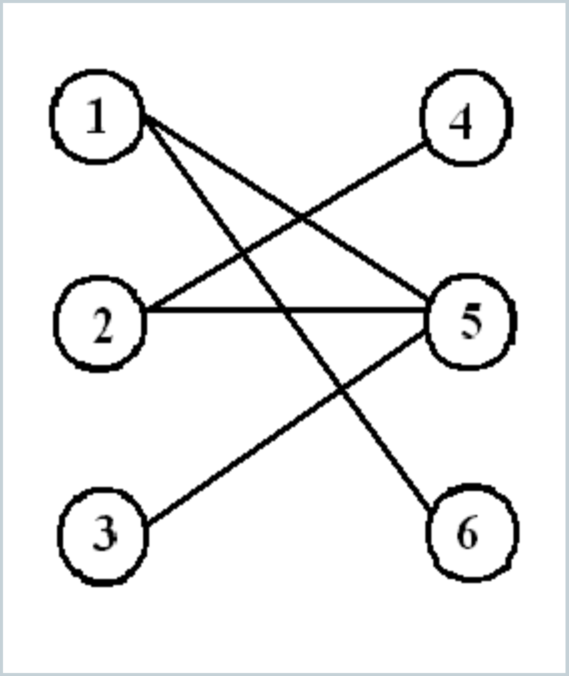
- is a graph for which each edge has an associated ***weight***





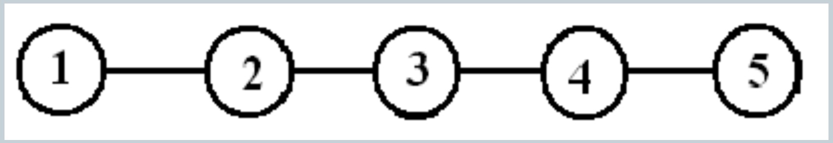
Bipartite Graph

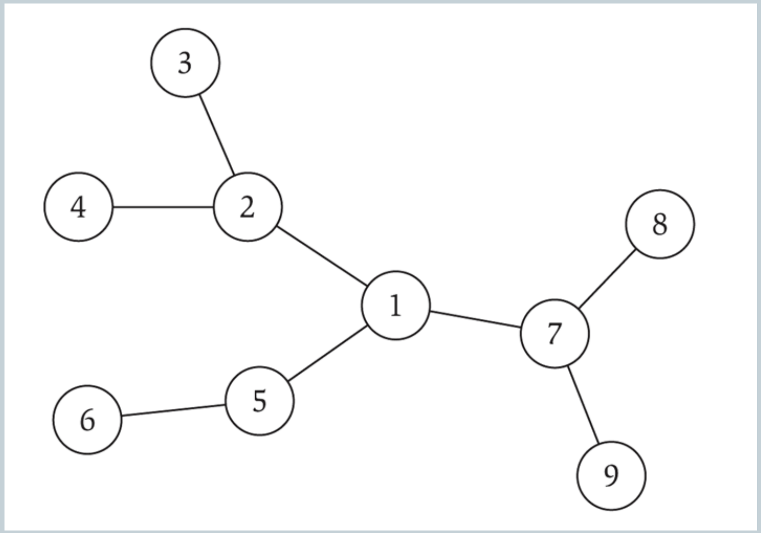
V can be partitioned into 2 sets V_1 and V_2
such that $(u,v) \in E$ implies
either $u \in V_1$ and $v \in V_2$
OR $v \in V_1$ and $u \in V_2$.

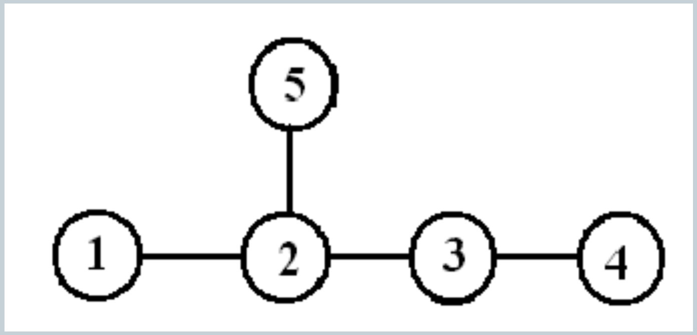


Trees

- An undirected graph is a **tree** if it is connected and does not contain a cycle (Connected Acyclic Graph)
- Two nodes have *exactly* one path between them

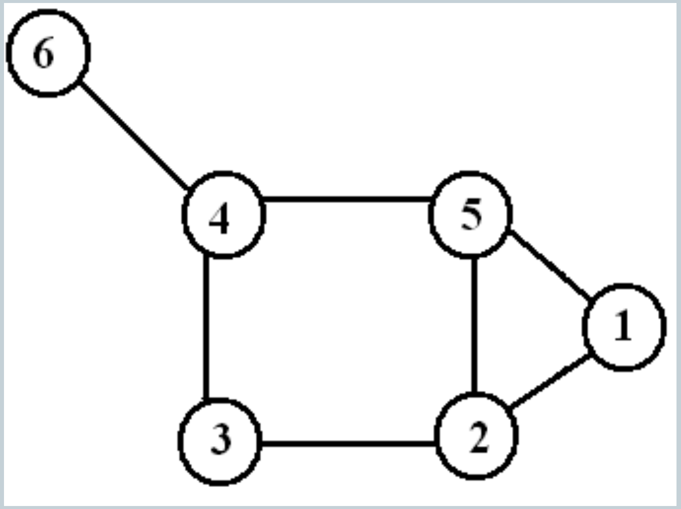


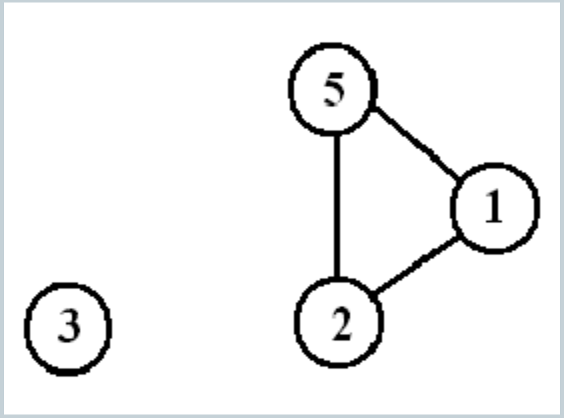




Subgraph

- Vertex and edge sets are subsets of those of G
  - a *supergraph* of a graph G is a graph that contains G as a subgraph.

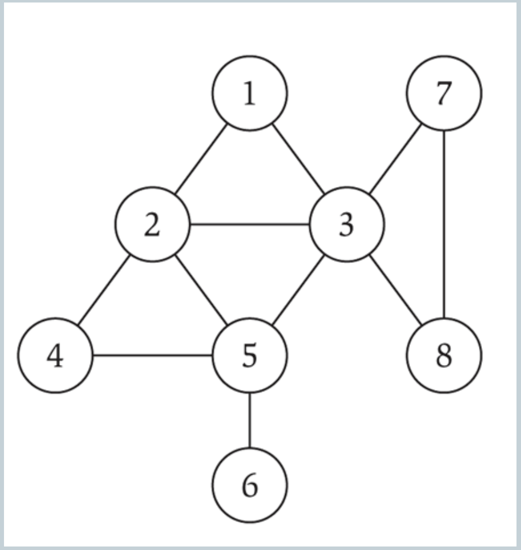




Graph Representations

1. Adjacency Matrix

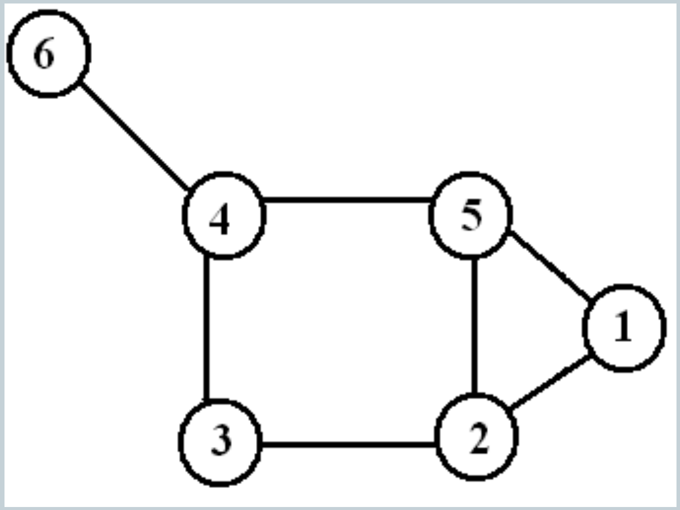
- n-by-n matrix with $A_{uv} = 1$ if (u, v) is an edge.
  - Diagonal Entries are self-links or loops
  - Symmetric matrix for undirected graphs



	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	1	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

2. Incidence Matrix

- V x E
- [vertex, edges] contains the edge's data



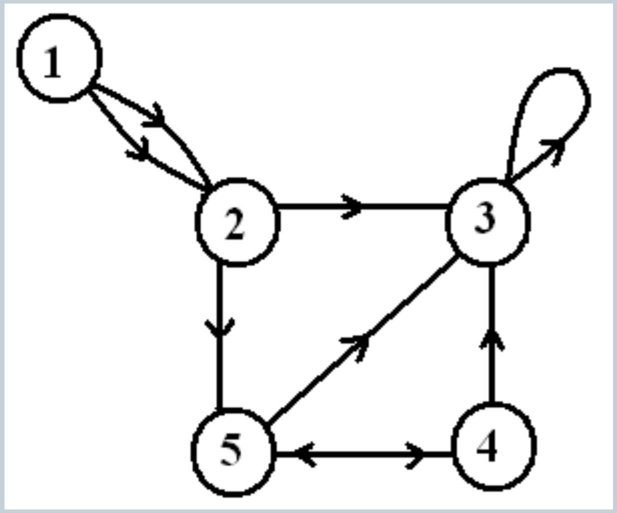
$$\begin{array}{c|ccccccc}
 & 1,2 & 1,5 & 2,3 & 2,5 & 3,4 & 4,5 & 4,6 \\
\hline
1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
2 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\
3 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
4 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
5 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
6 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{array}$$

3. Adjacency List

- Edge List

Edge List

1 2
1 2
2 3
2 5
3 3
4 3
4 5
5 3
5 4

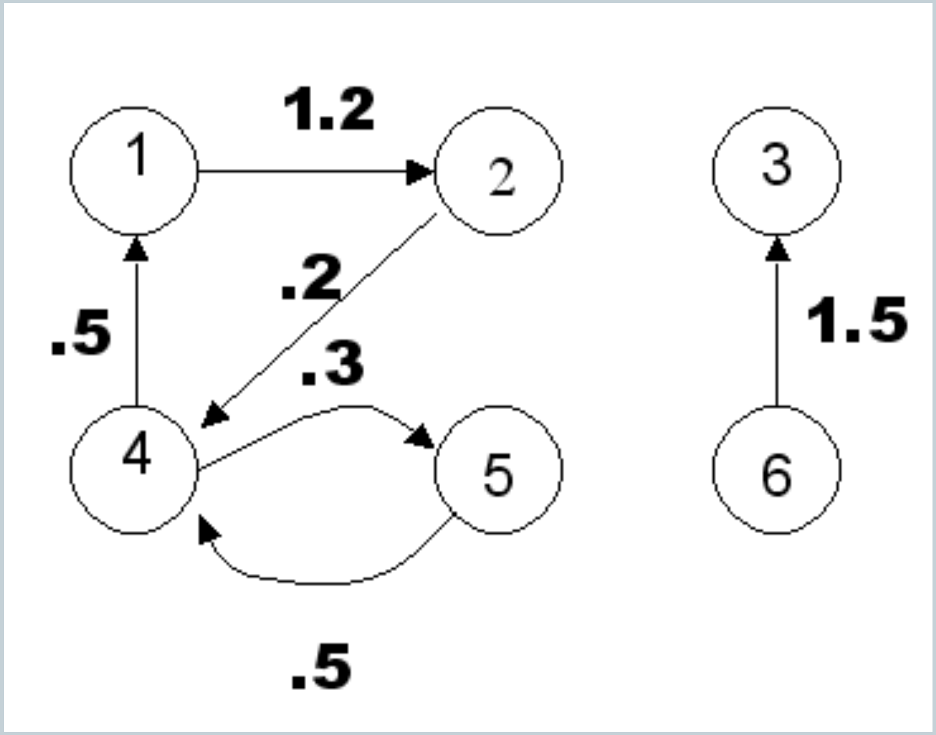


- Adjacency List (node list)

Node List

1 2 2
2 3 5
3 3
4 3 5
5 3 4

Edge Lists for Weighted Graphs



Edge List
1 2 1.2
2 4 0.2
4 5 0.3
4 1 0.5
5 4 0.5
6 3 1.5

Graph Terminologies

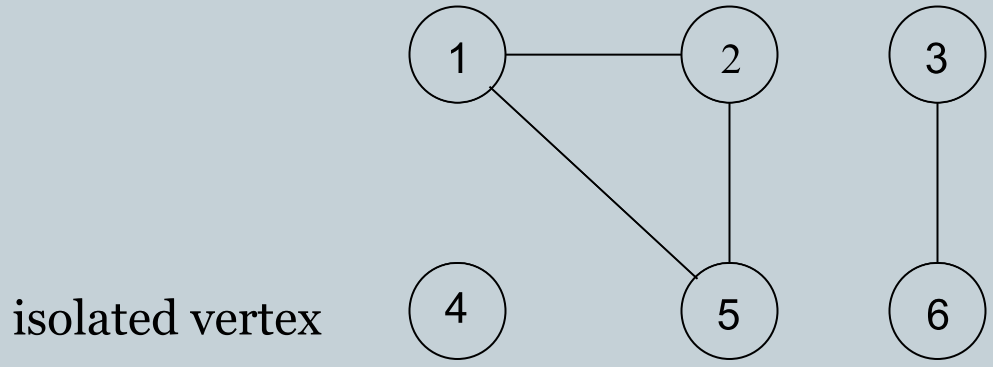
Classification of Graph Terms

- Global terms refer to a whole graph
- Local terms refer to a single node in a graph

Connected and Isolated vertex

- Two vertices are **connected** if there is a path between them

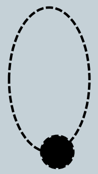
- Isolated vertex – not connected



Adjacent nodes

- **Adjacent nodes** **-**Two nodes are adjacent if they are connected via an edge.
  - If edge **e={*u*,*v*}** *∈ **E(G),*** we say that *u* and *v* are **adjacent** or **neigbors**

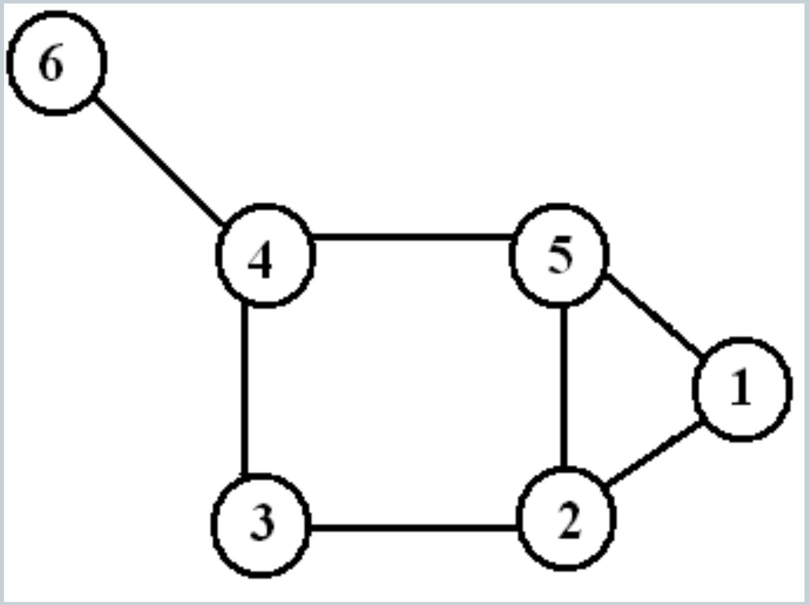
- **An edge where the two end vertices are the same is called a loop, or a self-loop**



Degree (Un Directed Graphs)

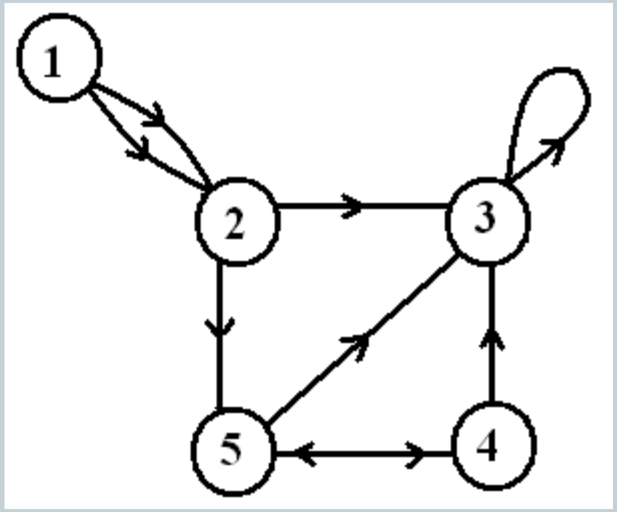
- Number of edges incident on a node

The degree of 5 is 3



Degree (Directed Graphs)

- In-degree: Number of edges entering
- Out-degree: Number of edges leaving
- Degree = indeg + outdeg



outdeg(1)=2
indeg(1)=0

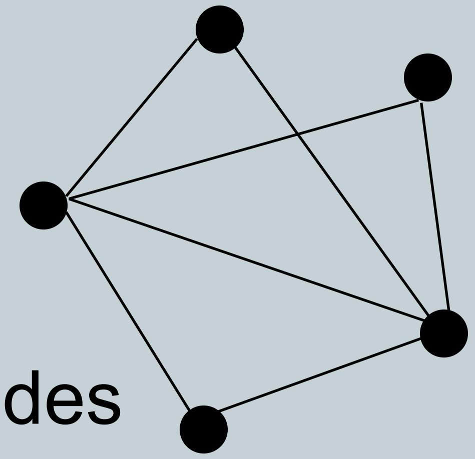
outdeg(2)=2
indeg(2)=2

outdeg(3)=1
indeg(3)=4

Walk

- **trail**: no edge can be repeat

  a-b-c-d-e-b-d



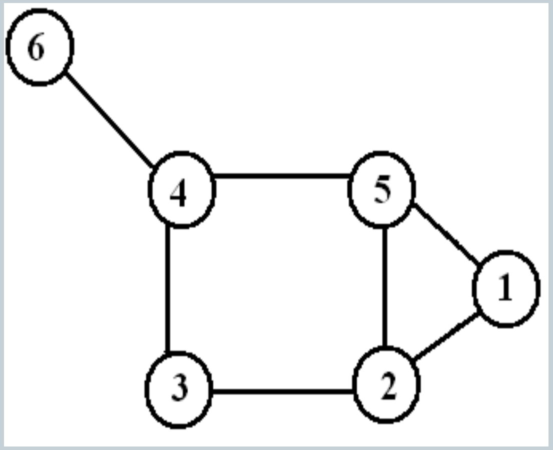
- **walk**: a path in which edges/nodes can be repeated.

  a-b-d-a-b-c

- A walk is ***closed*** is a=c

Paths

- **Path**: is a sequence P of nodes v_1, v_2, …, v_{k-1}, v_k
- No vertex can be repeated
- A closed path is called a cycle
- The length of a path or cycle is the number of edges visited in the path or cycle

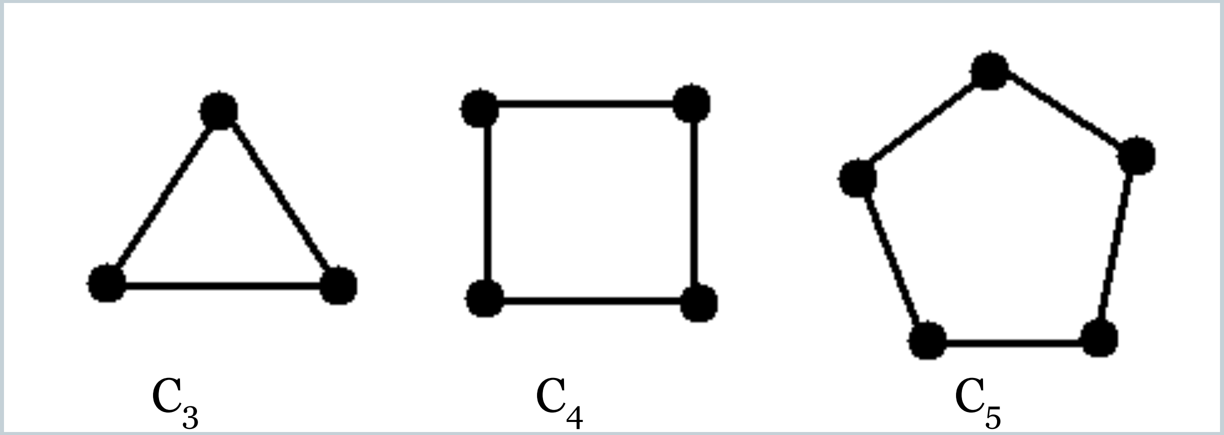


Walks and Paths

1,2,5,2,3,4	1,2,5,2,3,2,1	1,2,3,4,6
walk of length 5	CW of length 6	path of length 4

Cycle

- Cycle - closed path: **cycle (a-b-c-d-a)** , closed if x=y
- Cycles denoted by C_k, where k is the number of nodes in the cycle



Shortest Path

- **Shortest Path is the path between two nodes that has the shortest length**
- **Length** – number of edges.
- **Distance** between u and v is the length of a shortest path between them
- The diameter of a graph is the length of the longest shortest path between any pairs of nodes in the graph

THANK YOU

PREM SANKAR C

M Tech Technology Management

Dept of Futures Studies

Kerala University

3: Nodal Analysis

Aim of Nodal Analysis

The aim of nodal analysis is to determine the voltage at each node relative to the reference node (or ground). Once you have done this you can easily work out anything else you need.

Aim of Nodal Analysis

The aim of nodal analysis is to determine the voltage at each node relative to the reference node (or ground). Once you have done this you can easily work out anything else you need.

There are two ways to do this:

(1) Nodal Analysis - systematic; always works

(2) Circuit Manipulation - ad hoc; but can be less work and clearer

Aim of Nodal Analysis

The aim of nodal analysis is to determine the voltage at each node relative to the reference node (or ground). Once you have done this you can easily work out anything else you need.

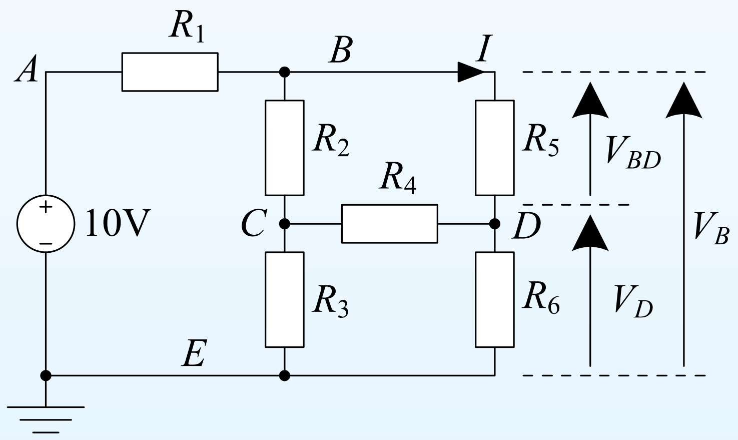
There are two ways to do this:

(1) Nodal Analysis - systematic; always works

(2) Circuit Manipulation - ad hoc; but can be less work and clearer

Reminders:

A node is all the points in a circuit that are directly interconnected. We assume the interconnections have zero resistance so all points within a node have the same voltage. Five nodes: $A, \cdots, E$.



Aim of Nodal Analysis

The aim of nodal analysis is to determine the voltage at each node relative to the reference node (or ground). Once you have done this you can easily work out anything else you need.

There are two ways to do this:

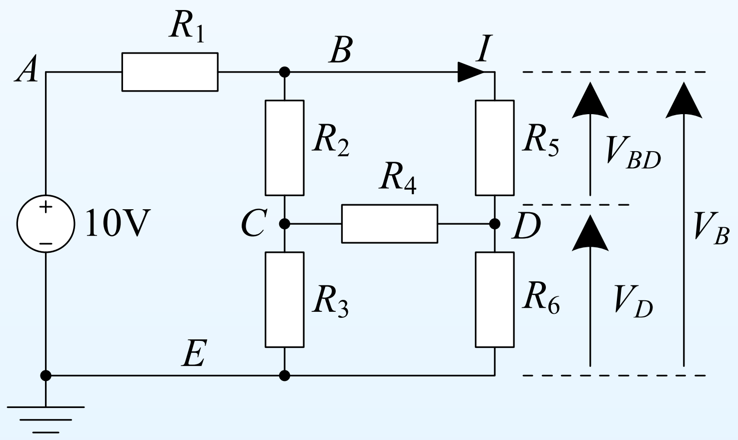
(1) Nodal Analysis - systematic; always works

(2) Circuit Manipulation - ad hoc; but can be less work and clearer

Reminders:

A node is all the points in a circuit that are directly interconnected. We assume the interconnections have zero resistance so all points within a node have the same voltage. Five nodes: $A, \cdots, E$.

Ohm's Law: $V_{BD} = IR_5$



Aim of Nodal Analysis

The aim of nodal analysis is to determine the voltage at each node relative to the reference node (or ground). Once you have done this you can easily work out anything else you need.

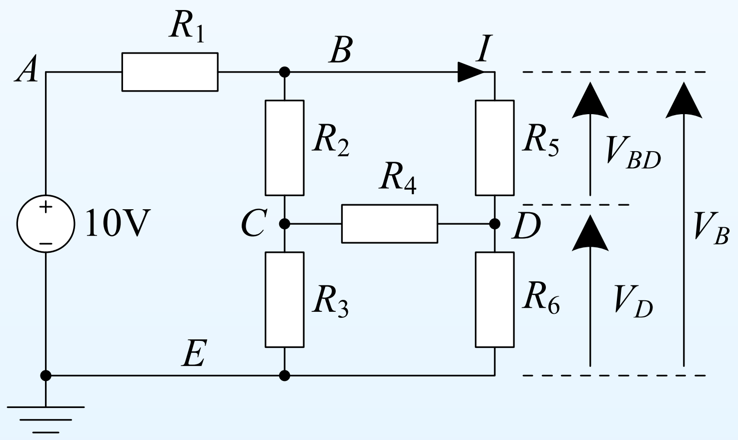
There are two ways to do this:

(1) Nodal Analysis - systematic; always works

(2) Circuit Manipulation - ad hoc; but can be less work and clearer

Reminders:

A node is all the points in a circuit that are directly interconnected. We assume the interconnections have zero resistance so all points within a node have the same voltage. Five nodes: $A, \cdots, E$.



Ohm's Law: $V_{BD} = IR_5$

KVL: $V_{BD} = V_B - V_D$

Aim of Nodal Analysis

The aim of nodal analysis is to determine the voltage at each node relative to the reference node (or ground). Once you have done this you can easily work out anything else you need.

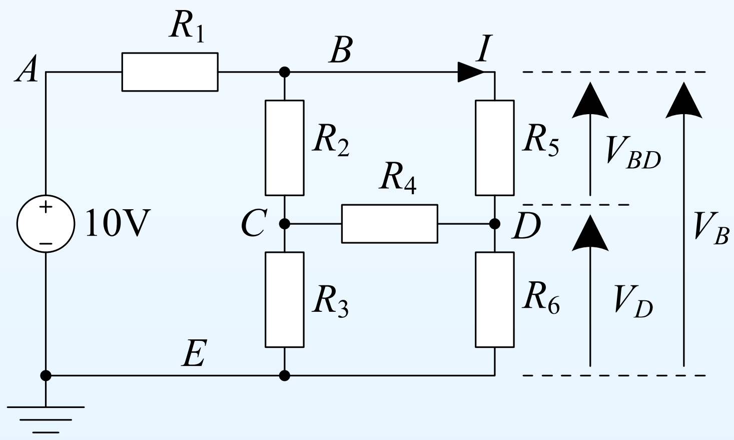
There are two ways to do this:

(1) Nodal Analysis - systematic; always works

(2) Circuit Manipulation - ad hoc; but can be less work and clearer

Reminders:

A node is all the points in a circuit that are directly interconnected. We assume the interconnections have zero resistance so all points within a node have the same voltage. Five nodes: $A, \cdots, E$.



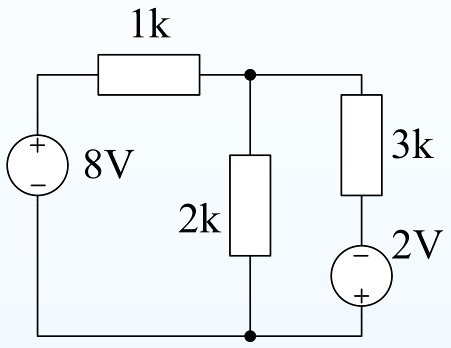
Ohm's Law: $V_{BD} = IR_5$

KVL: $V_{BD} = V_B - V_D$

KCL: Total current exiting any closed region is zero.

Nodal Analysis Stage 1: Label Nodes

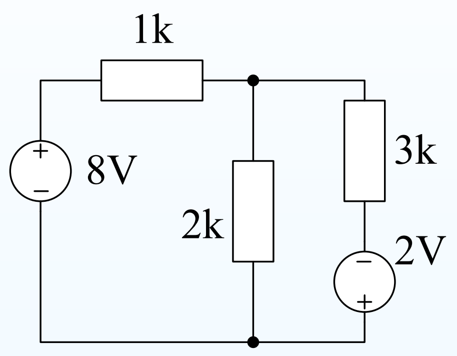
To find the voltage at each node, the first step is to label each node with its voltage as follows



(1) Pick any node as the voltage reference. Label its voltage as 0 V.
(2) If any fixed voltage sources are connected to a labelled node, label their other ends by adding the value of the source onto the voltage of the labelled end.
(3) Pick an unlabelled node and label it with X, Y, $\ldots$, then go back to step (2) until all nodes are labelled.

Nodal Analysis Stage 1: Label Nodes

To find the voltage at each node, the first step is to label each node with its voltage as follows



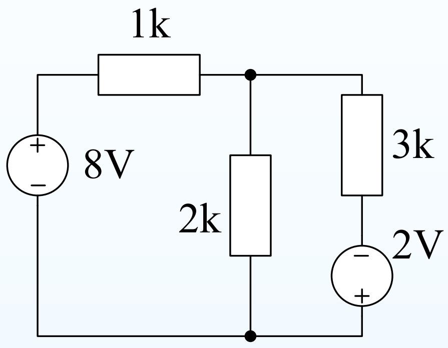
(1) Pick any node as the voltage reference. Label its voltage as $0\ \text{V}$.

(2) If any fixed voltage sources are connected to a labelled node, label their other ends by adding the value of the source onto the voltage of the labelled end.

(3) Pick an unlabelled node and label it with X, Y, $\ldots$, then go back to step (2) until all nodes are labelled.

Nodal Analysis Stage 1: Label Nodes

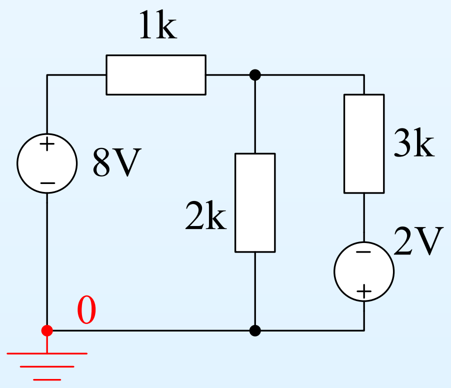
To find the voltage at each node, the first step is to label each node with its voltage as follows



(1) Pick any node as the voltage reference. Label its voltage as $0\ \mathrm{V}$.

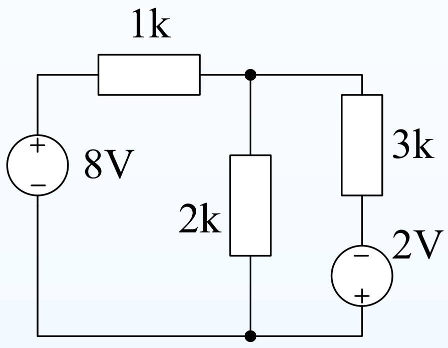
(2) If any fixed voltage sources are connected to a labelled node, label their other ends by adding the value of the source onto the voltage of the labelled end.

(3) Pick an unlabelled node and label it with X, Y, $\ldots$, then go back to step (2) until all nodes are labelled.



Nodal Analysis Stage 1: Label Nodes

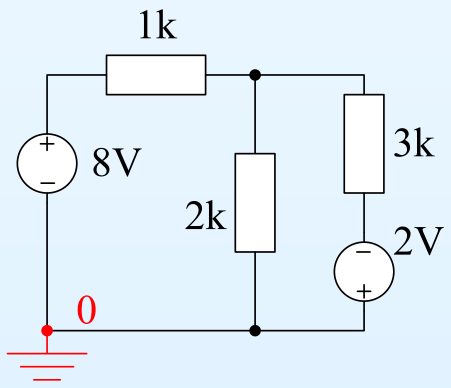
To find the voltage at each node, the first step is to label each node with its voltage as follows



(1) Pick any node as the voltage reference. Label its voltage as 0 V.

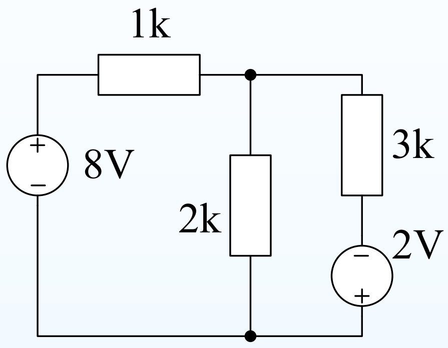
(2) If any fixed voltage sources are connected to a labelled node, label their other ends by adding the value of the source onto the voltage of the labelled end.

(3) Pick an unlabelled node and label it with X, Y, $\ldots$, then go back to step (2) until all nodes are labelled.



Nodal Analysis Stage 1: Label Nodes

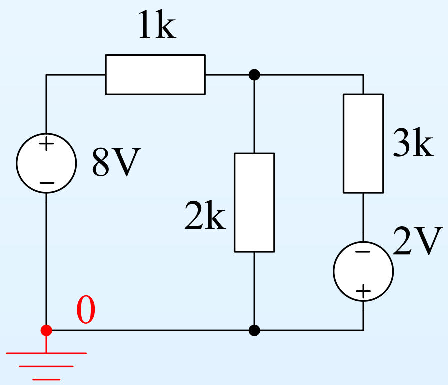
To find the voltage at each node, the first step is to label each node with its voltage as follows

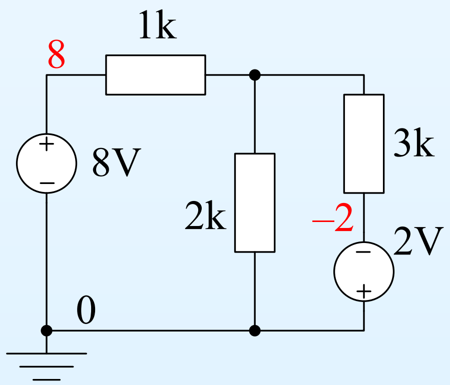


(1) Pick any node as the voltage reference. Label its voltage as 0 V.

(2) If any fixed voltage sources are connected to a labelled node, label their other ends by adding the value of the source onto the voltage of the labelled end.

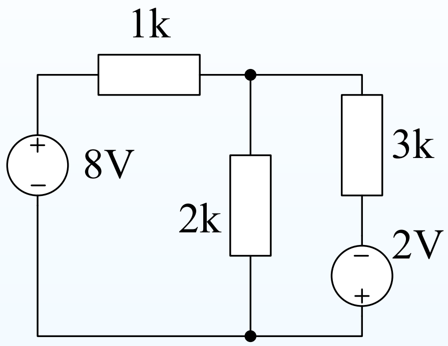
(3) Pick an unlabelled node and label it with X, Y, $\ldots$, then go back to step (2) until all nodes are labelled.





Nodal Analysis Stage 1: Label Nodes

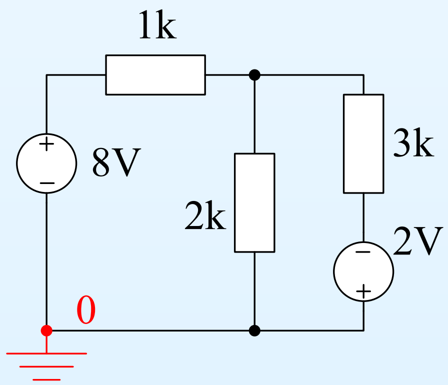
To find the voltage at each node, the first step is to label each node with its voltage as follows

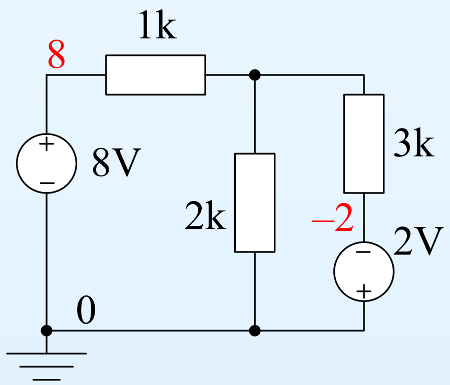


(1) Pick any node as the voltage reference. Label its voltage as 0 V.

(2) If any fixed voltage sources are connected to a labelled node, label their other ends by adding the value of the source onto the voltage of the labelled end.

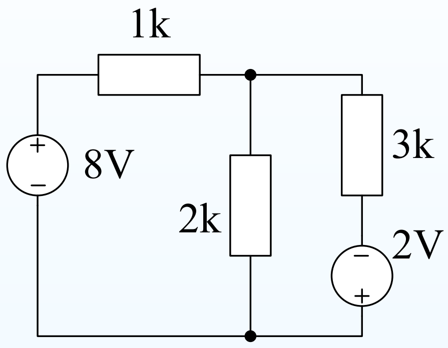
(3) Pick an unlabelled node and label it with X, Y, $\ldots$, then go back to step (2) until all nodes are labelled.





Nodal Analysis Stage 1: Label Nodes

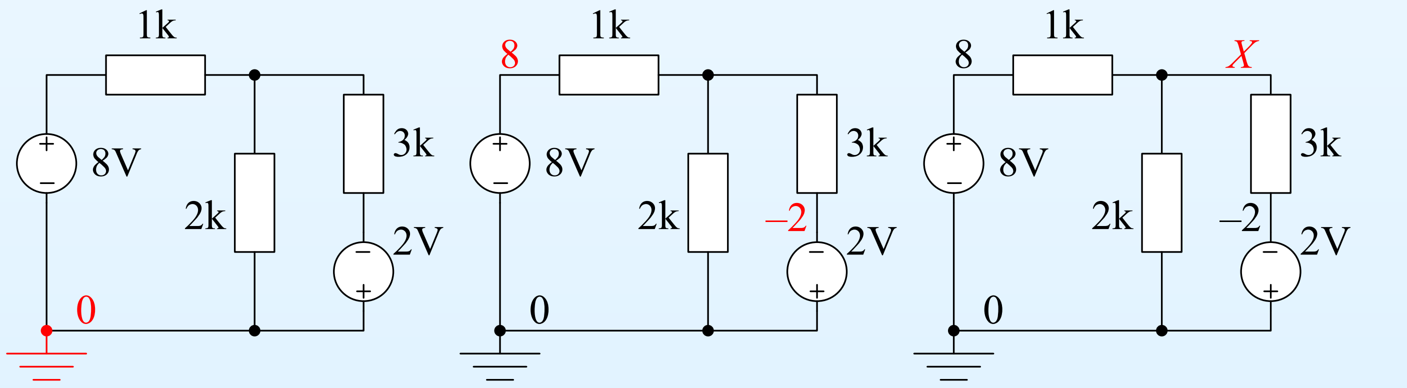
To find the voltage at each node, the first step is to label each node with its voltage as follows



(1) Pick any node as the voltage reference. Label its voltage as 0 V.

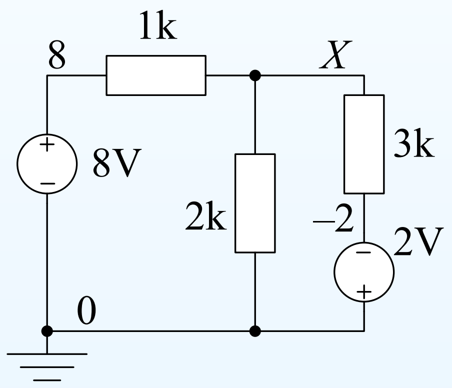
(2) If any fixed voltage sources are connected to a labelled node, label their other ends by adding the value of the source onto the voltage of the labelled end.

(3) Pick an unlabelled node and label it with X, Y, $\ldots$, then go back to step (2) until all nodes are labelled.



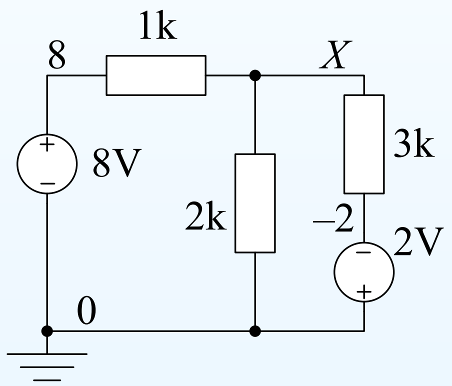
Nodal Analysis Stage 2: KCL Equations

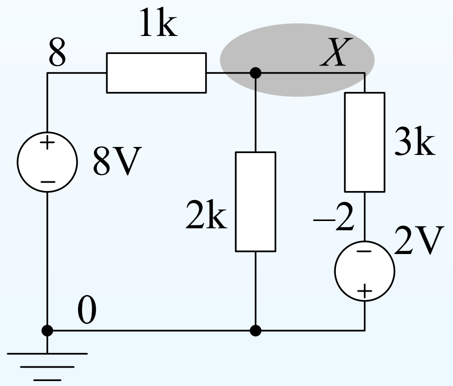
The second step is to write down a KCL equation for each node labelled with a variable by setting the total current flowing out of the node to zero. For a circuit with N nodes and S voltage sources you will have $N - S - 1$ simultaneous equations to solve.



Nodal Analysis Stage 2: KCL Equations

The second step is to write down a KCL equation for each node labelled with a variable by setting the total current flowing out of the node to zero. For a circuit with N nodes and S voltage sources you will have $N - S - 1$ simultaneous equations to solve.



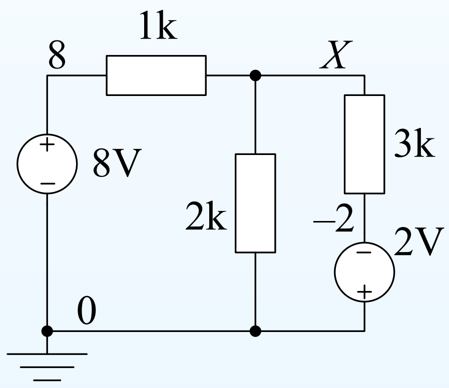


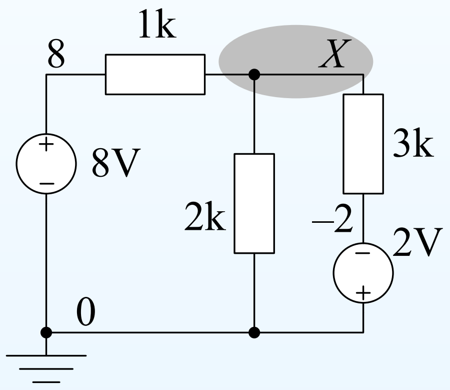
We only have one variable:

$$\frac{X-8}{1\text{ k}} + \frac{X-0}{2\text{ k}} + \frac{X-(-2)}{3\text{ k}} = 0$$

Nodal Analysis Stage 2: KCL Equations

The second step is to write down a KCL equation for each node labelled with a variable by setting the total current flowing out of the node to zero. For a circuit with N nodes and S voltage sources you will have $N - S - 1$ simultaneous equations to solve.





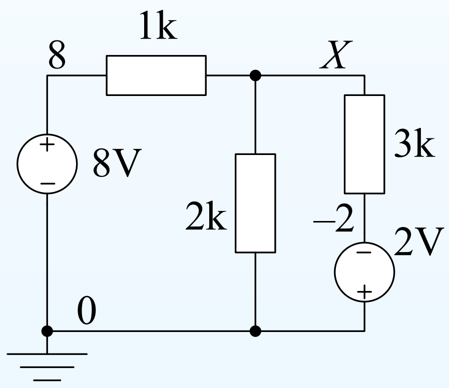
We only have one variable:

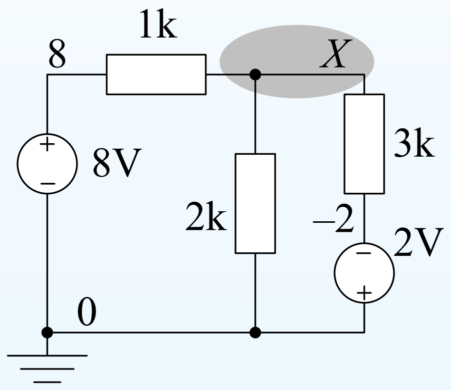
$$\frac{X-8}{1\text{ k}} + \frac{X-0}{2\text{ k}} + \frac{X-(-2)}{3\text{ k}} = 0$$

Numerator for a resistor is always of the form $X - V_N$ where V_N is the voltage on the other side of the resistor.

Nodal Analysis Stage 2: KCL Equations

The second step is to write down a KCL equation for each node labelled with a variable by setting the total current flowing out of the node to zero. For a circuit with N nodes and S voltage sources you will have $N - S - 1$ simultaneous equations to solve.





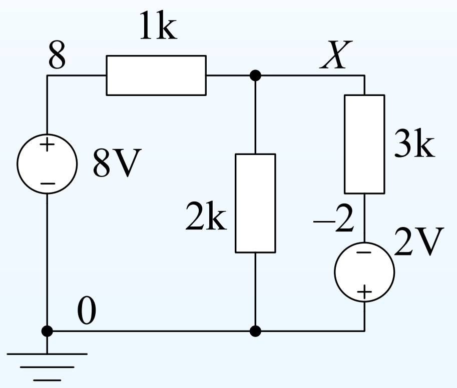
We only have one variable:

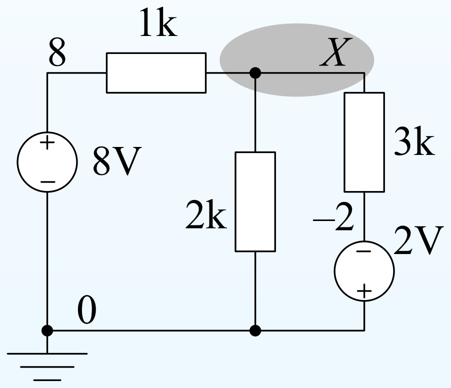
$$\frac{X-8}{1\text{ k}} + \frac{X-0}{2\text{ k}} + \frac{X-(-2)}{3\text{ k}} = 0 \quad \Rightarrow \quad (6X - 48) + 3X + (2X + 4) = 0$$

Numerator for a resistor is always of the form $X - V_N$ where V_N is the voltage on the other side of the resistor.

Nodal Analysis Stage 2: KCL Equations

The second step is to write down a KCL equation for each node labelled with a variable by setting the total current flowing out of the node to zero. For a circuit with N nodes and S voltage sources you will have $N - S - 1$ simultaneous equations to solve.





We only have one variable:

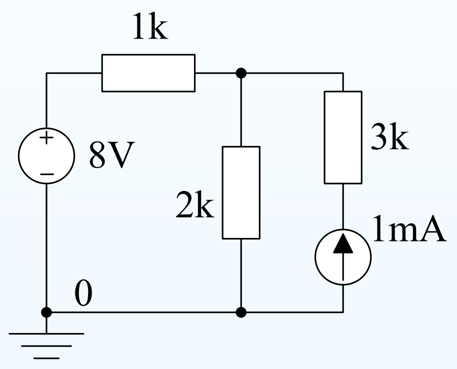
$$\frac{X-8}{1\text{ k}} + \frac{X-0}{2\text{ k}} + \frac{X-(-2)}{3\text{ k}} = 0 \quad \Rightarrow \quad (6X - 48) + 3X + (2X + 4) = 0$$

$$11X = 44 \quad \Rightarrow \quad X = 4$$

Numerator for a resistor is always of the form $X - V_N$ where V_N is the voltage on the other side of the resistor.

Current Sources

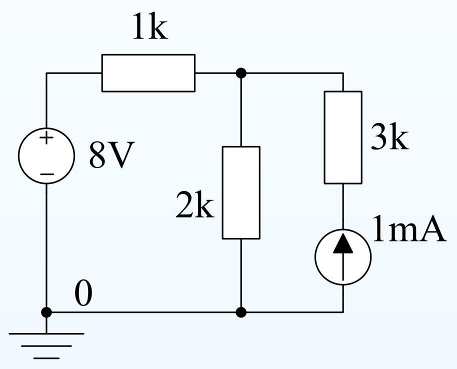
Current sources cause no problems.



Current Sources

Current sources cause no problems.

(1) Pick reference node.

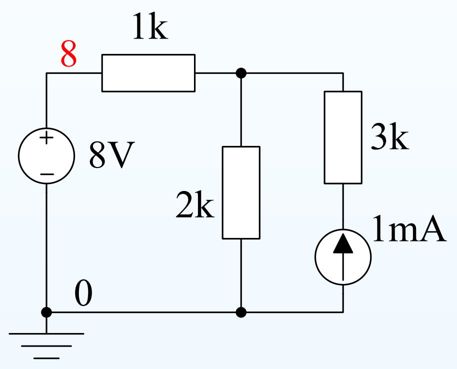


Current Sources

Current sources cause no problems.

(1) Pick reference node.

(2) Label nodes: 8

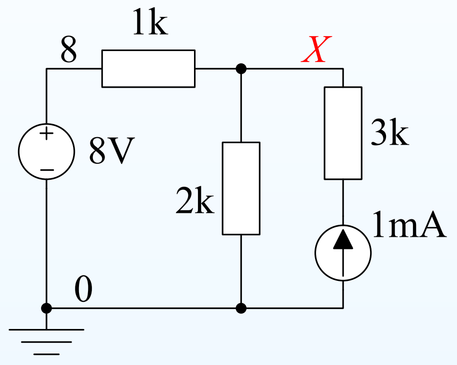


Current Sources

Current sources cause no problems.

(1) Pick reference node.

(2) Label nodes: 8, X

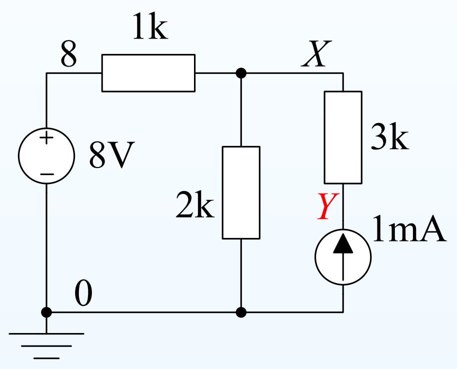


Current Sources

Current sources cause no problems.

(1) Pick reference node.

(2) Label nodes: 8, X and Y.

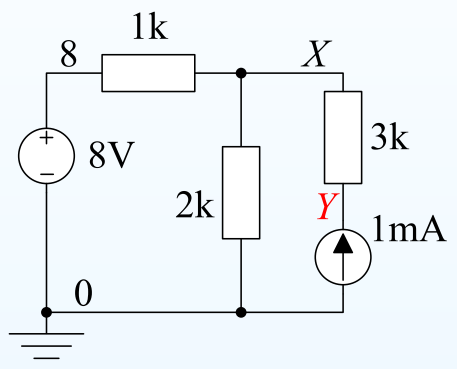


Current Sources

Current sources cause no problems.

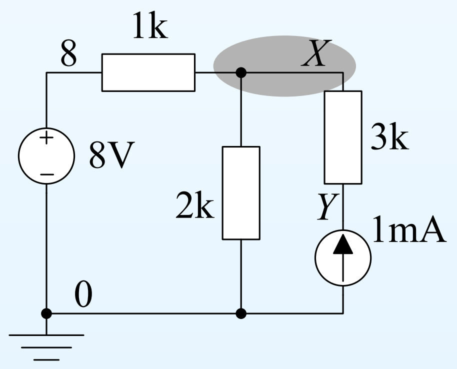
(1) Pick reference node.

(2) Label nodes: 8, X and Y.



(3) Write equations

$$\frac{X-8}{1}+\frac{X}{2}+\frac{X-Y}{3}=0$$

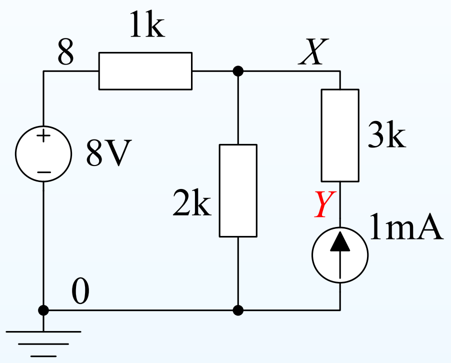


Current Sources

Current sources cause no problems.

(1) Pick reference node.

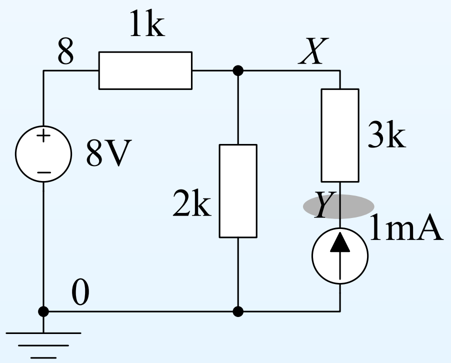
(2) Label nodes: 8, X and Y.



(3) Write equations

$$\frac{X-8}{1}+\frac{X}{2}+\frac{X-Y}{3}=0$$

$$\frac{Y-X}{3}+(-1)=0$$

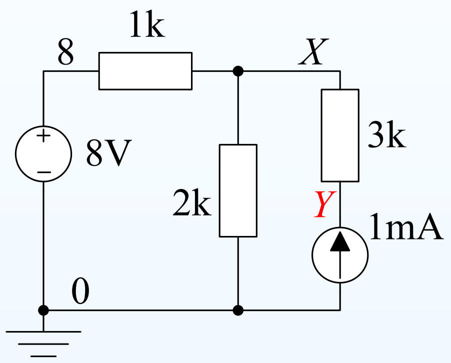


Current Sources

Current sources cause no problems.

(1) Pick reference node.

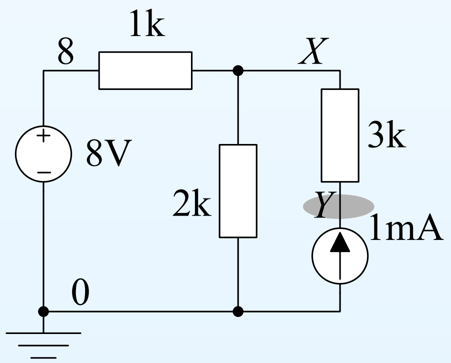
(2) Label nodes: 8, X and Y.



(3) Write equations

$$\frac{X-8}{1}+\frac{X}{2}+\frac{X-Y}{3}=0$$

$$\frac{Y-X}{3}+(-1)=0$$



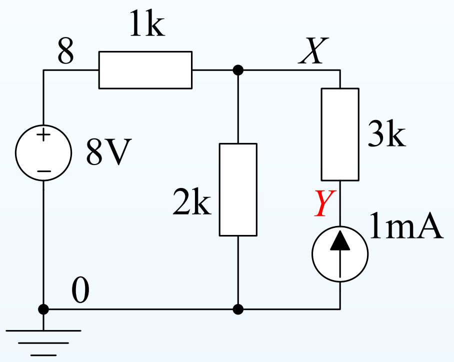
Ohm's law works OK if all resistors are in $\mathrm{k}\Omega$ and all currents in mA.

Current Sources

Current sources cause no problems.

(1) Pick reference node.

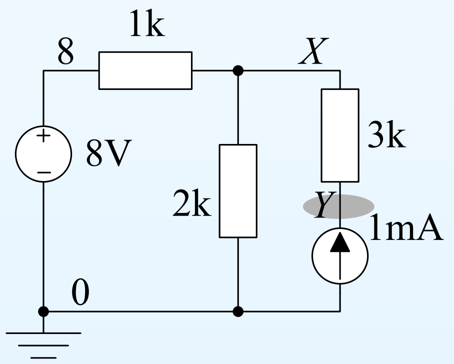
(2) Label nodes: 8, X and Y.



(3) Write equations

$$\frac{X-8}{1}+\frac{X}{2}+\frac{X-Y}{3}=0$$

$$\frac{Y-X}{3}+(-1)=0$$

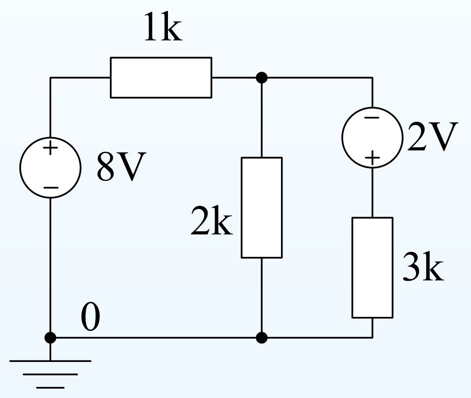


Ohm's law works OK if all resistors are in $\mathrm{k\Omega}$ and all currents in mA.

(4) Solve the equations: $X=6,\ Y=9$

Floating Voltage Sources

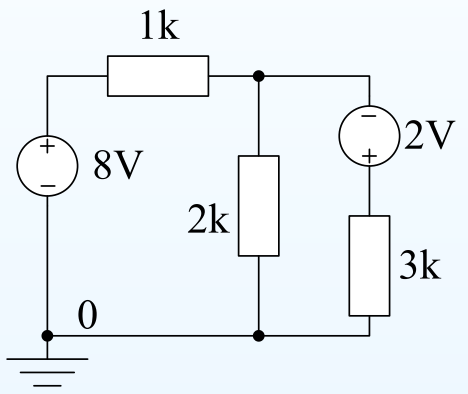
Floating voltage sources have neither end connected to a known fixed voltage. We have to change how we form the KCL equations slightly.



Floating Voltage Sources

Floating voltage sources have neither end connected to a known fixed voltage. We have to change how we form the KCL equations slightly.

(1) Pick reference node.

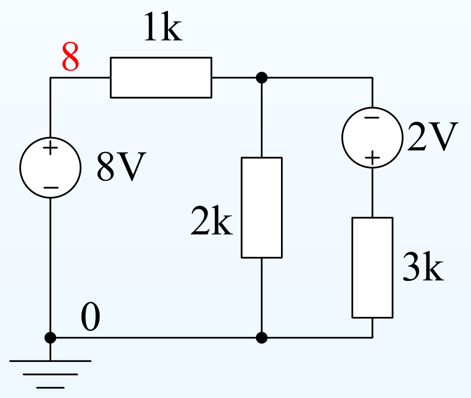


Floating Voltage Sources

Floating voltage sources have neither end connected to a known fixed voltage. We have to change how we form the KCL equations slightly.

(1) Pick reference node.

(2) Label nodes: 8

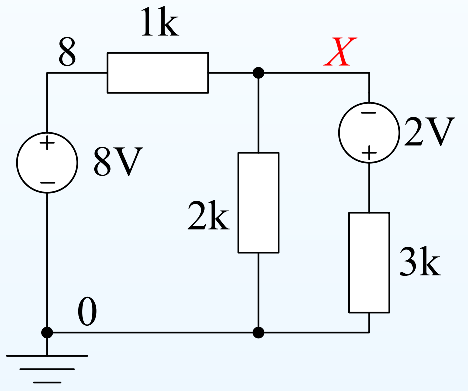


Floating Voltage Sources

Floating voltage sources have neither end connected to a known fixed voltage. We have to change how we form the KCL equations slightly.

(1) Pick reference node.

(2) Label nodes: 8, X

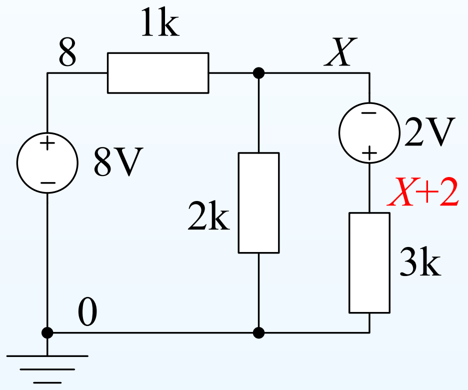


Floating Voltage Sources

Floating voltage sources have neither end connected to a known fixed voltage. We have to change how we form the KCL equations slightly.

(1) Pick reference node.

(2) Label nodes: 8, X and $X+2$ since it is joined to X via a voltage source.



Floating Voltage Sources

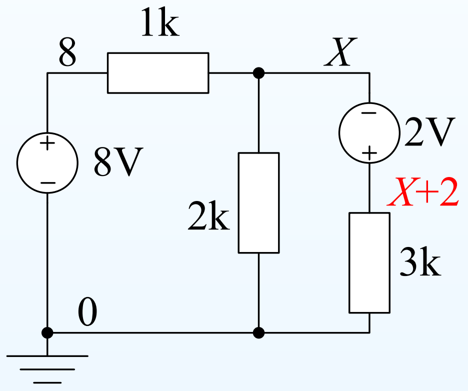
Floating voltage sources have neither end connected to a known fixed voltage. We have to change how we form the KCL equations slightly.

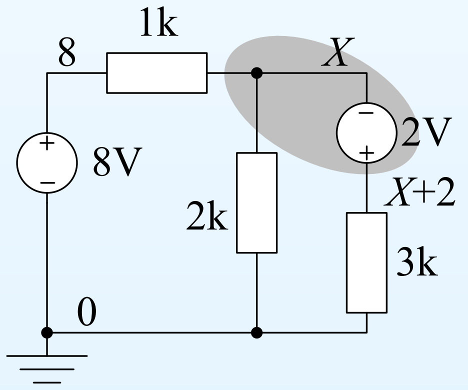
(1) Pick reference node.

(2) Label nodes: 8, X and $X + 2$ since it is joined to X via a voltage source.

(3) Write KCL equations but count all the nodes connected via floating voltage sources as a single "super-node" giving one equation

$$\frac{X-8}{1} + \frac{X}{2} + \frac{(X+2)-0}{3} = 0$$





Floating Voltage Sources

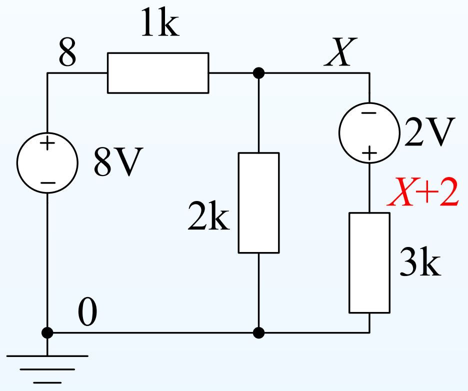
Floating voltage sources have neither end connected to a known fixed voltage. We have to change how we form the KCL equations slightly.

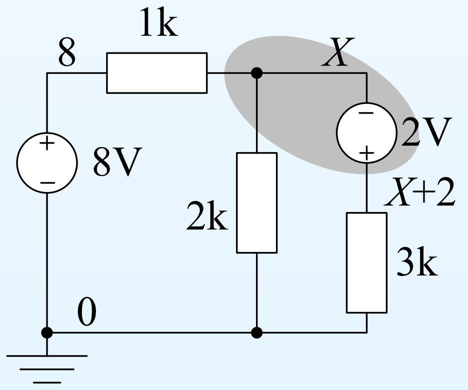
(1) Pick reference node.

(2) Label nodes: 8, X and $X+2$ since it is joined to X via a voltage source.

(3) Write KCL equations but count all the nodes connected via floating voltage sources as a single “super-node” giving one equation

$$\frac{X-8}{1}+\frac{X}{2}+\frac{(X+2)-0}{3}=0$$





Ohm’s law always involves the difference between the voltages at either end of a resistor. (Obvious but easily forgotten)

Floating Voltage Sources

Floating voltage sources have neither end connected to a known fixed voltage. We have to change how we form the KCL equations slightly.

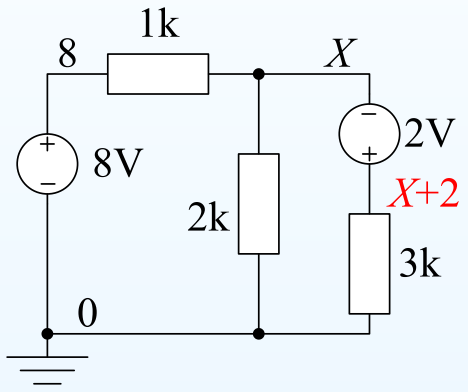
(1) Pick reference node.

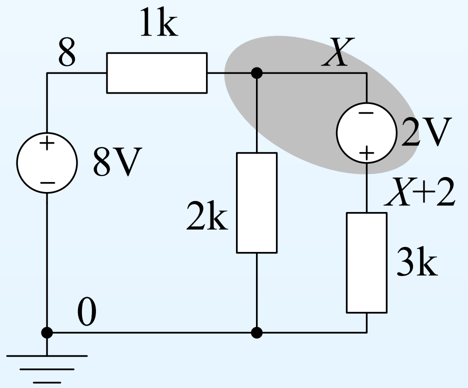
(2) Label nodes: 8, X and $X+2$ since it is joined to X via a voltage source.

(3) Write KCL equations but count all the nodes connected via floating voltage sources as a single "super-node" giving one equation

$$\frac{X-8}{1}+\frac{X}{2}+\frac{(X+2)-0}{3}=0$$

(4) Solve the equations: $X=4$

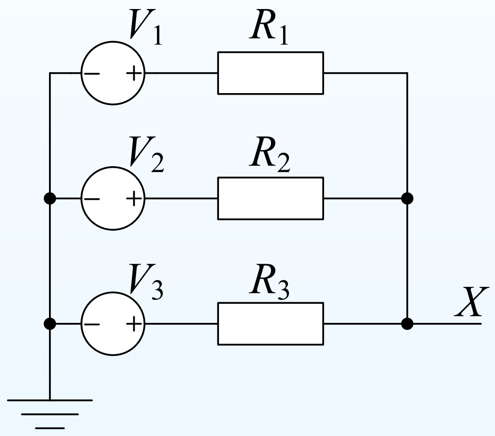




Ohm's law always involves the difference between the voltages at either end of a resistor. (Obvious but easily forgotten)

Weighted Average Circuit

A very useful sub-circuit that calculates the weighted average of any number of voltages.

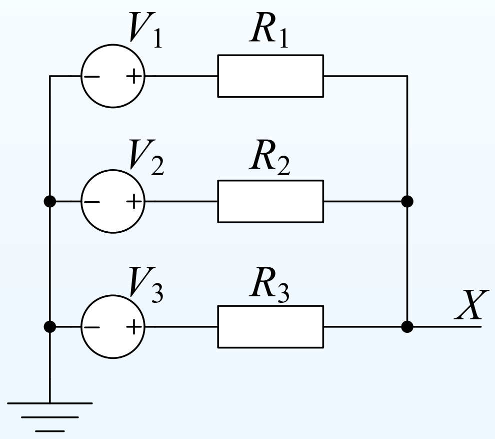


Weighted Average Circuit

A very useful sub-circuit that calculates the weighted average of any number of voltages.

KCL equation for node X:

$$\frac{X-V_1}{R_1} + \frac{X-V_2}{R_2} + \frac{X-V_3}{R_3} = 0$$

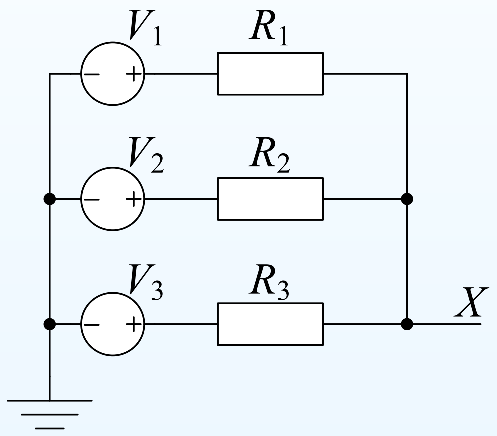


Weighted Average Circuit

A very useful sub-circuit that calculates the weighted average of any number of voltages.

KCL equation for node X:

$$\frac{X-V_1}{R_1}+\frac{X-V_2}{R_2}+\frac{X-V_3}{R_3}=0$$



Or using conductances:

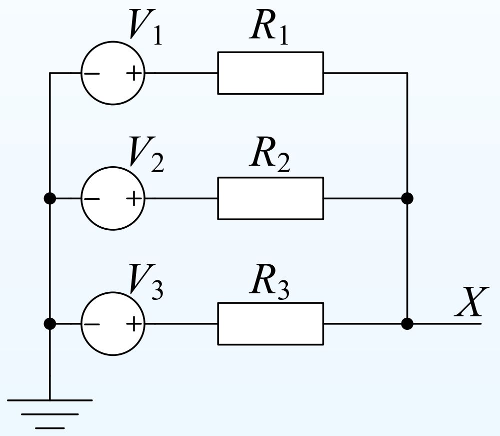
$$(X-V_1)G_1+(X-V_2)G_2+(X-V_3)G_3=0$$

Weighted Average Circuit

A very useful sub-circuit that calculates the weighted average of any number of voltages.

KCL equation for node X:

$$\frac{X-V_1}{R_1}+\frac{X-V_2}{R_2}+\frac{X-V_3}{R_3}=0$$



Or using conductances:

$$(X-V_1)G_1+(X-V_2)G_2+(X-V_3)G_3=0$$

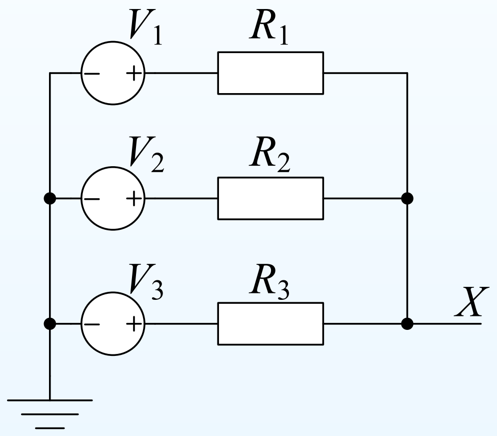
$$X(G_1+G_2+G_3)=V_1G_1+V_2G_2+V_3G_3$$

Weighted Average Circuit

A very useful sub-circuit that calculates the weighted average of any number of voltages.

KCL equation for node X:

$$\frac{X-V_1}{R_1}+\frac{X-V_2}{R_2}+\frac{X-V_3}{R_3}=0$$



Or using conductances:

$$(X-V_1)G_1+(X-V_2)G_2+(X-V_3)G_3=0$$

$$X(G_1+G_2+G_3)=V_1G_1+V_2G_2+V_3G_3$$

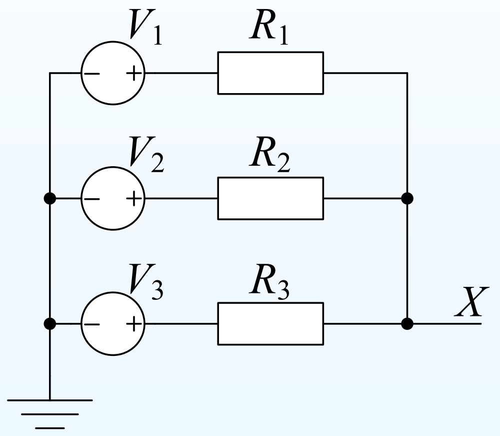
$$X=\frac{V_1G_1+V_2G_2+V_3G_3}{G_1+G_2+G_3}=\frac{\sum V_iG_i}{\sum G_i}$$

Weighted Average Circuit

A very useful sub-circuit that calculates the weighted average of any number of voltages.

KCL equation for node X:

$$\frac{X-V_1}{R_1}+\frac{X-V_2}{R_2}+\frac{X-V_3}{R_3}=0$$



Or using conductances:

$$(X-V_1)G_1+(X-V_2)G_2+(X-V_3)G_3=0$$

$$X(G_1+G_2+G_3)=V_1G_1+V_2G_2+V_3G_3$$

$$X=\frac{V_1G_1+V_2G_2+V_3G_3}{G_1+G_2+G_3}=\frac{\sum V_iG_i}{\sum G_i}$$

Voltage X is the average of $V_1,\ V_2,\ V_3$ weighted by the conductances.

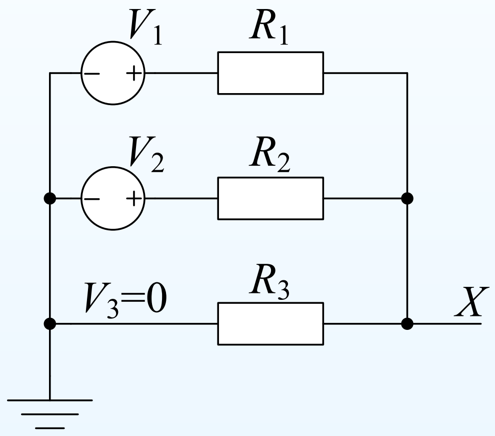
Weighted Average Circuit

A very useful sub-circuit that calculates the weighted average of any number of voltages.

KCL equation for node X:

$$\frac{X-V_1}{R_1} + \frac{X-V_2}{R_2} + \frac{X-V_3}{R_3} = 0$$

Still works if $V_3 = 0$.



Or using conductances:

$$(X - V_1)G_1 + (X - V_2)G_2 + (X - V_3)G_3 = 0$$

$$X(G_1 + G_2 + G_3) = V_1G_1 + V_2G_2 + V_3G_3$$

$$X = \frac{V_1G_1+V_2G_2+V_3G_3}{G_1+G_2+G_3} = \frac{\sum V_iG_i}{\sum G_i}$$

Voltage X is the average of V_1, V_2, V_3 weighted by the conductances.

Digital-to-Analog Converter

A 3-bit binary number, b, has bit-weights of 4, 2 and 1. Thus 110 has a value 6 in decimal.

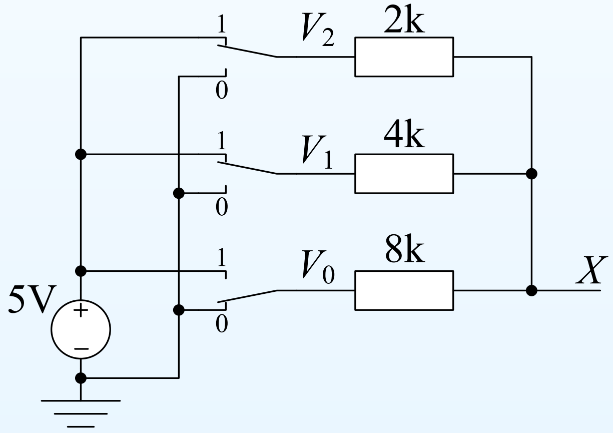
Digital-to-Analog Converter

A 3-bit binary number, b, has bit-weights of 4, 2 and 1. Thus 110 has a value 6 in decimal. If we label the bits $b_2b_1b_0$, then $b = 4b_2 + 2b_1 + b_0$.

Digital-to-Analog Converter

A 3-bit binary number, b, has bit-weights of 4, 2 and 1. Thus 110 has a value 6 in decimal. If we label the bits $b_2b_1b_0$, then $b = 4b_2 + 2b_1 + b_0$.

We use $b_2b_1b_0$ to control the switches which determine whether $V_i = 5\text{ V}$ or $V_i = 0\text{ V}$. Thus $V_i = 5b_i$. Switches shown for $b = 6$.

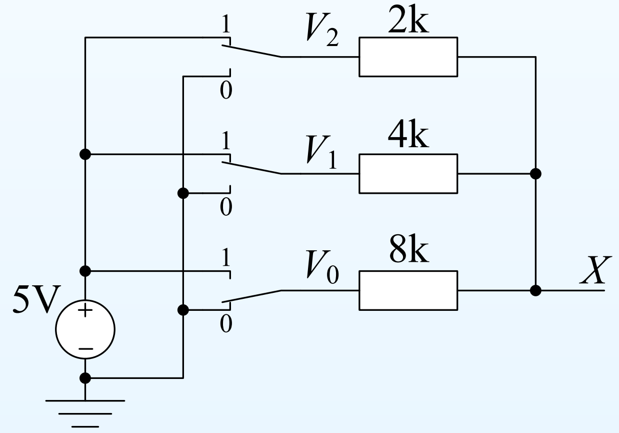


Digital-to-Analog Converter

A 3-bit binary number, b, has bit-weights of 4, 2 and 1. Thus 110 has a value 6 in decimal. If we label the bits $b_2 b_1 b_0$, then $b = 4b_2 + 2b_1 + b_0$.

We use $b_2 b_1 b_0$ to control the switches which determine whether $V_i = 5\text{ V}$ or $V_i = 0\text{ V}$. Thus $V_i = 5b_i$. Switches shown for $b = 6$.

$$X = \frac{\frac{1}{2}V_2 + \frac{1}{4}V_1 + \frac{1}{8}V_0}{\frac{1}{2} + \frac{1}{4} + \frac{1}{8}}$$



$$G_2 = \frac{1}{R_2} = \frac{1}{2\,k} = \frac{1}{2}\text{mS}, \ldots$$

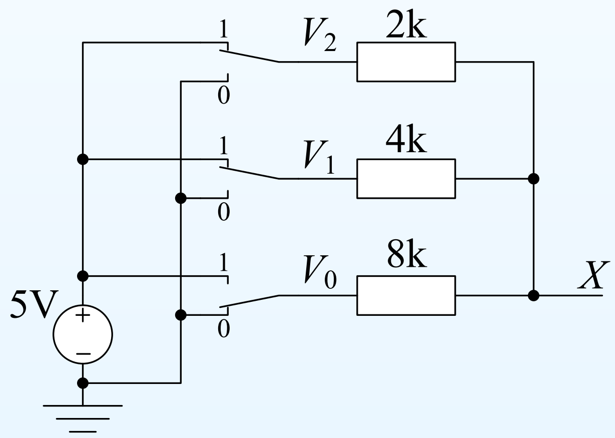
Digital-to-Analog Converter

A 3-bit binary number, b, has bit-weights of 4, 2 and 1. Thus 110 has a value 6 in decimal. If we label the bits $b_2b_1b_0$, then $b = 4b_2 + 2b_1 + b_0$.

We use $b_2b_1b_0$ to control the switches which determine whether $V_i = 5\text{ V}$ or $V_i = 0\text{ V}$. Thus $V_i = 5b_i$. Switches shown for $b = 6$.

$$X = \frac{\frac{1}{2}V_2 + \frac{1}{4}V_1 + \frac{1}{8}V_0}{\frac{1}{2} + \frac{1}{4} + \frac{1}{8}}$$

$$= \frac{1}{7}\left(4V_2 + 2V_1 + V_0\right)$$



$$G_2 = \frac{1}{R_2} = \frac{1}{2\,k} = \frac{1}{2}\,\text{mS}, \ldots$$

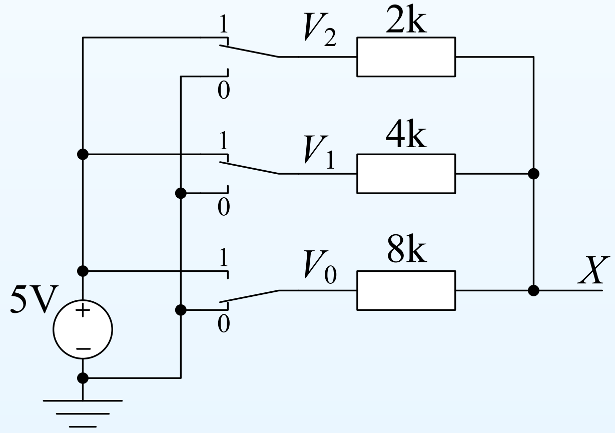
Digital-to-Analog Converter

A 3-bit binary number, b, has bit-weights of 4, 2 and 1. Thus 110 has a value 6 in decimal. If we label the bits $b_2b_1b_0$, then $b = 4b_2 + 2b_1 + b_0$.

We use $b_2b_1b_0$ to control the switches which determine whether $V_i = 5\text{ V}$ or $V_i = 0\text{ V}$. Thus $V_i = 5b_i$. Switches shown for $b = 6$.

$$X = \frac{\frac{1}{2}V_2+\frac{1}{4}V_1+\frac{1}{8}V_0}{\frac{1}{2}+\frac{1}{4}+\frac{1}{8}}$$

$$= \frac{1}{7}\left(4V_2 + 2V_1 + V_0\right)$$



but $V_i = 5 \times b_i$ since it connects to either 0 V or 5 V

$$G_2 = \frac{1}{R_2} = \frac{1}{2\,k} = \frac{1}{2}\text{mS}, \ldots$$

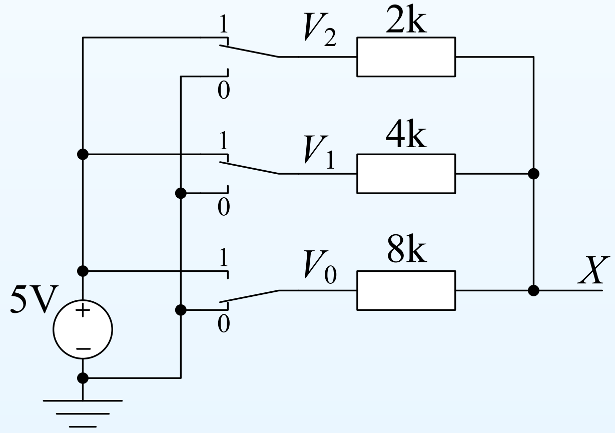
Digital-to-Analog Converter

A 3-bit binary number, b, has bit-weights of 4, 2 and 1. Thus 110 has a value 6 in decimal. If we label the bits $b_2b_1b_0$, then $b = 4b_2 + 2b_1 + b_0$.

We use $b_2b_1b_0$ to control the switches which determine whether $V_i = 5\text{ V}$ or $V_i = 0\text{ V}$. Thus $V_i = 5b_i$. Switches shown for $b = 6$.

$$X = \frac{\frac{1}{2}V_2+\frac{1}{4}V_1+\frac{1}{8}V_0}{\frac{1}{2}+\frac{1}{4}+\frac{1}{8}}$$

$$= \frac{1}{7}\left(4V_2 + 2V_1 + V_0\right)$$



but $V_i = 5 \times b_i$ since it connects to either 0 V or 5 V

$$= \frac{5}{7}\left(4b_2 + 2b_1 + b_0\right) = \frac{5}{7}b$$

$$G_2 = \frac{1}{R_2} = \frac{1}{2\,k} = \frac{1}{2}\text{mS}, \ldots$$

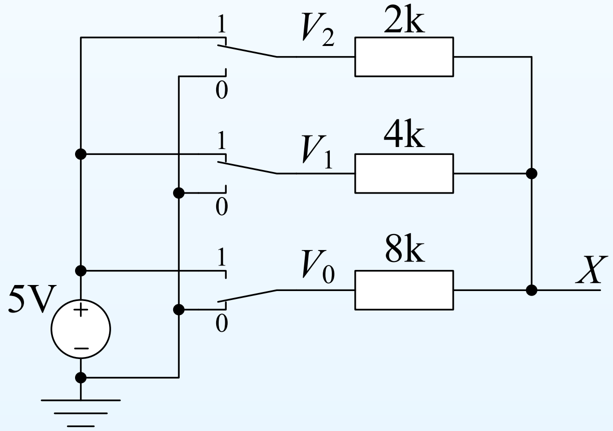
Digital-to-Analog Converter

A 3-bit binary number, b, has bit-weights of 4, 2 and 1. Thus 110 has a value 6 in decimal. If we label the bits $b_2b_1b_0$, then $b = 4b_2 + 2b_1 + b_0$.

We use $b_2b_1b_0$ to control the switches which determine whether $V_i = 5\text{ V}$ or $V_i = 0\text{ V}$. Thus $V_i = 5b_i$. Switches shown for $b = 6$.

$$X = \frac{\frac{1}{2}V_2+\frac{1}{4}V_1+\frac{1}{8}V_0}{\frac{1}{2}+\frac{1}{4}+\frac{1}{8}}$$

$$= \frac{1}{7}\left(4V_2 + 2V_1 + V_0\right)$$



but $V_i = 5 \times b_i$ since it connects to either 0 V or 5 V

$$= \frac{5}{7}\left(4b_2 + 2b_1 + b_0\right) = \frac{5}{7}b$$

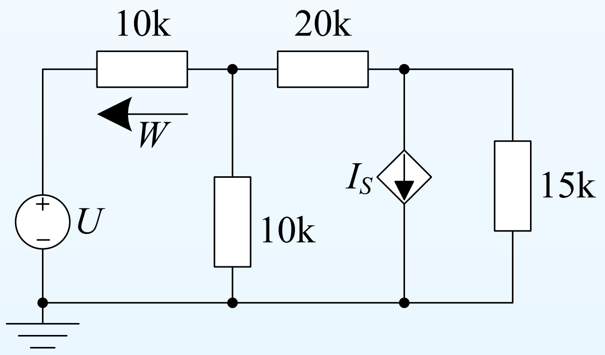
$$G_2 = \frac{1}{R_2} = \frac{1}{2\,k} = \frac{1}{2}\text{mS}, \ldots$$

So we have made a circuit in which X is proportional to a binary number b.

Dependent Sources

A *dependent* voltage or current source is one whose value is determined by voltages or currents elsewhere in the circuit. These are most commonly used when modelling the behaviour of transistors or op-amps. Each dependent source has a defining equation.

In this circuit: $I_S = 0.2W$ mA where W is in volts.

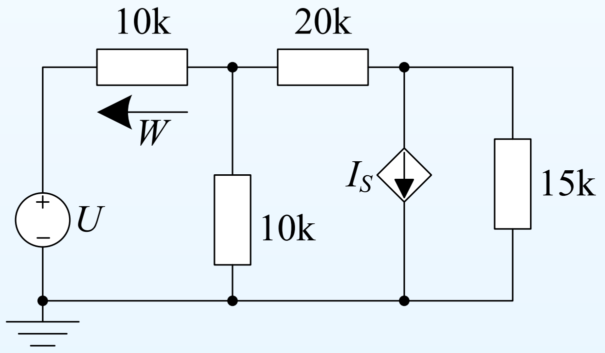


Dependent Sources

A *dependent* voltage or current source is one whose value is determined by voltages or currents elsewhere in the circuit. These are most commonly used when modelling the behaviour of transistors or op-amps. Each dependent source has a defining equation.

In this circuit: $I_S = 0.2W$ mA where W is in volts.

(1) Pick reference node.



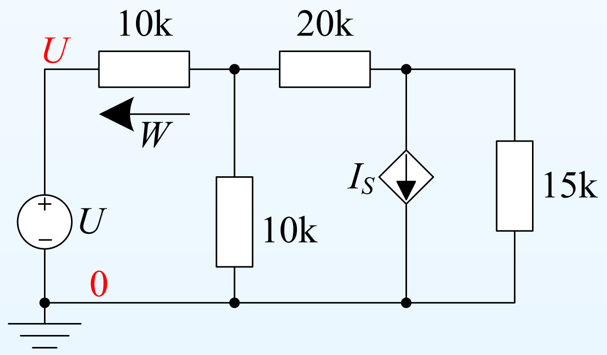
Dependent Sources

A *dependent* voltage or current source is one whose value is determined by voltages or currents elsewhere in the circuit. These are most commonly used when modelling the behaviour of transistors or op-amps. Each dependent source has a defining equation.

In this circuit: $I_S = 0.2W$ mA where W is in volts.

(1) Pick reference node.

(2) Label nodes: $0,\ U$



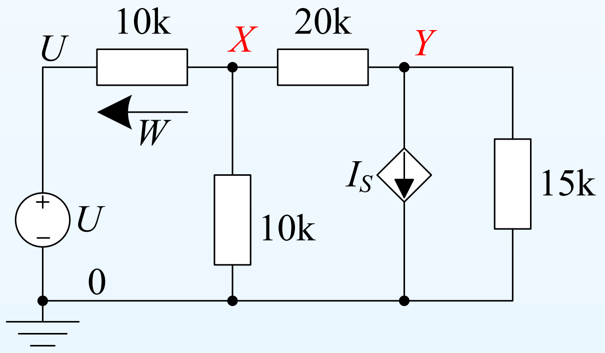
Dependent Sources

A *dependent* voltage or current source is one whose value is determined by voltages or currents elsewhere in the circuit. These are most commonly used when modelling the behaviour of transistors or op-amps. Each dependent source has a defining equation.

In this circuit: $I_S = 0.2W$ mA where W is in volts.

(1) Pick reference node.

(2) Label nodes: 0, U, X and Y .



Dependent Sources

A *dependent* voltage or current source is one whose value is determined by voltages or currents elsewhere in the circuit. These are most commonly used when modelling the behaviour of transistors or op-amps. Each dependent source has a defining equation.

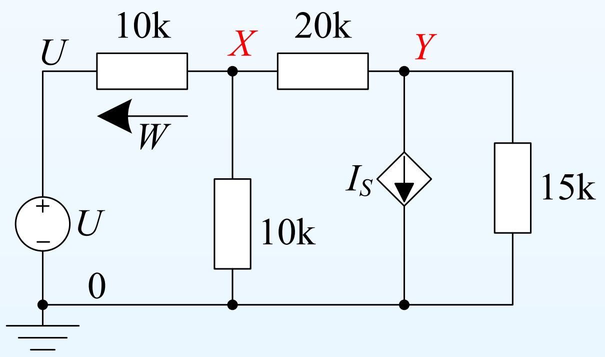
In this circuit: $I_S = 0.2W$ mA where W is in volts.

(1) Pick reference node.

(2) Label nodes: 0, U, X and Y .

(3) Write equation for the dependent source, I_S, in terms of node voltages:

$$I_S = 0.2\,(U - X)$$

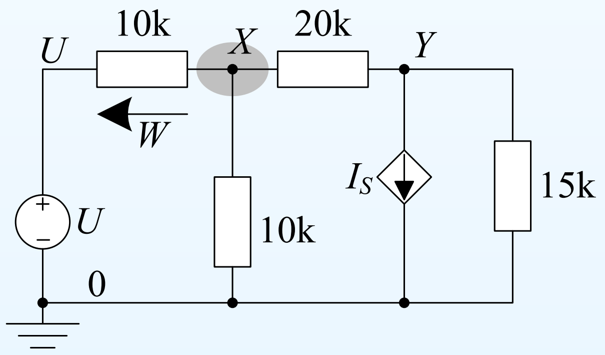


Dependent Sources

A *dependent* voltage or current source is one whose value is determined by voltages or currents elsewhere in the circuit. These are most commonly used when modelling the behaviour of transistors or op-amps. Each dependent source has a defining equation.

In this circuit: $I_S = 0.2W$ mA where W is in volts.

(1) Pick reference node.



(2) Label nodes: 0, U, X and Y .

(3) Write equation for the dependent source, I_S, in terms of node voltages:

$$I_S = 0.2\,(U - X)$$

(4) Write KCL equations:

$$\frac{X-U}{10} + \frac{X}{10} + \frac{X-Y}{20} = 0$$

Dependent Sources

A *dependent* voltage or current source is one whose value is determined by voltages or currents elsewhere in the circuit. These are most commonly used when modelling the behaviour of transistors or op-amps. Each dependent source has a defining equation.

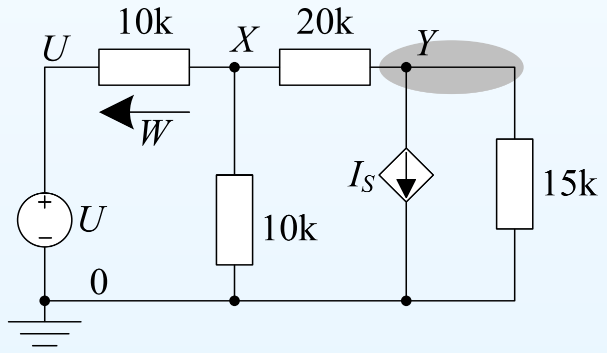
In this circuit: $I_S = 0.2W$ mA where W is in volts.

(1) Pick reference node.

(2) Label nodes: 0, U, X and Y.

(3) Write equation for the dependent source, I_S, in terms of node voltages:

$$I_S = 0.2\,(U - X)$$



(4) Write KCL equations:

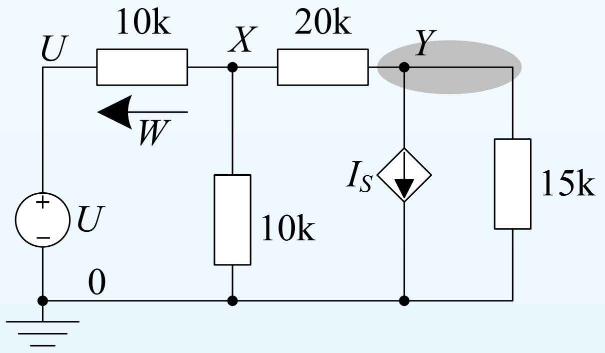
$$\frac{X-U}{10} + \frac{X}{10} + \frac{X-Y}{20} = 0 \qquad \frac{Y-X}{20} + I_S + \frac{Y}{15} = 0$$

Dependent Sources

A *dependent* voltage or current source is one whose value is determined by voltages or currents elsewhere in the circuit. These are most commonly used when modelling the behaviour of transistors or op-amps. Each dependent source has a defining equation.

In this circuit: $I_S = 0.2W$ mA where W is in volts.

(1) Pick reference node.



(2) Label nodes: 0, U, X and Y .

(3) Write equation for the dependent source, I_S, in terms of node voltages:

$$I_S = 0.2\,(U - X)$$

(4) Write KCL equations:

$$\frac{X-U}{10} + \frac{X}{10} + \frac{X-Y}{20} = 0 \qquad\qquad \frac{Y-X}{20} + I_S + \frac{Y}{15} = 0$$

(5) Solve all three equations to find X, Y and I_S in terms of U:

Dependent Sources

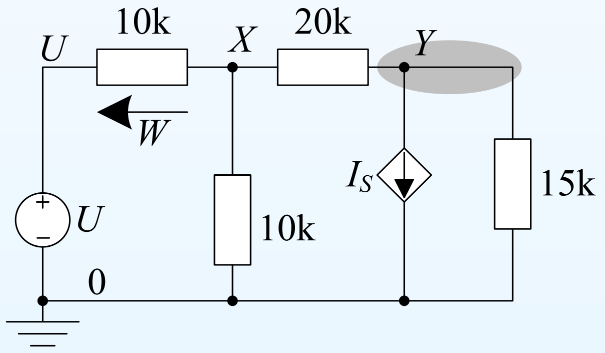
A *dependent* voltage or current source is one whose value is determined by voltages or currents elsewhere in the circuit. These are most commonly used when modelling the behaviour of transistors or op-amps. Each dependent source has a defining equation.

In this circuit: $I_S = 0.2W$ mA where W is in volts.

(1) Pick reference node.

(2) Label nodes: 0, U, X and Y .

(3) Write equation for the dependent source, I_S, in terms of node voltages:
$$I_S = 0.2\left(U - X\right)$$



(4) Write KCL equations:

$$\frac{X-U}{10} + \frac{X}{10} + \frac{X-Y}{20} = 0 \qquad \frac{Y-X}{20} + I_S + \frac{Y}{15} = 0$$

(5) Solve all three equations to find X, Y and I_S in terms of U:
$$X = 0.1U,\ Y = -1.5U,\ I_S = 0.18U$$

Dependent Sources

A *dependent* voltage or current source is one whose value is determined by voltages or currents elsewhere in the circuit. These are most commonly used when modelling the behaviour of transistors or op-amps. Each dependent source has a defining equation.

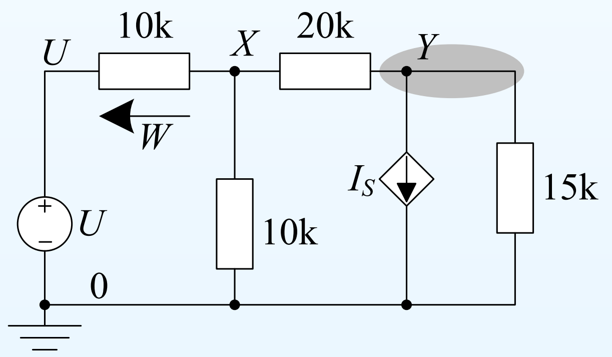
In this circuit: $I_S = 0.2W$ mA where W is in volts.

(1) Pick reference node.

(2) Label nodes: 0, U, X and Y.

(3) Write equation for the dependent source, I_S, in terms of node voltages:

$$I_S = 0.2\,(U - X)$$



(4) Write KCL equations:

$$\frac{X-U}{10} + \frac{X}{10} + \frac{X-Y}{20} = 0 \qquad \frac{Y-X}{20} + I_S + \frac{Y}{15} = 0$$

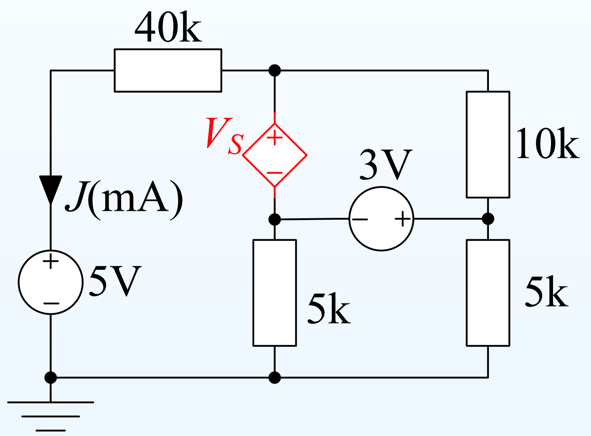
(5) Solve all three equations to find X, Y and I_S in terms of U:

$$X = 0.1U,\ Y = -1.5U,\ I_S = 0.18U$$

Note that the value of U is assumed to be known.

Dependent Voltage Sources

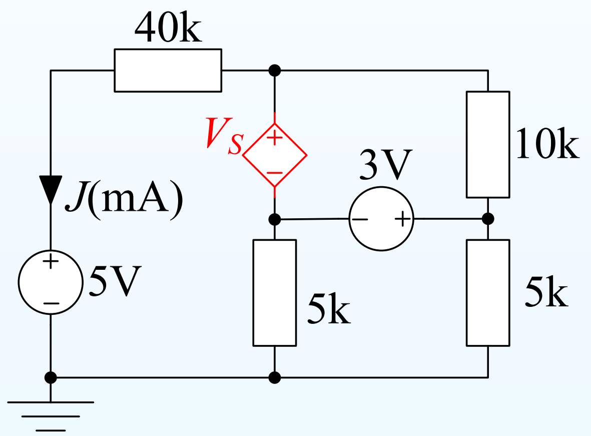
The value of the highlighted dependent voltage source is $V_S = 10J$ Volts where J is the indicated current in mA.



Dependent Voltage Sources

The value of the highlighted dependent voltage source is $V_S = 10J$ Volts where J is the indicated current in mA.

(1) Pick reference node.

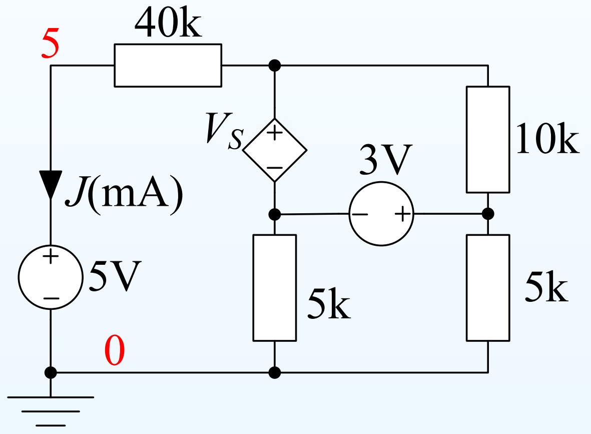


Dependent Voltage Sources

The value of the highlighted dependent voltage source is $V_S = 10J$ Volts where J is the indicated current in mA.

(1) Pick reference node.

(2) Label nodes: $0,\ 5$

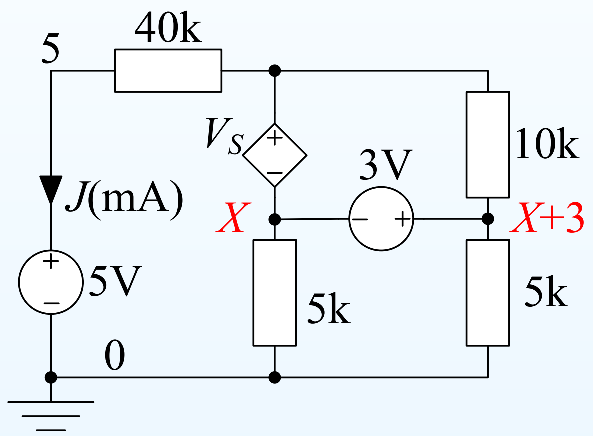


Dependent Voltage Sources

The value of the highlighted dependent voltage source is $V_S = 10J$ Volts where J is the indicated current in mA.

(1) Pick reference node.

(2) Label nodes: $0,\ 5, X,\ X+3$

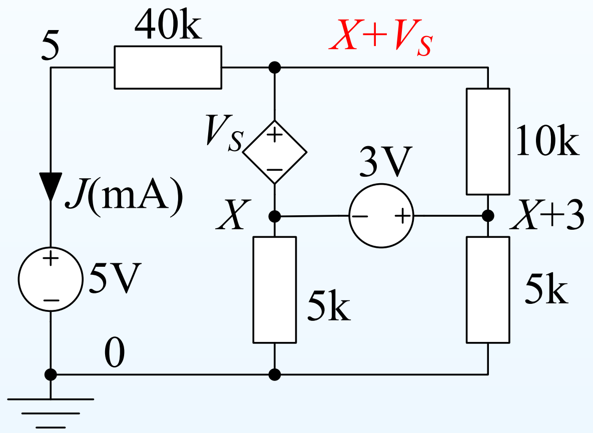


Dependent Voltage Sources

The value of the highlighted dependent voltage source is $V_S = 10J$ Volts where J is the indicated current in mA.

(1) Pick reference node.

(2) Label nodes: $0,\ 5$, $X,\ X+3$ and $X+V_S$.



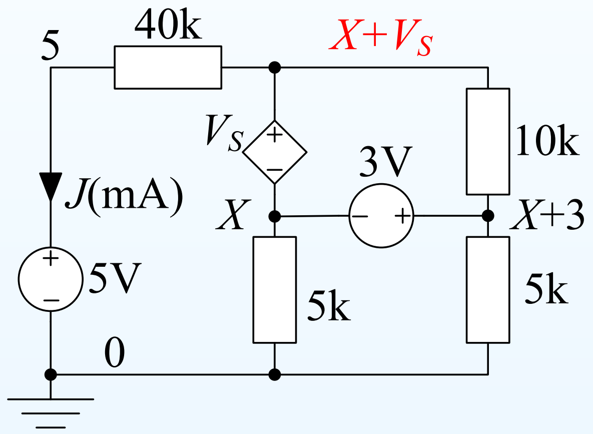
Dependent Voltage Sources

The value of the highlighted dependent voltage source is $V_S = 10J$ Volts where J is the indicated current in mA.

(1) Pick reference node.

(2) Label nodes: $0,\ 5, X,\ X+3$ and $X+V_S$.

(3) Write equation for the dependent source, V_S, in terms of node voltages:



$$V_S = 10J = 10 \times \frac{X+V_S-5}{40} \Rightarrow 3V_S = X-5$$

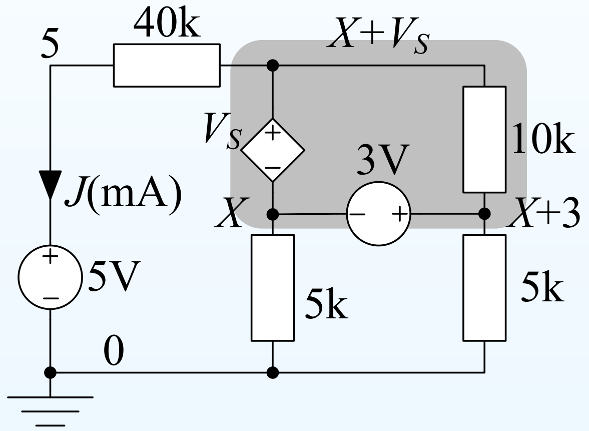
Dependent Voltage Sources

The value of the highlighted dependent voltage source is $V_S = 10J$ Volts where J is the indicated current in mA.

(1) Pick reference node.

(2) Label nodes: $0,\ 5,\ X,\ X+3$ and $X+V_S$.

(3) Write equation for the dependent source, V_S, in terms of node voltages:



$$V_S = 10J = 10 \times \frac{X+V_S-5}{40} \Rightarrow 3V_S = X-5$$

(4) Write KCL equations: all nodes connected by floating voltage sources and all components connecting these nodes are in the same “super-node”

$$\frac{X+V_S-5}{40} + \frac{X}{5} + \frac{X+3}{5} = 0$$

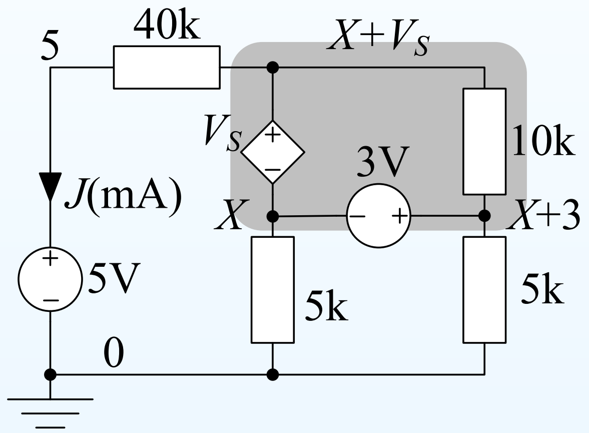
Dependent Voltage Sources

The value of the highlighted dependent voltage source is $V_S = 10J$ Volts where J is the indicated current in mA.

(1) Pick reference node.

(2) Label nodes: $0,\ 5,\ X,\ X+3$ and $X+V_S$.

(3) Write equation for the dependent source, V_S, in terms of node voltages:



$$V_S = 10J = 10 \times \frac{X+V_S-5}{40} \Rightarrow 3V_S = X-5$$

(4) Write KCL equations: all nodes connected by floating voltage sources and all components connecting these nodes are in the same "super-node"

$$\frac{X+V_S-5}{40} + \frac{X}{5} + \frac{X+3}{5} = 0$$

(5) Solve the two equations: $X = -1$ and $V_S = -2$

Universal Nodal Analysis Algorithm

(1) Pick any node as the voltage reference. Label its voltage as 0 V. Label any dependent sources with $V_S,\ I_S,\ \ldots$.

(2) If any voltage sources are connected to a labelled node, label their other ends by adding the value of the source onto the voltage of the labelled end. Repeat as many times as possible.

(3) Pick an unlabelled node and label it with $X,\ Y,\ \ldots$, then loop back to step (2) until all nodes are labelled.

(4) For each **dependent source**, write down an equation that expresses its value in terms of other node voltages.

(5) Write down a KCL equation for each “normal” node (i.e. one that is not connected to a floating voltage source).

(6) Write down a KCL equation for each “super-node”. A super-node consists of a set of nodes that are joined by floating voltage sources and includes any other components joining these nodes.

(7) Solve the set of simultaneous equations that you have written down.

Universal Nodal Analysis Algorithm

(1) Pick any node as the voltage reference. Label its voltage as 0 V. Label any dependent sources with $V_S,\ I_S,\ \ldots$.

(2) If any voltage sources are connected to a labelled node, label their other ends by adding the value of the source onto the voltage of the labelled end. Repeat as many times as possible.

(3) Pick an unlabelled node and label it with $X,\ Y,\ \ldots$, then loop back to step (2) until all nodes are labelled.

(4) For each **dependent source**, write down an equation that expresses its value in terms of other node voltages.

(5) Write down a KCL equation for each “normal” node (i.e. one that is not connected to a floating voltage source).

(6) Write down a KCL equation for each “super-node”. A super-node consists of a set of nodes that are joined by floating voltage sources and includes any other components joining these nodes.

(7) Solve the set of simultaneous equations that you have written down.

Universal Nodal Analysis Algorithm

(1) Pick any node as the voltage reference. Label its voltage as 0 V. Label any dependent sources with $V_S,\ I_S,\ \ldots$.

(2) If any voltage sources are connected to a labelled node, label their other ends by adding the value of the source onto the voltage of the labelled end. Repeat as many times as possible.

(3) Pick an unlabelled node and label it with $X,\ Y,\ \ldots$, then loop back to step (2) until all nodes are labelled.

(4) For each **dependent source**, write down an equation that expresses its value in terms of other node voltages.

(5) Write down a KCL equation for each “normal” node (i.e. one that is not connected to a floating voltage source).

(6) Write down a KCL equation for each “super-node”. A super-node consists of a set of nodes that are joined by floating voltage sources and includes any other components joining these nodes.

(7) Solve the set of simultaneous equations that you have written down.

Universal Nodal Analysis Algorithm

(1) Pick any node as the voltage reference. Label its voltage as 0 V. Label any dependent sources with $V_S,\ I_S,\ \ldots$.

(2) If any voltage sources are connected to a labelled node, label their other ends by adding the value of the source onto the voltage of the labelled end. Repeat as many times as possible.

(3) Pick an unlabelled node and label it with $X,\ Y,\ \ldots$, then loop back to step (2) until all nodes are labelled.

(4) For each **dependent source**, write down an equation that expresses its value in terms of other node voltages.

(5) Write down a KCL equation for each “normal” node (i.e. one that is not connected to a floating voltage source).

(6) Write down a KCL equation for each “super-node”. A super-node consists of a set of nodes that are joined by floating voltage sources and includes any other components joining these nodes.

(7) Solve the set of simultaneous equations that you have written down.

Universal Nodal Analysis Algorithm

(1) Pick any node as the voltage reference. Label its voltage as 0 V. Label any dependent sources with $V_S,\ I_S,\ \ldots$.

(2) If any voltage sources are connected to a labelled node, label their other ends by adding the value of the source onto the voltage of the labelled end. Repeat as many times as possible.

(3) Pick an unlabelled node and label it with $X,\ Y,\ \ldots$, then loop back to step (2) until all nodes are labelled.

(4) For each **dependent source**, write down an equation that expresses its value in terms of other node voltages.

(5) Write down a KCL equation for each “normal” node (i.e. one that is not connected to a floating voltage source).

(6) Write down a KCL equation for each “super-node”. A super-node consists of a set of nodes that are joined by floating voltage sources and includes any other components joining these nodes.

(7) Solve the set of simultaneous equations that you have written down.

Universal Nodal Analysis Algorithm

(1) Pick any node as the voltage reference. Label its voltage as 0 V. Label any dependent sources with $V_S,\ I_S,\ \ldots$.

(2) If any voltage sources are connected to a labelled node, label their other ends by adding the value of the source onto the voltage of the labelled end. Repeat as many times as possible.

(3) Pick an unlabelled node and label it with $X,\ Y,\ \ldots$, then loop back to step (2) until all nodes are labelled.

(4) For each **dependent source**, write down an equation that expresses its value in terms of other node voltages.

(5) Write down a KCL equation for each “normal” node (i.e. one that is not connected to a floating voltage source).

(6) Write down a KCL equation for each “super-node”. A super-node consists of a set of nodes that are joined by floating voltage sources and includes any other components joining these nodes.

(7) Solve the set of simultaneous equations that you have written down.

Universal Nodal Analysis Algorithm

(1) Pick any node as the voltage reference. Label its voltage as 0 V. Label any dependent sources with $V_S,\ I_S,\ \ldots$.

(2) If any voltage sources are connected to a labelled node, label their other ends by adding the value of the source onto the voltage of the labelled end. Repeat as many times as possible.

(3) Pick an unlabelled node and label it with $X,\ Y,\ \ldots$, then loop back to step (2) until all nodes are labelled.

(4) For each **dependent source**, write down an equation that expresses its value in terms of other node voltages.

(5) Write down a KCL equation for each “normal” node (i.e. one that is not connected to a floating voltage source).

(6) Write down a KCL equation for each “super-node”. A super-node consists of a set of nodes that are joined by floating voltage sources and includes any other components joining these nodes.

(7) Solve the set of simultaneous equations that you have written down.

Summary

- Nodal Analysis
  - Simple Circuits (no floating or dependent voltage sources)

Summary

- Nodal Analysis
  - Simple Circuits (no floating or dependent voltage sources)
  - Floating Voltage Sources
    - use supernodes: all the nodes connected by floating voltage sources (independent or dependent)

Summary

- Nodal Analysis
  - Simple Circuits (no floating or dependent voltage sources)
  - Floating Voltage Sources
    - use supernodes: all the nodes connected by floating voltage sources (independent or dependent)
  - Dependent Voltage and Current Sources
    - Label each source with a variable
    - Write extra equations expressing the source values in terms of node voltages
    - Write down the KCL equations as before

Summary

- Nodal Analysis
  - Simple Circuits (no floating or dependent voltage sources)
  - Floating Voltage Sources
    - use supernodes: all the nodes connected by floating voltage sources (independent or dependent)
  - Dependent Voltage and Current Sources
    - Label each source with a variable
    - Write extra equations expressing the source values in terms of node voltages
    - Write down the KCL equations as before
- Mesh Analysis (in most textbooks)
  - Alternative to nodal analysis but doesn't work for all circuits
  - No significant benefits $\Rightarrow$ ignore it

Summary

- Nodal Analysis
  - Simple Circuits (no floating or dependent voltage sources)
  - Floating Voltage Sources
    - use supernodes: all the nodes connected by floating voltage sources (independent or dependent)
  - Dependent Voltage and Current Sources
    - Label each source with a variable
    - Write extra equations expressing the source values in terms of node voltages
    - Write down the KCL equations as before
- Mesh Analysis (in most textbooks)
  - Alternative to nodal analysis but doesn't work for all circuits
  - No significant benefits $\Rightarrow$ ignore it

For further details see Hayt et al. Chapter 4.

Electricity flows in two ways: either in an alternating current (AC) or in a direct current (DC). Electricity or "current" is nothing but the movement of electrons through a conductor, like a wire. The difference between AC and DC lies in the direction in which the electrons flow. In DC, the electrons flow steadily in a single direction, or "forward." In AC, electrons keep switching directions, sometimes going "forward" and then going "backward."

Alternating current is the best way to transmit electricity over large distances.

Comparison chart

	Alternating Current	Direct Current
Amount of energy that can be carried	Safe to transfer over longer city distances and can provide more power.	Voltage of DC cannot travel very far until it begins to lose energy.
Cause of the direction of flow of electrons	Rotating magnet along the wire.	Steady magnetism along the wire.
Frequency	The frequency of alternating current is 50Hz or 60Hz depending upon the country.	The frequency of direct current is zero.
Direction	It reverses its direction while flowing in a circuit.	It flows in one direction in the circuit.
Current	It is the current of magnitude varying with time	It is the current of constant magnitude.
Flow of Electrons	Electrons keep switching directions - forward and backward.	Electrons move steadily in one direction or 'forward'.
Obtained from	A.C Generator and mains.	Cell or Battery.
Passive Parameters	Impedance.	Resistance only
Power Factor	Lies between 0 & 1.	It is always 1.
Types	Sinusoidal, Trapezoidal, Triangular, Square.	Pure and pulsating.

1.1 Electric Circuit

An electric circuit is a closed conducting path through which an electric current either flows or is intended to flow. The basic electric circuit consists of
a) Source of energy
b) Two conductors connecting the source and the load to transfer the energy.

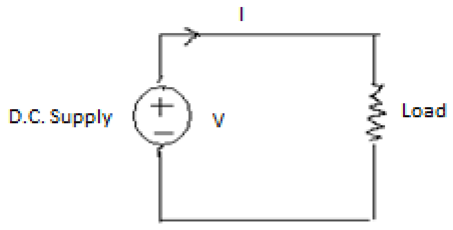


Fig.1.1 Electric circuit with resistive load

Network:

Component of circuit elements are resistor, inductor and capacitor, voltage source and current source. An interconnection of circuit elements is called a network.

Linear Circuit:
The parameters of linear element remains constant i.e the parameters do not change with current or voltage applied to the element(i.e R,L,C).The linear element shows linear characteristics of voltage vs. current (for constant temperature and frequency).

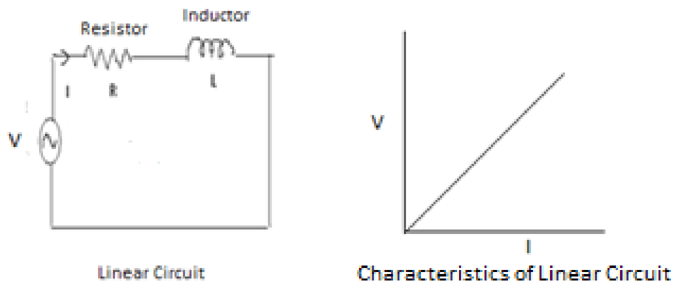
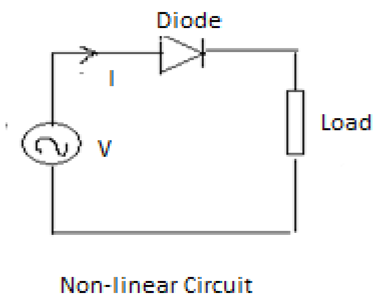


Fig. 1.2

Non Linear Circuit:

In a non linear electric circuit is an electrical element which does not have a linear relationship between current and voltage. Examples are diode, transistors and semiconductor devices. The current I through a diode is a non-linear function.



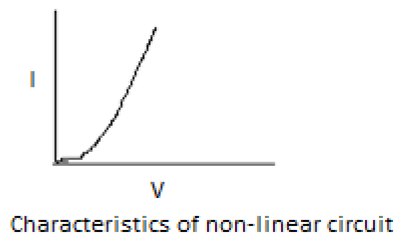


Fig. 1.3

Bilateral circuit:

Bilateral circuit is one whose properties and characteristics are same in either direction. For example, a resistor , if it is connected right to left or left to right whose properties and characteristics are same.

Unilateral Circuit:

Unilateral circuit is the circuit where properties and characteristics change with the direction of operation (direction current). Diode rectifier is the best example of unilateral circuit.

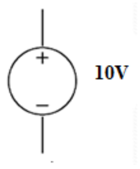
1.2 Electrical Sources

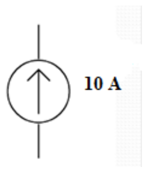
There are two types of electrical sources.

- Independent Sources and
- Dependent sources.

Independent Sources:

The strength of voltage or current is not changed by any variation in the connected network the source is said to be either independent voltage or independent current source. In this, the value of voltage or current is fixed and is not adjustable.





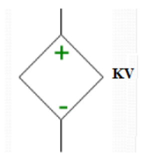
a) Independent Voltage Source Symbol b) Independent Current Source Symbol

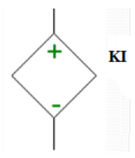
Fig.1.4

Dependent Sources:

The output voltage or current of a dependent source is determined by one of the parameters associated with another component in the circuit.

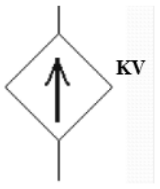
Dependent Sources are classified as:

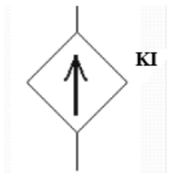




i) Voltage controlled voltage source

ii) Current controlled voltage source





iii) Voltage controlled current source

iv) Current controlled current source

Fig.1.5

Node: A node is the point of connection between two or more branches. A node is a point where two or more circuit elements meet.

Branch: A branch represents a single element such as a voltage source or a resistor.

Active Element: The active elements generate energy. Batteries, generators, operational amplifiers etc are active elements.

Passive Element: A passive **element** is an electrical component that does not generate power, but instead dissipates, stores, and/or releases it. Passive **elements** include resistances, capacitors and inductors.

Loop: A loop is a closed path in a circuit where two nodes are not traversed twice except the initial point.

Mesh: A mesh is a closed path in a circuit with no other paths inside it, in other words, a loop with no other loops inside it.

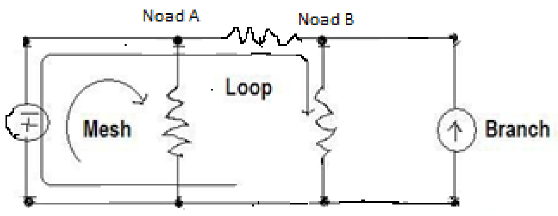


Fig.1.6

In the above fig.1.6, we find that the circuit has ***2 nodes*** and ***3 meshes (independent loops)***

1.3 Kirchoff's Laws

There are some simple relationships between currents and voltages of different branches of an electrical circuit.

Kirchhoff's Current Law (KCL):

If we consider all the currents enter in the junction are considered as positive sign, then convention of all the branch currents leaving the junction are negative.

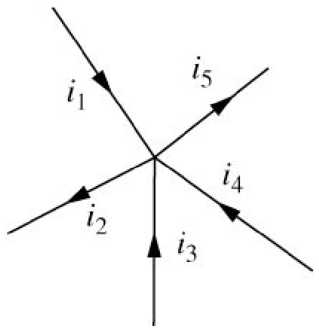


Fig.1.7

Mathematically we can write, $i_1 + (-i_2) + i_3 + i_4 + (-i_5) = 0$

Kirchhoff's Voltage Law (KVL):

This law deals with the voltage drops at various branches.

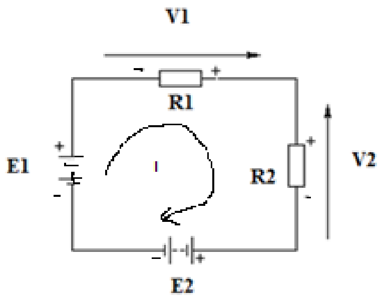


Fig.1.8

Mathematically we can write, E1+ V1-V2-E2 =0

Sign of Battery

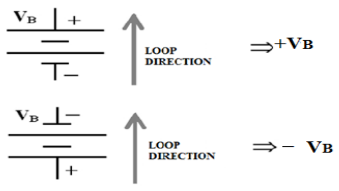


Fig.1.9

For a battery, the polarity is usually indicated on the battery with "+" or "-" near one of the terminals.

Sign of Resistor Voltage Drop Polarity

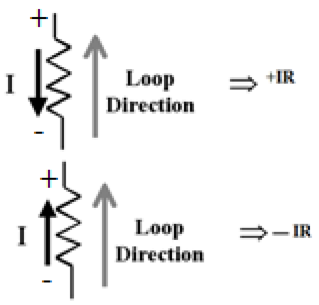


Fig.1.10

The direction of current flow through a resistor determines the polarity of resistors

1.4 Source Equivalence and Conversion

Source transformation is simplifying a circuit solution by transforming a voltage into a current source, and vice versa.

Conversion of voltage source to current source

Convert the constant voltage source shown in figure 1.11 to constant current source.

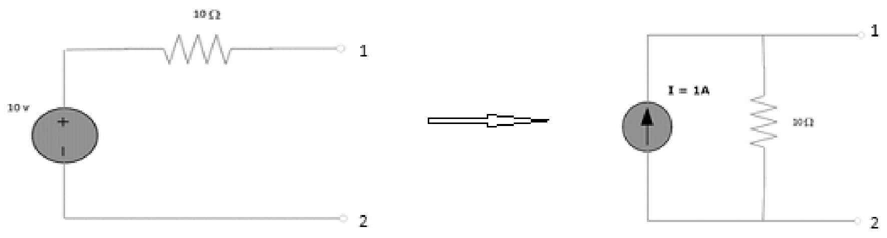


Fig.1.11

Conversion of current source to voltage source

Convert the constant current source shown in figure 1.12 to voltage source.

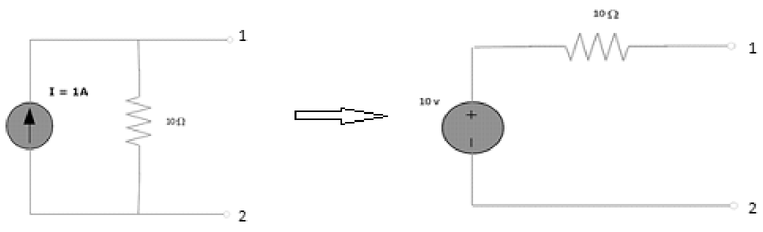


Fig.1.12

We have to do same inverse procedure.

Norton's Theorem

Statement:

Procedure to solve any network using Norton's Theorem:

Step-I:

To find I_{SC}:

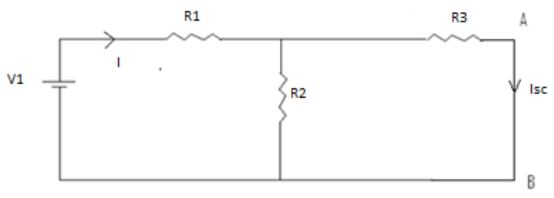


Fig.1.18

Calculation:

$$I = \frac{V1}{R_{eq}} = \frac{V1}{R_1 + \frac{R_2 R_3}{R_2 + R_3}}$$

$$I_{SC} = I \, X \, \frac{R_2}{R_2 + R_3}$$

Step-II: To find R_N:

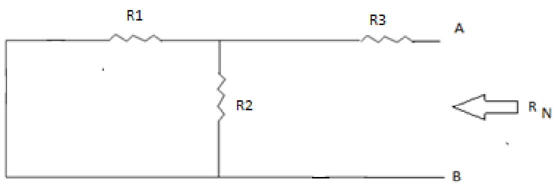


Fig.1.19

Calculation:

$$R_N = \frac{R_1 R_2}{R_1 + R_2} + R_3$$

Norton's Equivalent Circuit:

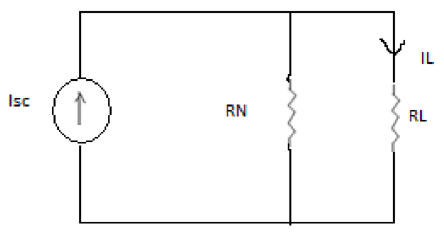


Fig.1.20

Calculation:

$$I_L = Isc\ X\ \frac{R_N}{R_N+R_L}$$

Maximum Power Transfer Theorem:

This theorem is used to value of load resistance for which maximum power will be transfer from source to load.

Statements of Maximum Power Transfer Theorem:

A resistive load abstracts maximum power from a network when the load resistance equals to the internal resistance of the network as viewed back to the network from output terminals, with all energy sources removed, leaving behind their internal resistances.

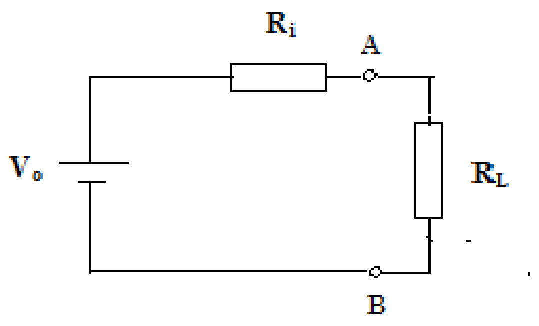


Fig.1.25

Proof of Maximum Power Transfer Theorem:

A variable resistance R_L is connected to a dc source network. The aim is to determine the value of R_L such that it receives maximum power from the dc source.

$I_L = V_0/ (R_i + R_L)$ ………(i)

The power delivered to the resistive load is given by

$P_L = I_{L2} R_L = [V_0/ (R_i + R_L)]^2 R_L$ ………..(ii) [substituting IL from equation (i)]

For PL to be maximum,

$dP_L / dR_L = 0$

$R_i = R_L$

Hence, it has been prove that the power transfer from a dc source network to a resistive load is maximum when the internal resistance of the dc source network is equal to the load resistance.

$P_{max} = = V_0^2 / 4 R_{th}$

The maximum power transfer theorem defines the condition under which the maximum power is transferred to the load in a circuit.

Star – Delta Conversion:

A **Star** connected network which has the symbol of the letter, Y (wye) and a **Delta** connected network which has the symbol of a triangle, Δ (delta).

Star Delta Transformations allow us to convert impedances connected together in a 3-phase configuration from one type of connection to another. These circuit transformations allow us to change the three connected resistances (or impedances) by their equivalents measured between the terminals 1-2, 1-3 or 2-3 for either a star or delta connected circuit

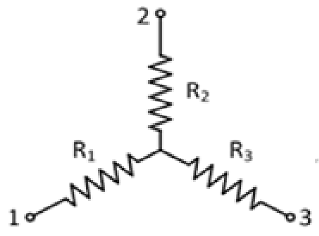


Fig.1.21 Star Connection

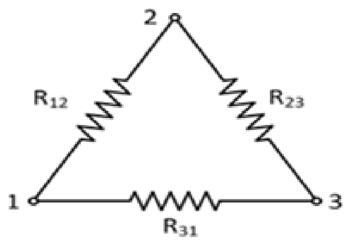


Fig.1.22 Delta Connection

From fig. (1) and fig. (2)

Resistance between terminal 1 and 2 for Star network = resistance between terminal 1 and 2 for Delta network

$R_1 + R_2 = R_{12} \parallel (R_{23} + R_{31})$

$= R_{12} (R_{23} + R_{31}) / (R_{12} + R_{23} + R_{31})$ ....(i)

Similarly ,

$R_2 + R_3 = R_{23} (R_{31} + R_{12}) / (R_{12} + R_{23} + R_{31})$ ....(ii)

$R_3 + R_1 = R_{31} (R_{12} + R_{23}) / (R_{12} + R_{23} + R_{31})$ ....(iii)

Equations for the transformation from Δ to Y :

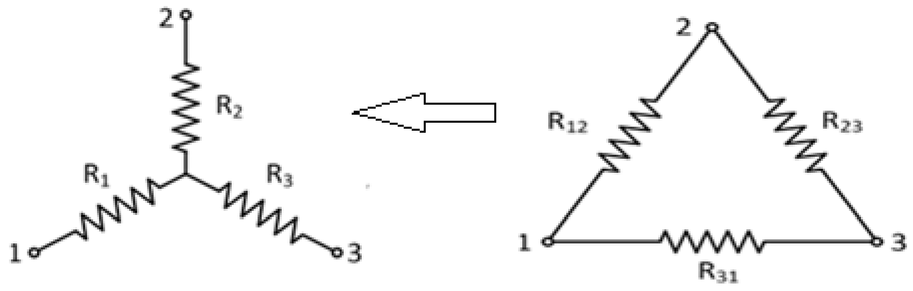


Fig.1.23

Adding the equation (i) , (ii) and (iii)

$(R_1 + R_2 + R_3) = (R_{12} R_{23} + R_{23} R_{31} + R_{31} R_{12}) / (R_{12} + R_{23} + R_{31})$ ……….(iv)

Subtraction of equation (ii) from equation (iv)

$R_1 = R_{12} R_{31} / (R_{12} + R_{23} + R_{31})$ ……………….. (v)

Similarly ,

$R_2 = R_{12} R_{23} / (R_{12} + R_{23} + R_{31})$ ………………….(vi)

$R_3 = R_{23} R_{31} / (R_{12} + R_{23} + R_{31})$ …………………..(vii)

Equations for the transformation from Y to Δ :

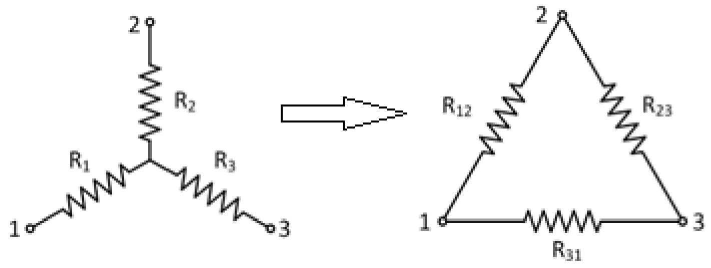


Fig.1.24

From the equation (v) , (vi) and (vii)

We gate,

$R_1 R_2 + R_2 R_3 + R_3 R_1 = R_{12} R_{23} R_{31} / (R_{12} + R_{23} + R_{31})$ ……………………….(viii)

Division of equation (viii) by equation (viii) ,

$R_{12} = (R_1 R_2 + R_2 R_3 + R_3 R_1) / R_3$

Similarly ,

$R_{23} = (R_1 R_2 + R_2 R_3 + R_3 R_1) / R_1$

$R_{31} = (R_1 R_2 + R_2 R_3 + R_3 R_1) / R_2$

3.1 Concept of magnetic circuit:

Magnetic field

Magnetic fields can be created due to the permanent magnet and current passes through in a solenoid. Magnetic field is represented by lines of force for static electric field. This static electric field produced by dc current or permanent magnet.

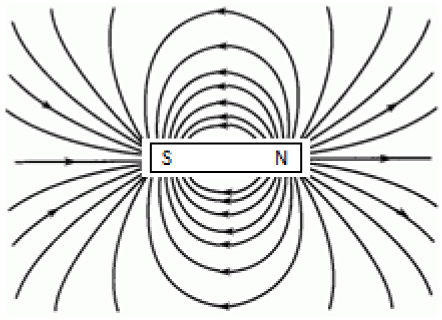


Fig.3.1

A **magnetic circuit** is created by one or more closed loop paths containing a **magnetic** flux. The magnetic flux is analogous to the electric current.
Magnetic flux = Φ

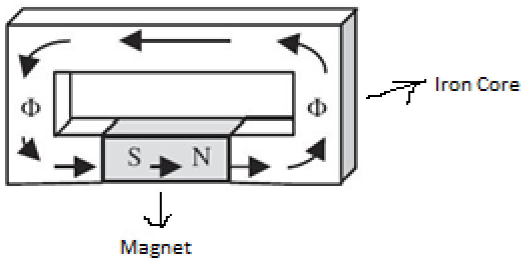


Fig.3.2

3.2 B-H curve

Magnetic Hysteresis Curves commonly is known as **B-H Curves**. Magnetic hysteresis is an important phenomenon of the magnetization and demagnetization process of material.
This curve is commonly formed by ac supply on the material.
The magnetic flux density = B = Φ/ A, B = µ H, where H is called magnetic field intensity and M is called magnetization and µ is the permeability of the medium.

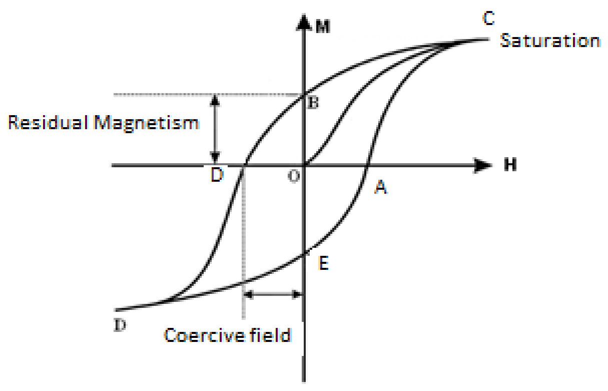


Fig.3.3 (B-H curve)

3.3 Analogous quantities in magnetic and electric circuits

Differences:

Electric field	**Magnetic field**
1.Magnetic field is produced by a moving charge.	1.Electric field is produced by a charge whether at rest or in motion
2.The total electric flux through any closed surface is equal to the net charge enclosed by the surface.	2.The total magnetic flux through any closed surface is always zero,(**Φ =0)**
1. Electric field lines are discontinuous. They have start from point (+ charge) and an ending point (- charge).	3. Magnetic field lines are continuous, they always make closed loops.

Similarities:

Both are attractive as well as repulsive (Like poles repel, like charges repel; unlike poles attract, unlike charges attract)

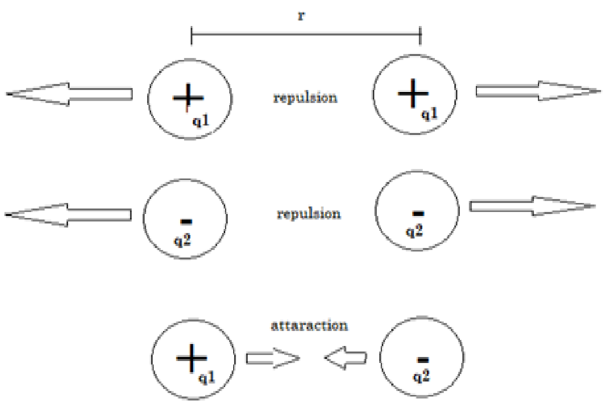


Fig.3. 4 force between like and unlike charges

3.4 Iron losses

Hysteresis loss and eddy current loss consists of iron loss or core loss.

Hysteresis loss in transformer is , $W_h = K_h f (B_m)^{1.6} \quad watts$

Eddy current loss in transformer is , $W_e = K_e f^2 K_f^2 B_m^2 \; watts$

Where, K_h = Hysteresis constant. K_e = Eddy current constant. K_f = form constant, B_m = Maximum flux density

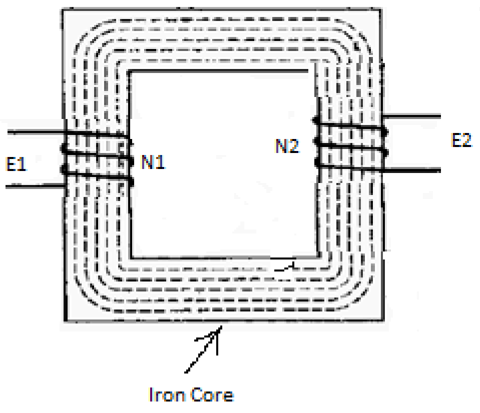


Fig.3.5

In this above figure (Fig.5) E_1 ,E_2 , N_1 and N_2 are the primary induced emf, secondary induced emf, primary number of turn and secondary number of turn respectively.
Eddy currents are currents induced in conductors to oppose the change in flux that generated them. It is caused when a conductor is exposed to a changing magnetic field due to relative motion of the field source and conductor; or due to field variation with time. This cause circulation of current, within the body of the conductor.
Eddy currents can o be minimized by using laminated thin plates of conductor .

3.5 Hysteresis loss

During each A.C. cycle, current flowing in the forward and reverse directions magnetizes and demagnetizes the core alternatively. Energy is lost in each hysteresis cycle within the magnetic core. Energy loss is dependent on the properties (e.g. coercively) of particular core material and is proportional to the area of the hysteresis loop (B-H curve).

3.6 Faraday's law

This law explains the working principle of the electrical motors, generators, transformers and inductors. According to Faraday's law "magnitude of induced emf is proportional rate of change of magnetic flux."

$V_{ind} = - N \; d\Phi / dt$, where N is number of turn of coil

Magnet and coil are required to perform the Faraday's law.

3.7 Lenz's Law

Lenz law states that the polarity of the induced emf generated in a coil by a changing magnetic flux is such that it produces a current whose magnetic field oppose the cause of its production.

3.8 Self Inductance

The property of self-inductance is a particular form of electromagnetic induction. Self inductance is defined as the induction of a voltage in a current-carrying wire when the current in the wire itself is changing. This current generates a magnetic flux density $\mathbf{B}$ which gives rise to a magnetic flux Φ linking the circuit. We expect the flux Φ to be directly proportional to the current i.

Mathematically,

$\Phi \propto i$

$\Phi = L\,i$, where is the magnetic flux , i is current and proportionality constant L is called self inductance.

Induced emf $(\mathcal{E}) = L\,di/dt$.

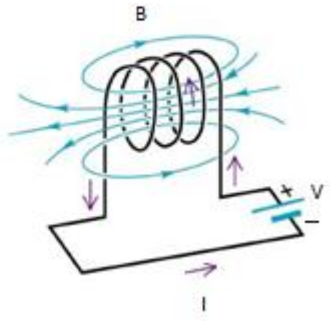


Fig.3.6

3.9 Mutual Inductance

If two coils of wire are brought into close proximity with each other so the magnetic field from one links with the other, a voltage will be generated in the second coil as a result changing flux of first coil. This is called mutual inductance.

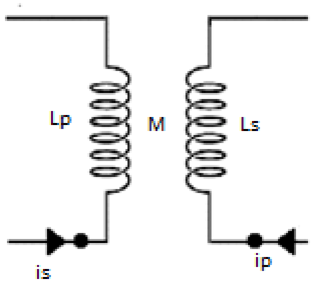


Fig.3.7

In this above figure (Fig.7) i_p ,i_s, L_p and L_s are primary current, secondary current ,primary self –inductance and secondary self inductance.

The time rate of change of magnetic flux Φ_{12} in coil 2 is proportional to the time rate of change of the current in coil 2.

$N_1\, d\Phi_{12}/dt = M_{12}\, d\, I_s/dt$

$M_{12} = \dfrac{ip \times \Phi 12}{is}$

Similarly,

$M_{21} = \dfrac{is \times \Phi 21}{ip}$

$M_{12} = M_{21} = M =$ mutual inductance

5.10 Energy stored in magnetic field of an inductor

Energy stored = $E = \int_0^I P dt = \int_0^I Vi\, dt = \int_0^I L\,(di/dt)\, i\, dt = \frac{1}{2} LI^2$, where P is the power.

Energy density = energy per unit volume = $\dfrac{\frac{1}{2}LI\ 2}{A\, l}$, where A is the area of the conductor and l is the length of the conductor.

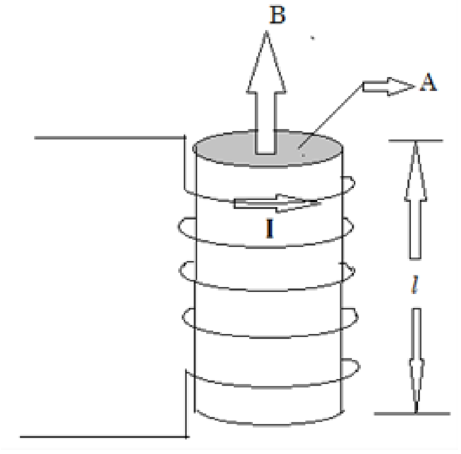


Fig.3.8

Energy stored in a magnetic field, $E = B^2 / 2\mu_0$, where B is magnetic flux density and μ_0 is permeability of the medium.

2.10 Complex Number:

Any phasor quantity can be represented in different forms.
i. Trigonomatric form, ii. Exponential form , iii. Polar form.

Trigonomatric form: V = V(cosΦ+sinΦ)
Polar form: V = V ∠+Φº
Rectangular form: V = a+jb

2.10.1 Mathematical Operation of Phasors:

❖ For addition and subtraction operation, rectangular forms are used. Such operations can't be performed in polar form.

❖ For multiplication and division operation, rectangular forms can be used, but not preferred. Such operations are performed in polar form.

❖ Let, A = X∠+Φº & B = Y∠+αº

Then, A*B = XY∠+(Φº+ αº) ; A/B = X/Y∠+(Φº- αº)

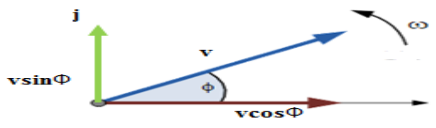
Let, A = a+jb & C = c+jd

Then, A+C = (a+c) + j(b+d)

2.10.2 Conversion of POLAR form into RECTANGULAR form

Let, **v** = v∠+Φº = v(cosΦ + j sinΦ) = a+jb

Where, a = vcosΦ b= vsinΦ



2.10.3 Conversion of RECTANGULAR form into POLAR form

Let, A = a+jb

Magnitude of A , X = $\sqrt{a^2 + b^2}$; Angle of A, Φ = $\tan^{-1}\frac{b}{a}$

2.11 Significance of j operator:

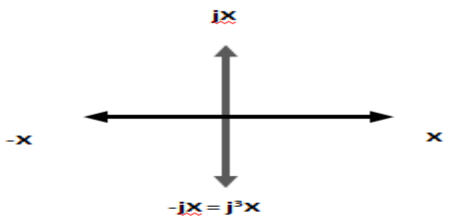


Fig. 12

Hence, if X is multiplied by j twice, the quantity will be rotated through 180° in anticlockwise direction. if X is multiplied by j once, the quantity will be rotated through 90° in anticlockwise direction. if X is multiplied by j thrice, the quantity will be rotated through 270° in anticlockwise direction.

Let, Z = R+jX, this indicates R and X are 90° apart and X leads R.
Z = R-jX, this indicates R and X are 90° apart and X lags behind R.

EXAMPLE I: An alternating voltage is given by the equation v = 10 sin (628 t + $\frac{\pi}{6}$) .Find the Rms value ,(b) frequency, (c)time period (d) form factor (e) peak factor (f) average value.

We know, $$v = V_m Sin\ (wt + \theta)$$

By comparison, $V_m = 10$ V; $\omega = 2\pi f = 628$; $\theta = \frac{\pi}{6}$

So, (a) $V_{rms} = \frac{V_m}{\sqrt{2}} = 7.07$ V (f) $V_{av} = 0.637 * V_m = 6.37$ V (b) $f = \frac{\omega}{2\pi} = 100$Hz

,

(c) $T = \frac{1}{f} = 0.01$ secs (d) $FF = \frac{V_{rms}}{V_{av}} = 1.11$ (e) Peak factor = $\frac{I_m}{I_{rms}}$ =1.414

KEY POINTS:

- ❖ DC has no frequency, constant magnitude.
- ❖ AC has specific frequency, changing magnitude.
- ❖ Average value of a sinusoidal ac signal should be calculated over a half cycle.
- ❖ RMS value of a sinusoidal ac signal should be calculated over a full cycle.
- ❖ All electrical machines, devices operate at 50Hz signal in India and rated in RMS value.
- ❖ For a better quality waveform, FF should be close to unity.
- ❖ Phasor addition & subtraction is not possible in polar form. Hence, POLAR to RECTANGULAR form conversion is necessary.
- ❖ Phasor multiplication and division is possible in rectangular form. But it is preferred to perform such operations in rectangular form. Hence, RECTANGULAR to POLAR form conversion is necessary.
- ❖ To rotate any quantity by 90° in anticlockwise direction, multiply that quantity by j.

Multiple Choice Type Questions:

1. The peak value of a sine wave is 100V. The average and RMS values are…………………. Respectively.
   (i) 63.7 V & 70.7 V (ii) 6.37 V & 7.07 V (iii) 0.637 V & 0.707 V (iv) none
2. Two alternating waveforms A & B have 10 secs & 20 secs time period. A has………………. Than B.
   (i) Less (ii) more (iii) equal (iv) none of these

Long Questions:

1. An alternating voltage is given by the equation v = 100 sin (314 t + $\frac{\pi}{3}$) .Find the
(a) Rms value ,(b) frequency, (c)time period (d) form factor (e) peak factor (f) average value
2. A = 10$\angle$30° & B = 20$\angle$60°. Find A+B.

3. An alternating voltage has amplitude of 100V at 50 Hz frequency. Write down equation of instantaneous voltage. Also find the magnitude of voltage at0.01 sec.

2.6 Average value or Mean value of a Sinusoidal waveform:

It is the amount of steady or direct current of an alternating current that transfer same amount of charge as transferred by that alternating current during same time.
Mathematically,

$$\textbf{Average value} = \frac{\textbf{Total Area covered by an alternating wave during certain period}}{\textbf{Total Time Period}}$$

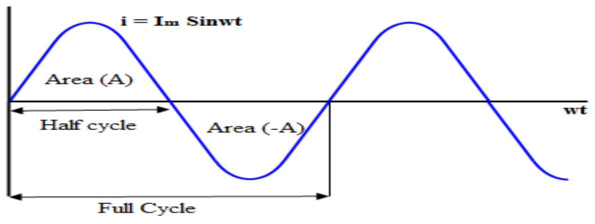


Fig 8

Average value of alternating wave is calculated over full cycle for unsymmetrical waves. For symmetrical waves average value is zero over a full cycle. Hence, for such waves it should be calculated over positive half cycle / negative half cycle.

The average value of sinusoidal alternating current is given by**:**

$\mathbf{I_{av}} = \frac{1}{\pi}\int_0^{\pi} i\, d(wt) = 0.637\, I_m$ [Substitute, $i=I_m$Sinwt]…………………………..(v)

2.9 RMS value or Effective value of a Sinusoidal waveform

It is the steady or direct current of an alternating quantity which produces same heat when passed through a circuit for certain period of time as produced by the alternating quantity.

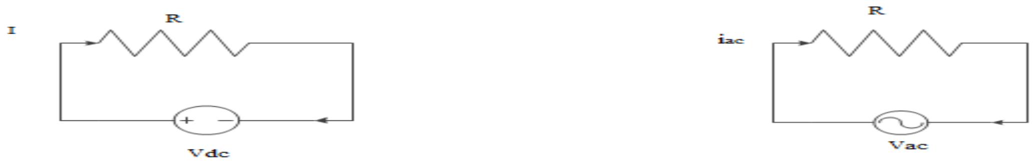


Fig. 9

Let, direct current I current passes through R resistance for t time and produces H_1 heat. Alternating current i_{ac} passes through the same circuit for same time and produces H_2 heat.

If, $H_1 = H_2$, then I is the RMS value of $i_{ac.}$

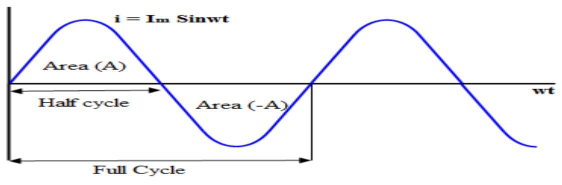


Fig. 10 AC waveform

$\mathbf{I = I_{rms}} = \frac{\sqrt{i_1^2+i_2^2 \ldots\ldots +i_n^2}}{2\pi} = \frac{\sqrt{\int_0^{2\pi} i^2 d(wt)}}{2\pi} = {I_m}/{\sqrt{2}} = \mathbf{0.707\ I_m}$ [Substitute i= I_m Sinwt]…….

(vi)

RMS value for any wave (symmetrical/ Unsymmetrical) is calculated over a full cycle. Although for the sinusoidal wave shown in above figure has zero average value over a full cycle, but rms value will have a positive value (As for RMS value we need to find squared values of area covered in a full cycle)

Form Factor: Ratio of RMS value and average value of an alternating quantity.

$$\text{Form Factor(FF)} = \frac{I_{rms}}{I_{av}} = 1.11 \text{(For sine wave)}$$

More closer the FF to unity (1), better is waveform quality.

Crest Factor/ Peak Factor: Ratio of peak value and RMS value of an alternating quantity.

$$\text{Crest Factor(CF)} = \frac{I_m}{I_{rms}} = 1.414$$

Phase Difference of a Sinusoidal Waveform

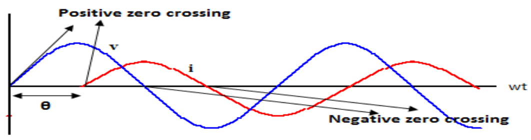


Fig. 11 Phase difference between voltage & current

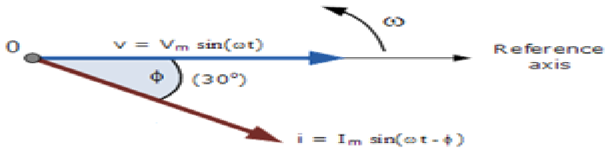
The generalized mathematical expression to define these two sinusoidal quantities will be written as:

$$v = V_m \text{Sin } wt$$

$$i = I_m \text{Sin } (wt - \theta)$$

From above two equations, it is found that phase difference between v & i is θ and –ve sign indicates i lags behind v.

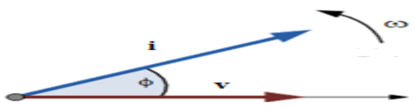
Phasor Diagram representing above equations:



Similarly, if current i leads voltage v, the phasor diagram can be written as:

$$v = V_m \text{Sin } wt$$

$$i = I_m \text{Sin } (wt + \theta)$$


i
v
Φ
ω

AC through Electric Circuit

2.12 Purely resistive circuit (R only)

Let, a pure sinusoidal voltage v is applied to a purely resistive circuit having resistance R.

$$i = \frac{v}{R} = \frac{V_m}{R}\, Sin\, wt \ldots\ldots\ldots\ldots\ldots\ (vii)$$

Where, Instantaneous supply voltage,

$$v = V_m Sin\, wt \ldots\ldots\ldots\ldots\ldots\ldots\ldots(viii)$$

$$I_m = \frac{V_m}{R}$$

I_m and V_m are the maximum values of current and voltage respectively.
V & I are RMS values of voltage & current respectively.

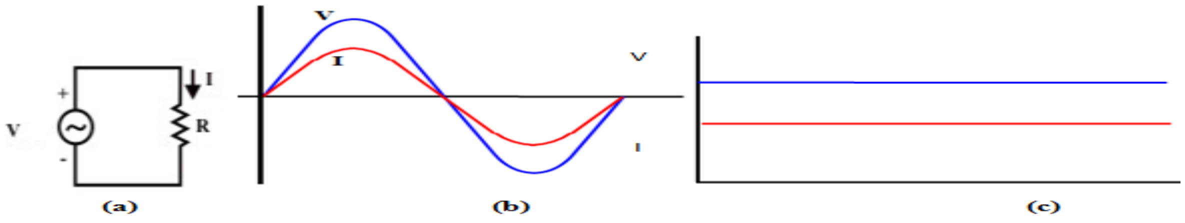


Fig. 13 (a) Purely resistive circuit with ac voltage
(b) voltage & current waveforms
(c) Phasor Diagram

- From equation (vii) & (viii), it is observed that phase angle between voltage (V) & current (I) is,Φ = 0°. In other words, V & I are in same phase.
- Hence, power factor, **cosΦ = 1 (Unity).**

2.13 Purely inductive circuit (L only)

If pure sinusoidal alternating voltage is applied to the circuit, the instantaneous voltage is given by,

$$v = V_m Sin\, wt \ldots\ldots\ldots\ldots\ldots\ldots(ix)$$

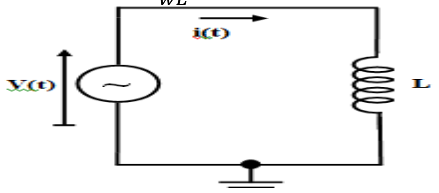
As we know, $v = L\frac{di}{dt}$,

$i = \frac{1}{L}\int v = V_m Sinwt$

Sol

$i = \frac{V_m}{wL} Sin(wt - 90°) = I_m\ Sin\ (wt\text{-}90°) \ldots\ldots\ldots\ldots(x)$

Where $I_m = \frac{V_m}{wL}$



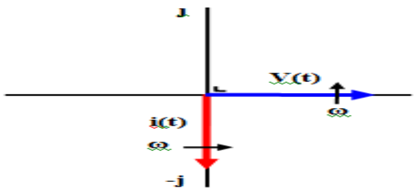


Fig . 14 Purely Inductive circuit **Fig 15 . Phasor Diagram**

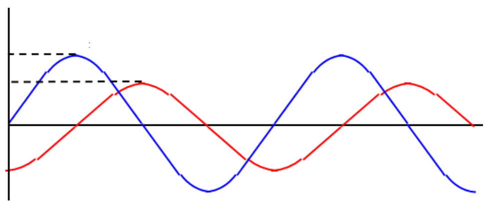


Fig. 16 Voltage & Current Waveforms

For purely inductive circuit, the term inductive reactance, $X_L = wL = 2\prod fL$ offers opposition to the flow of current. Its unit is ohm.

- ❖ For DC voltage source as frequency is zero, inductor behaves as a short circuit.
- ❖ From equation (ix) & (x), it is observed that phase angle between voltage (V) & current (I) is, $\Phi = 90^o$. **(-ve) sign indicates current is lagging**. Hence, power factor, **cosΦ = 0.**

2.14 Purely capacitive circuit (C only)

If pure sinusoidal alternating voltage is applied to the circuit, the instantaneous voltage is given by,

$$v = V_m Sin\, wt \ldots\ldots\ldots\ldots\ldots\ldots (xi)$$

As we know, $i = C\frac{dv}{dt}$

Solving this equation, we get ;

$i = \frac{V_m}{(^1/_{wc})}\, Sin(wt + 90^o) = I_m$ Sin $(wt+90^o)$ …………..(xii)

Where, $I_m = \frac{V_m}{(^1/_{wc})} = wcV_m$

For purely capacitive circuit, the term inductive reactance, $X_c = \frac{1}{wc} = \frac{1}{2\prod fC}$ offers opposition to the flow of current. Its unit is ohm.

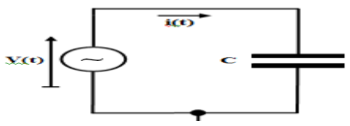


Fig. 17 Purely capacitive circuit

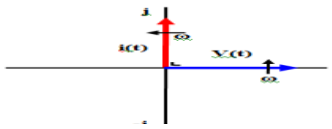


Fig. 18 Phasor Diagram

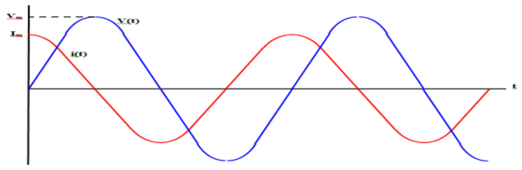


Fig. 19 Voltage & current waveforms

- ❖ Capacitive reactance decreases with increasing frequency.
- ❖ For DC voltage source as frequency is zero, capacitor behaves as an open circuit.
- ❖ From equation (xi) & (xii), it is observed that phase angle between voltage (V) & current (I) is,

  $\Phi = 90^o$. **(+ve) sign indicates current is leading voltage by 90 degree**.

- ❖ Hence, power factor, cosΦ = 0.

Advantages of Three Phase System over Single Phase System

Presently 3-ø AC system is very popular and being used worldwide for power generation, power transmission, distribution and for electric motors.

Three phase system has the following advantages as compare to single phase system:

1. Power to weight ratio of 3-ø alternator is high as compared to 1-ø alternator. That means for generation for same amount of Electric Power, the size of 3-ø alternator is small as compare to 1-ø Alternator. Hence, the overall cost of alternator is reduced for generation of same amount of power. Moreover, due to reduction in weight, transportation and installation of alternator become convenient and less space is required to accommodate the alternator in power houses.
2. For electric power transmission and distribution of same amount of power, the requirement of conductor material is less in 3-ø system as compare to 1-ø system. Hence, the 3-ø transmission and distribution system is economical as compare 1-ø system.
3. Let us consider the power produced by single phase supply and 3-phase supply at unity power factor. Wave form of power produce due 1-phase supply at unity power factor is shown in figure (A) and Wave form of power produced due to 3-phase supply is shown in figure (B) below.

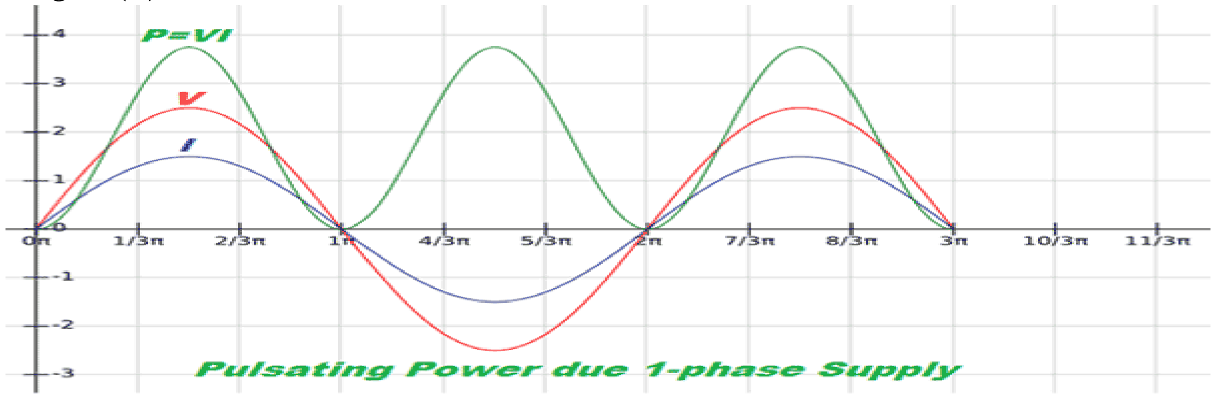


Figure (A)

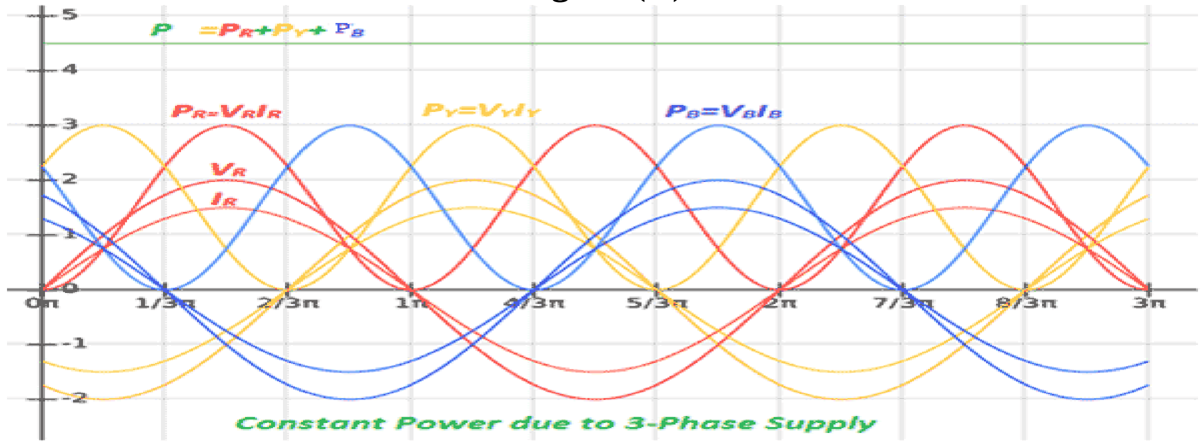


Figure (B)

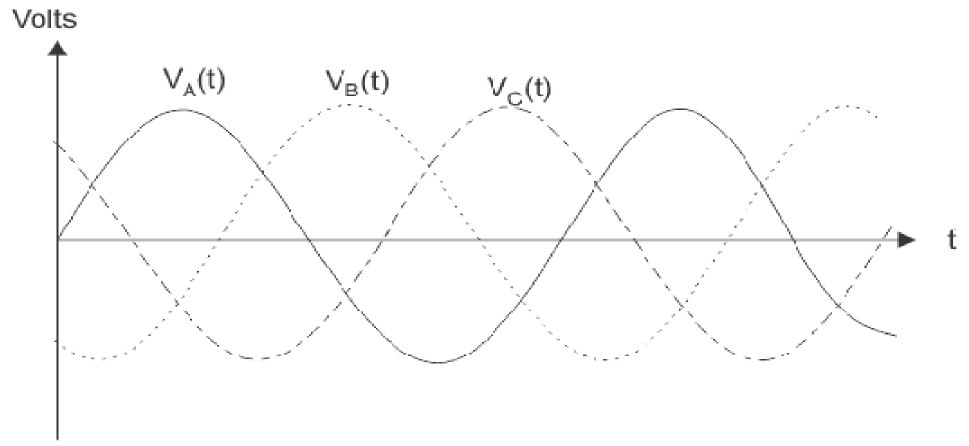
4. From power wave forms shown in figure (A) and (B) above it is clear that in 3-phase system, the instantaneous power is always constant over the cycle results in smooth and vibration free operation of machine. Whereas in 1-ø system the instantaneous power is pulsating hence change over the cycle, which leads to vibrations in machines.

5. Power to weight ratio of three phase induction motor is high as compare to single phase induction motor. Means for same amount of Mechanical Power, the size of three phase induction motor is small as compare to single phase induction motor. Hence, the overall cost of induction motor is reduced. Moreover, due to reduction in weight, transportation and installation of induction motor become convenient and less space is required to accommodate the Induction motor.
6. 3-phase induction motor is self-started as the magnetic flux produced by 3-phase supply is rotating in nature with constant magnitude. Whereas 1-ø induction motor is not self-started as the magnetic flux produced by 1-ø supply is pulsating in nature. Hence, we have to make some arrangement to make the 1-ø induction motor self-started which further increases the cost of 1-ø induction motor.
7. 3-phase motor is having better power factor and efficiency as compare to 1-ø motor.
8. Power to weight ratio of 3-phase transformer is high as compare to 1-ø transformer. Means for same amount of Electric Power, the size of 3-phase transformer is small as compared to 1-ø transformer. Hence, the overall cost of transformer is reduced. Moreover, due to reduction in weight, transportation and installation of transformer become convenient and less space is required to accommodate the transformer.
9. If fault occurs in any winding of 3-phase transformer, the rest of two winding can be used in open delta to serve the 3-phase load which is not possible in 1-ø transformer. This ability of 3-phase transformer further increases the reliability of 3-phase transformer.
10. A 3-phase system can be used to feed a 1-ø load, whereas vice-versa is not possible.
11. DC rectified from 3-phase supply is having the ripple factor 4% and DC rectified from 1-ø supply is having the ripple factor 48.2 %. Mean DC rectified from 3-ø supply contains less ripples as compare to DC rectified from 1-ø supply. Hence the requirement of filter is reduced for DC rectified from 3-phase supply. Which reduce the overall cost of converter.
12. From above it is clear the 3-phase system is more economical, efficient, reliable and convenient as compared to 1-ø system.

Three Phase Circuit | Star and Delta System

There are two types of system available in electric circuit, single phase and **three phase system**. In single phase circuit, there will be only one phase, i.e the current will flow through only one wire and there will be one return path called neutral line to complete the circuit. So in single phase minimum amount of power can be transported. Here the generating station and load station will also be single phase. This is an old system using from previous time.In 1882, new invention has been done on polyphase system, that more than one phase can be used for generating, transmitting and for load system. **Three phase circuit** is the polyphase system where three phases are send together from the generator to the load. Each phase are having a phase difference of 120°, i.e. 120° angle electrically. So from the total of 360°, three phases are equally divided into 120° each. The power in **three phase system** is continuous as all the three phases are involved in generating the total power. The sinusoidal waves for 3 phase system is shown below-

The three phases can be used as single phase each. So if the load is single phase, then one phase can be taken from the **three phase circuit** and the neutral can be used as ground to complete the circuit.



Why Three Phase is preferred Over Single Phase?

There are various reasons for this question because there are numbers of advantages over single phase circuit. The three phase system can be used as three single phase line so it can act as three single phase system. The three phase generation and single phase generation is same in the generator except the arrangement of coil in the generator to get 120° phase difference. The conductor needed in three phase circuit is 75% that of conductor needed in single phase circuit. And also the instantaneous power in single phase system falls down to zero as in single phase we can see from the sinusoidal curve but in three phase system the net power from all the phases gives a continuous power to the load.

Till now we can say that there are three voltage source connected together to form a three phase circuit and actually it is inside the generator. The generator is having three voltage sources which are acting together in 120° phase difference. If we can arrange three single phase circuit with 120° phase difference, then it will become a three phase circuit. So 120° phase difference is must otherwise the circuit will not work, the three phase load will not be able to get active and it may also cause damage to the system. The size or metal quantity of three phase devices is not having much difference. Now if we consider the transformer, it will be almost same size for both single phase and three phase because transformer will make only the linkage of flux. So the three phase system will have higher efficiency compared to single phase because for the same or little difference in mass of transformer, three phase line will be out whereas in single phase it will be only one. And losses will be minimum in three phase circuit. So overall in conclusion the three phase system will have better and higher efficiency compared to the single phase system. In three phase circuit, connections can be given in two types:

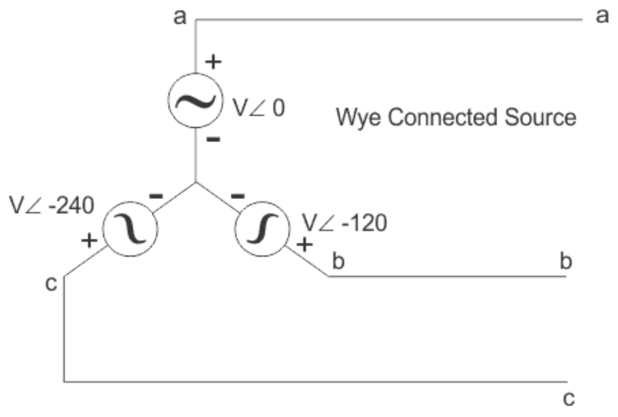
1. Star connection
2. Delta connection

1. Star Connection

In star connection, there is four wire, three wires are phase wire and fourth is neutral which is taken from the star point. Star connection is preferred for long distance power transmission because it is having the neutral point. In this we need to come to the concept of balanced and unbalanced current in power system.

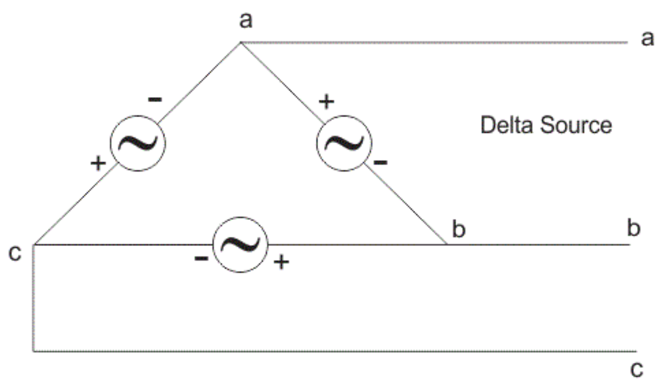
When equal current will flow through all the three phases, then it is called as balanced current. And when the current will not be equal in any of the phase, then it is unbalanced current. In this case, during balanced condition there will be no current flowing through the neutral line and hence there is no use of the neutral terminal. But when there will be unbalanced current flowing in the three phase circuit, neutral is having a vital role. It will take the unbalanced current through to the ground and protect the transformer. Unbalanced current affects transformer and it may also cause damage to the transformer and for this star connection is preferred for long distance transmission. The star connection is shown below- In star connection, the line voltage is √3 times of phase voltage. Line voltage is the voltage between two phases in three phase circuit and phase voltage is the voltage between one phase to the neutral line. And the current is same for both line and phase. It is shown as expression below

$$E_{Line} = \sqrt{3}E_{phase} \text{ and } I_{Line} = I_{Phase}$$



2. Delta Connection

In delta connection, there is three wires alone and no neutral terminal is taken. Normally delta connection is preferred for short distance due to the problem of unbalanced current in the circuit. The figure is shown below for delta connection. In the load station, ground can be used as neutral path if required.



In delta connection, the line voltage is same with that of phase voltage. And the line current is √3 times of phase current. It is shown as expression below,

$$E_{Line} = E_{phase}\ and\ I_{Line} = \sqrt{3} I_{Phase}$$

In three phase circuit, star and delta connection can be arranged in four different ways-

1. Star-Star connection
2. Star-Delta connection
3. Delta-Star connection
4. Delta-Delta connection

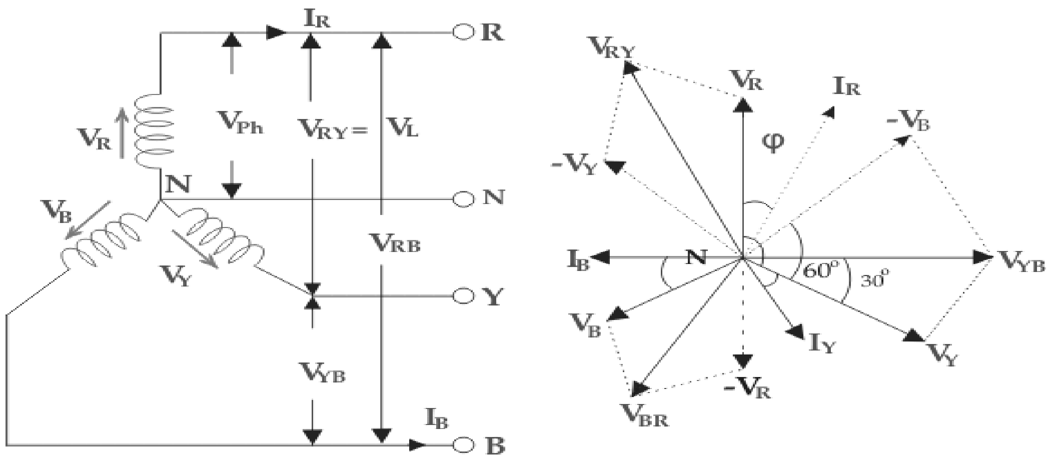
But the power is independent of the circuit arrangement of the three phase system. The net power in the circuit will be same in both star and delta connection. The power in three phase circuit can be calculated from the equation below,

$$P_{Total} = 3 \times E_{phase} \times I_{phase} \times PF$$

Since, there is three phases, so the multiple of 3 is made in the normal power equation and the PF is power factor. Power factor is a very important factor in three phase system and sometimes due to certain error; it is corrected by using capacitors.

Relationship of Line and Phase Voltages and Currents in a Star Connected System

To derive the relations between line and phase currents and voltages of a star connected system, we have first to draw a balanced star connected system.



Suppose due to load impedance the current lags the applied voltage in each phase of the system by an angle ϕ. As we have considered that the system is perfectly balanced, the magnitude of current and voltage of each phase is the same. Let us say, the magnitude of the voltage across the red phase i.e. magnitude of the voltage between neutral point (N) and red phase terminal (R) is V_R.

Similarly, the magnitude of the voltage across yellow phase is V_Y and the magnitude of the voltage across blue phase is V_B. In the balanced star system, magnitude of phase voltage in each phase is V_{ph}. $\therefore V_R = V_Y = V_B = V_{ph}$

We know in the star connection, line current is same as phase current. The magnitude of this current is same in all three phases and say it is I_L. $\therefore I_R = I_Y = I_B = I_L$, Where, I_R is line current of R phase, I_Y is line current of Y phase and I_B is line current of B phase. Again, phase current, I_{ph} of each phase is same as line current IL in star connected system. $\therefore I_R = I_Y = I_B = I_L = I_{ph}$.

Now, let us say, the voltage across R and Y terminal of the star connected circuit is V_{RY}. The voltage across Y and B terminal of the star connected circuit is V_{YB}. The voltage across B and R terminal of the star connected circuit is V_{BR}. From the diagram, it is found that $V_{RY} = V_R + (-V_Y)$ Similarly, $V_{YB} = V_Y + (-V_B)$ And, $V_{BR} = V_B + (-V_R)$ Now, as angle between V_R and V_Y is 120°(electrical), the angle between V_R and $-V_Y$ is 180° – 120° = 60°(electrical)

$$V_L = |V_{RY}| = \sqrt{V_R^2 + V_Y^2 + 2V_R V_Y \cos 60^o}$$
$$= \sqrt{V_{ph}^2 + V_{ph}^2 + 2V_{ph}V_{ph} \times \frac{1}{2}}$$
$$= \sqrt{3}V_{ph}$$
$$\therefore V_L = \sqrt{3}V_{ph}$$

Thus, for the star-connected system line voltage = √3 × phase voltage. Line current = Phase current as, the angle between voltage and current per phase is φ, the electric power per phase is

$$V_{ph}I_{ph}cos\phi = \frac{V_L}{\sqrt{3}}I_L\cos\phi$$

So the total power of three phase system is

$$3 \times \frac{V_L}{\sqrt{3}}I_L\cos\phi = \sqrt{3}V_LI_L\cos\phi$$

Delta Connection (Δ)

In this system of interconnection, the starting ends of the three phases or coils are connected to the finishing ends of the coil. Or the starting end of the first coil is connected to the finishing end of the second coil and so on (for all three coils) and it looks like a closed mesh or circuit as shown in fig (1).

In more clear words, all three coils are connected in series to form a close mesh or circuit. Three wires are taken out from three junctions and the all outgoing currents from junction assumed to be positive.

In Delta connection, the three windings interconnection looks like a short circuit, but this is not true, if the system is balanced, then the value of the algebraic sum of all voltages around the mesh is zero.

When a terminal is open, then there is no chance of flowing currents with basic frequency around the closed mesh.

Good to Remember: at any instant, the EMF value of one phase is equal to the resultant of the other two phases EMF values but in the opposite direction.

Delta or Mesh Connection System is also called Three Phase Three Wire System (3-Phase 3 Wire) and it is the best and suitable system for AC Power Transmission.

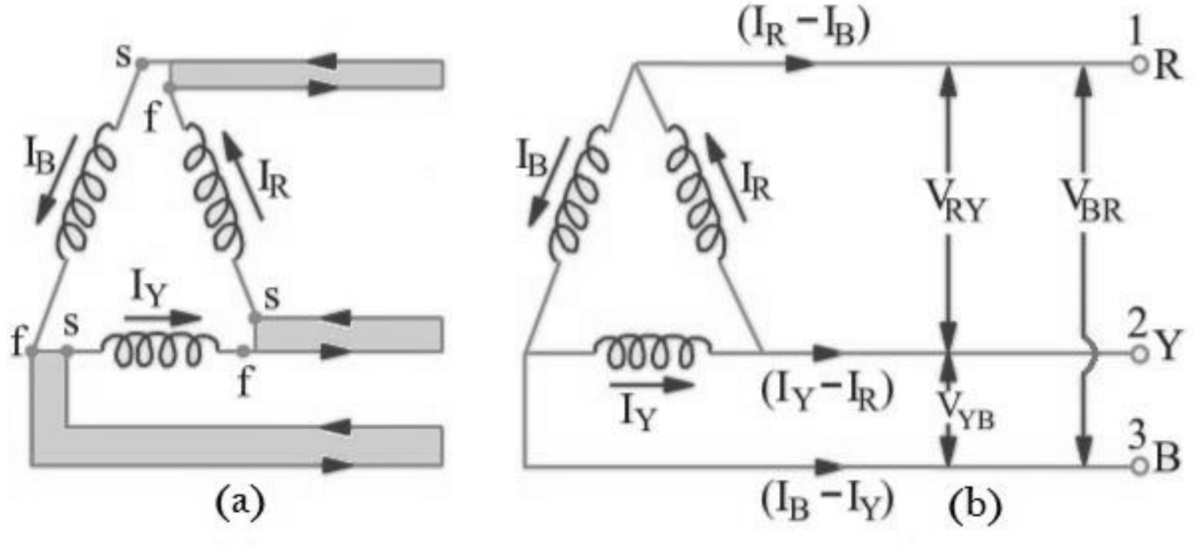


Fig 1:Delta Connection (Δ): 3 Phase Power, Voltage & Current Values

Voltage, Current and Power Values in Delta Connection (Δ)

1. Line Voltages and Phase Voltages in Delta Connection

It is seen from fig 2 that there is only one phase winding between two terminals (i.e. there is one phase winding between two wires). Therefore, in Delta Connection, the voltage between (any pair of) two lines is equal to the phase voltage of the phase winding which is connected between two lines. Since the phase sequence is R → Y → B, therefore, the direction of voltage from R phase towards Y phase is positive (+),and the voltage of R phase is leading by 120°from Y phase voltage. Likewise, the voltage of Y phase is leading by 120° from the phase voltage of B and its direction is positive from Y towards B.

If the line voltage between;

Line 1 and Line 2 = V_{RY}

Line 2 and Line 3 = V_{YB}

Line 3 and Line 1 = V_{BR}
Then, we see that V_{RY} leads VYB by 120° and V_{YB} leads V_{BR} by 120°.
Let's suppose,
$V_{RY} = V_{YB} = V_{BR} = V_L$ …………… (Line Voltage)
Then
$V_L = V_{PH}$
I.e. in Delta connection, the Line Voltage is equal to the Phase Voltage.
2. Line Currents and Phase Currents in Delta Connection
It will be noted from the below (fig-2) that the total current of each Line is equal to the vector difference between two phase currents flowing through that line. i.e.;
Current in Line 1= $I_1 = I_R – I_B$
Current in Line 2 =$I_2 = I_Y – I_R$
Current in Line 3 =$I_3 = I_B – I_Y$
{Vector Difference}

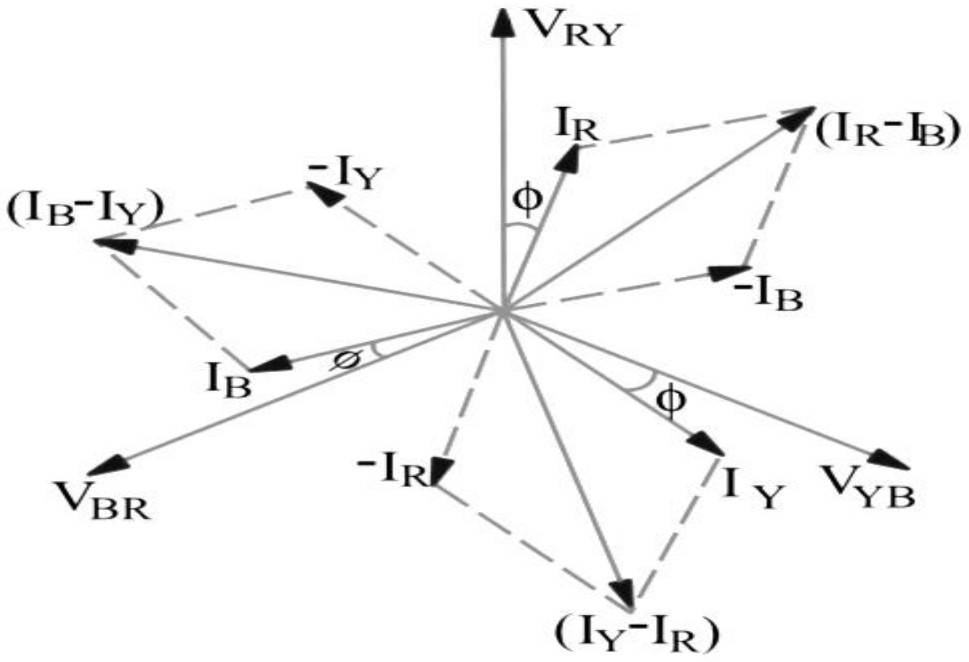


Fig 2: Line & Phase Current and Line & Phase Voltage in Delta (Δ) Connection

The current of Line 1 can be found by determining the vector difference between I_R and I_B and we can do that by increasing the I_B Vector in reverse, so that, I_R and I_B makes a parallelogram. The diagonal of that parallelogram shows the vector difference of I_R and I_B which is equal to Current in Line 1= I_1. Moreover,
by reversing the vector of I_B, it may indicate as ($-I_B$), therefore, the angle between I_R and $-I_B$ (I_B, when reversed = $-I_B$) is 60°. If,
$I_R = I_Y = I_B = I_{PH}$ …. The phase currents
Then;
The current flowing in Line 1 would be;
I_L or $I_1 = 2 \times I_{PH} \times \text{Cos}\ (60°/2)$
$= 2 \times I_{PH} \times \text{Cos}\ 30°$
$= 2 \times I_{PH} \times (\sqrt{3}/2)$ …… Since Cos 30° = $\sqrt{3}/2 = \sqrt{3}\ I_{PH}$

i.e. In Delta Connection, The Line current is √3 times of Phase Current.
Similarly, we can find the reaming two Line currents as same as above. i.e.,
$I_2 = I_Y - I_R$ … Vector Difference = $\sqrt{3}\ I_{PH}$
$I_3 = I_B - I_Y$ … Vector difference = $\sqrt{3}\ I_{PH}$
As, all the Line current are equal in magnitude i.e.
$I_1 = I_2 = I_3 = I_L$
Hence
$I_L = \sqrt{3}\ I_{PH}$
It is seen from the fig above that;
The Line Currents are 120° apart from each other
Line currents are lagging by 30° from their corresponding Phase Currents
The angle Φ between line currents and respective line voltages is (30°+Φ), i.e. each line current is lagging by (30°+Φ) from the corresponding line voltage.

3. Power in Delta Connection

We know that the power of each phase
Power / Phase = $V_{PH} \times I_{PH} \times \cos\Phi$
And the total power of three phases;
Total Power = $P = 3 \times V_{PH} \times I_{PH} \times \cos\Phi$ ….. (1)
We know that the values of Phase Current and Phase Voltage in Delta Connection;
$I_{PH} = I_L / \sqrt{3}$ ….. (From $I_L = \sqrt{3}\ I_{PH}$)
$V_{PH} = V_L$
Putting these values in power eqn……. (1)
$P = 3 \times V_L \times (I_L/\sqrt{3}) \times \cos\Phi$ …… ($I_{PH} = I_L$ / /√3)
$P = \sqrt{3} \times\sqrt{3} \times V_L \times (I_L/\sqrt{3}) \times \cos\Phi$ …{ $3 = \sqrt{3}\times\sqrt{3}$ }
$P = \sqrt{3} \times V_L\times I_L \times \cos\Phi$ …
Hence proved;
Power in Delta Connection,
$P = 3 \times V_{PH} \times I_{PH} \times \cos\Phi$ …. or
$P = \sqrt{3} \times V_L \times I_L \times \cos\Phi$

Good to Know: Where Cos Φ = Power factor = the phase angle between Phase Voltage and Phase Current and not between Line current and line voltage.

Good to Remember:
In both Star and Delta Connections, The total power on balanced load is same.
I.e. total power in a Three Phase System = P = √3 x VL x IL x CosΦ

Good to know:
Balanced System is a system where:

- All three phase voltages are equal in magnitude.
- All phase voltages are in phase by each other i.e. 360°/3 = 120°.
- All three phase Currents are equal in magnitude.
- All phase Currents are in phase by each other i.e. 360°/3 = 120°.
- A three phase balanced load is a system in which the load connected across three phases are identical.

1.5 Network Theorem

Superposition Theorem:

Superposition theorem states that, "In a network of linear resistances containing more than one source of e.m.f., the current which flows at any point is the sum of all the currents which would flow at that point if each source of e.m.f where considered separately and all the other source of e.m.f replaced for the time being by resistances equal to their internal resistances."

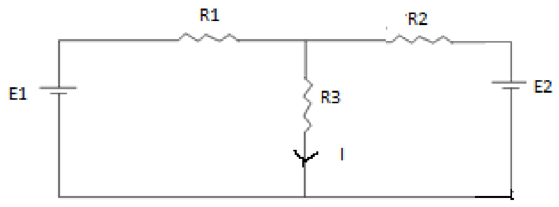


Fig.1.13

- Replace all other independent voltage sources with a short circuit and current sources with an open circuit.
  - Once voltage drops and/or currents have been determined for each individual source working separately, the values are added algebraically to find the actual voltage drops/currents with all sources active.

Step 1 : To find I_1

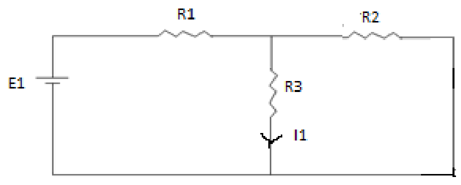


Fig.1.14

Step 2 : To find I_2

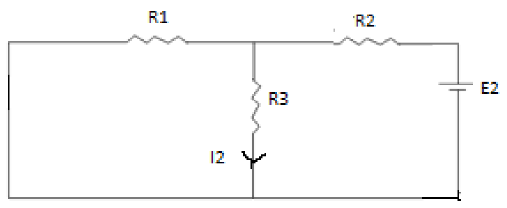


Fig.1.15

Step 3 : Total current I = $I_1 + I_2$

Thevenin's Theorem

Statement: The current flowing through a load resistance R_L connected across any two terminals X and Y of a bilateral, linear network is given by $\frac{V_{TH}}{R_{TH}+R_L}$, where V_{TH} is the open circuit voltage and R_{TH} is the Thevenin's resistance of the network as viewed back into the open circuited network from terminals XY deactivating all the independent sources.

Procedure to solve any network using Thevenin's Theorem:

Step-I:

To find V_{TH}:

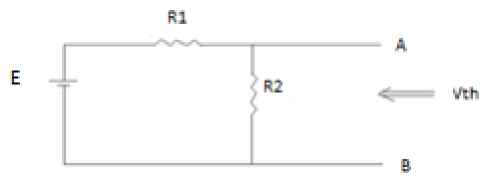


Fig.1.16

Calculation:

$V_{TH}=IR_2$ .... (i)

Where, $I = \frac{E}{R_1+R_2}$ .... (ii)

$V_{TH} = \frac{E}{R_1+R_2}R_2$ .... (iii)

Step-II:

To find R_{TH}:

Calculation:

$R_{TH}=\frac{R_1R_2}{R_1+R_2}$ .... (iv)

Step-III:

To find I_L:

Thevnin's Equivalent Circuit:

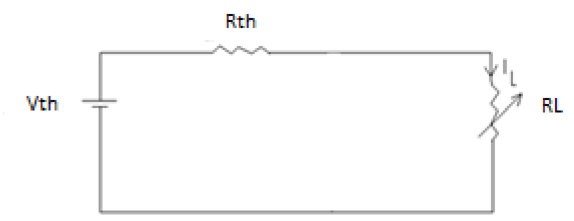


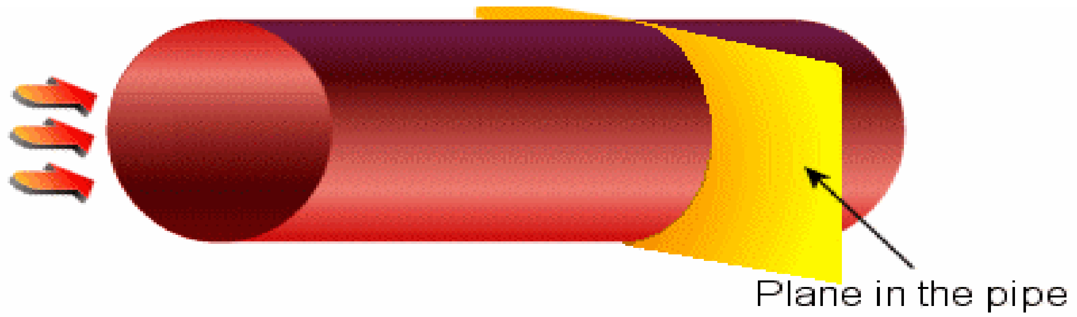
Fig.1.17

Calculation:

$$I_L = \frac{V_{TH}}{R_{TH}+R_L}$$

ELECTRICAL TECHNOLOGY (EE-101F)

SECTION A : DC NETWORK LAWS AND THEOREMS



Section A

Part A

- Ohm’s Law
- Kirchhoff’s Laws: KVL and KCL
- Nodal and Loop methods of analysis,
- Star to Delta and Delta to Star transformations

Part B

Thevenin’s Theorem
Norton’s Theorem
Superposition Theorem
Maximum Power Transfer Theorem
Milman’s Theorem

DCE,Gurgaon

Circuit Elements

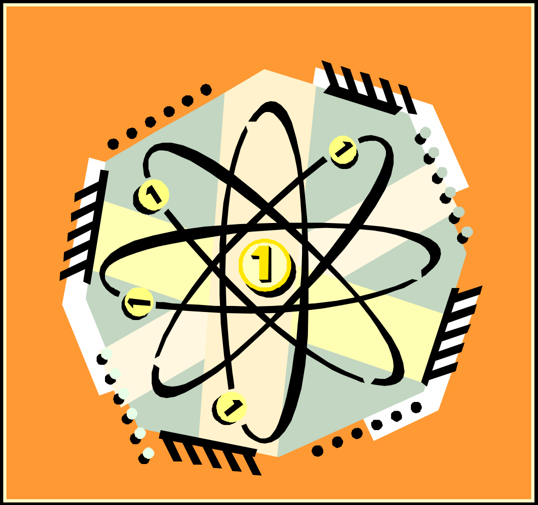
Overview of this Part

In this part, we will cover the following topics:

- What a circuit element is
- Independent and dependent voltage sources and current sources
- Resistors and Ohm's Law

Circuit Elements

- In circuits, we think about basic circuit elements that are the “building blocks” of our circuits. This is similar to what we do in Chemistry with chemical elements like oxygen or nitrogen.
- A circuit element cannot be broken down or subdivided into other circuit elements.
- A circuit element can be defined in terms of the behavior of the voltage and current at its terminals.



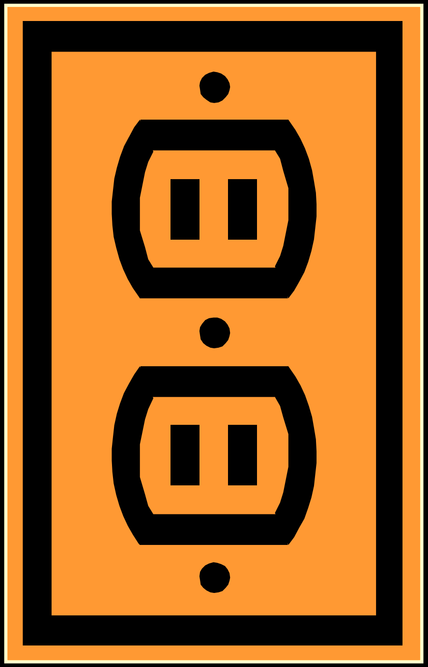
The 5 Basic Circuit Elements

There are 5 basic circuit elements:

1. Voltage sources
2. Current sources
3. Resistors
4. Inductors
5. Capacitors

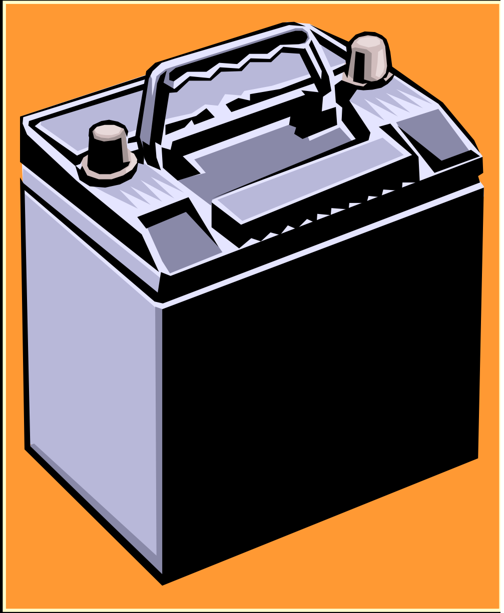
Voltage Sources

- A voltage source is a two-terminal circuit element that maintains a voltage across its terminals.
- The value of the voltage is the defining characteristic of a voltage source.
- Any value of the current can go through the voltage source, in any direction. The current can also be zero. The voltage source does not “care about” current. It “cares” only about voltage.



Voltage Sources – Ideal and Practical

- A voltage source maintains a voltage across its terminals no matter what you connect to those terminals.
- We often think of a battery as being a voltage source. For many situations, this is fine. Other times it is not a good model. A real battery will have different voltages across its terminals in some cases, such as when it is supplying a large amount of current. As we have said, a voltage source should not change its voltage as the current changes.
- We sometimes use the term ideal voltage source for our circuit elements, and the term practical voltage source for things like batteries. We will find that a more accurate model for a battery is an ideal voltage source in series with a resistor. More on that later.



Voltage Sources – 2 kinds

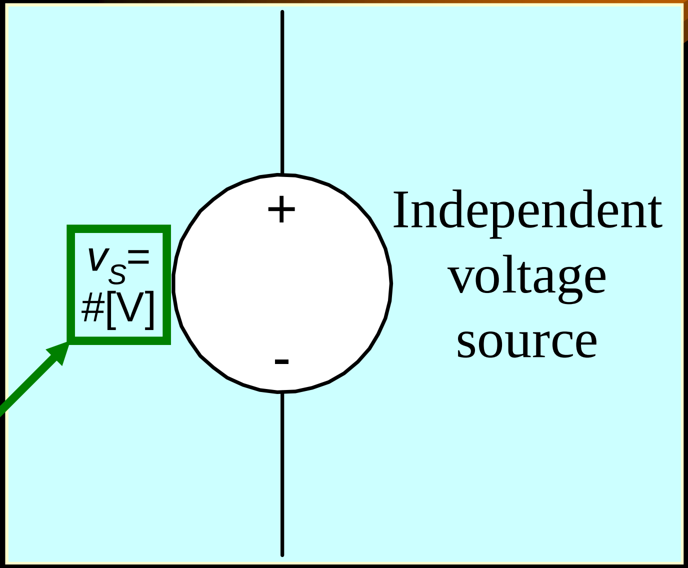
There are 2 kinds of voltage sources:

1. Independent voltage sources
2. Dependent voltage sources, of which there are 2 forms:
   i. Voltage-dependent voltage sources
   ii. Current-dependent voltage sources

DCE,Gurgaon

Voltage Sources – Schematic Symbol for Independent Sources

The schematic symbol that we use for independent voltage sources is shown here.



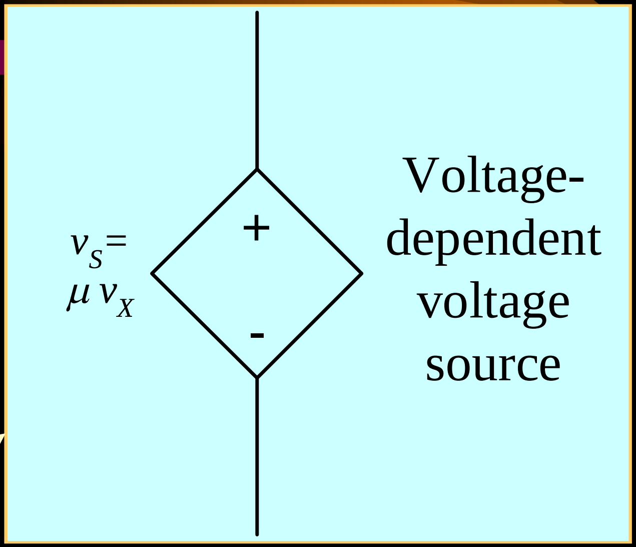
This is intended to indicate that the schematic symbol can be labeled either with a variable, like v_S, or a value, with some number, and units. An example might be 1.5[V]. It could also be labeled with both.

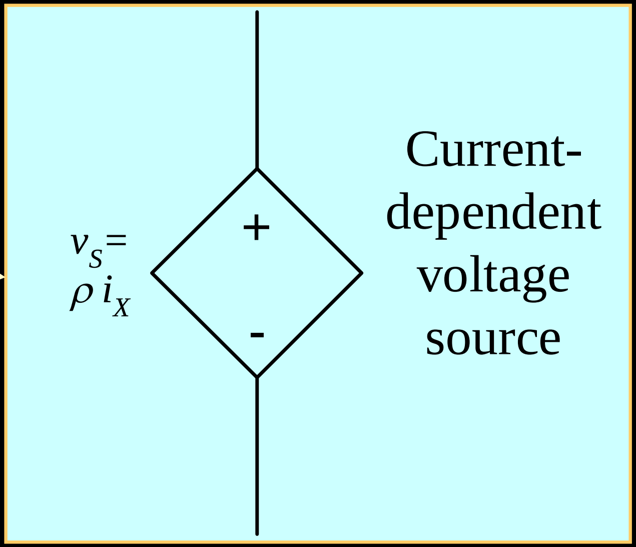
Voltage Sources – Schematic Symbols for Dependent Voltage Sources

The schematic symbols that we use for dependent voltage sources are shown here, of which there are 2 forms:

i. Voltage-dependent voltage sources

ii. Current-dependent voltage sources





DCE,Gurgaon

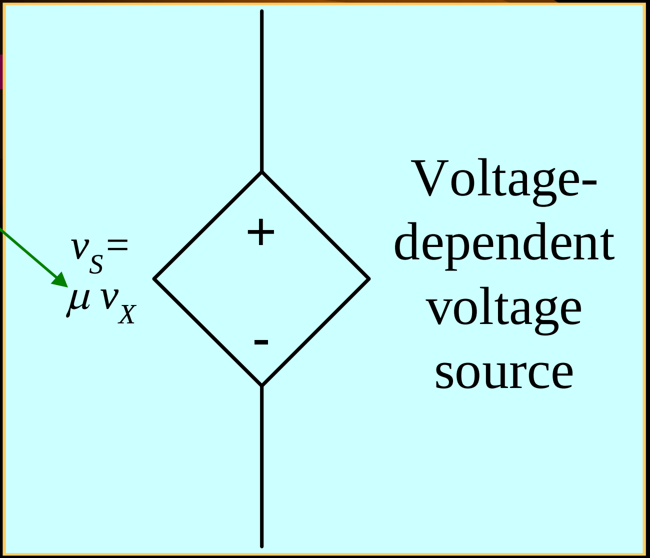
Notes on Schematic Symbols for Dependent Voltage Sources

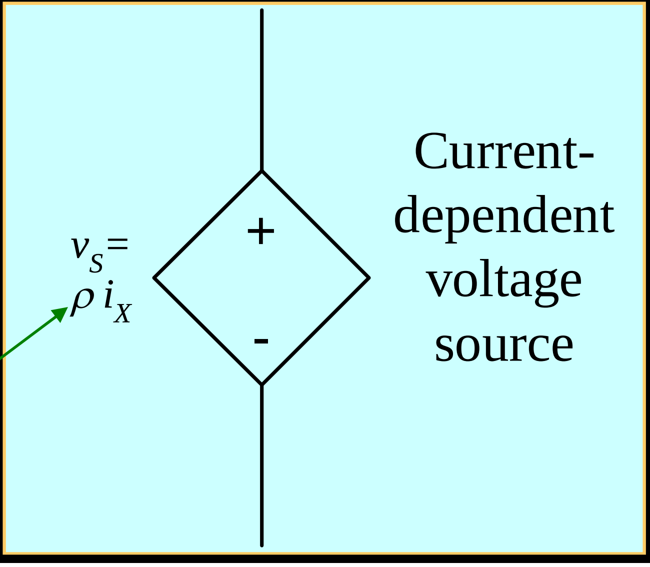
The symbol m is the coefficient of the voltage v_X. It is dimensionless. For example, it might be 4.3 v_X. The v_X is a voltage somewhere in the circuit.

The schematic symbols that we use for dependent voltage sources are shown here, of which there are 2 forms:

i. Voltage-dependent voltage sources
ii. Current-dependent voltage sources

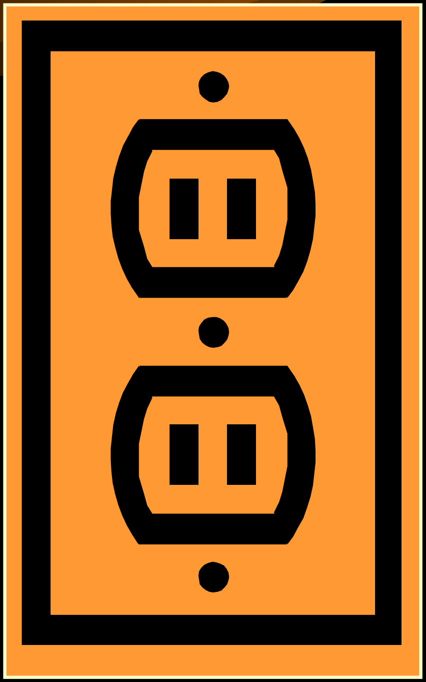
The symbol r is the coefficient of the current i_X. It has dimensions of [voltage/current]. For example, it might be 4.3[V/A] i_X. The i_X is a current somewhere in the circuit.





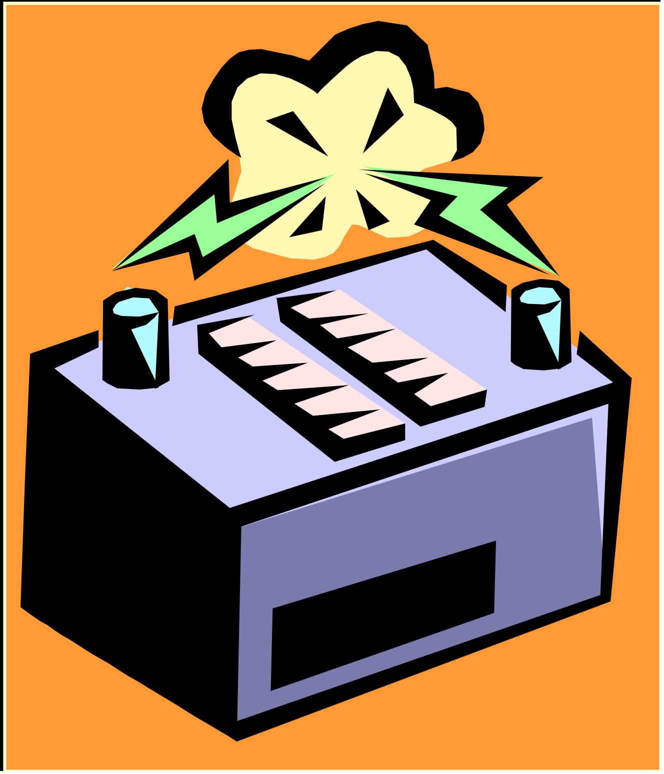
Current Sources

- A current source is a two-terminal circuit element that maintains a current through its terminals.
- The value of the current is the defining characteristic of the current source.
- Any voltage can be across the current source, in either polarity. It can also be zero. The current source does not “care about” voltage. It “cares” only about current.



Current Sources - Ideal

- A current source maintains a current through its terminals no matter what you connect to those terminals.
- While there will be devices that reasonably model current sources, these devices are not as familiar as batteries.
- We sometimes use the term <u>ideal current source</u> for our circuit elements, and the term <u>practical current source</u> for actual devices. We will find that a good model for these devices is an ideal current source in parallel with a resistor. More on that later.



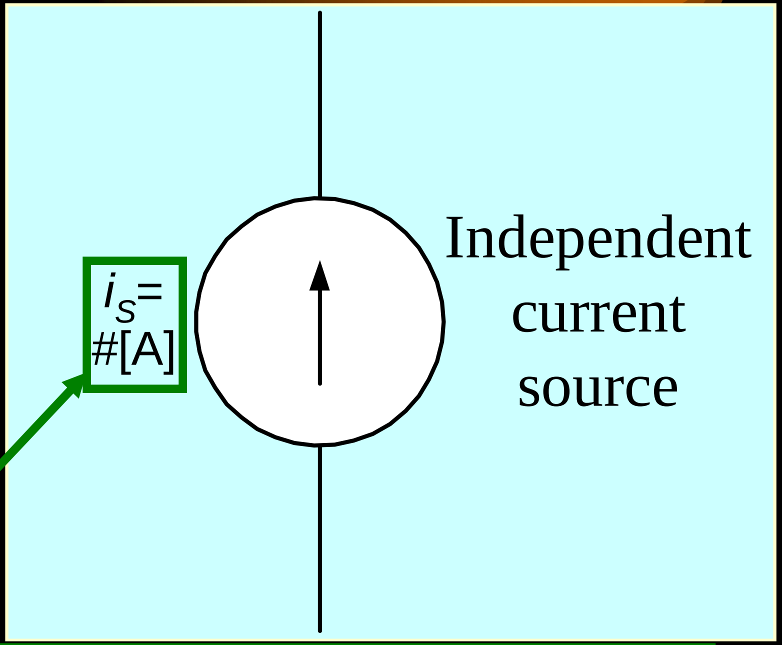
Current Sources – 2 kinds

There are 2 kinds of current sources:

1. Independent current sources
2. Dependent current sources, of which there are 2 forms:
   i. Voltage-dependent current sources
   ii. Current-dependent current sources

Current Sources – Schematic Symbol for Independent Sources

The schematic symbols that we use for current sources are shown here.



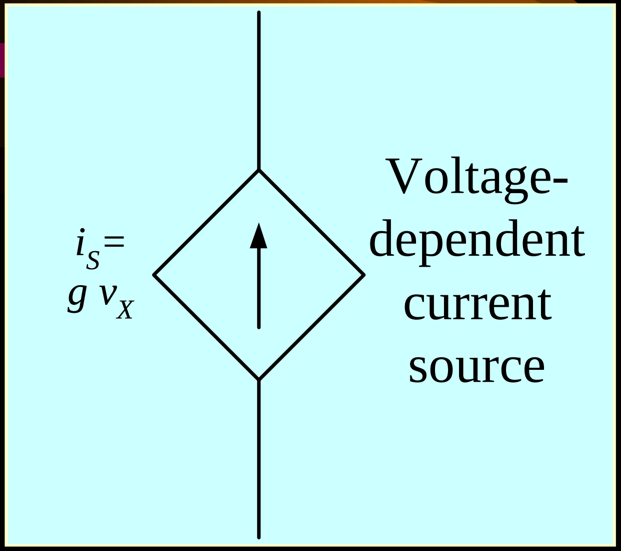
This is intended to indicate that the schematic symbol can be labeled either with a variable, like i_S, or a value, with some number, and units. An example might be 0.2[A]. It could also be labeled with both.

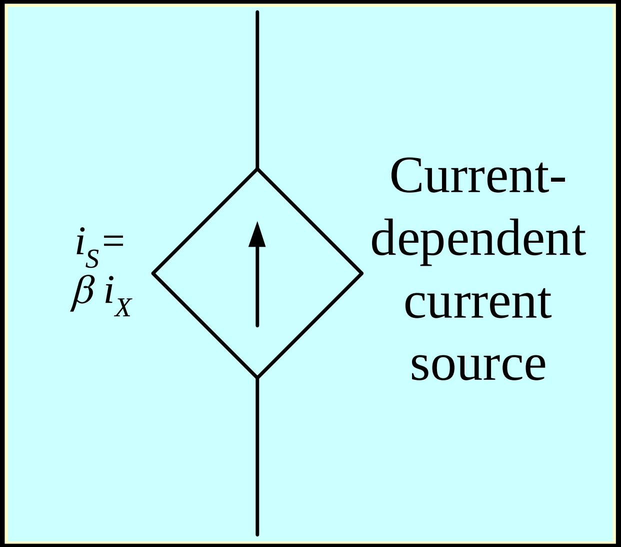
Current Sources – Schematic Symbols for Dependent Current Sources

The schematic symbols that we use for dependent current sources are shown here, of which there are 2 forms:

i. Voltage-dependent current sources

ii. Current-dependent current sources





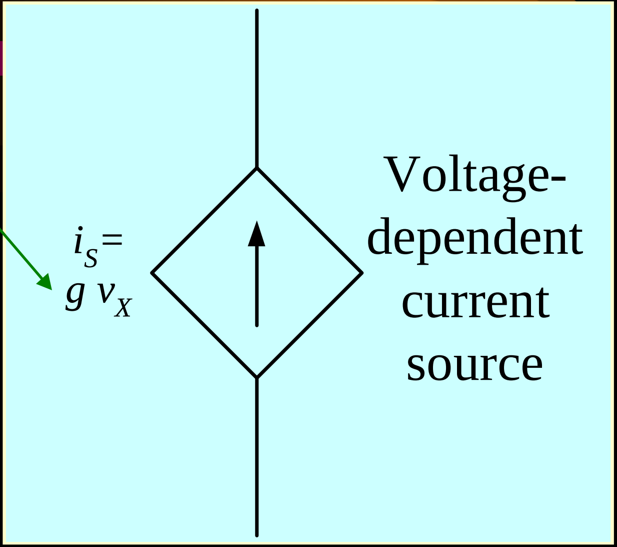
Notes on Schematic Symbols for Dependent Current Sources

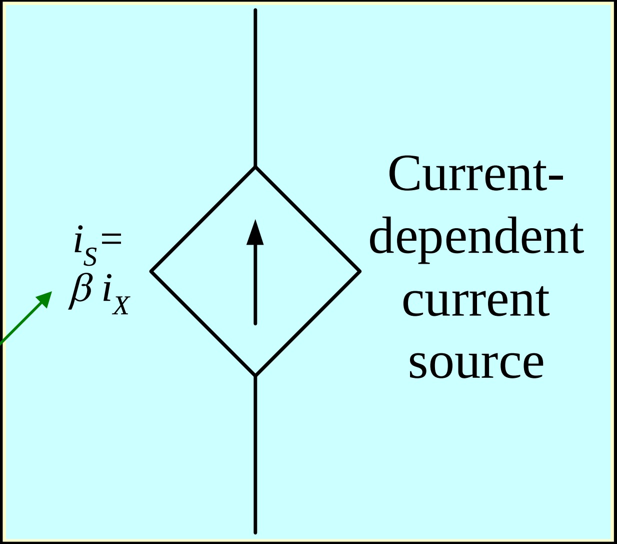
The symbol g is the coefficient of the voltage v_X. It has dimensions of [current/voltage]. For example, it might be 16[A/V] v_X. The v_X is a voltage somewhere in the circuit.

The schematic symbols that we use for dependent current sources are shown here, of which there are 2 forms:

i. Voltage-dependent current sources
ii. Current-dependent current sources

The symbol b is the coefficient of the current i_X. It is dimensionless. For example, it might be 53.7 i_X. The i_X is a current somewhere in the circuit.





Voltage and Current Polarities

- Previously, we have emphasized the importance of reference polarities of currents and voltages.
- Notice that the schematic symbols for the voltage sources and current sources indicate these polarities.
- The voltage sources have a “+” and a “–” to show the voltage reference polarity. The current sources have an arrow to show the current reference polarity.

Dependent Voltage and Current Sources – Coefficients

- Some textbooks use symbols other than the ones we have used here (`m`, `b`, `r`, and `g`). There are no firm standards. We hope this is not confusing!!!

DCE,Gurgaon

UPPERCASE vs lowercase – Part 1

In this course, we use UPPERCASE variables for quantities that do not change with time. For example, resistance, capacitance, and inductance are assumed to be constant in this course, and so are represented as UPPERCASE variables.

- For example, we will have things such as $R_X = 120[\Omega]$ and $C_{23} = 4.76[F]$.

UPPERCASE vs lowercase – Part 2

In this course, we use lowercase variables for quantities that do change with time. For example, voltage, current, energy, and power are assumed to be able to change with time, and so are represented as lowercase variables, with UPPERCASE subscripts.

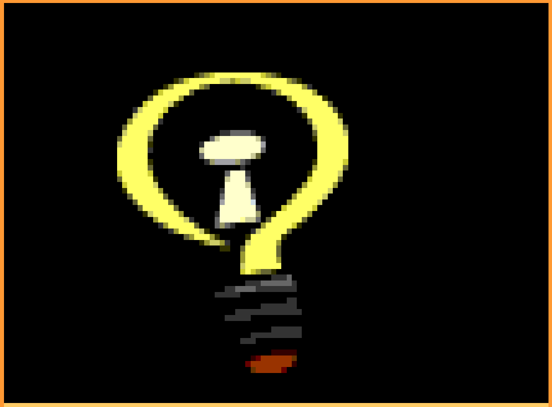
•For example, we will have things such as $v_X = 120[\text{V}]$ and $p_{ABS,TRUCK} = 4.76[\text{W}]$.

Why do we have these dependent sources?

- Students who are new to circuits often question why dependent sources are included. Some students find these to be confusing, and they do add to the complexity of our solution techniques.
- However, there is no way around them. We need dependent sources to be able to model amplifiers, and amplifier-like devices. Amplifiers are crucial in electronics. Therefore, we simply need to understand and be able to work with dependent sources.

Resistors

- A resistor is a two terminal circuit element that has a constant ratio of the voltage across its terminals to the current through its terminals.
- The value of the ratio of voltage to current is the defining characteristic of the resistor.



In many cases a light bulb can be modeled with a resistor.

Resistors – Definition and Units

- A resistor obeys the expression

$$R = \frac{v_R}{i_R}$$

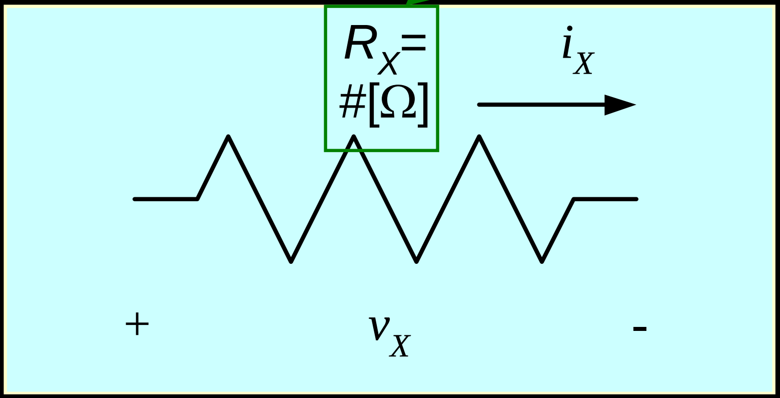
where R is the *resistance*.
- If something obeys this expression, we can think of it, and model it, as a resistor.
- This expression is called Ohm's Law. The unit ([Ohm] or [W]) is named for Ohm, and is equal to a [Volt/Ampere].
- IMPORTANT: use Ohm's Law only on resistors. It does not hold for sources.

To a first-order approximation, the body can modeled as a resistor. Our goal will be to avoid applying large voltages across our bodies, because it results in large currents through our body. This is not good.

Schematic Symbol for Resistors

The schematic symbols that we use for resistors are shown here.

This is intended to indicate that the schematic symbol can be labeled either with a variable, like R_X, or a value, with some number, and units. An example might be 390[W]. It could also be labeled with both.



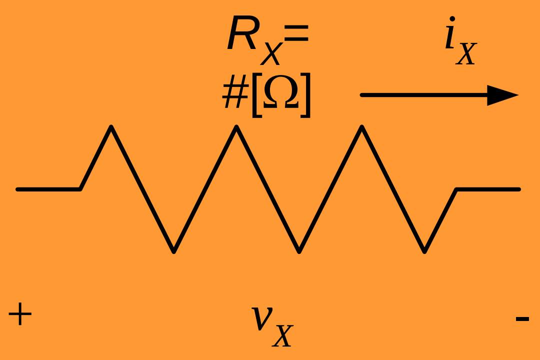
Resistor Polarities

- Previously, we have emphasized the important of reference polarities of current sources and voltages sources. There is no corresponding polarity to a resistor. You can flip it end-for-end, and it will behave the same way.
- However, even in a resistor, direction matters in one sense; we need to have defined the voltage and current in the passive sign convention to use the Ohm's Law equation the way we have it listed here.

Getting the Sign Right with Ohm's Law

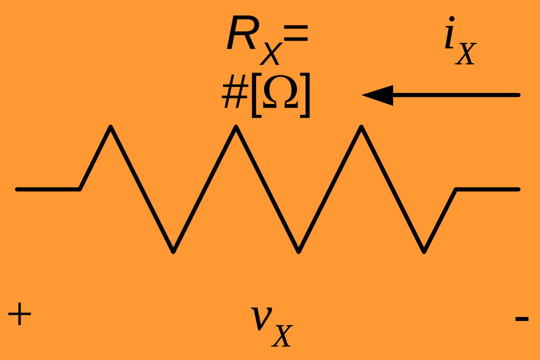
If the reference current is in the direction of the reference voltage drop (Passive Sign Convention), then…

$$R_X = \frac{v_X}{i_X}$$



If the reference current is in the direction of the reference voltage rise (Active Sign Convention), then…

$$R_X = -\frac{v_X}{i_X}$$



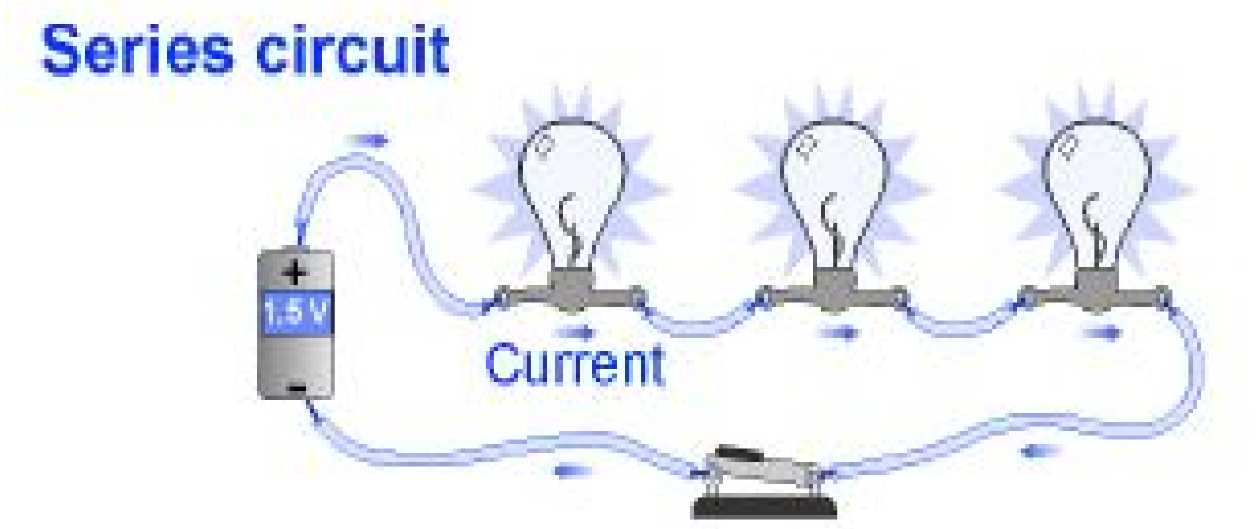
*Why do we have to worry about the sign in **Everything**?*

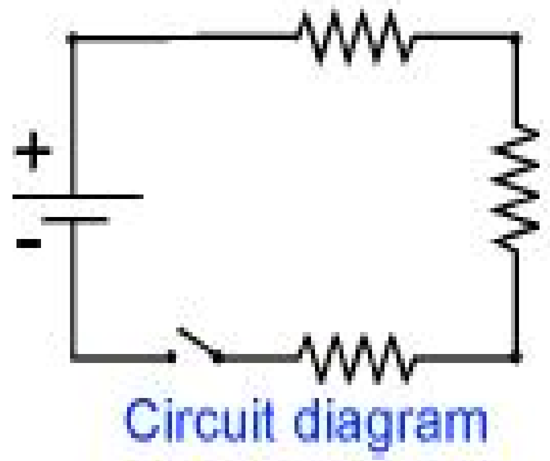
- This is one of the central themes in circuit analysis. The polarity, and the sign that goes with that polarity, matters. The key is to find a way to get the sign correct every time.
- This is why we need to define reference polarities for every voltage and current.
- This is why we need to take care about what relationship we have used to assign reference polarities (passive sign convention and active sign convention).

An analogy: Suppose I was going to give you $10,000. This would probably be fine with you. However, it will matter a great deal which direction the money flows. You will care a great deal about the sign of the $10,000 in this transaction. If I give you -$10,000, it means that you are giving $10,000 to me. This would probably not be fine with you!

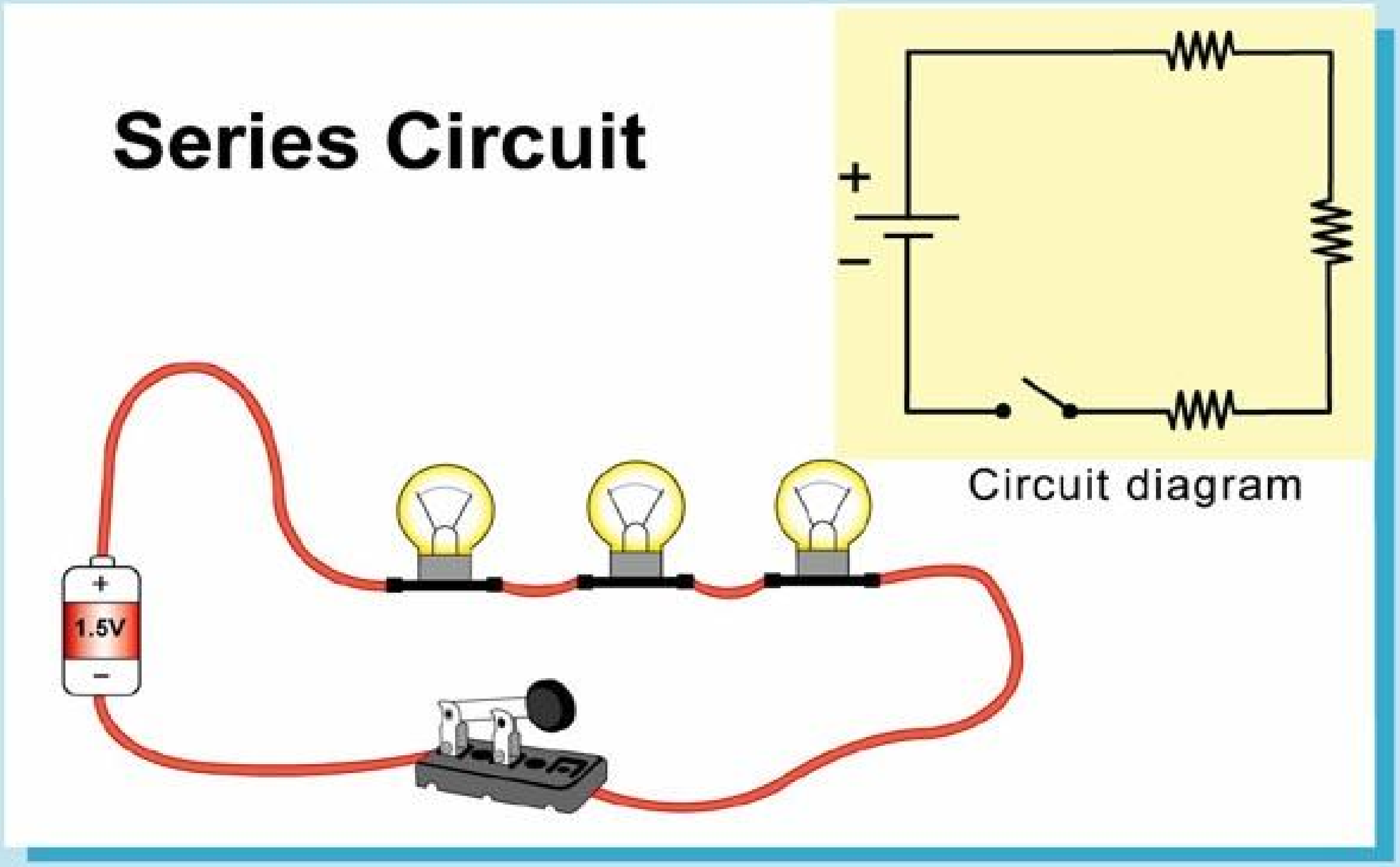
Series and Parallel Circuits

- In series circuits, current can only take one path.
- The amount of current is the same at all points in a series circuit.

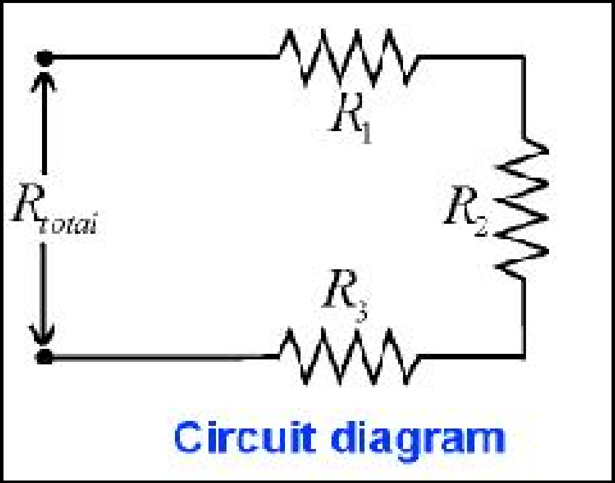




DCE,Gurgaon



Adding resistances in series

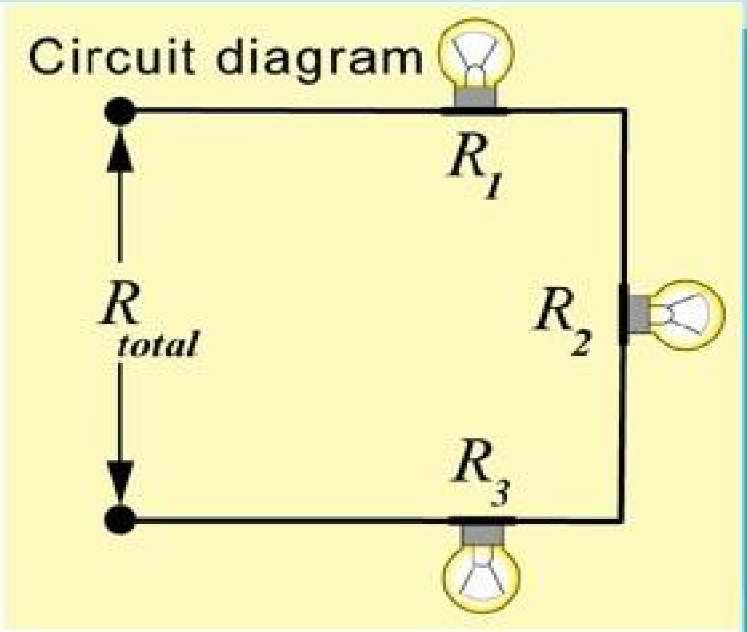


- Each resistance in a series circuit adds to the total resistance of the circuit.

$$\underset{\text{Total resistance (ohms)}}{R_{total}} = \underbrace{R_1 + R_2 + R_3 \ldots}_{\text{Individual resistances (W)}}$$

DCE,Gurgaon

Adding Resistances in Series



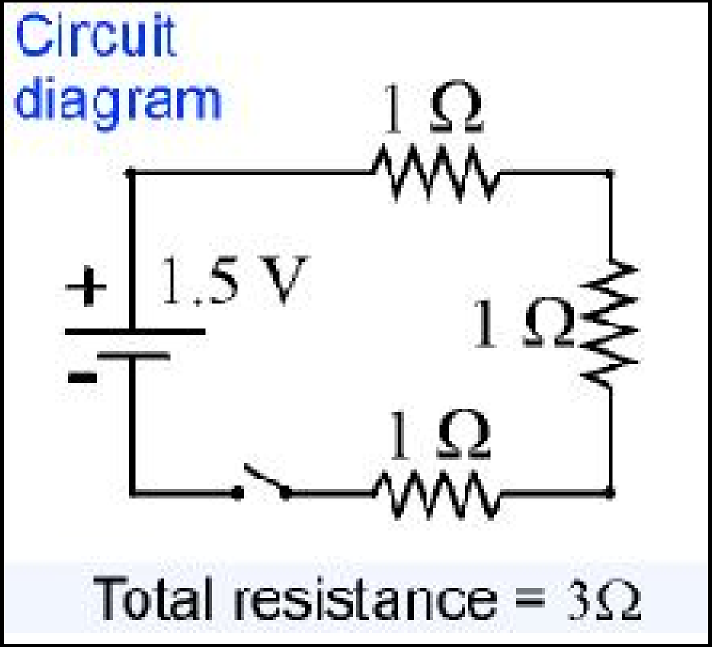
$$R_{total} = \underbrace{R_1 + R_2 + R_3}_{\text{Individual resistances } (\Omega)} + \cdots$$

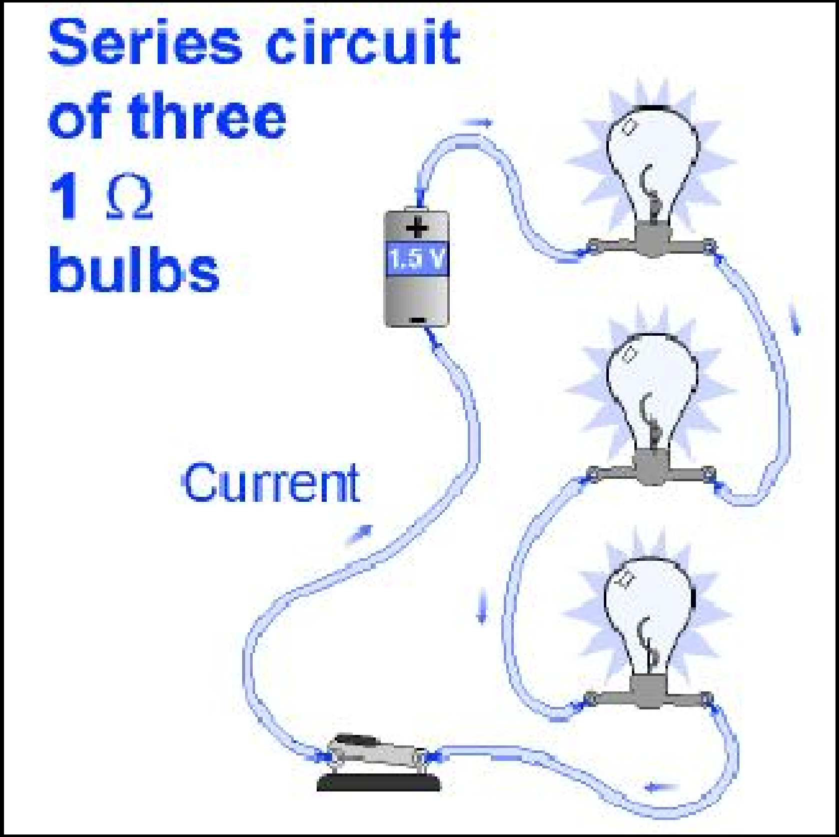
Total resistance (Ω)

Individual resistances (Ω)

Total resistance in a series circuit

- Light bulbs, resistors, motors, and heaters usually have much greater resistance than wires and batteries.

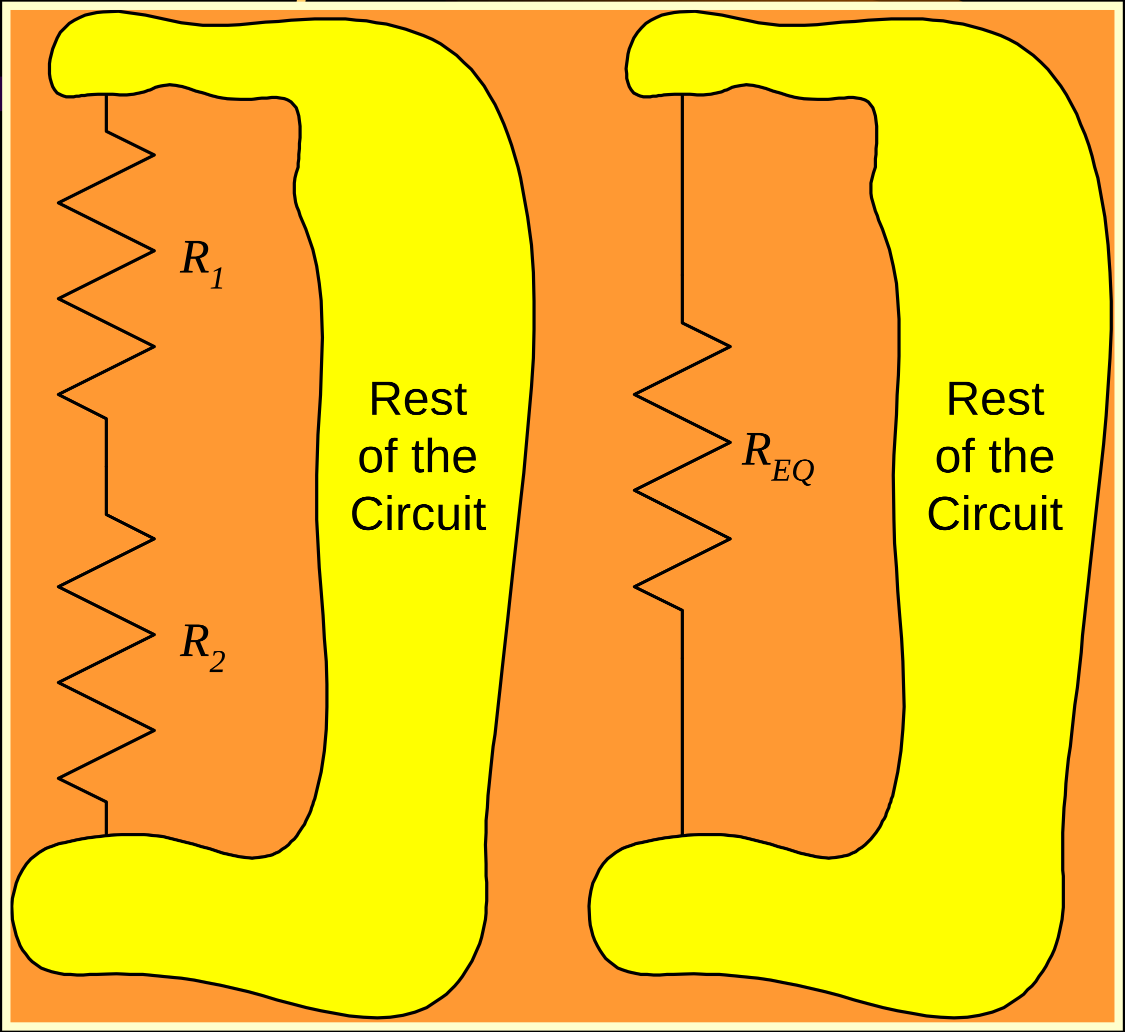




Series Resistors Equivalent Circuits

Two series resistors, R_1 and R_2, can be replaced with an equivalent circuit with a single resistor R_{EQ}, as long as

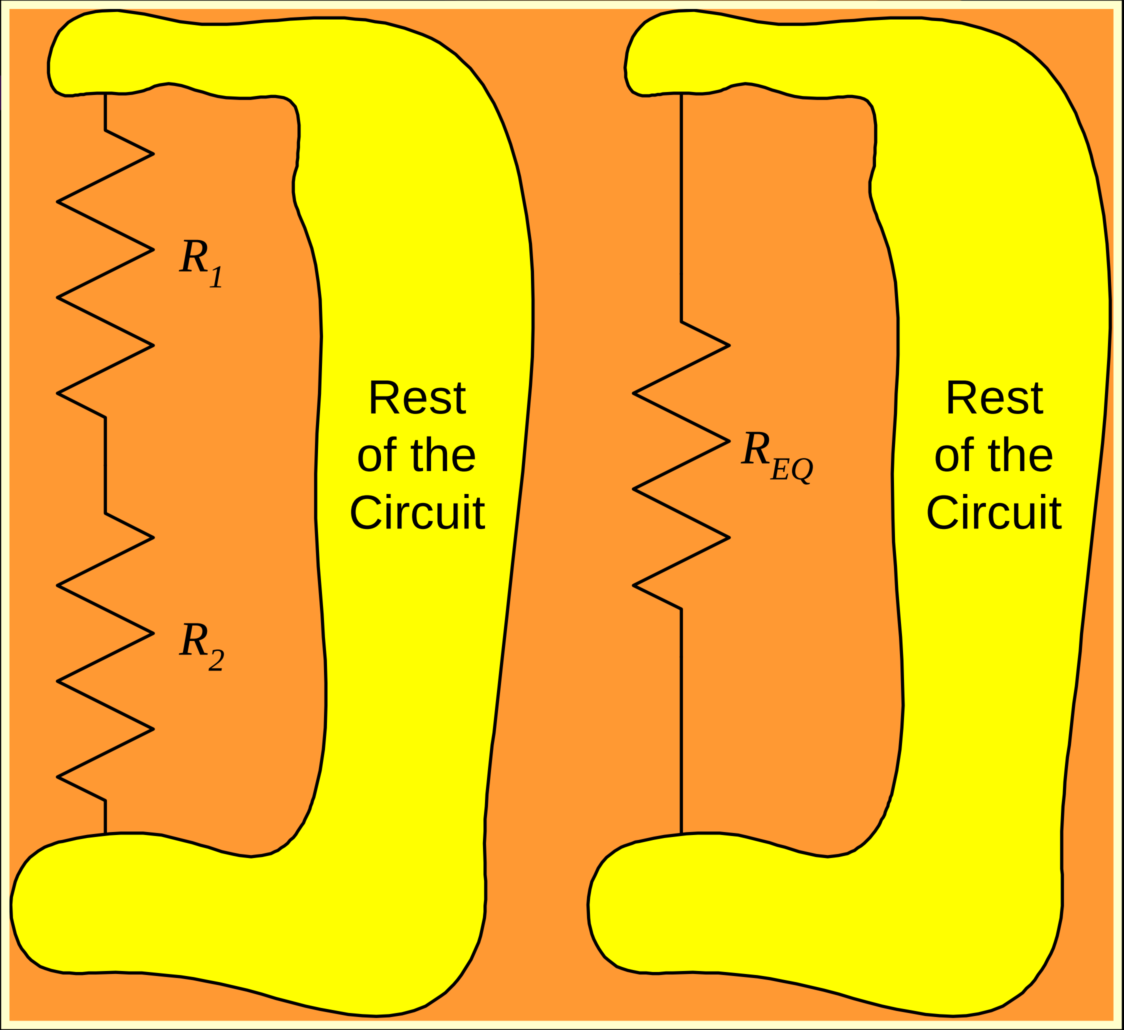
$$R_{EQ} = R_1 + R_2 .$$



More than 2 Series Resistors

This rule can be extended to more than two series resistors. In this case, for N series resistors, we have

$$R_{EQ} = R_1 + R_2 + \ldots + R_N .$$

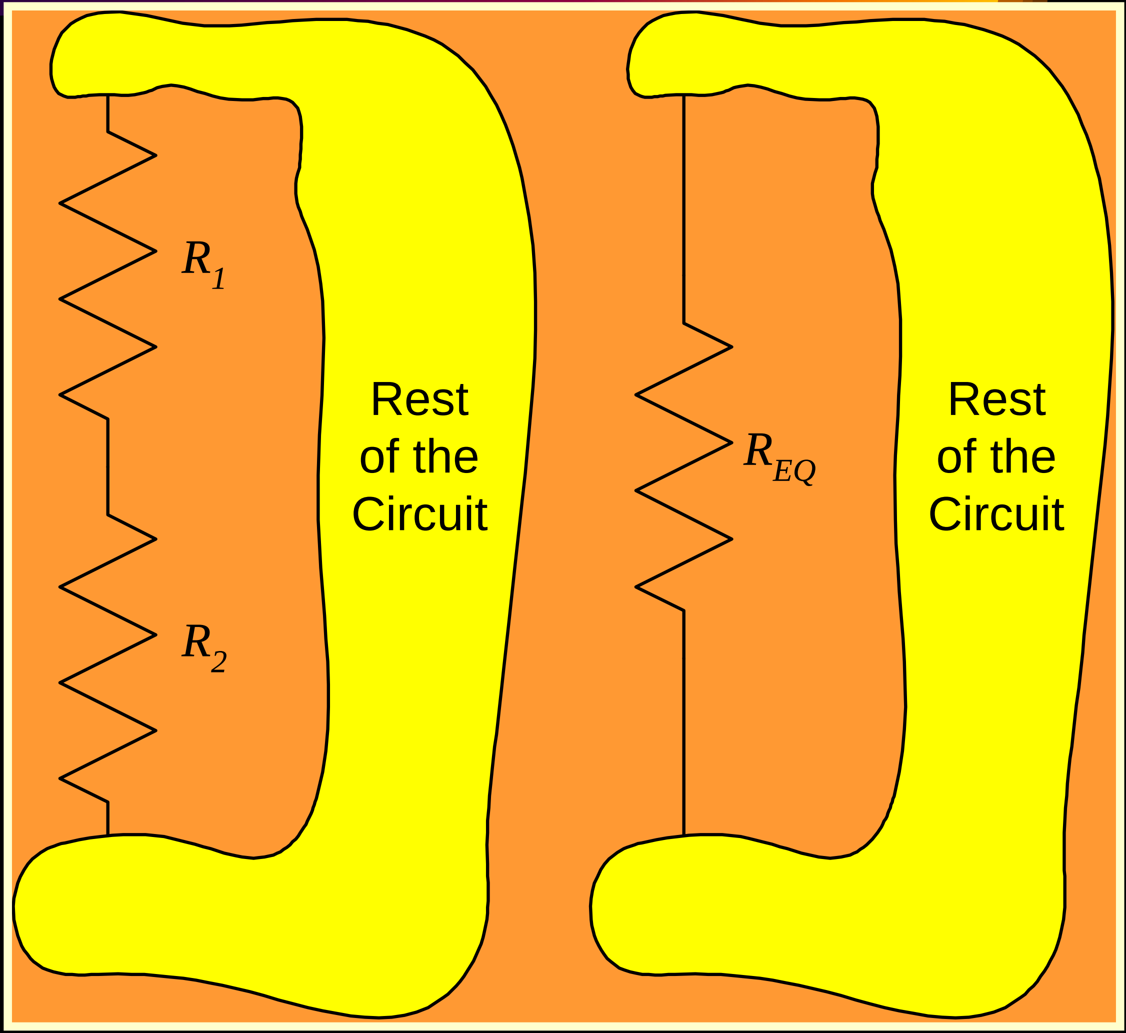


Series Resistors Equivalent Circuits: A Reminder

Two series resistors, R_1 and R_2, can be replaced with an equivalent circuit with a single resistor R_{EQ}, as long as

$$R_{EQ} = R_1 + R_2.$$

Remember that these two equivalent circuits are equivalent only with respect to the circuit connected to them. (In yellow here.)

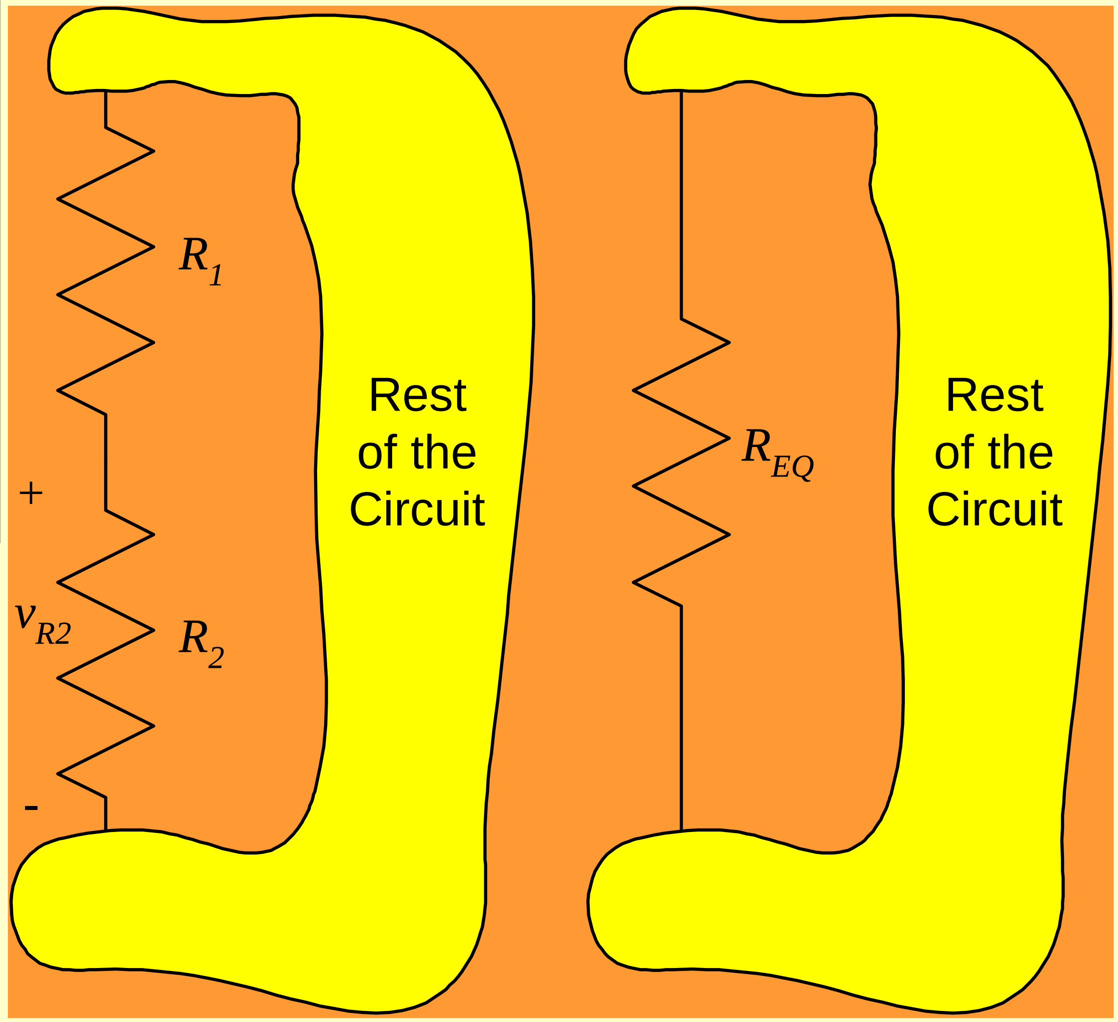


Series Resistors Equivalent Circuits: Another Reminder

Resistors R_1 and R_2 can be replaced with a single resistor R_{EQ}, as long as

$$R_{EQ} = R_1 + R_2 .$$

Remember that these two equivalent circuits are equivalent only with respect to the circuit connected to them. (In yellow here.) ***The voltage v_{R2} does not exist in the right hand equivalent.***



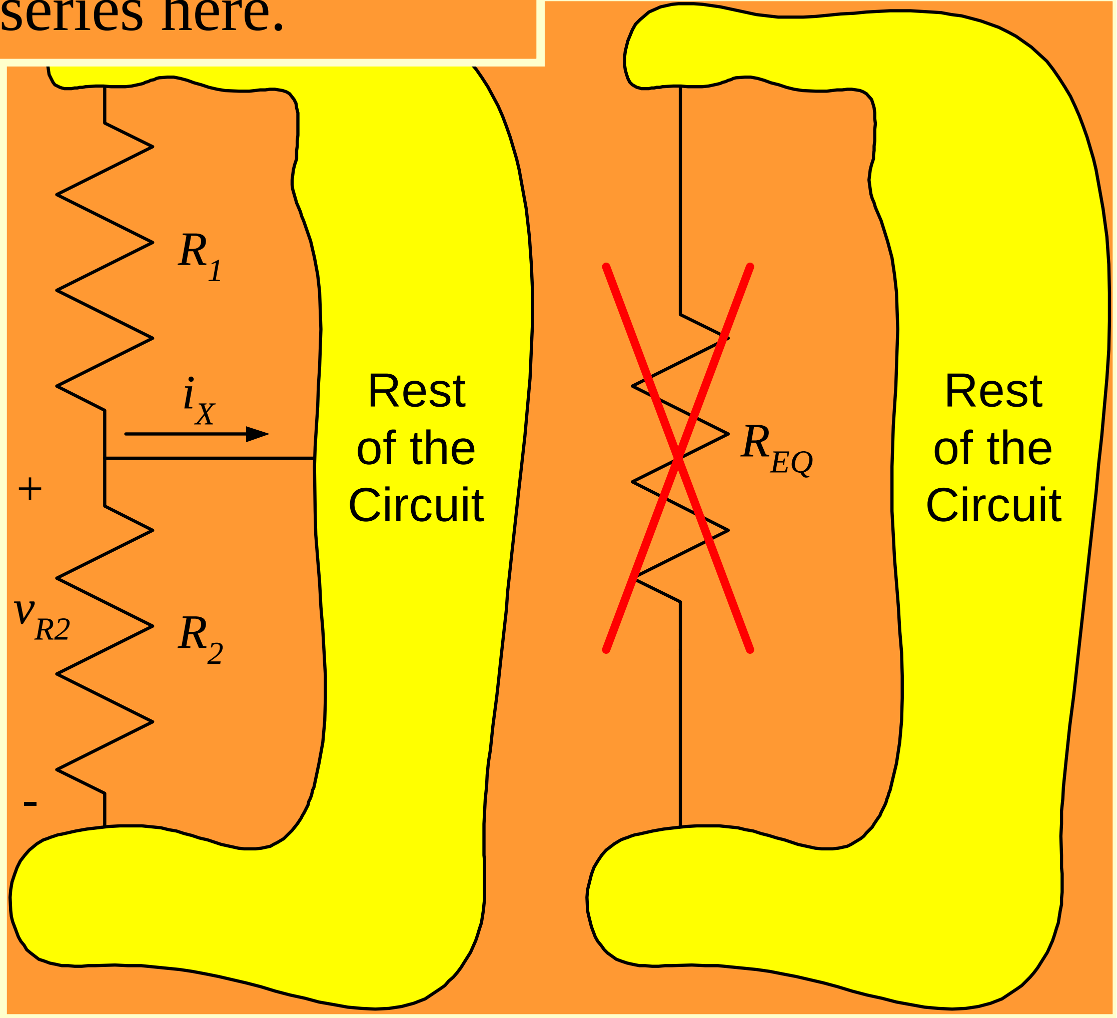
The Resistors Must be in Series

Resistors R_1 and R_2 can be replaced with a single resistor R_{EQ}, as long as

$$R_{EQ} = R_1 + R_2.$$

Remember also that these two equivalent circuits are equivalent ***only when R_1 and R_2 are in series***. If there is something connected to the node between them, and it carries current, ($i_X \neq 0$) then this does not work.

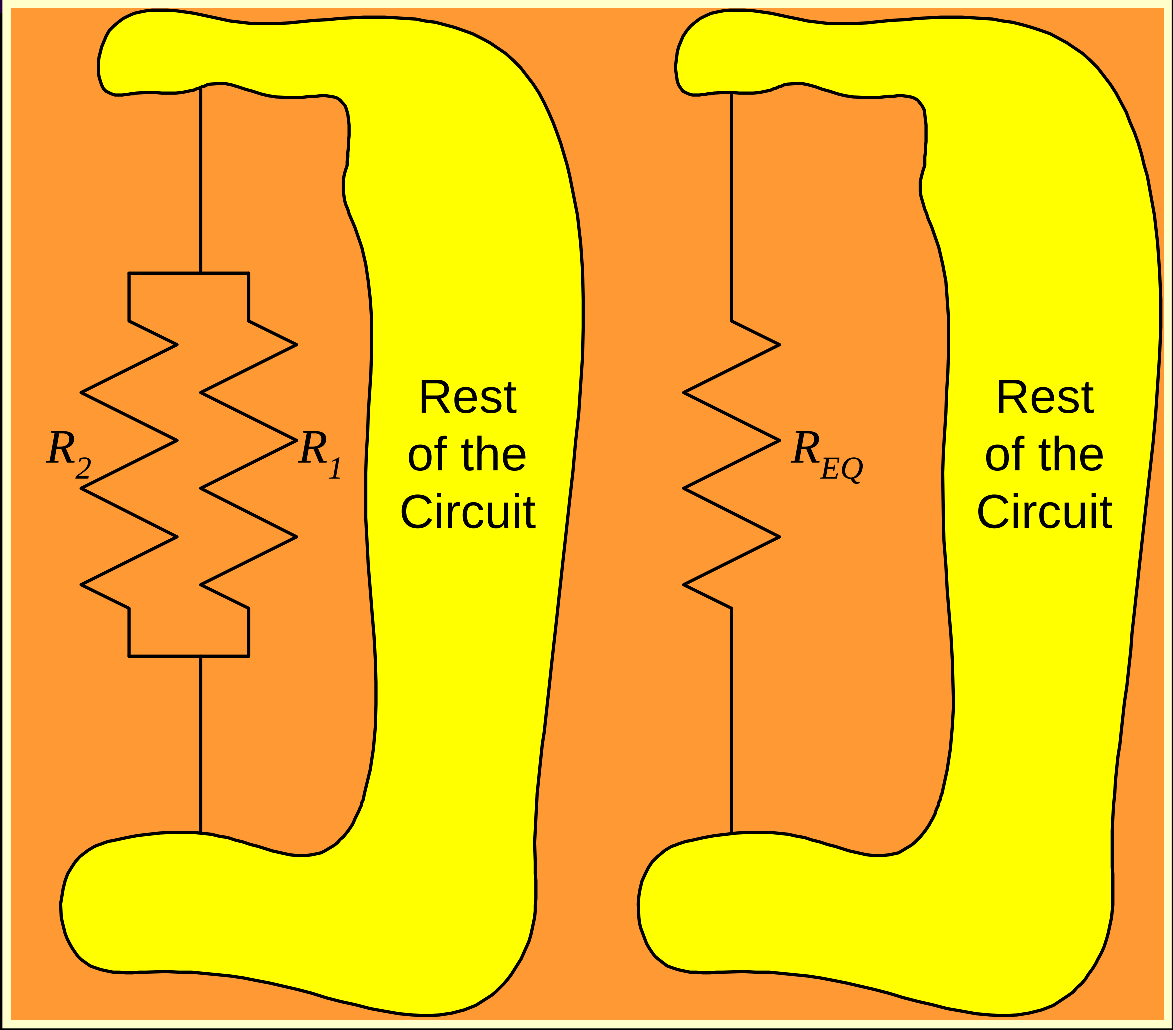
R_1 and R_2 are not in series here.



Parallel Resistors Equivalent Circuits

Two parallel resistors, R_1 and R_2, can be replaced with an equivalent circuit with a single resistor R_{EQ}, as long as

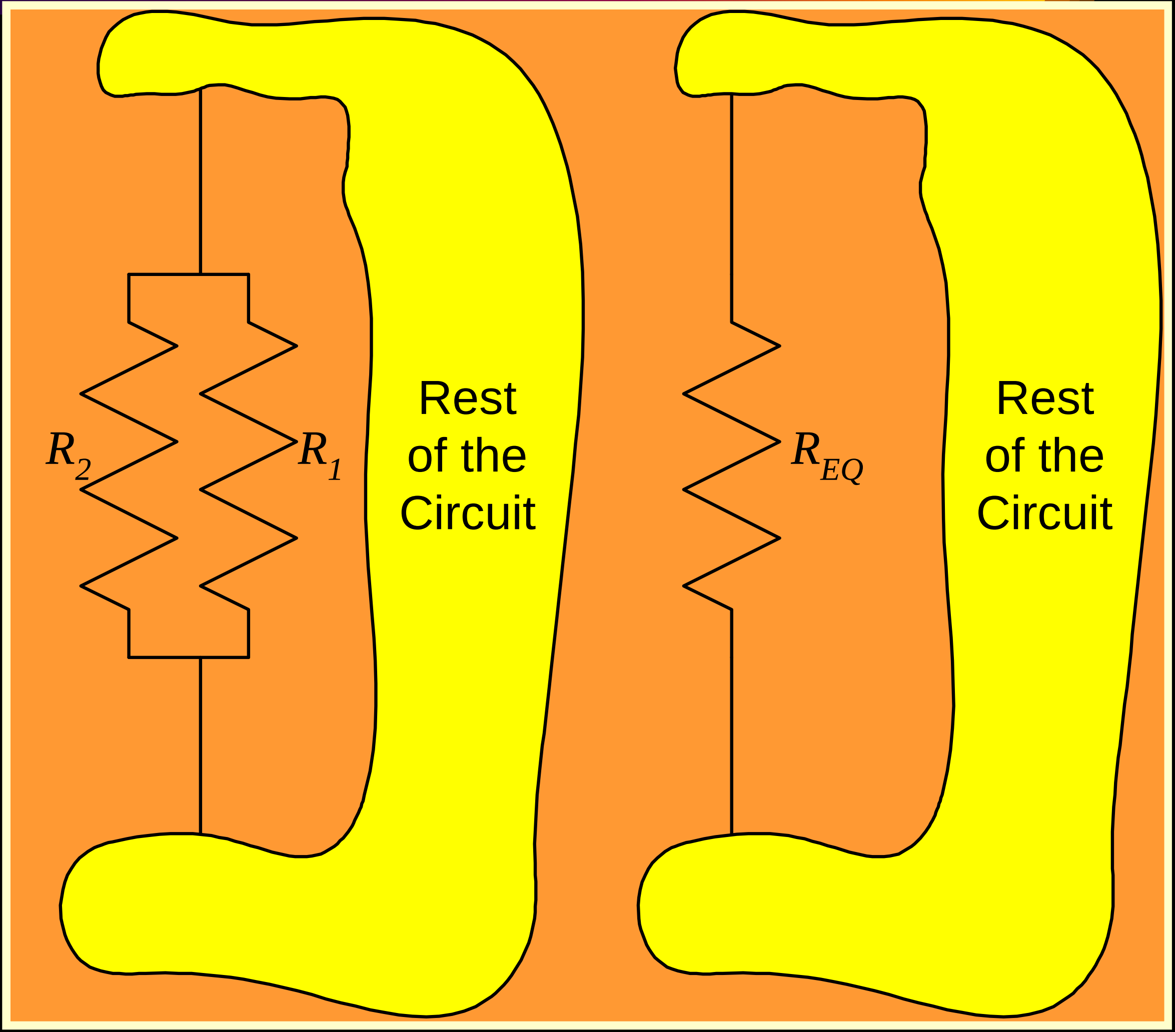
$$\frac{1}{R_{EQ}} = \frac{1}{R_1} + \frac{1}{R_2}.$$



More than 2 Parallel Resistors

This rule can be extended to more than two parallel resistors. In this case, for N parallel resistors, we have

$$\frac{1}{R_{EQ}} = \frac{1}{R_1} + \frac{1}{R_2} + \ldots + \frac{1}{R_N}.$$

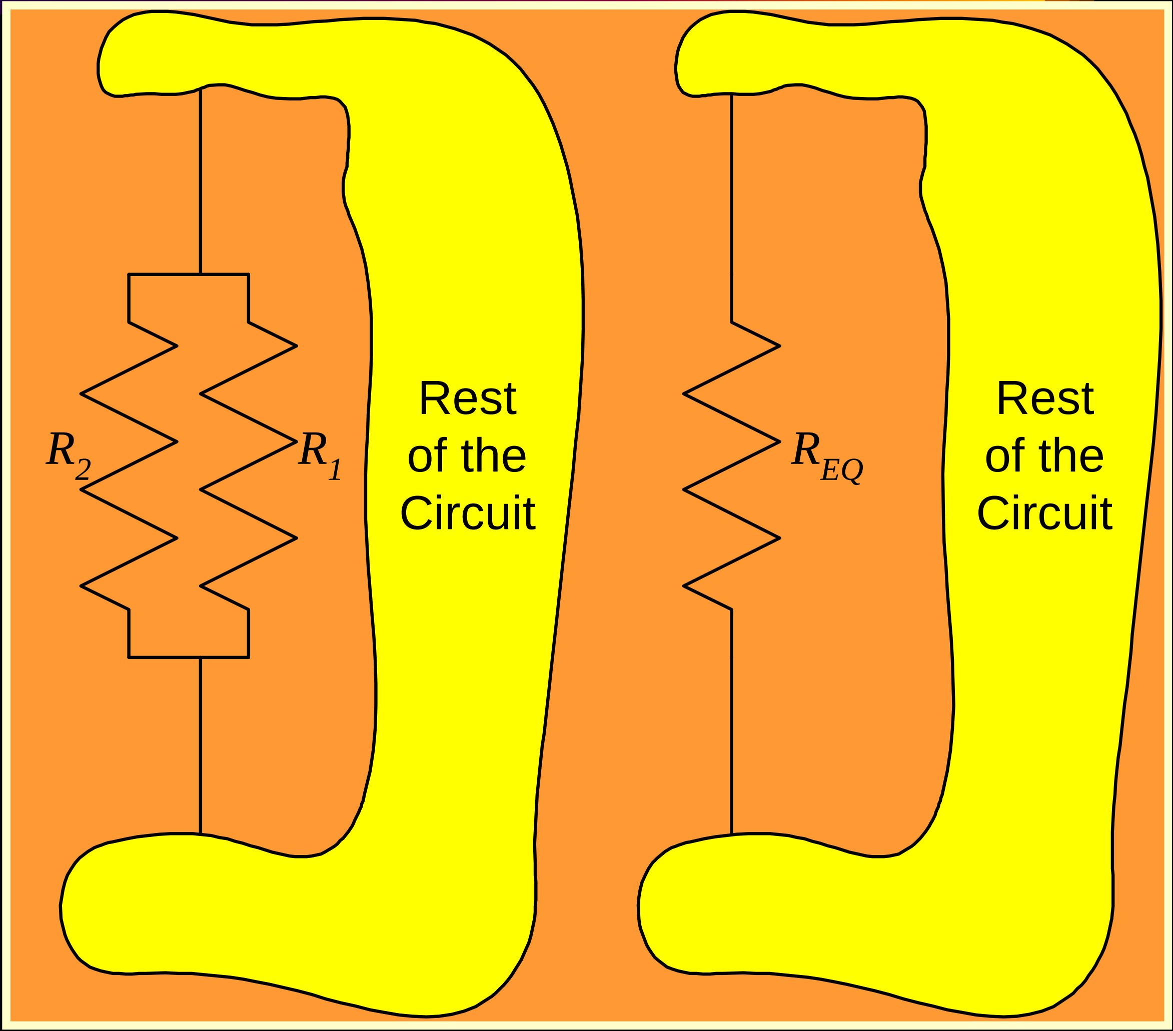


Parallel Resistors Notation

We have a special notation for this operation. When two things, Thing1 and Thing2, are in parallel, we write Thing1||Thing2 to indicate this. So, we can say that

$$\text{if } \frac{1}{R_{EQ}} = \frac{1}{R_1} + \frac{1}{R_2},$$

$$\text{then } R_{EQ} = R_1 \,||\, R_2.$$

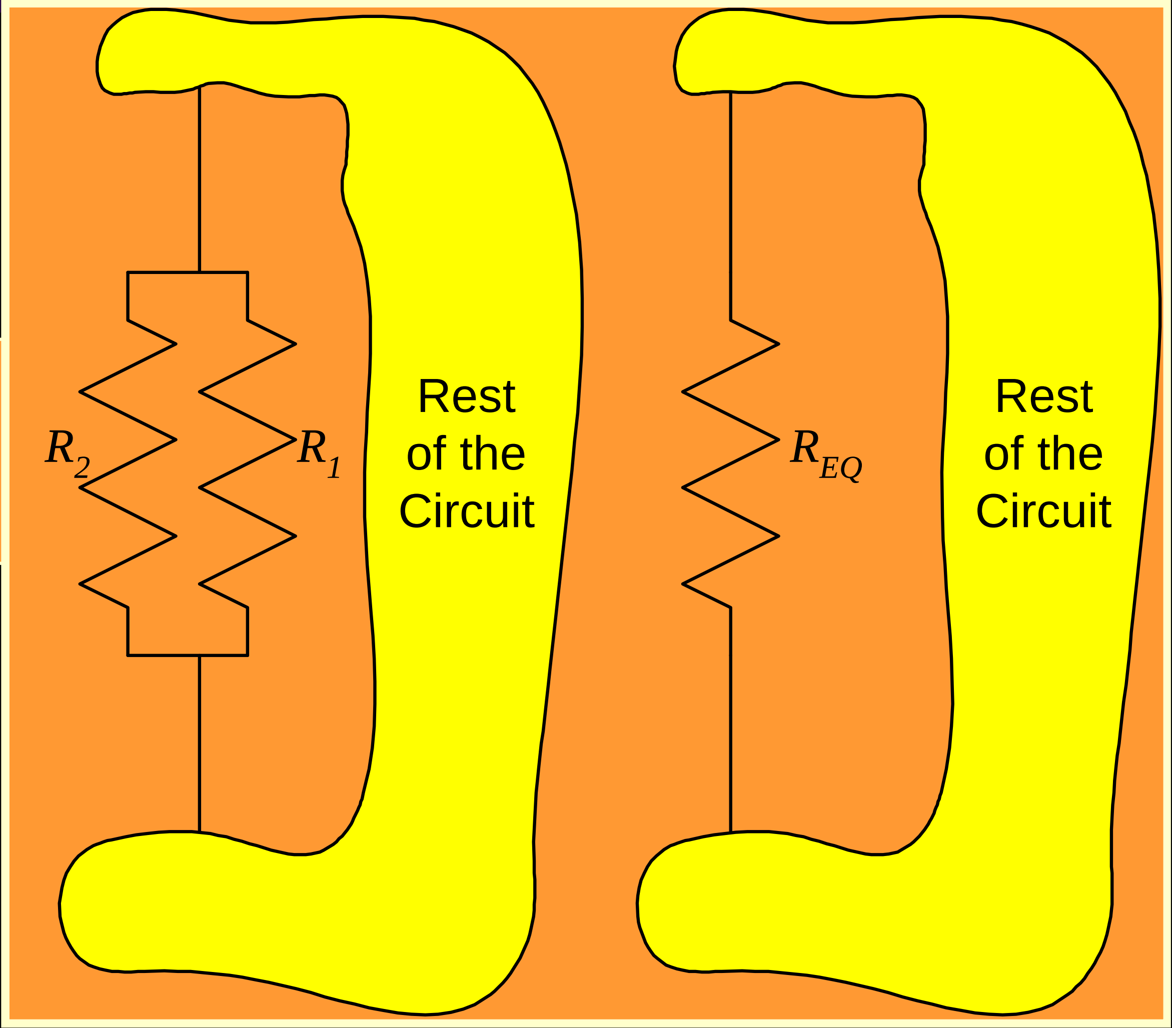


Parallel Resistor Rule for 2 Resistors

When there are only two resistors, then you can perform the algebra, and find that

$$R_{EQ} = R_1 \parallel R_2 = \frac{R_1 R_2}{R_1 + R_2}.$$

This is called the product-over-sum rule for parallel resistors. Remember that the product-over-sum rule ***only works for two resistors***, not for three or more.

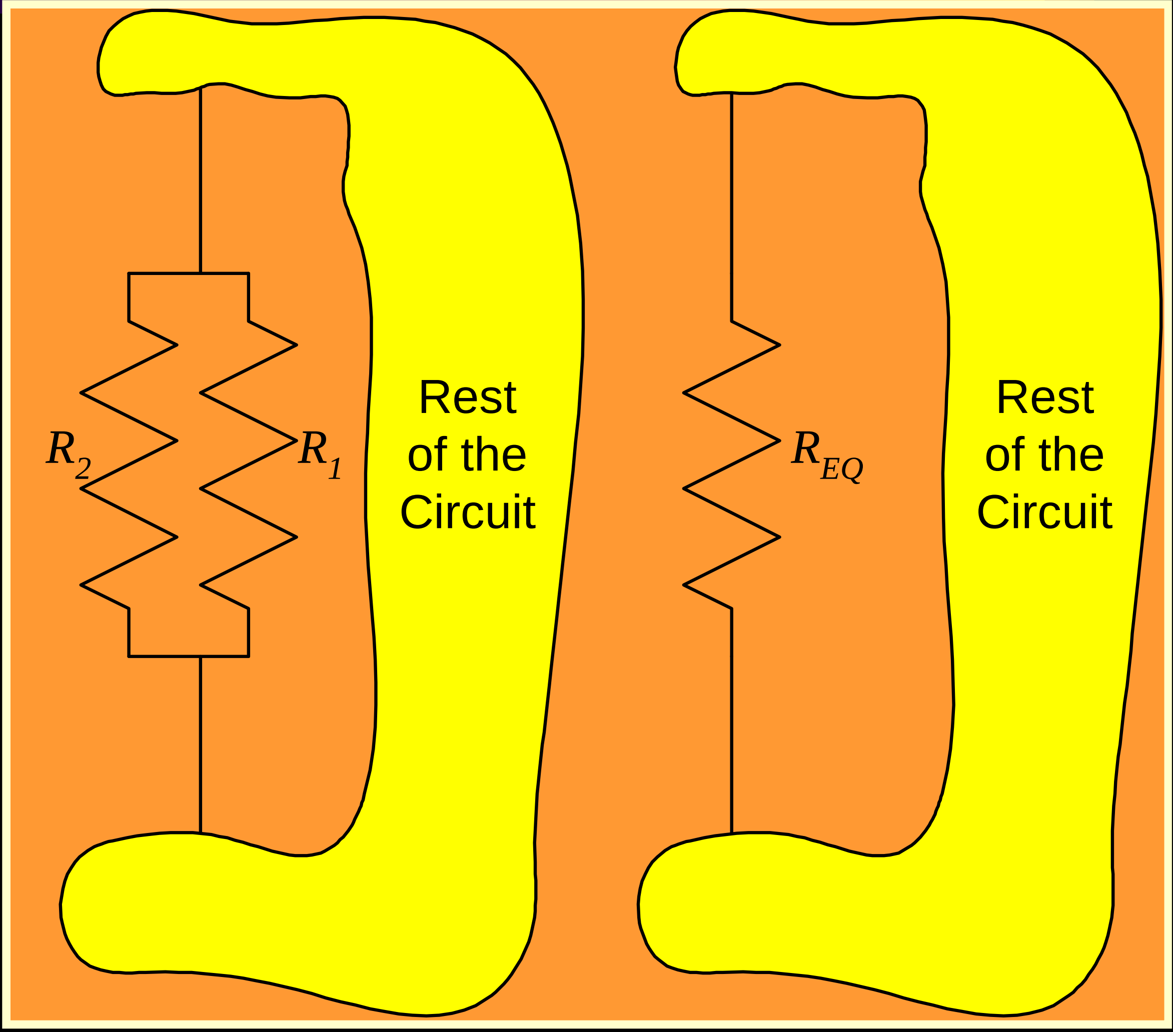


Parallel Resistors Equivalent Circuits: A Reminder

Two parallel resistors, R_1 and R_2, can be replaced with a single resistor R_{EQ}, as long as

$$\frac{1}{R_{EQ}} = \frac{1}{R_1} + \frac{1}{R_2}.$$

Remember that these two equivalent circuits are equivalent only with respect to the circuit connected to them. (In yellow here.)

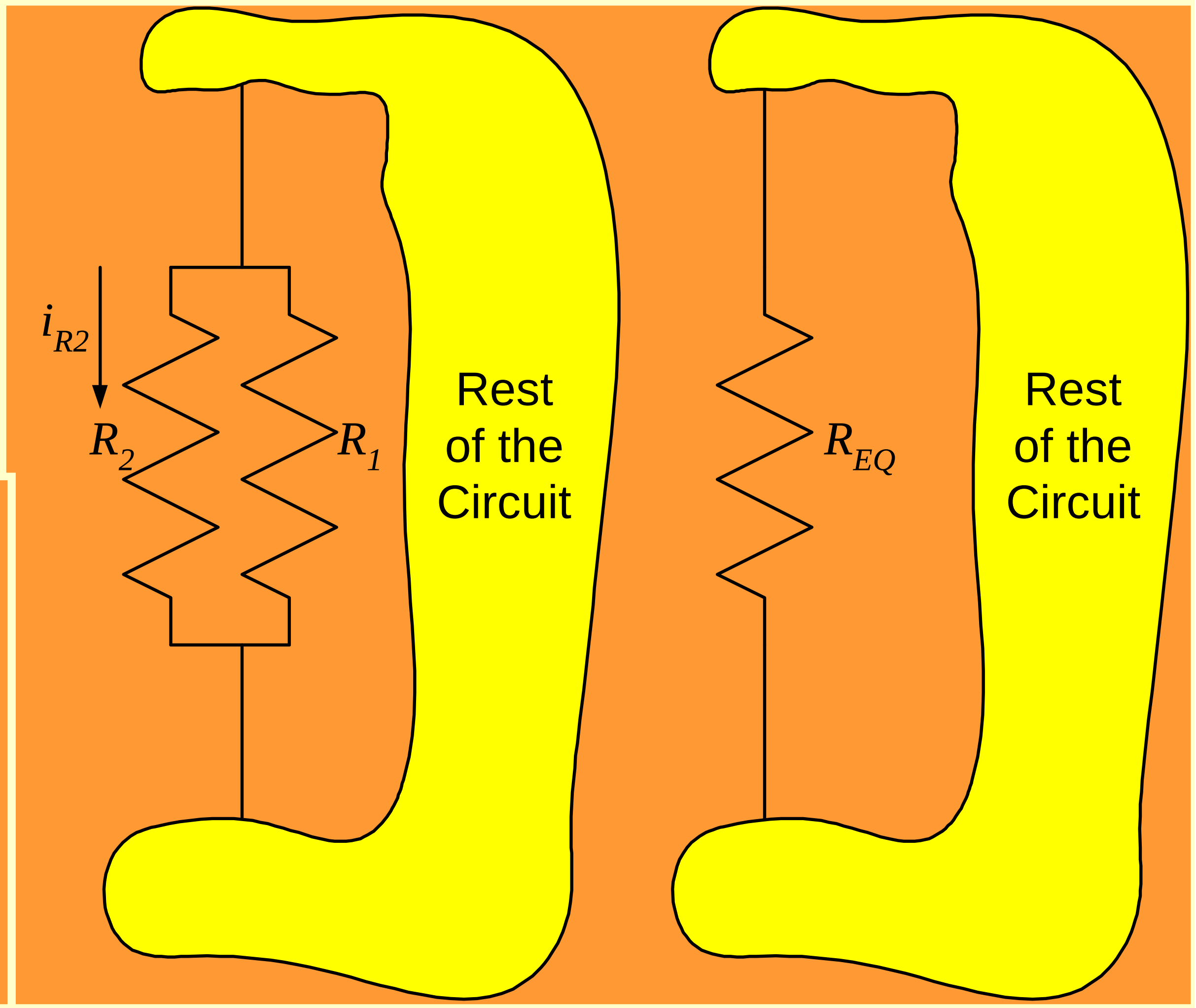


Parallel Resistors Equivalent Circuits: Another Reminder

Two parallel resistors, R_1 and R_2, can be replaced with R_{EQ}, as long as

$$\frac{1}{R_{EQ}} = \frac{1}{R_1} + \frac{1}{R_2}.$$

Remember that these two equivalent circuits are equivalent only with respect to the circuit connected to them. (In yellow here.) ***The current i_{R2} does not exist in the right hand equivalent.***



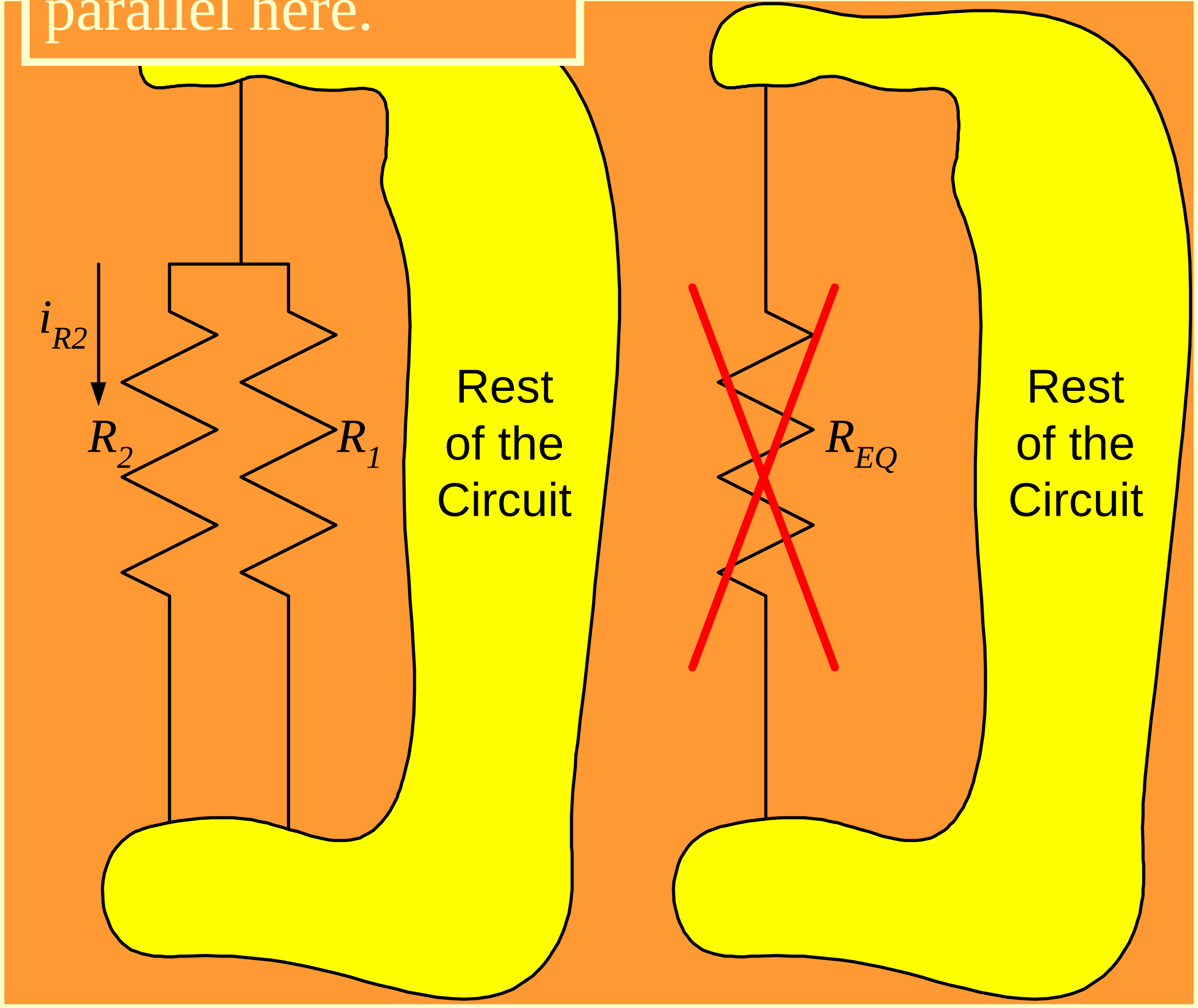
The Resistors Must be in Parallel

Go back to Overview slide.

Two parallel resistors, R_1 and R_2, can be replaced with R_{EQ}, as long as

$$\frac{1}{R_{EQ}} = \frac{1}{R_1} + \frac{1}{R_2}.$$

R_1 and R_2 are not in parallel here.



Remember also that these two equivalent circuits are equivalent ***only when R_1 and R_2 are in parallel***. If the two terminals of the resistors are not connected together, then this does not work.

Why are we doing this? Isn’t all this obvious?

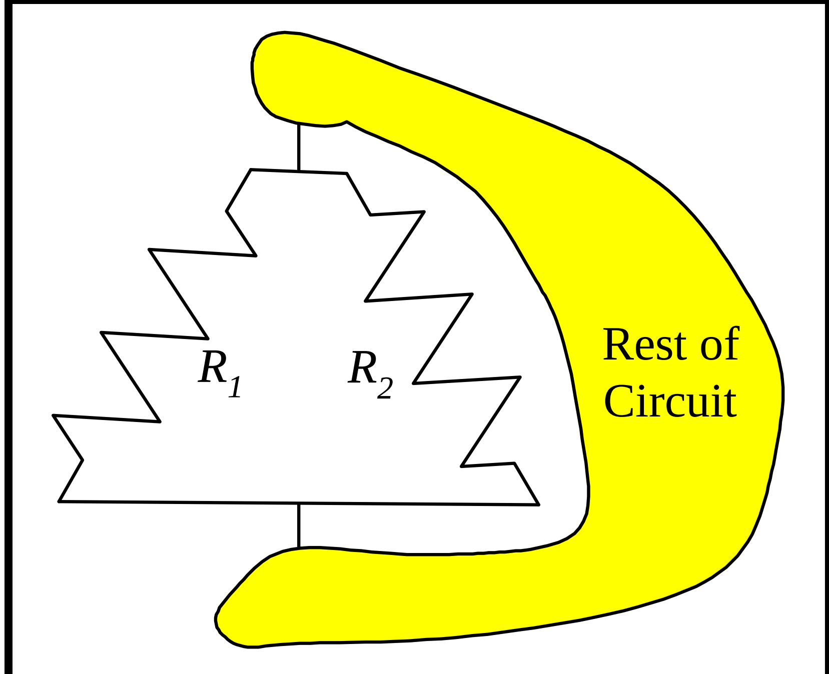
- This is a good question.
- Indeed, most students come to the study of engineering circuit analysis with a little background in circuits. Among the things that they believe that they do know is the concept of series and parallel.
- However, once complicated circuits are encountered, the simple rules that some students have used to identify series and parallel combinations can fail. We need rules that will always work.

Why It Isn't Obvious

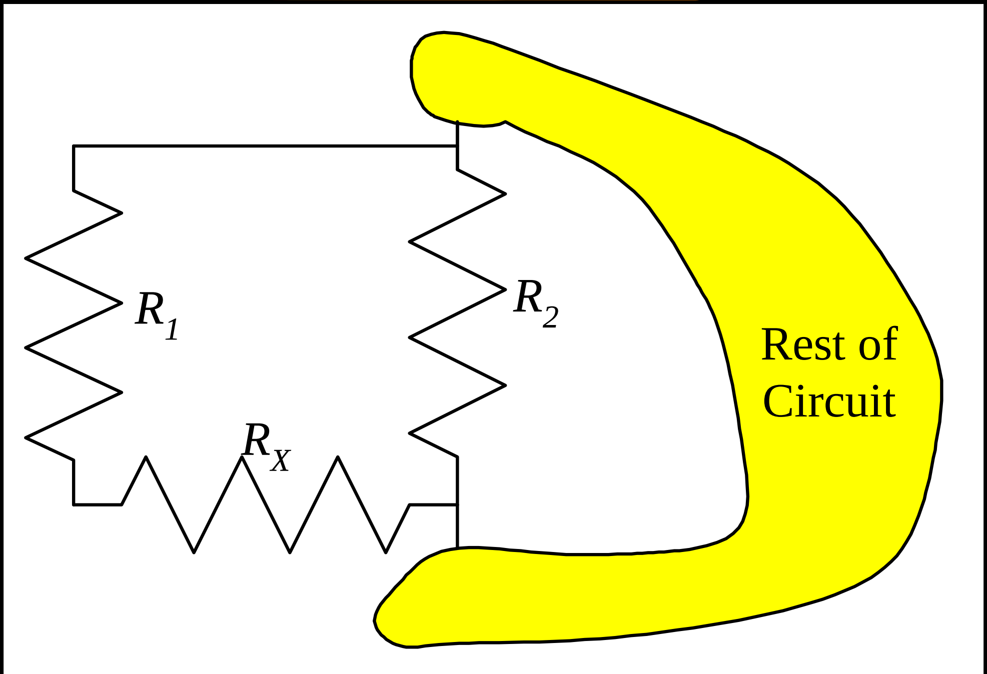
- The problems for students in many cases that they identify series and parallel by the orientation and position of the resistors, and not by the way they are connected.
- In the case of parallel resistors, the resistors do not have to be drawn "parallel", that is, along lines with the same slope. The angle does not matter. Only the nature of the connection matters.
- In the case of series resistors, they do not have to be drawn along a single line. The alignment does not matter. Only the nature of the connection matters.

Examples (Parallel)

- Some examples are given here.



R_1 and R_2 are in parallel

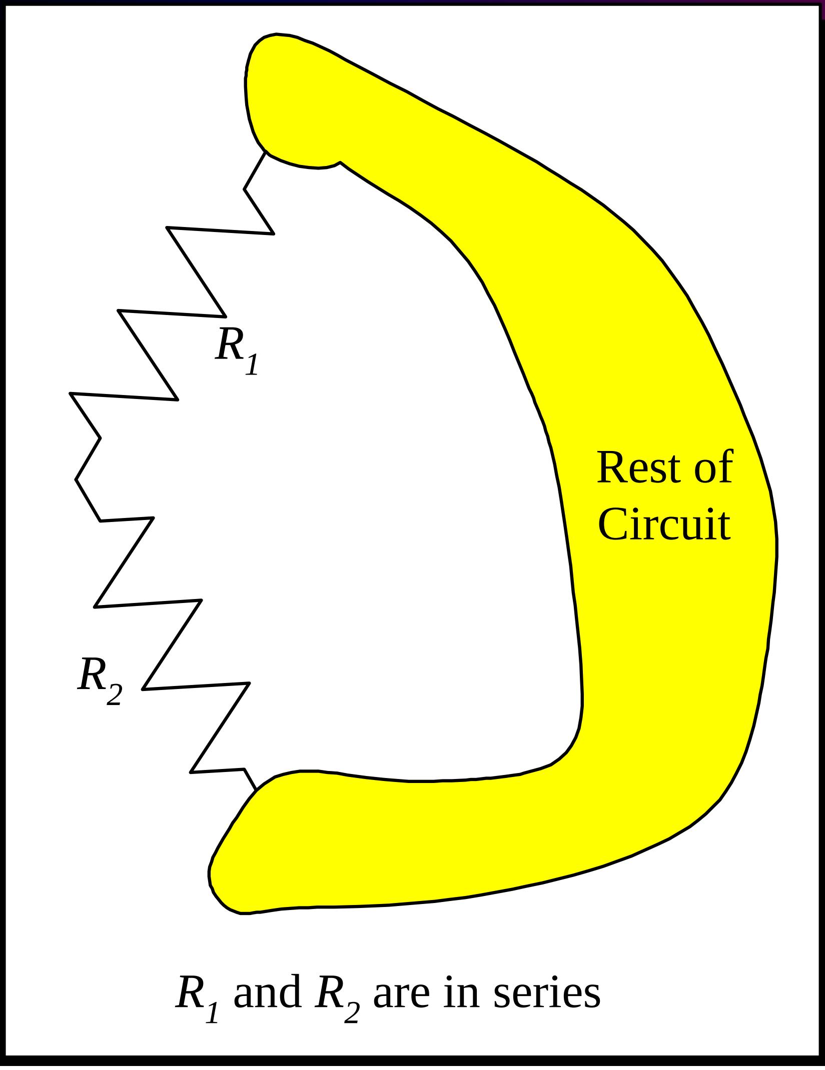


R_1 and R_2 are ***not*** in parallel

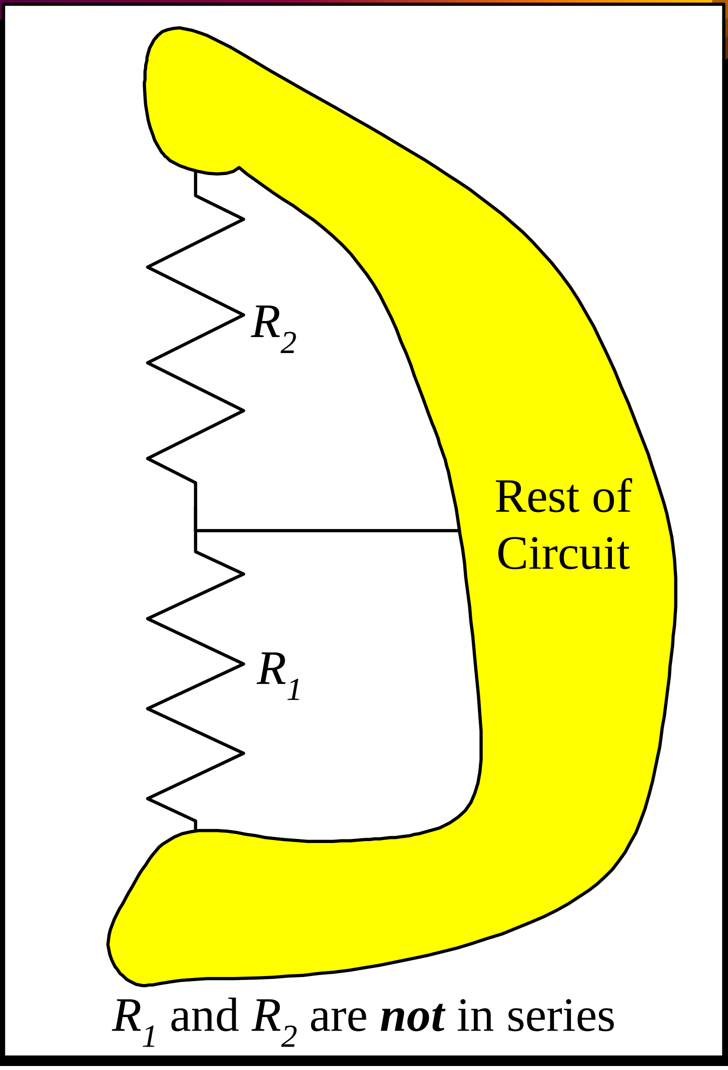
DCE,Gurgaon

Examples (Series)

- Some more examples are given here.



R_1 and R_2 are in series



R_1 and R_2 are ***not*** in series

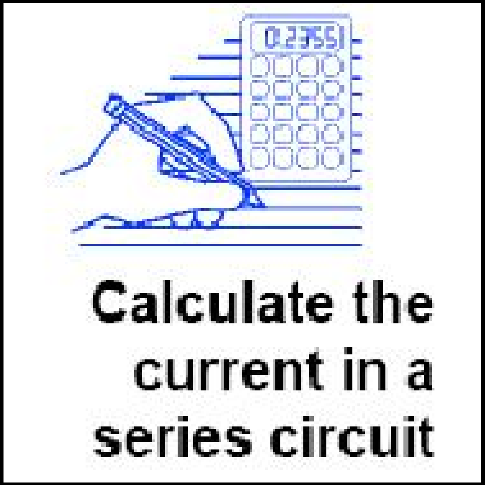
How do we use equivalent circuits?

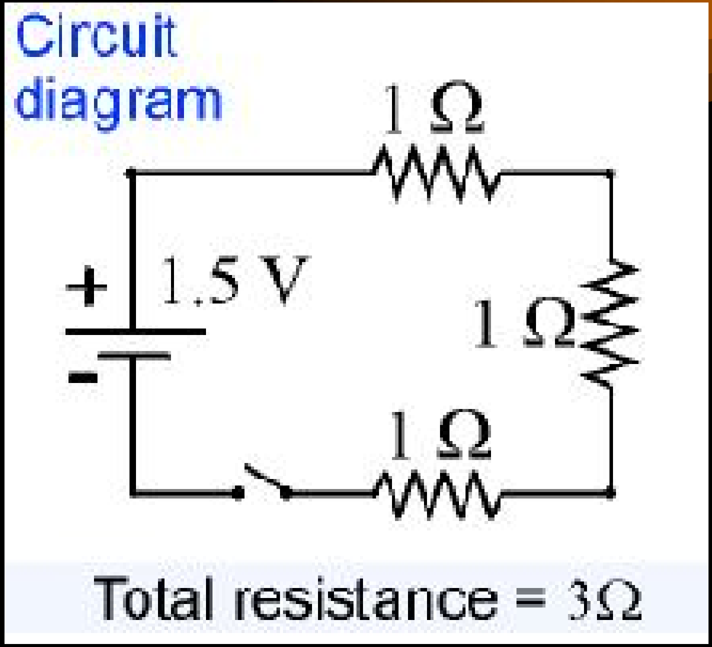
- This is yet another good question.

- We will use these equivalents to simplify circuits, making them easier to solve. Sometimes, equivalent circuits are used in other ways. In some cases, one equivalent circuit is not simpler than another; rather one of them fits the needs of the particular circuit better.

- The key point is this: Equivalent circuits are used throughout circuits and electronics. We need to use them correctly. ***Equivalent circuits are equivalent only with respect to the circuit outside them.***

Calculate Current

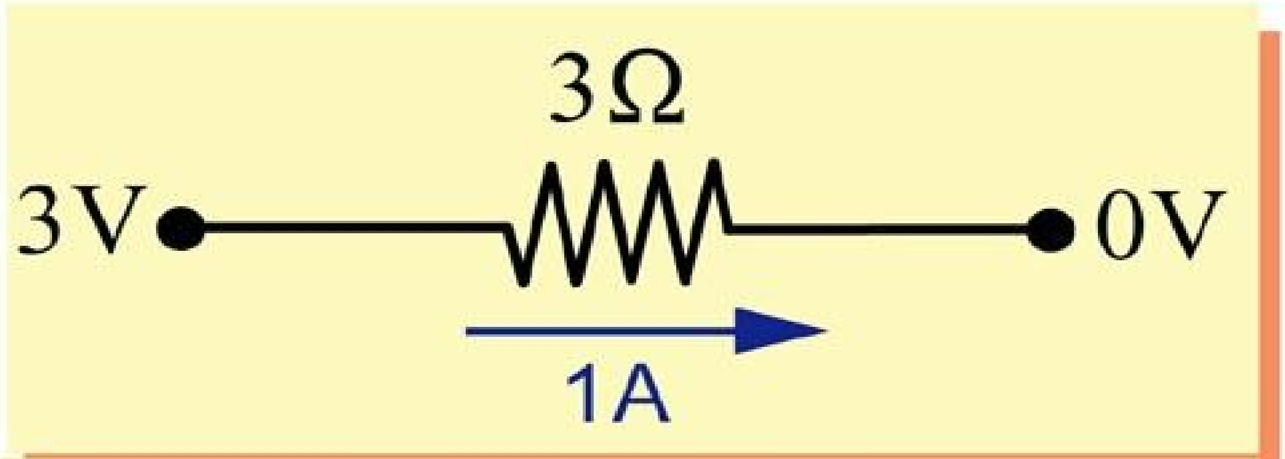




- How much current flows in a circuit with a 1.5-volt battery and three 1 ohm resistances (bulbs) in series?

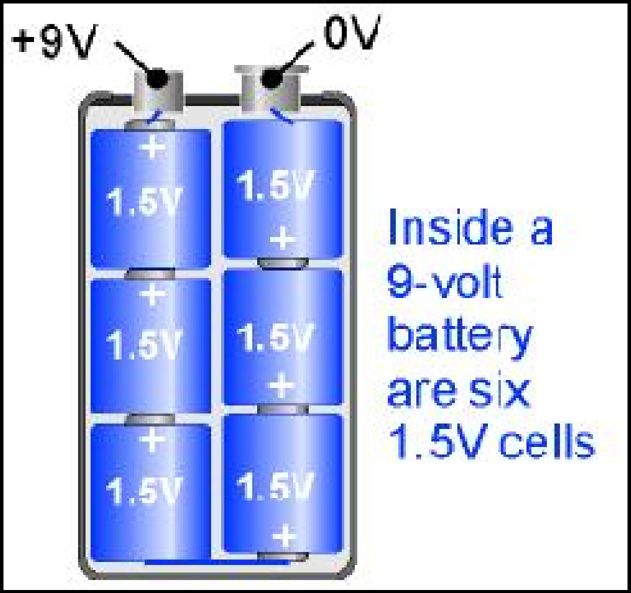
DCE,Gurgaon

Ohm's Law



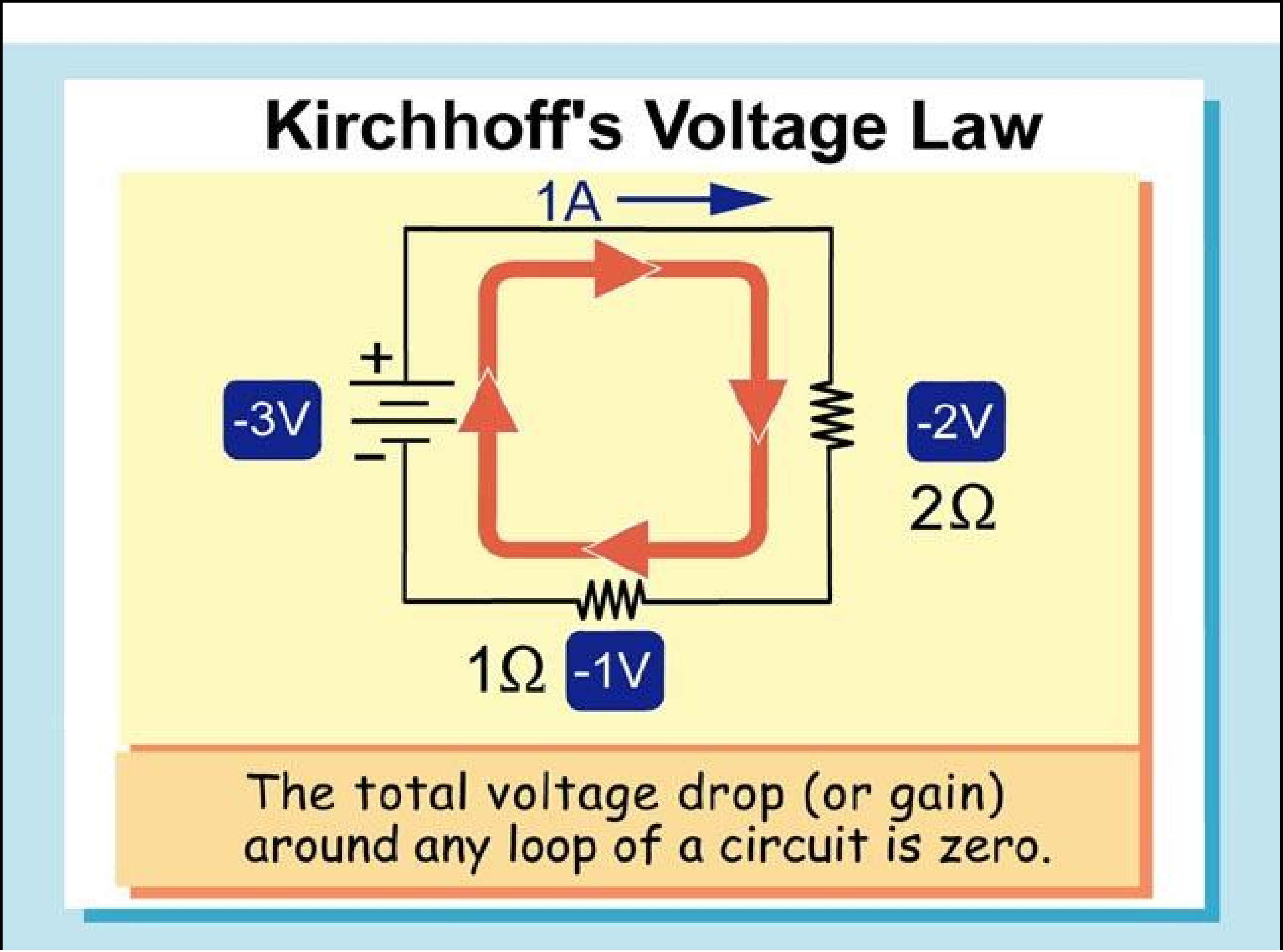
$$I = \frac{V}{R} \qquad \frac{3V}{3\Omega} = 1A$$

Voltage in a series circuit



- Each separate resistance creates a voltage drop as the current passes through.
- As current flows along a series circuit, each type of resistor transforms some of the electrical energy into another form of energy
- Ohm’s law is used to calculate the voltage drop across each resistor.

DCE,Gurgaon



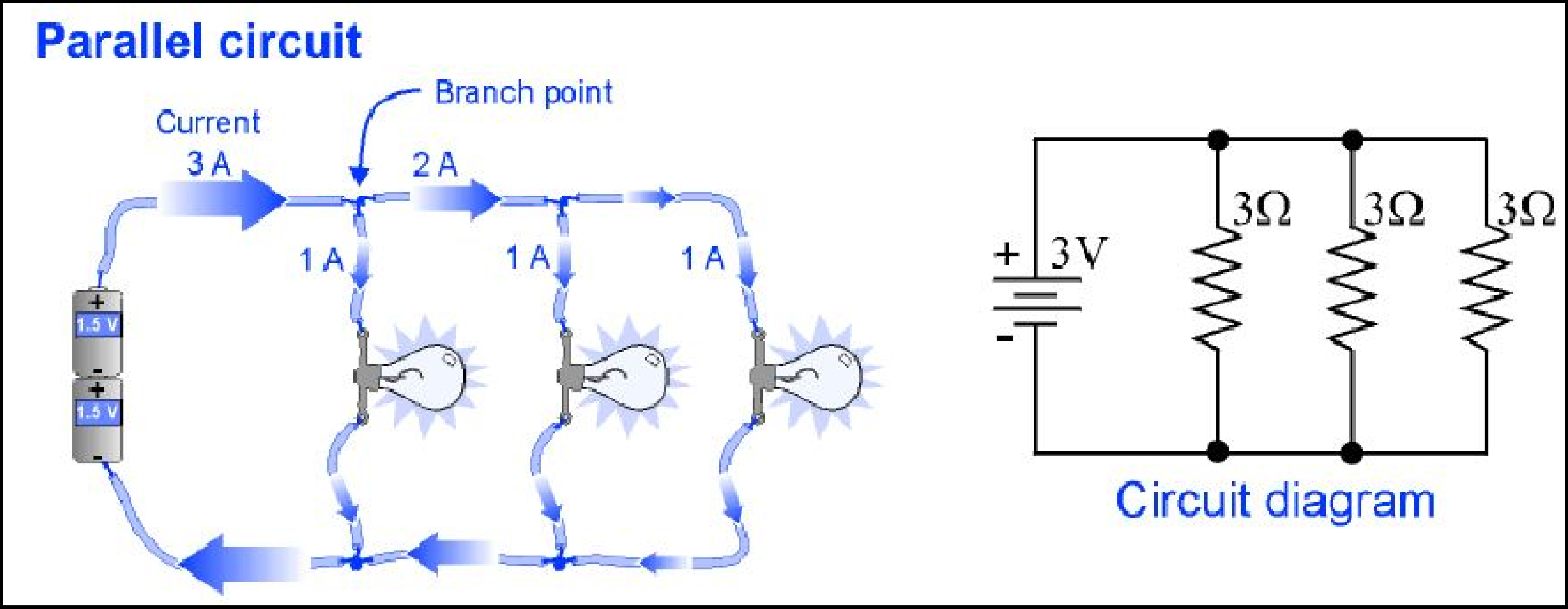
Kirchhoff's Voltage Law

The total voltage drop (or gain) around any loop of a circuit is zero.

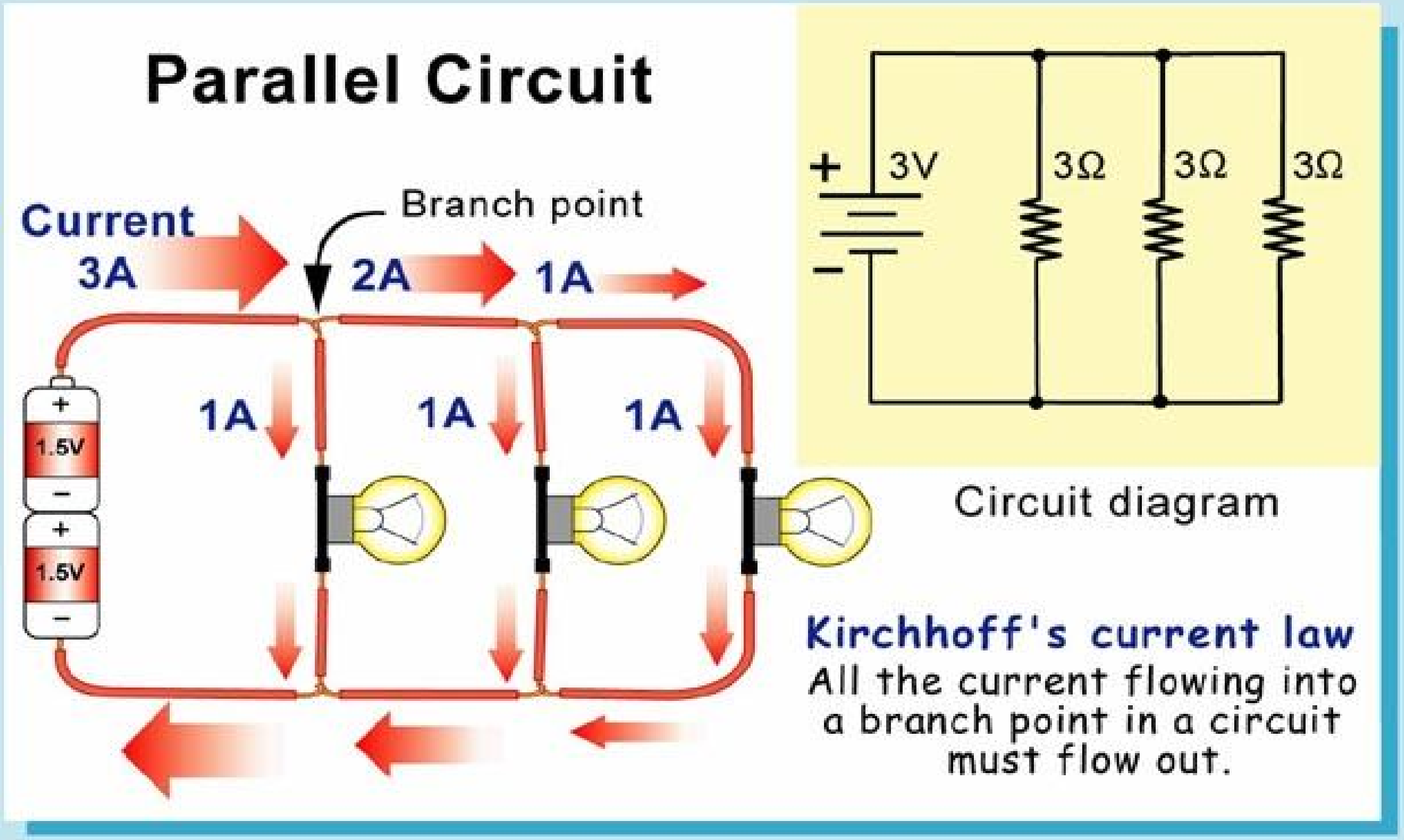
DCE,Gurgaon

Series and Parallel Circuits

- In parallel circuits the current can take more than one path.
- Because there are multiple branches, the current is not the same at all points in a parallel circuit.



DCE,Gurgaon

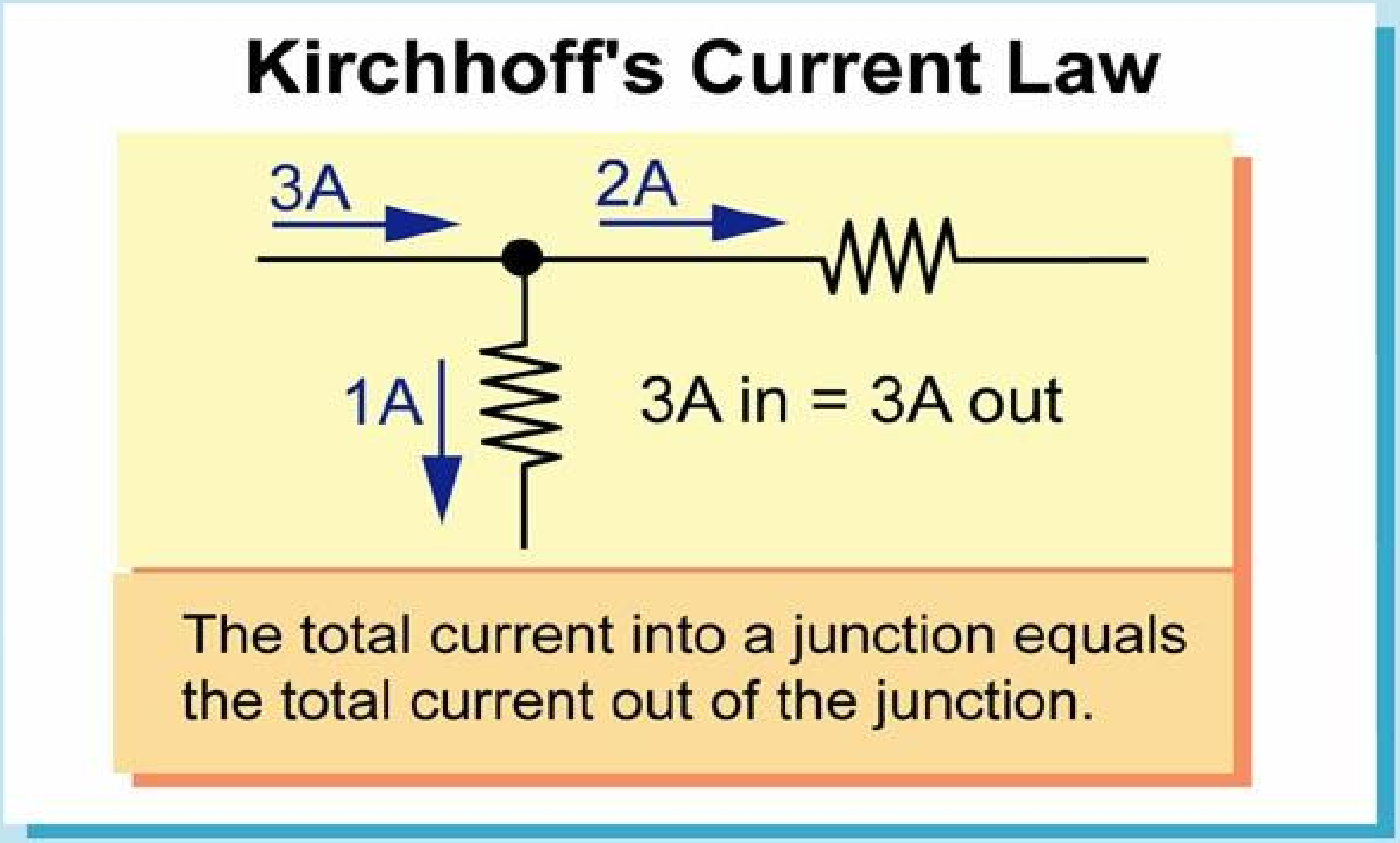


Kirchhoff's current law
All the current flowing into a branch point in a circuit must flow out.

Series and Parallel Circuits

- Sometimes these paths are called branches.
- The current through a branch is also called the branch current.
- When analyzing a parallel circuit, remember that the current always has to go somewhere.
- The total current in the circuit is the sum of the currents in all the branches.
- At every branch point the current flowing out must equal the current flowing in.
- This rule is known as Kirchhoff’s current law.

DCE,Gurgaon



Kirchhoff's Current Law

The total current into a junction equals the total current out of the junction.

Voltage and current in a parallel circuit

- In a parallel circuit the voltage is the same across each branch because each branch has a low resistance path back to the battery.
- The amount of current in each branch in a parallel circuit is not necessarily the same.
- The resistance in each branch determines the current in that branch.

DCE,Gurgaon

20.1 Advantages of parallel circuits

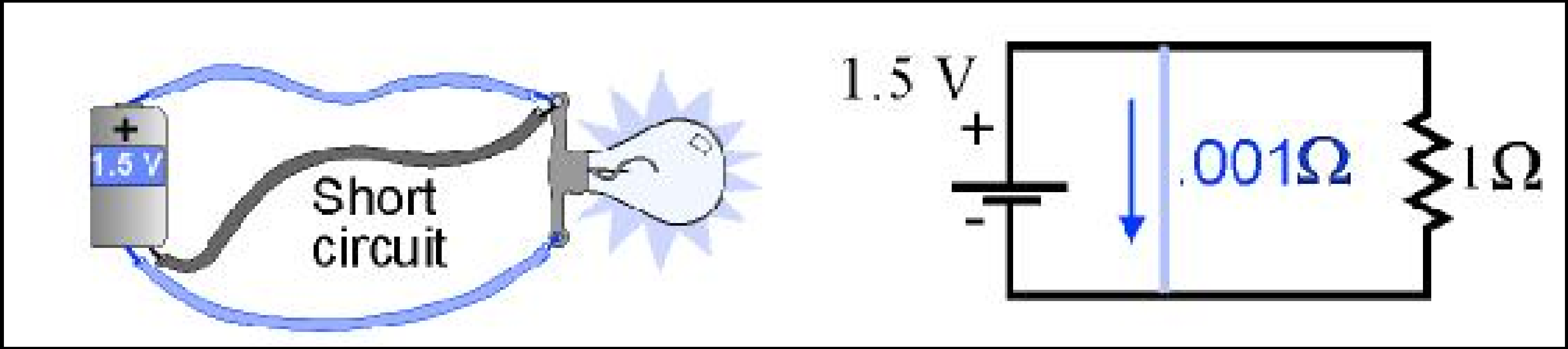
Parallel circuits have two big advantages over series circuits:

1. Each device in the circuit sees the full battery voltage.

2. Each device in the circuit may be turned off independently without stopping the current flowing to other devices in the circuit.

Short circuit

- A short circuit is a parallel path in a circuit with zero or very low resistance.
- Short circuits can be made accidentally by connecting a wire between two other wires at different voltages.
- Short circuits are dangerous because they can draw huge amounts of current.

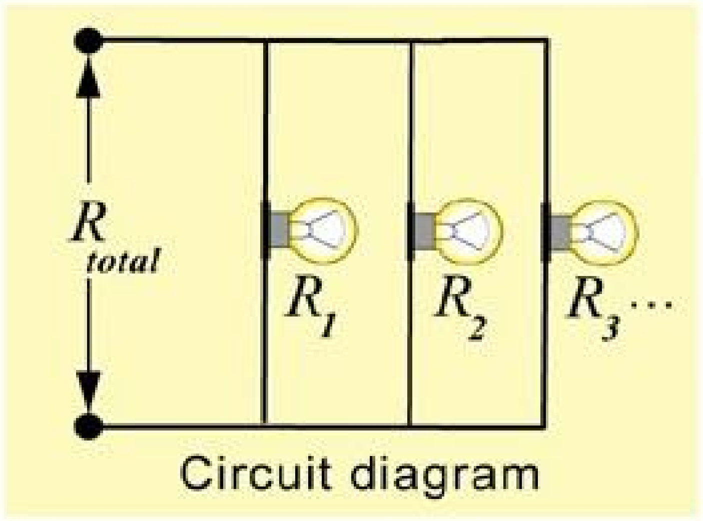


Resistance in parallel circuits

- Adding resistance in parallel provides another path for current, and more current flows.
- When more current flows for the same voltage, the total resistance of the circuit decreases.
- This happens because every new path in a parallel circuit allows more current to flow for the same voltage.

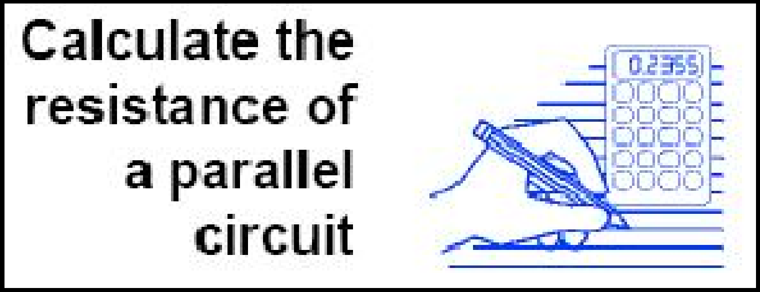
DCE,Gurgaon

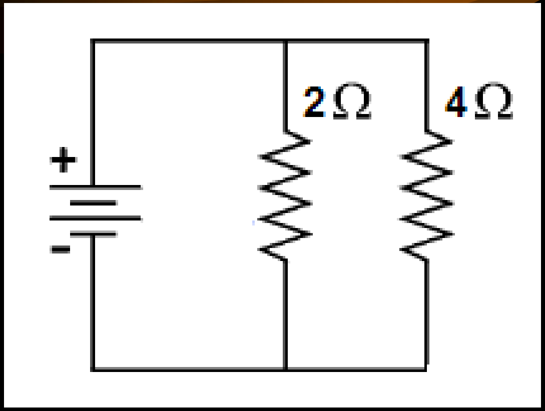
Adding Resistances in Parallel



$$\underbrace{\frac{1}{R_{total}}}_{\text{Total resistance }(\Omega)} = \underbrace{\frac{1}{R_1} + \frac{1}{R_2} + \frac{1}{R_3}}_{\text{Individual resistances }(\Omega)} + \cdots$$

Adding resistance in parallel circuits



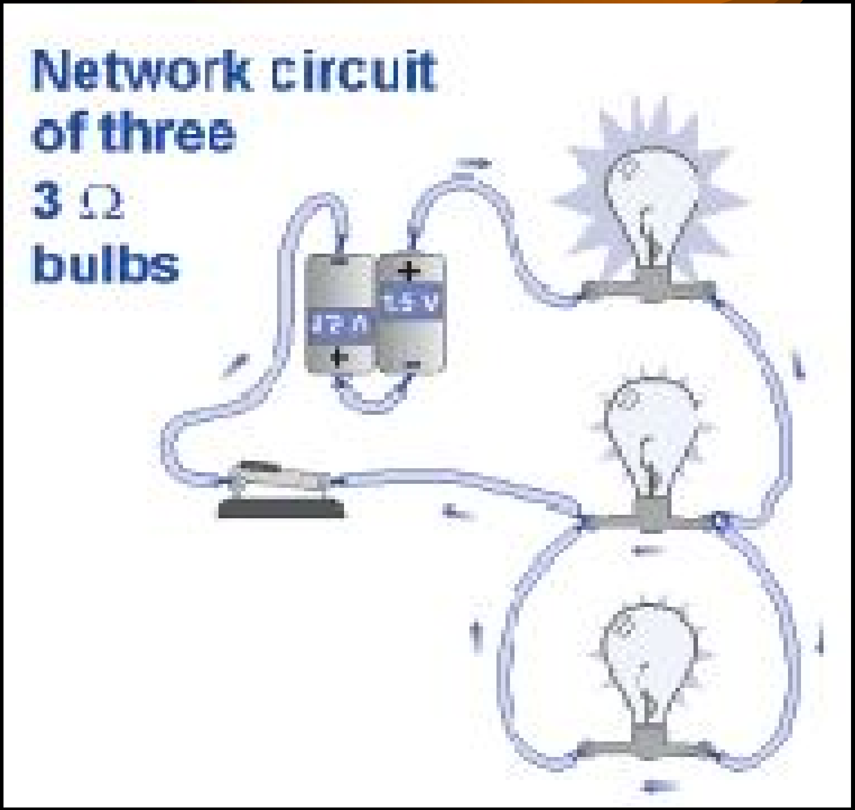


- A circuit contains a 2 ohm resistor and a 4 ohm resistor in parallel.
- Calculate the total resistance of the circuit.

Analysis of Circuits

Key Question:

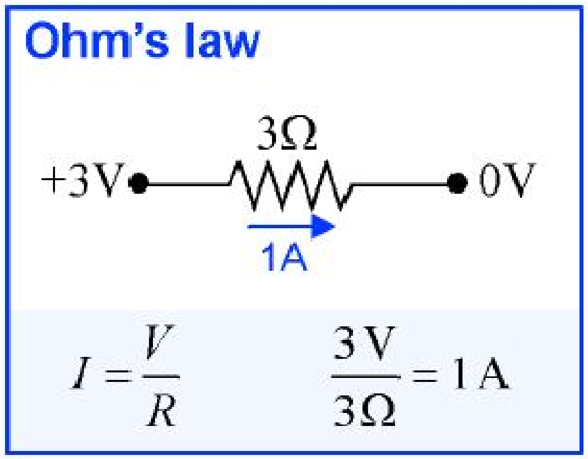
How do we analyze network circuits?



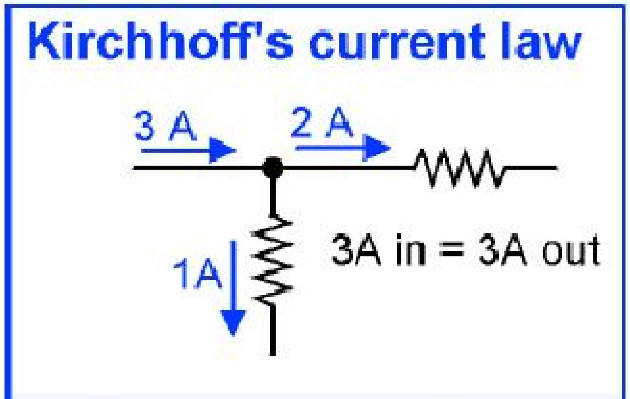
Analysis of Circuits

- All circuits work by manipulating currents and voltages.
- The process of circuit analysis means figuring out what the currents and voltages in a circuit are, and also how they are affected by each other.
- Three basic laws are the foundation of circuit analysis.

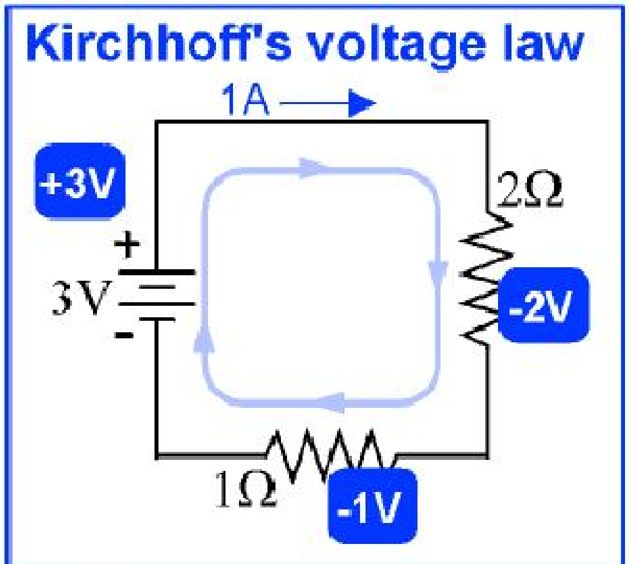
Three circuit laws



$$I = \frac{V}{R} \qquad \frac{3\,\text{V}}{3\,\Omega} = 1\,\text{A}$$



The total current into a junction equals the total current out of the junction.



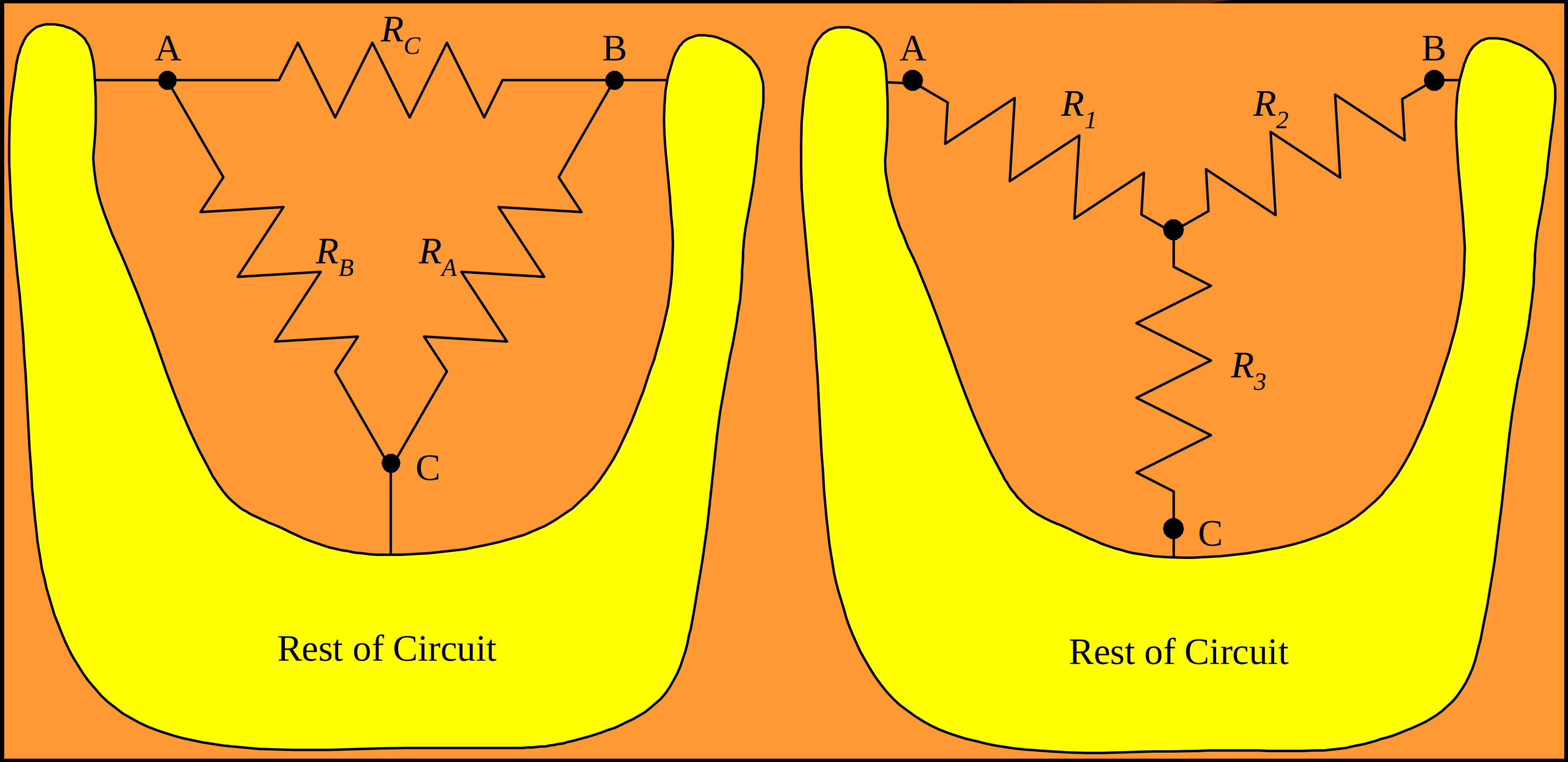
The total voltage drop (or gain) around any loop of a circuit is zero.

Delta-to-Wye Transformations

- The transformations, or equivalent circuits, that we cover next are called delta-to-wye, or wye-to-delta transformations. They are also sometimes called pi-to-tee or tee-to-pi transformations.
- These are equivalent circuit pairs. They apply for parts of circuits that have three terminals. Each version of the equivalent circuit has three resistors.

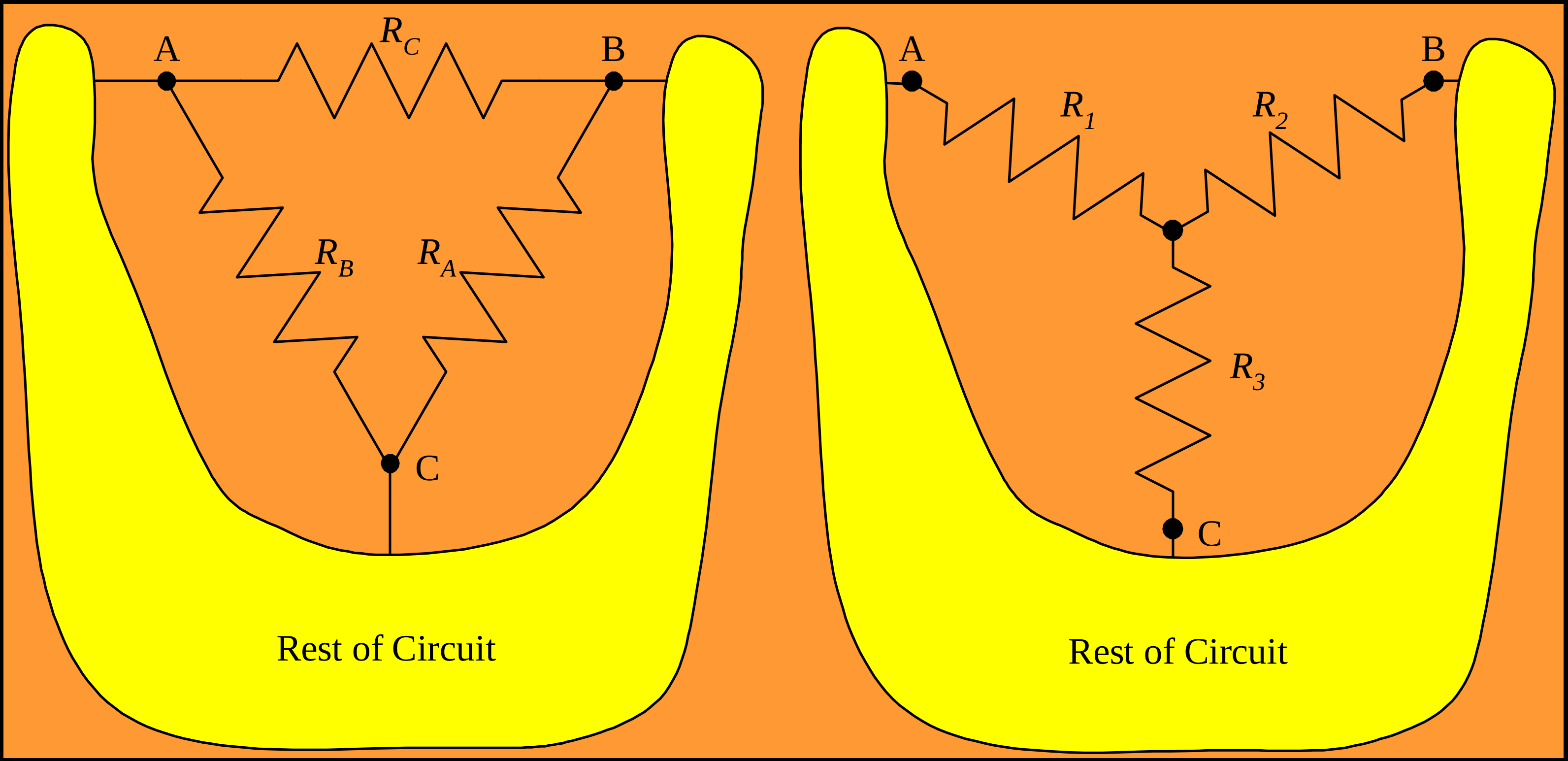
Delta-to-Wye Transformations

Three resistors in a part of a circuit with three terminals can be replaced with another version, also with three resistors. The two versions are shown here. Note that none of these resistors is in series with any other resistor, nor in parallel with any other resistor. The three terminals in this example are labeled A, B, and C.



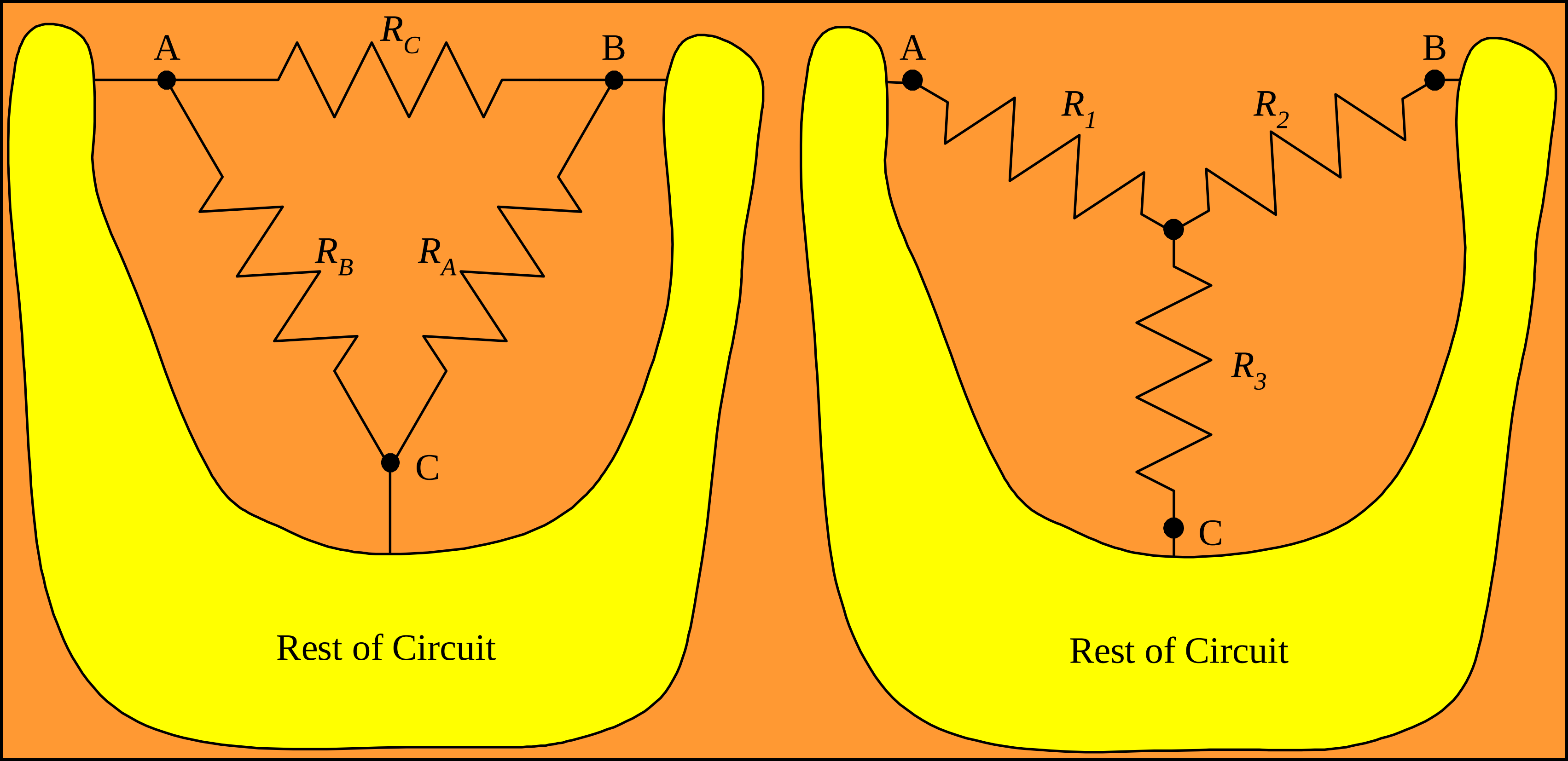
Delta-to-Wye Transformations (Notes on Names)

The version on the left hand side is called the delta connection, for the Greek letter D. The version on the right hand side is called the wye connection, for the letter Y. The delta connection is also called the pi (p) connection, and the wye interconnection is also called the tee (T) connection. All these names come from the shapes of the drawings.



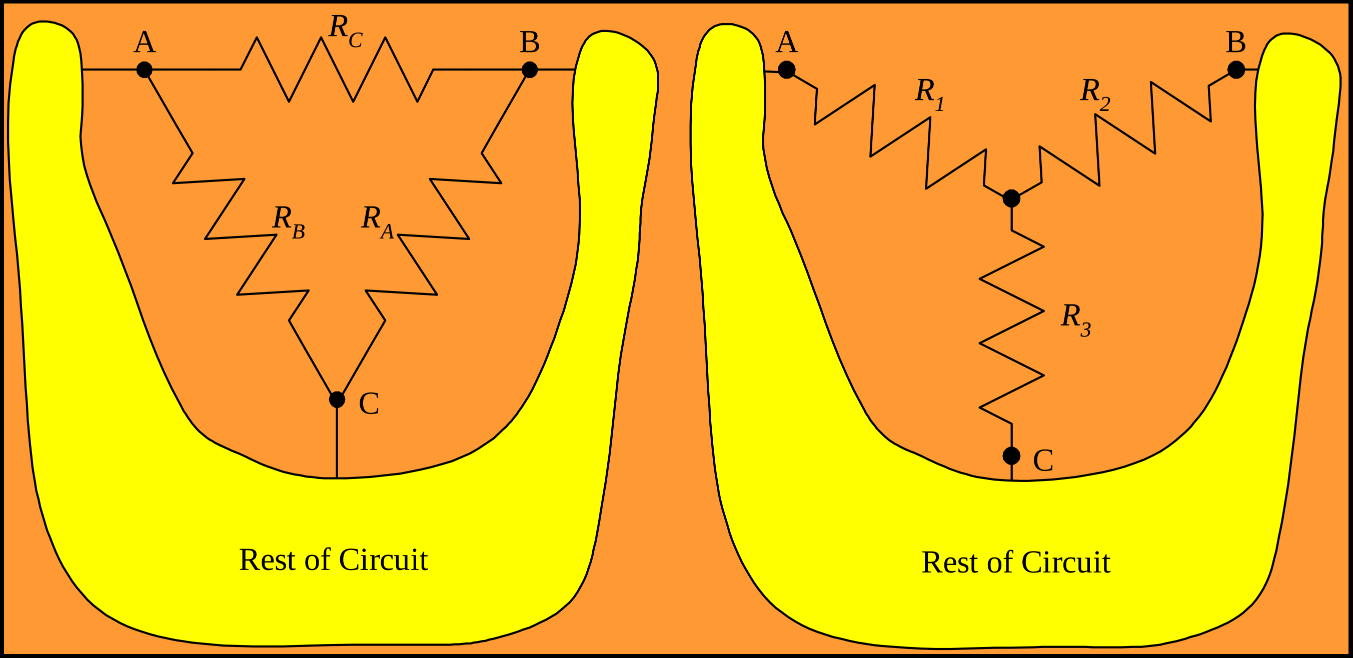
Delta-to-Wye Transformations (More Notes)

When we go from the delta connection (on the left) to the wye connection (on the right), we call this the delta-to-wye transformation. Going in the other direction is called the wye-to-delta transformation. One can go in either direction, as needed. These are equivalent circuits.



Delta-to-Wye Transformation Equations

When we perform the delta-to-wye transformation (going from left to right) we use the equations given below.



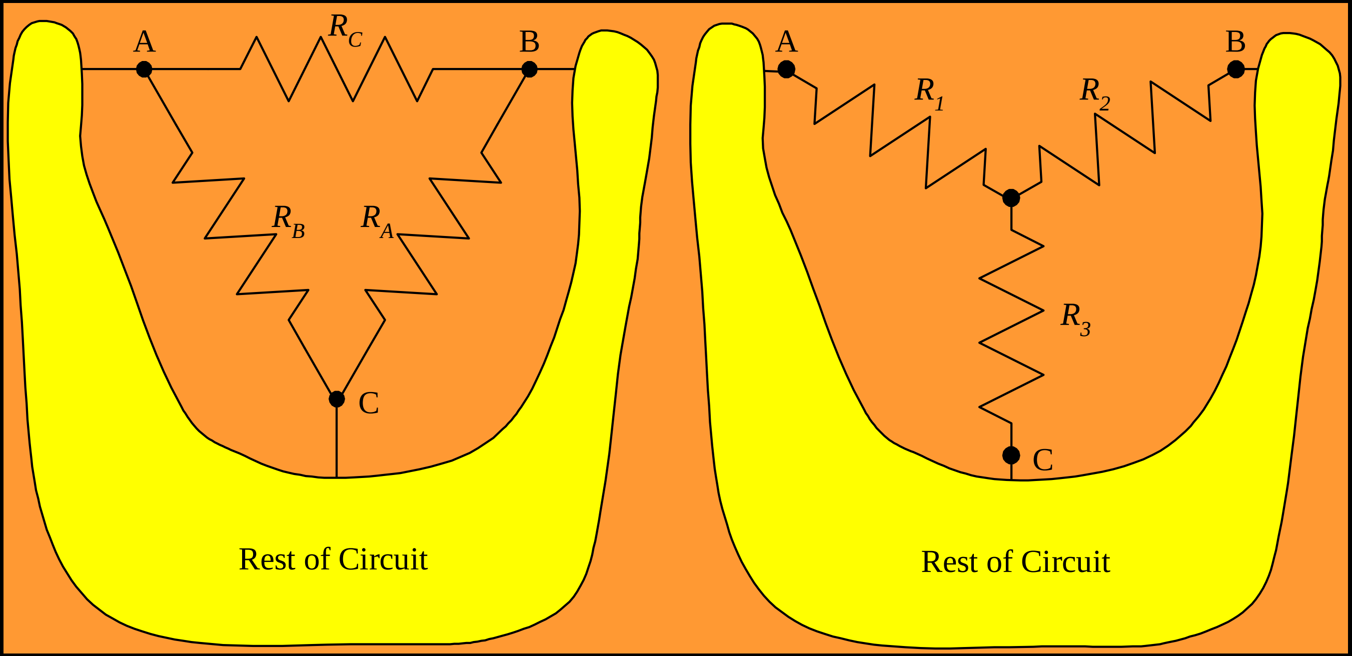
$$R_1 = \frac{R_B R_C}{R_A + R_B + R_C}$$

$$R_2 = \frac{R_A R_C}{R_A + R_B + R_C}$$

$$R_3 = \frac{R_A R_B}{R_A + R_B + R_C}$$

Wye-to-Delta Transformation Equations

When we perform the wye-to-delta transformation (going from right to left) we use the equations given below.



$$R_A = \frac{R_1R_2 + R_2R_3 + R_1R_3}{R_1}$$

$$R_B = \frac{R_1R_2 + R_2R_3 + R_1R_3}{R_2}$$

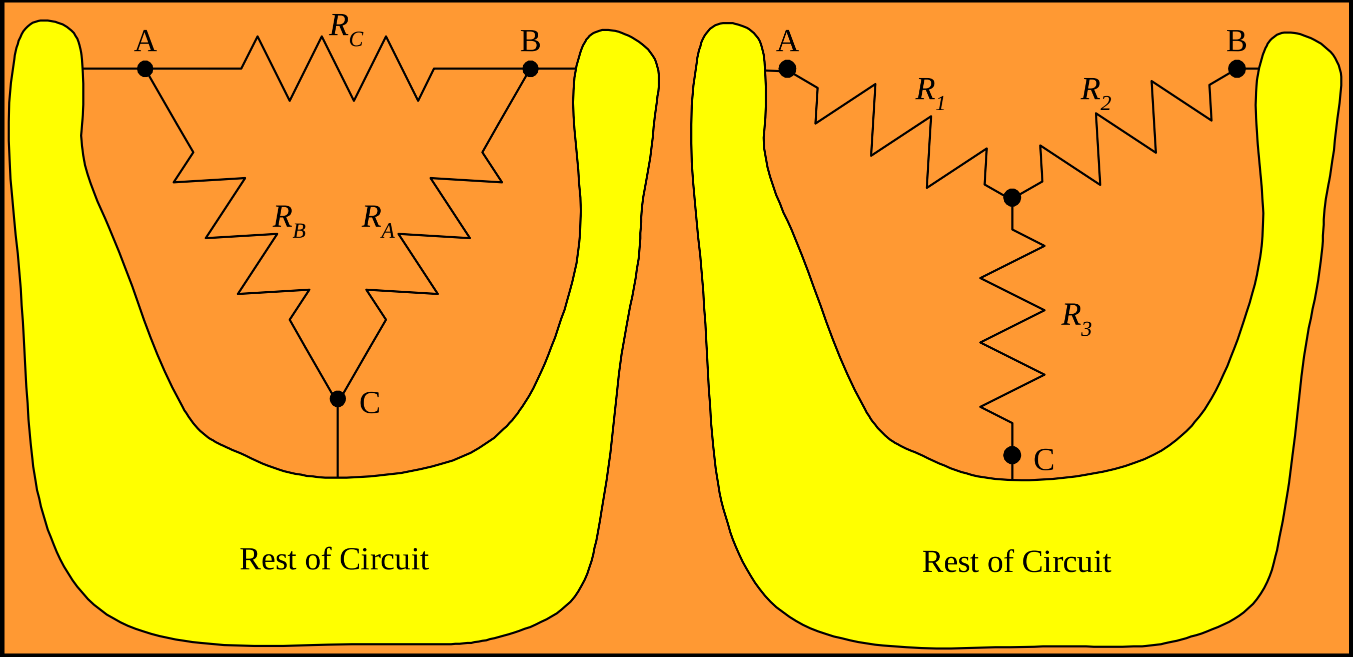
$$R_C = \frac{R_1R_2 + R_2R_3 + R_1R_3}{R_3}$$

DCE,Gurgaon

Deriving the Equations

While these equivalent circuits are useful, perhaps the most important insight is gained from asking where these useful equations come from. How were these equations derived?

The answer is that they were derived using the fundamental rule for equivalent circuits. These two equivalent circuits have to behave the same way no matter what circuit is connected to them. So, we can choose specific circuits to connect to the equivalents. We make the derivation by solving for equivalent resistances, using our series and parallel rules, under different, specific conditions.



$$R_1 = \frac{R_B R_C}{R_A + R_B + R_C}$$

$$R_2 = \frac{R_A R_C}{R_A + R_B + R_C}$$

$$R_3 = \frac{R_A R_B}{R_A + R_B + R_C}$$

$$R_A = \frac{R_1 R_2 + R_2 R_3 + R_1 R_3}{R_1}$$

$$R_B = \frac{R_1 R_2 + R_2 R_3 + R_1 R_3}{R_2}$$

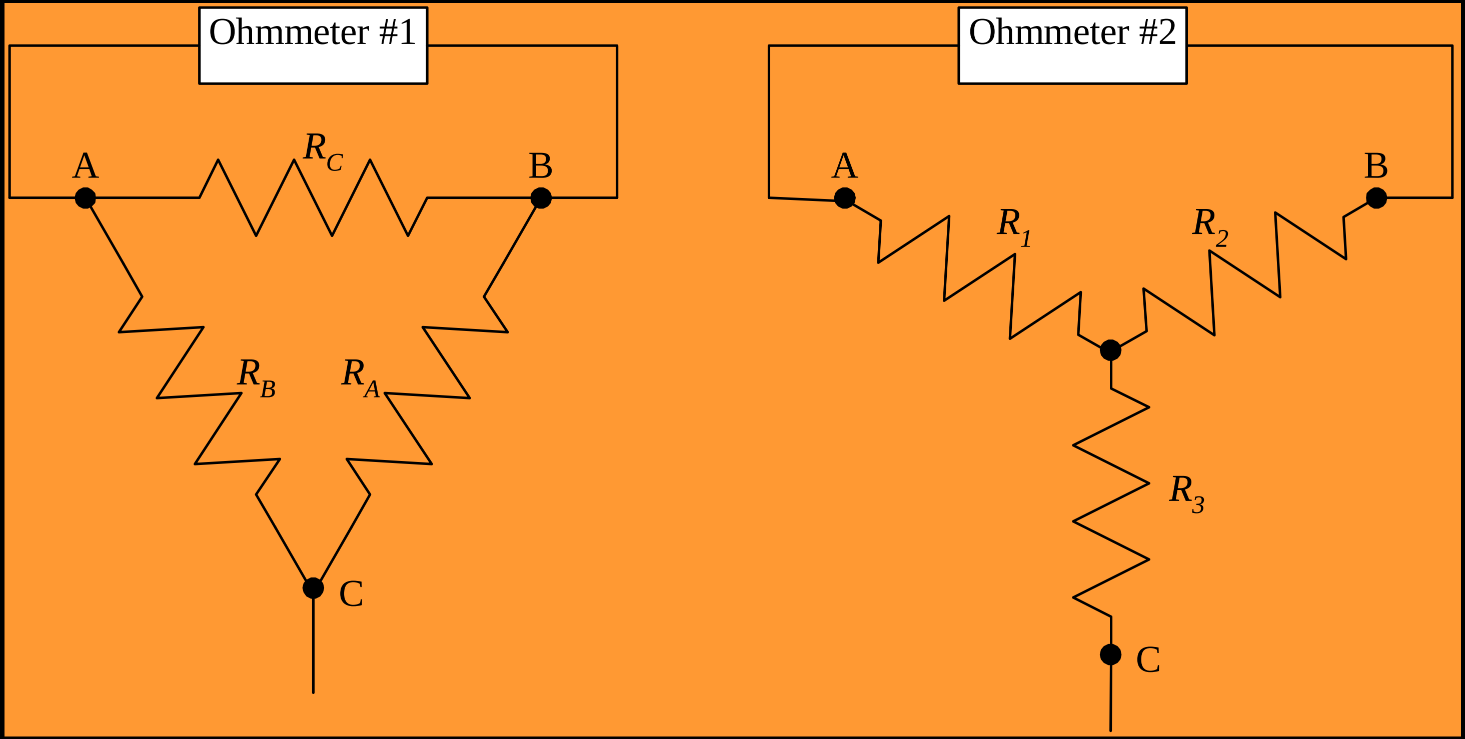
$$R_C = \frac{R_1 R_2 + R_2 R_3 + R_1 R_3}{R_3}$$

Equation 1

We can calculate the equivalent resistance between terminals A and B, when C is not connected anywhere. The two cases are shown below. This is the same as connecting an ohmmeter, which measures resistance, between terminals A and B, while terminal C is left disconnected.

Ohmmeter #1 reads $R_{EQ1} = R_C \,||\, (R_A + R_B)$. Ohmmeter #2 reads $R_{EQ2} = R_1 + R_2$.

These must read the same value, so $R_C \,||\, (R_A + R_B) = R_1 + R_2$.



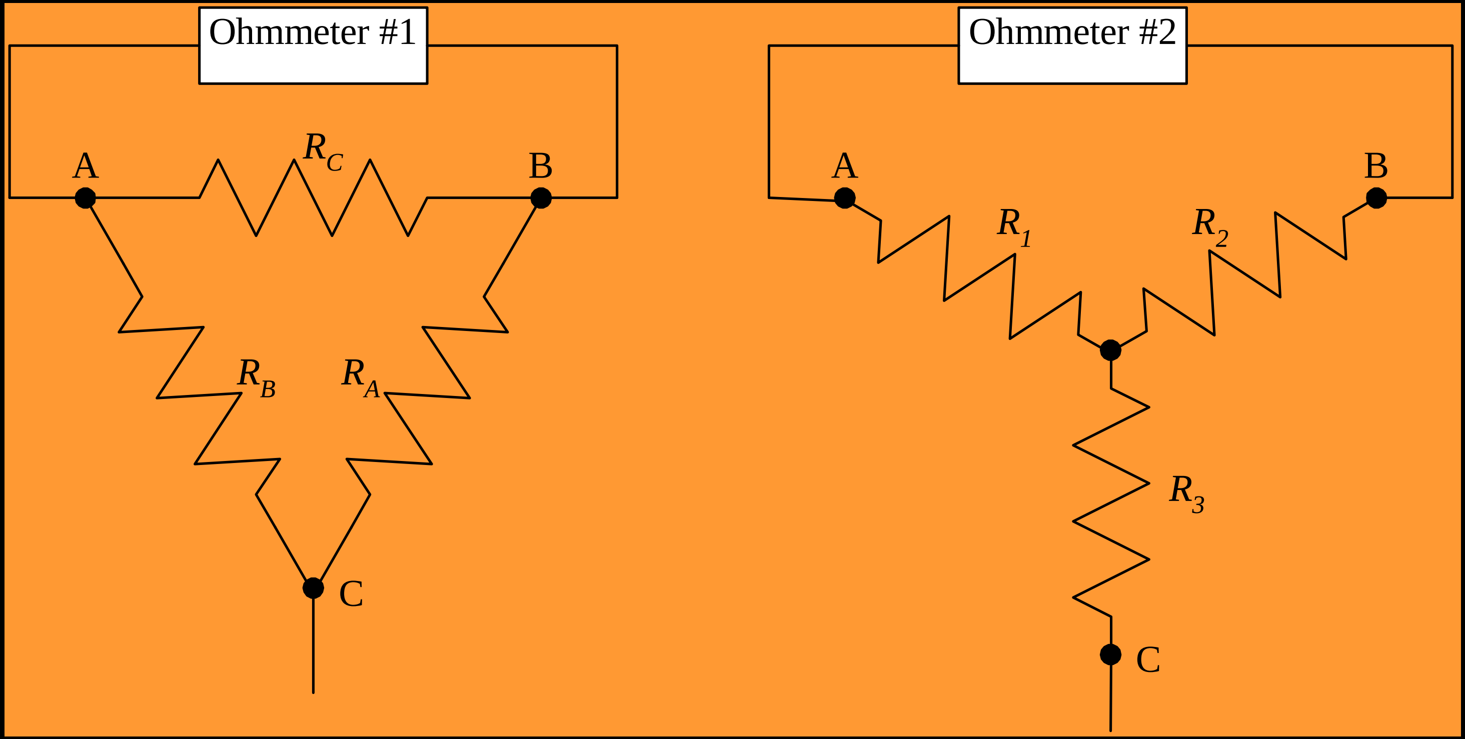
DCE,Gurgaon

Equations 2 and 3

So, the equation that results from the first situation is

$$R_C \parallel (R_A + R_B) = R_1 + R_2.$$

We can make this measurement two other ways, and get two more equations. Specifically, we can measure the resistance between A and C, with B left open, and we can measure the resistance between B and C, with A left open.



All Three Equations

The three equations we can obtain are

$$R_C \,||\, (R_A + R_B) = R_1 + R_2,$$
$$R_B \,||\, (R_A + R_C) = R_1 + R_3, \text{ and}$$
$$R_A \,||\, (R_B + R_C) = R_2 + R_3.$$

This is all that we need. These three equations can be manipulated algebraically to obtain either the set of equations for the delta-to-wye transformation (by solving for R_1, R_2, and R_3), or the set of equations for the wye-to-delta transformation (by solving for R_A, R_B, and R_C).

Why Are Delta-to-Wye Transformations Needed?

- This is a good question. In fact, it should be pointed out that these transformations are not necessary. Rather, they are like many other aspects of circuit analysis in that they allow us to solve circuits more quickly and more easily. They are used in cases where the resistors are neither in series nor parallel, so to simplify the circuit requires something more.
- One key in applying these equivalents is to get the proper resistors in the proper place in the equivalents and equations. We recommend that you name the terminals each time, on the circuit diagrams, to help you get these things in the right places.

DCE,Gurgaon

Voltage Divider and Current Divider Rules

Overview of this Part *Series, Parallel, and other Resistance Equivalent Circuits*

In this part, we will cover the following topics:

- Voltage Divider Rule
- Current Divider Rule
- Signs in the Voltage Divider Rule
- Signs in the Current Divider Rule

Voltage Divider Rule – Our First Circuit Analysis Tool

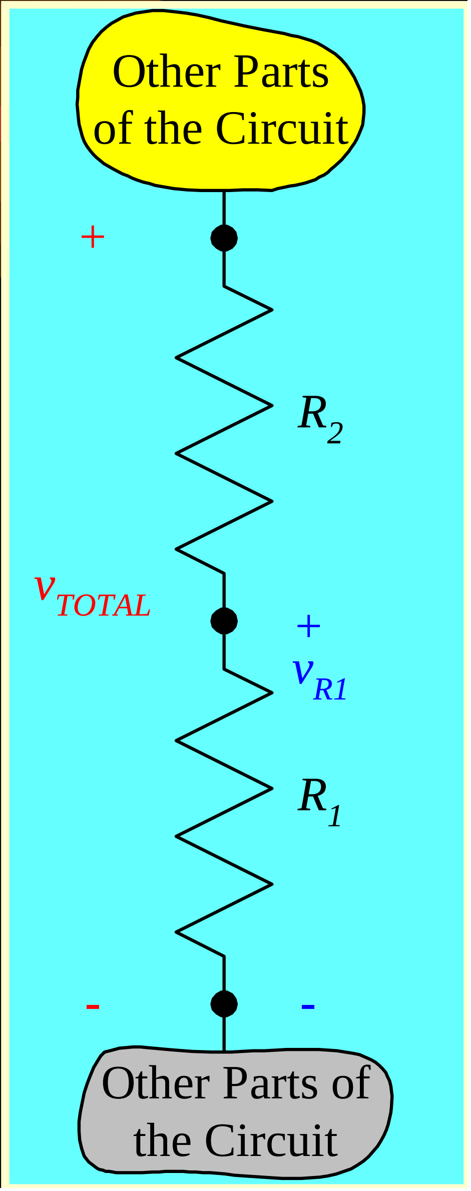
The Voltage Divider Rule (VDR) is the first of long list of tools that we are going to develop to make circuit analysis quicker and easier. The idea is this: if the same situation occurs often, we can derive the solution once, and use it whenever it applies. As with any tools, the keys are:

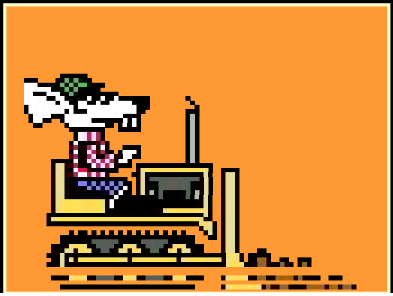
1. Recognizing when the tool works and when it doesn't work.
2. Using the tool properly.

DCE,Gurgaon

Voltage Divider Rule – Setting up the Derivation

The Voltage Divider Rule involves the voltages across series resistors. Let's take the case where we have two resistors in series. Assume for the moment that the voltage across these two resistors, v_{TOTAL}, is known. Assume that we want the voltage across one of the resistors, shown here as v_{R1}. Let's find it.



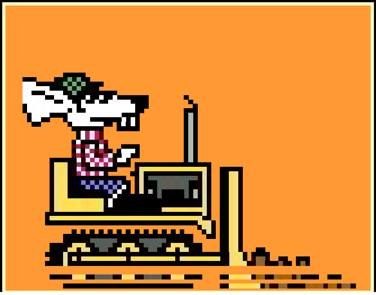


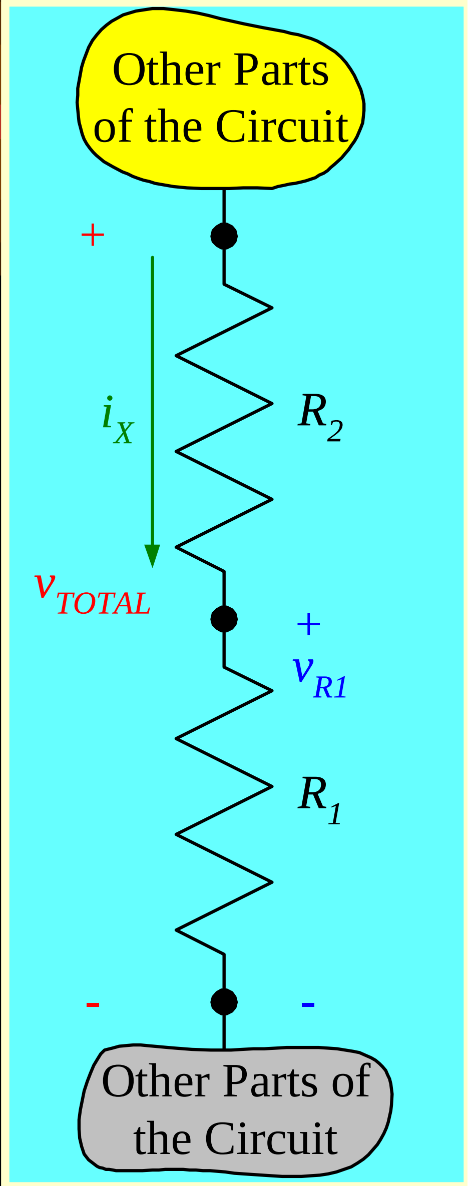
DCE,Gurgaon

Voltage Divider Rule – Derivation Step 1

The current through both of these resistors is the same, since the resistors are in series. The current, i_X, is

$$i_X = \frac{v_{TOTAL}}{R_1 + R_2}.$$



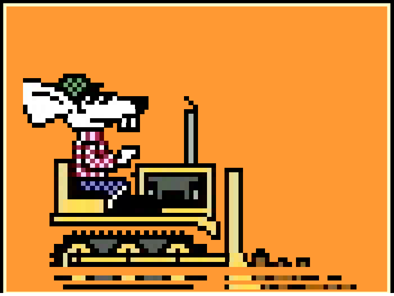


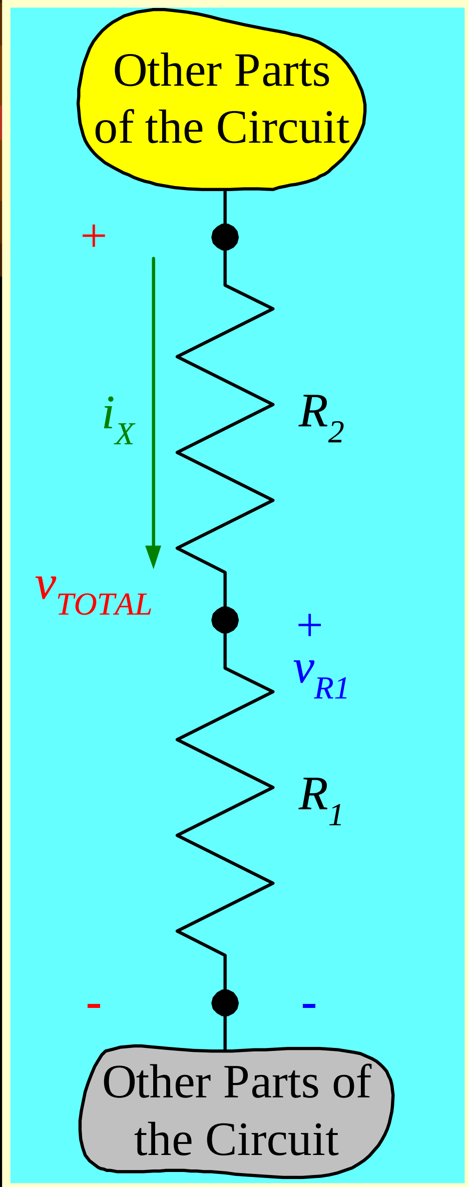
DCE,Gurgaon

Voltage Divider Rule – Derivation Step 2

The current through resistor R_1 is the same current. The current, i_X, is

$$\boxed{i_X = \frac{v_{R1}}{R_1}.}$$



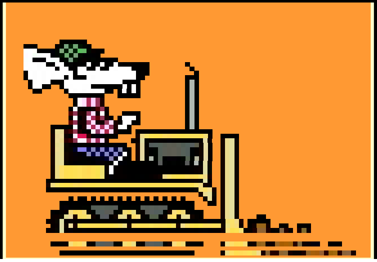


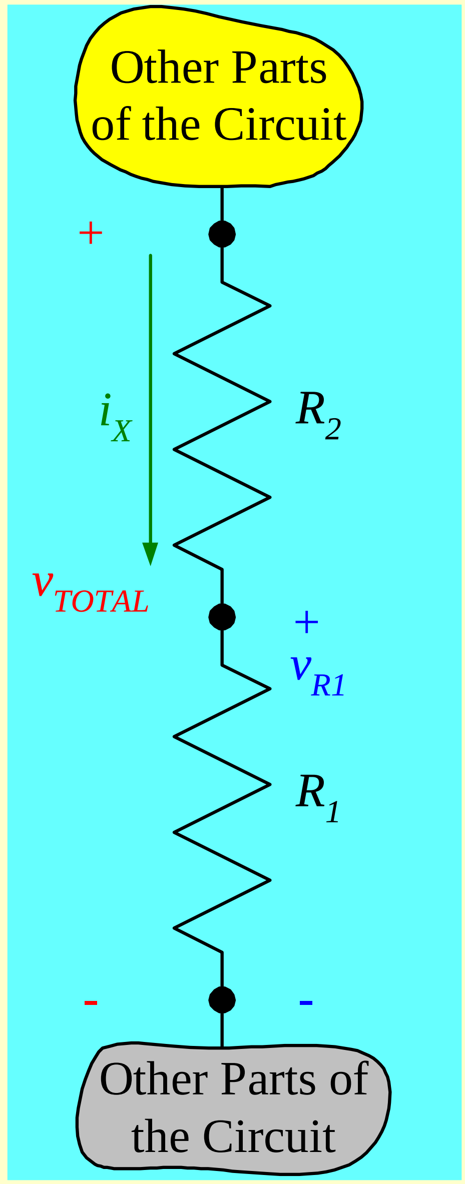
Voltage Divider Rule – Derivation Step 3

These are two expressions for the same current, so they must be equal to each other. Therefore, we can write

$\frac{v_{R1}}{R_1} = \frac{v_{TOTAL}}{R_1 + R_2}$. Solving for v_{R1}, we get

$$v_{R1} = v_{TOTAL} \frac{R_1}{R_1 + R_2}.$$

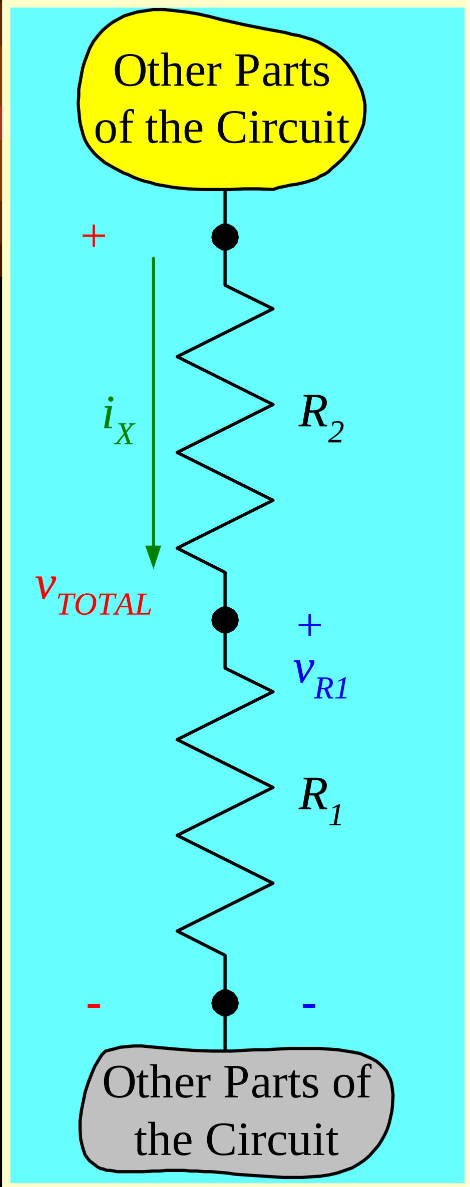


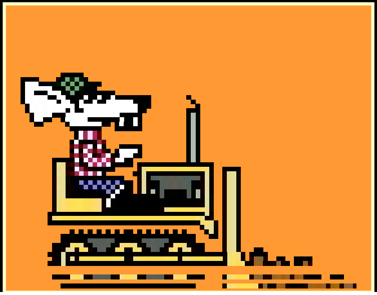


The Voltage Divider Rule

This is the expression we wanted. We call this the Voltage Divider Rule (VDR).

$$\boxed{v_{R1} = v_{TOTAL} \frac{R_1}{R_1 + R_2}.}$$





DCE,Gurgaon

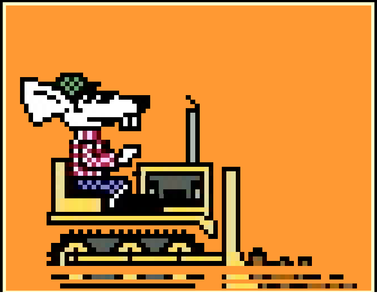
Voltage Divider Rule – For Each Resistor

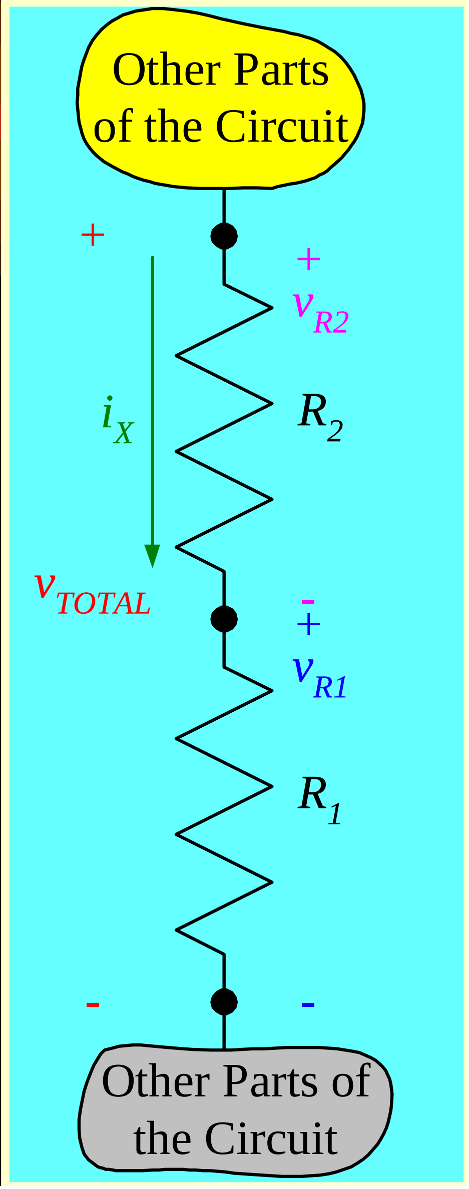
Go back to Overview slide.

This is easy enough to remember that most people just memorize it. Remember that it only works for resistors that are in series. Of course, there is a similar rule for the other resistor. For the voltage across one resistor, we put that resistor value in the numerator.

$$v_{R1} = v_{TOTAL} \frac{R_1}{R_1 + R_2}.$$

$$v_{R2} = v_{TOTAL} \frac{R_2}{R_1 + R_2}.$$

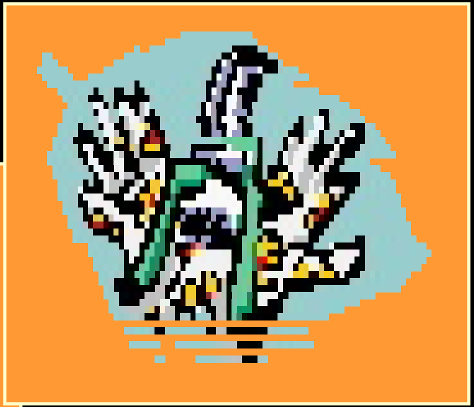
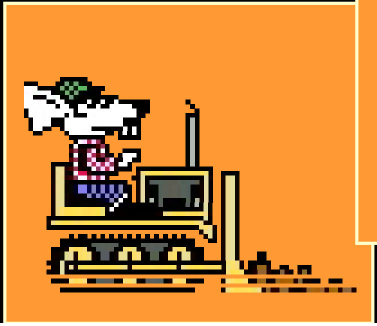




Current Divider Rule – Our Second Circuit Analysis Tool

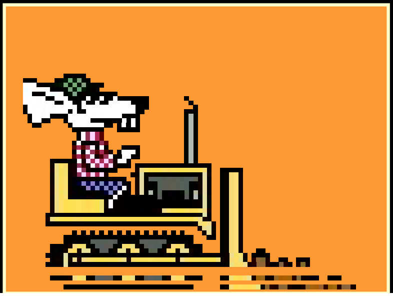
The Current Divider Rule (CDR) is the first of long list of tools that we are going to develop to make circuit analysis quicker and easier. Again, if the same situation occurs often, we can derive the solution once, and use it whenever it applies. As with any tools, the keys are:

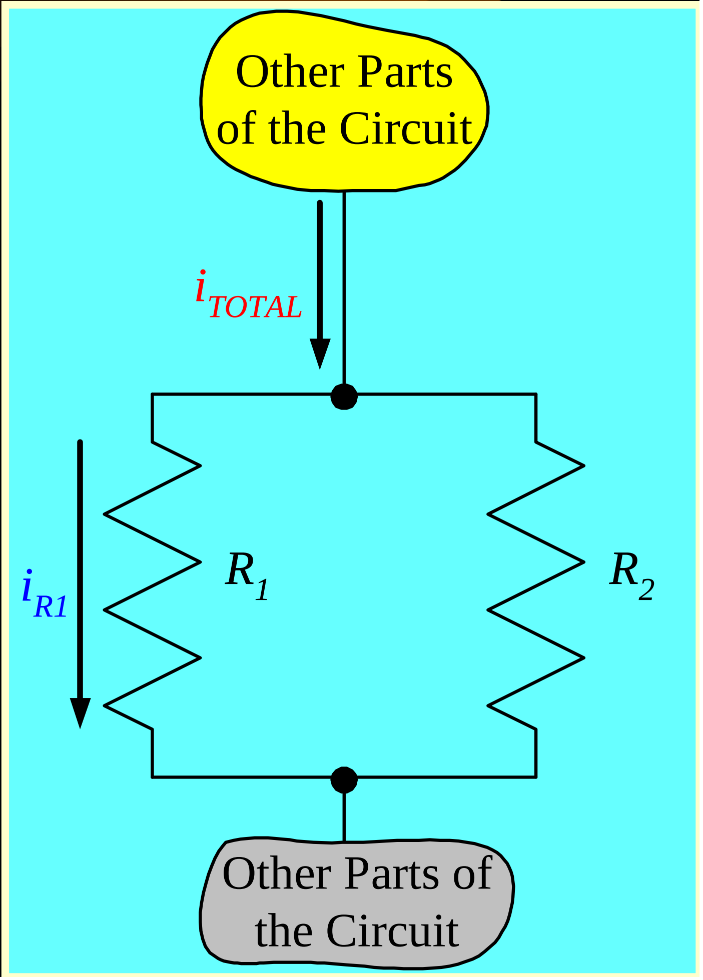
1. Recognizing when the tool works and when it doesn't work.
2. Using the tool properly.

Current Divider Rule – Setting up the Derivation

The Current Divider Rule involves the currents through parallel resistors. Let's take the case where we have two resistors in parallel. Assume for the moment that the current feeding these two resistors, i_{TOTAL}, is known. Assume that we want the current through one of the resistors, shown here as i_{R1}. Let's find it.





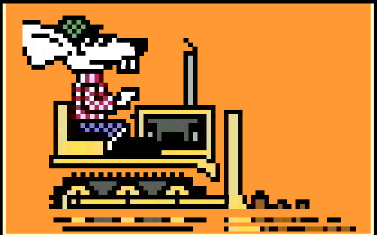
DCE,Gurgaon

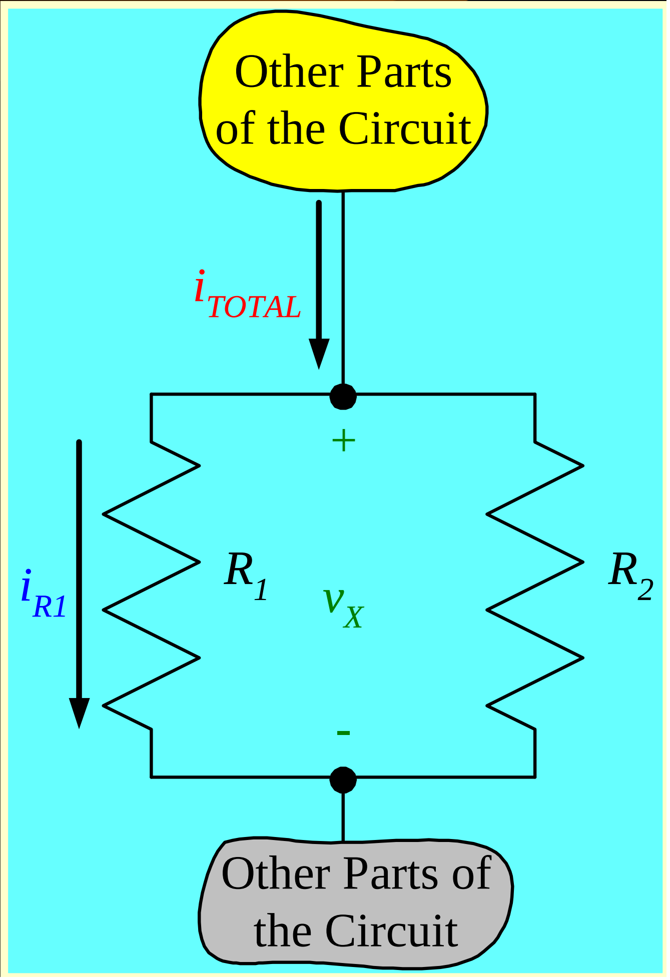
Current Divider Rule – Derivation Step 1

The voltage across both of these resistors is the same, since the resistors are in parallel. The voltage, v_X, is the current multiplied by the equivalent parallel resistance,

$$v_X = i_{TOTAL}\left(R_1 \| R_2\right), \text{ or}$$

$$v_X = i_{TOTAL}\left(\frac{R_1 R_2}{R_1 + R_2}\right).$$

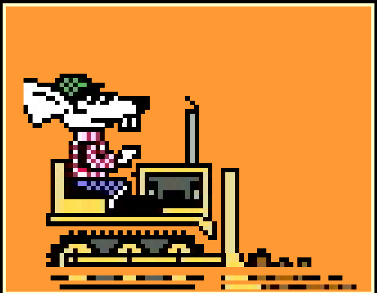


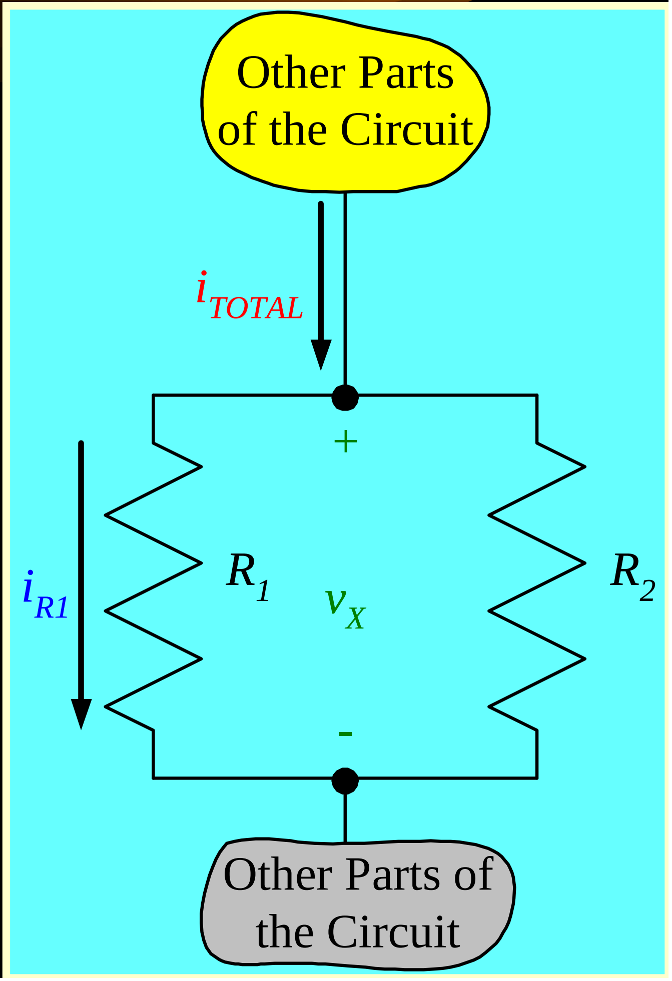


Current Divider Rule – Derivation Step 2

The voltage across resistor R_1 is the same voltage, v_X. The voltage, v_X, is

$$v_X = i_{R1} R_1 .$$



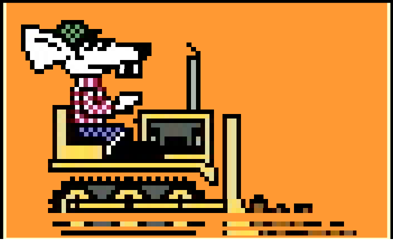


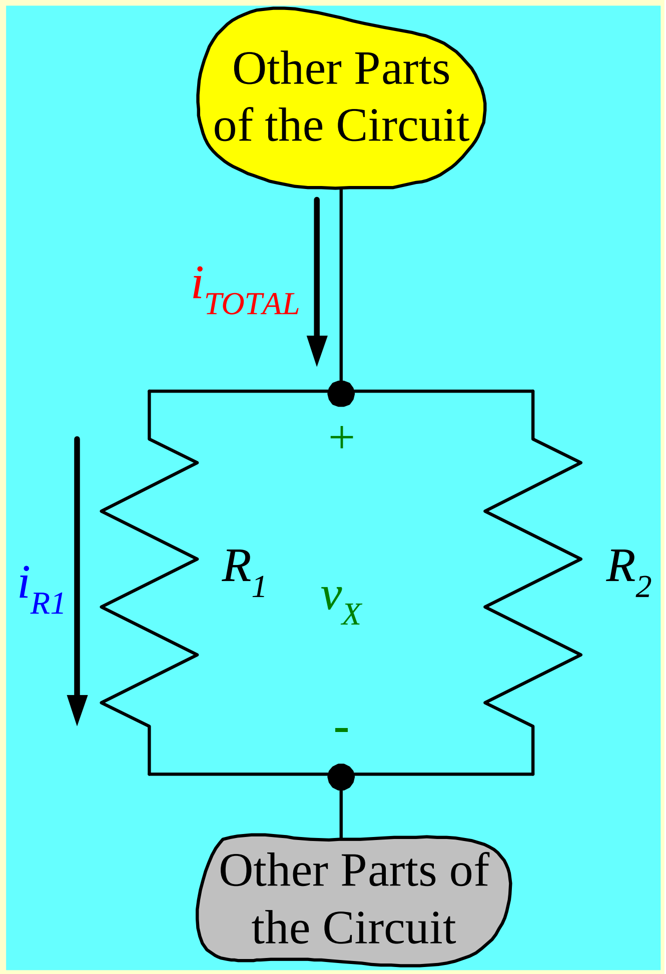
Current Divider Rule – Derivation Step 3

These are two expressions for the same voltage, so they must be equal to each other. Therefore, we can write

$$i_{R1}R_1 = i_{TOTAL}\frac{R_1R_2}{R_1+R_2}. \quad \text{Solve for } i_{R1};$$

$$i_{R1} = i_{TOTAL}\frac{R_2}{R_1+R_2}.$$

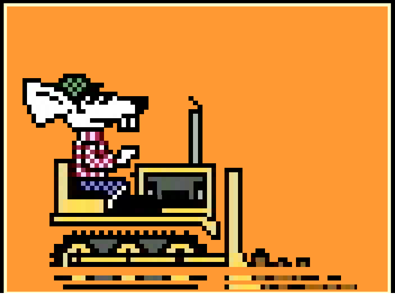


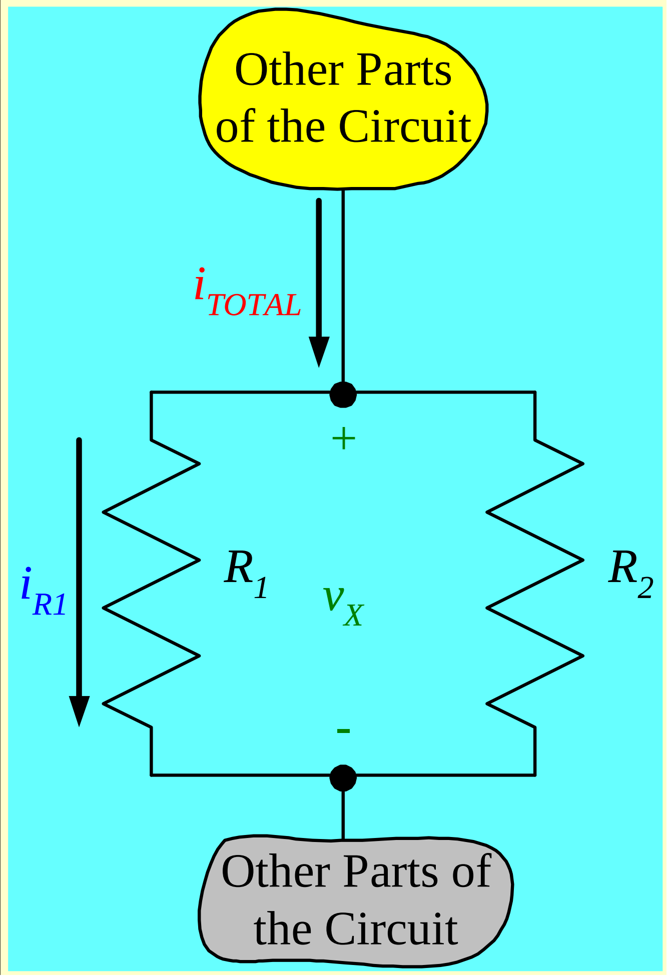


The Current Divider Rule

This is the expression we wanted. We call this the Current Divider Rule (CDR).

$$i_{R1} = i_{TOTAL} \frac{R_2}{R_1 + R_2}.$$

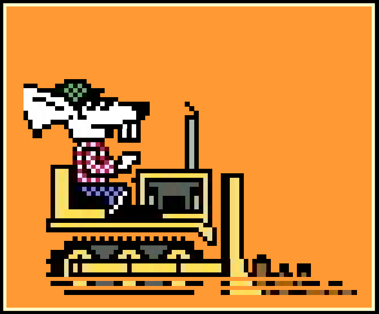




Go back to Overview slide.

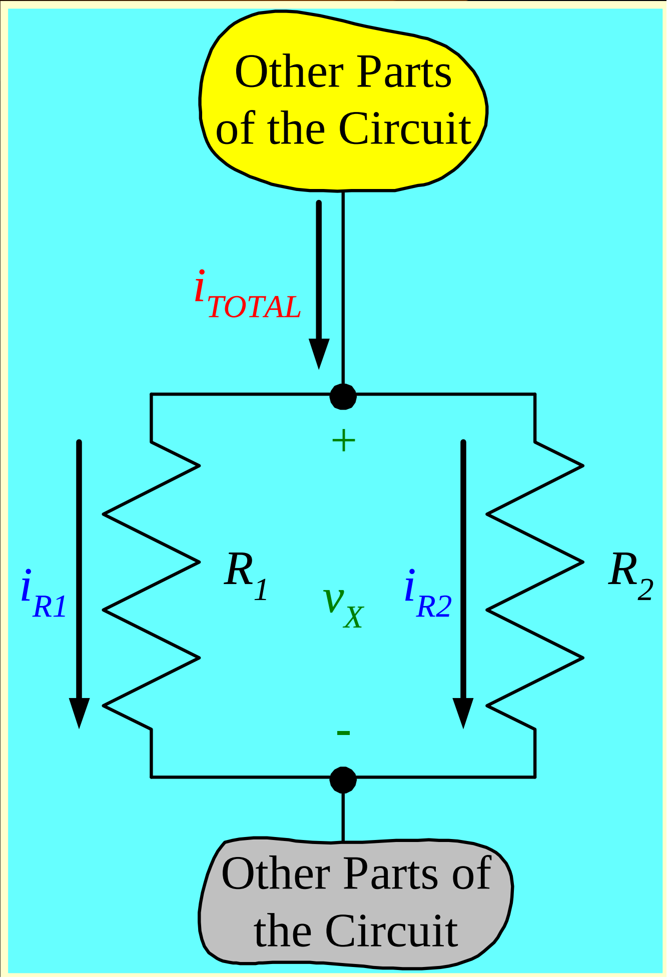
Current Divider Rule – For Each Resistor

Most people just memorize this. Remember that it only works for resistors that are in parallel. Of course, there is a similar rule for the other resistor. For the current through one resistor, we put the opposite resistor value in the numerator.



$$i_{R1} = i_{TOTAL} \frac{R_2}{R_1 + R_2}.$$

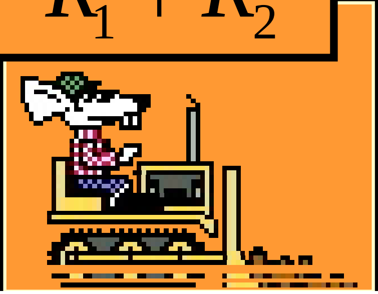
$$i_{R2} = i_{TOTAL} \frac{R_1}{R_1 + R_2}.$$

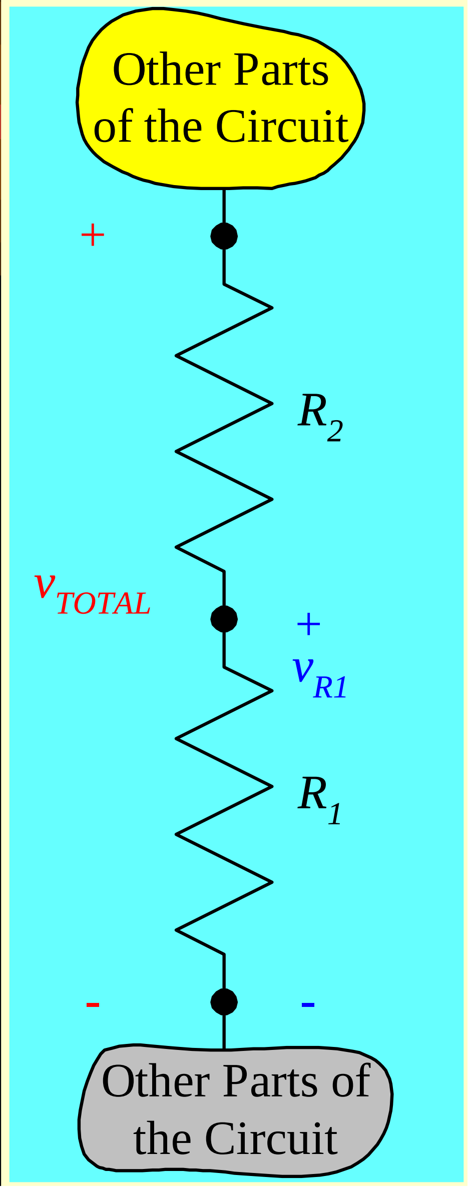


Signs in the Voltage Divider Rule

As in most equations we write, we need to be careful about the sign in the Voltage Divider Rule (VDR). Notice that when we wrote this expression, there is a positive sign. This is because the voltage v_{TOTAL} is in the same relative polarity as v_{R1}.

$$v_{R1} = +v_{TOTAL} \frac{R_1}{R_1 + R_2}.$$

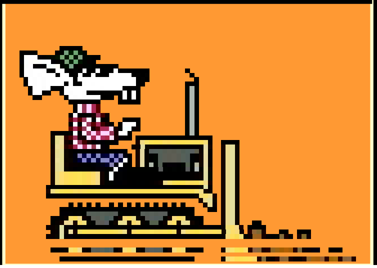


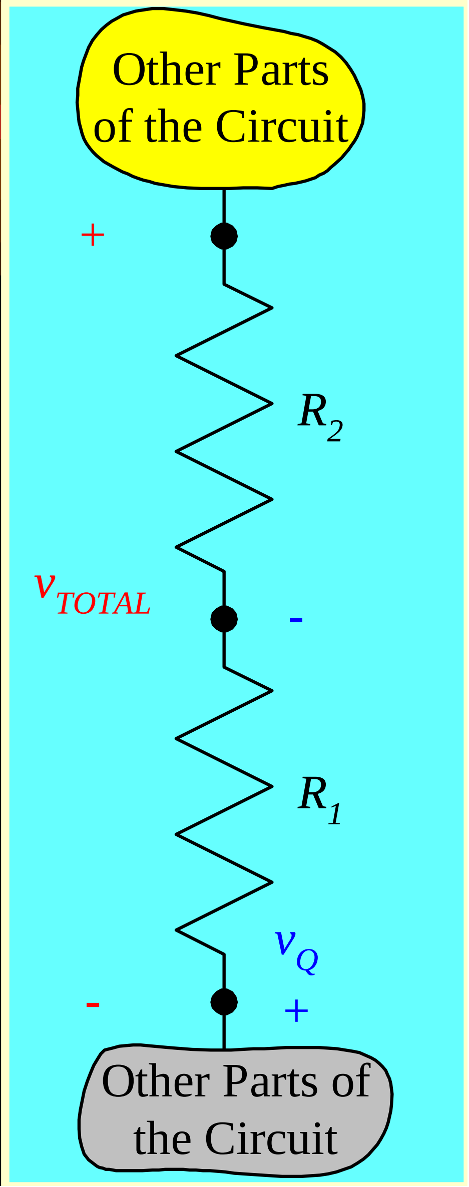


Negative Signs in the Voltage Divider Rule

If, instead, we had solved for v_Q, we would need to change the sign in the equation. This is because the voltage v_{TOTAL} is in the opposite relative polarity from v_Q.

$$v_Q = -v_{TOTAL} \frac{R_1}{R_1 + R_2}.$$





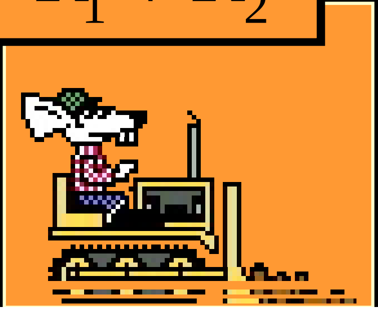
DCE,Gurgaon

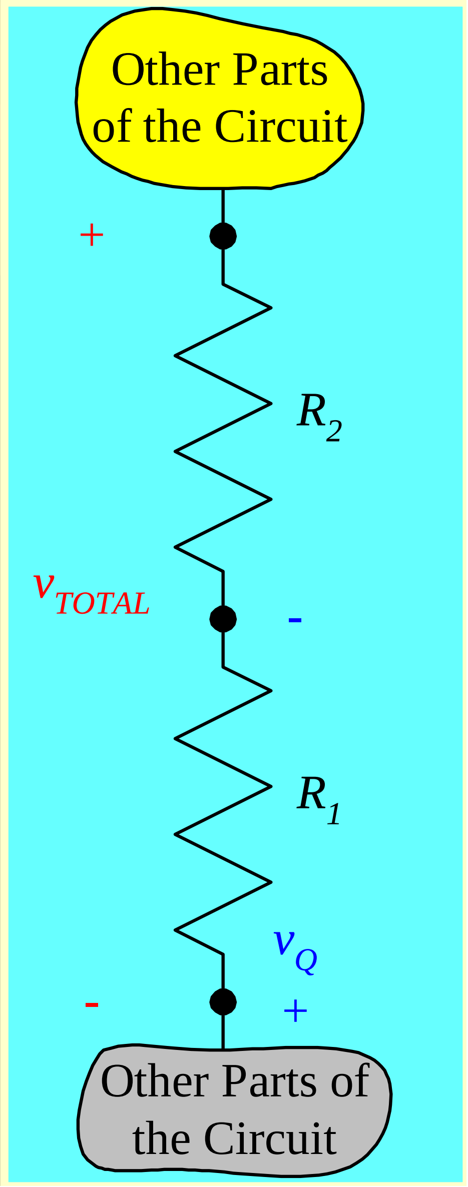
Go back to
Overview
slide.

Check for Signs in the Voltage Divider Rule

The rule for proper use of this tool, then, is to check the relative polarity of the voltage across the series resistors, and the voltage across one of the resistors.

$$v_Q = -v_{TOTAL}\frac{R_1}{R_1 + R_2}.$$

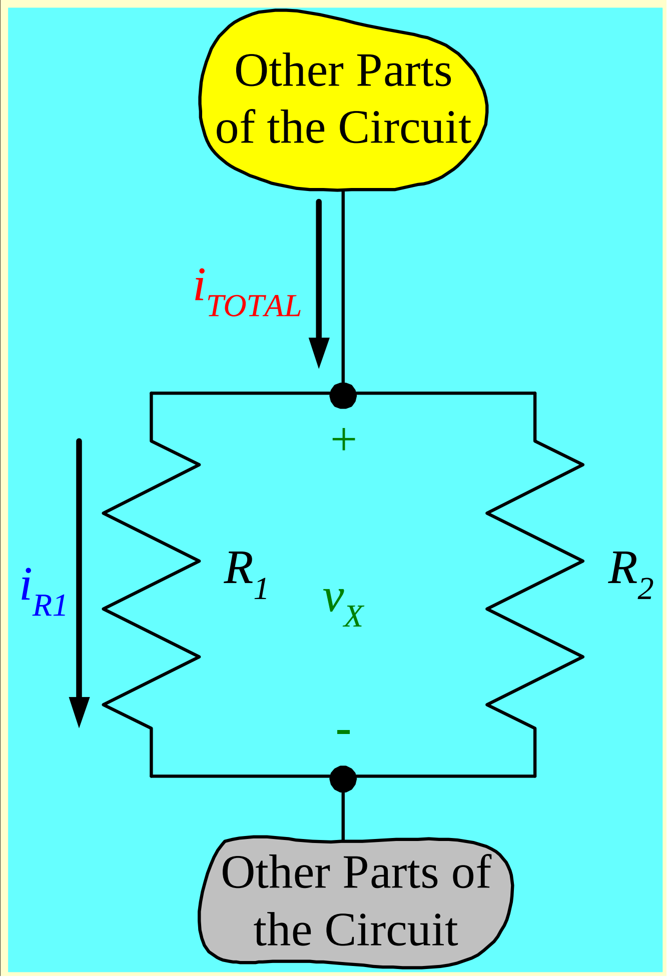




Signs in the Current Divider Rule

As in every equations we write, we need to be careful about the sign in the Current Divider Rule (CDR). Notice that when we wrote this expression, there is a positive sign. This is because the current i_{TOTAL} is in the same relative polarity as i_{R1}.

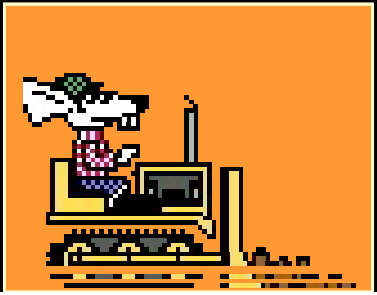
$$i_{R1} = +i_{TOTAL} \frac{R_2}{R_1 + R_2}.$$

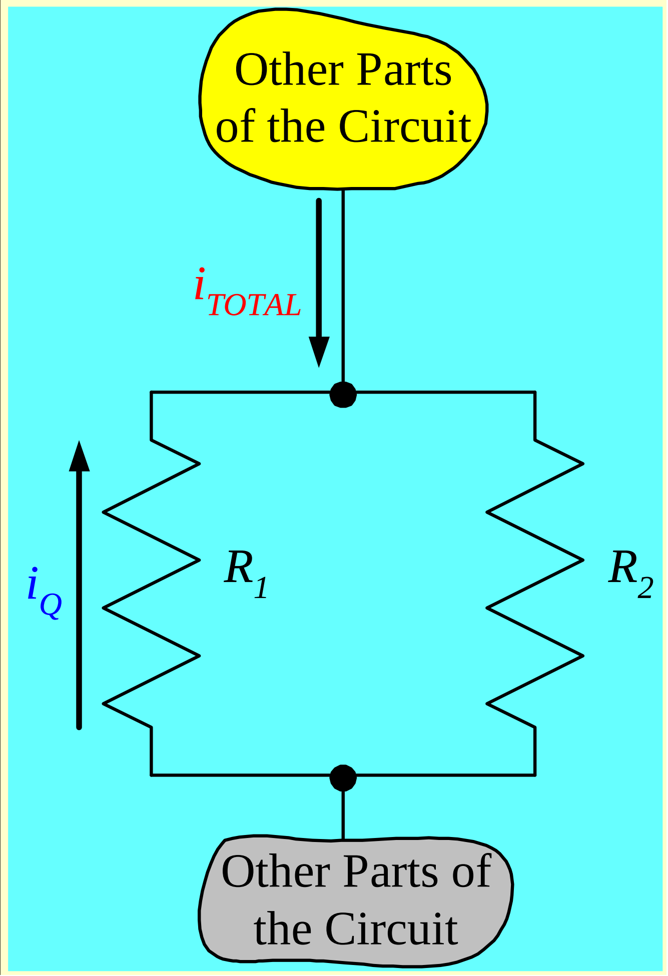


Negative Signs in the Current Divider Rule

If, instead, we had solved for i_Q, we would need to change the sign in the equation. This is because the current i_{TOTAL} is in the opposite relative polarity from i_Q.

$$i_Q = -i_{TOTAL}\frac{R_2}{R_1 + R_2}.$$



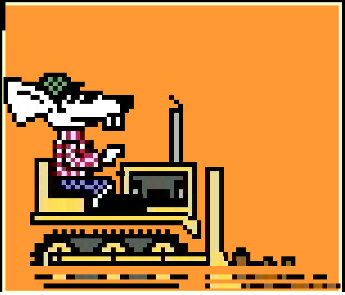


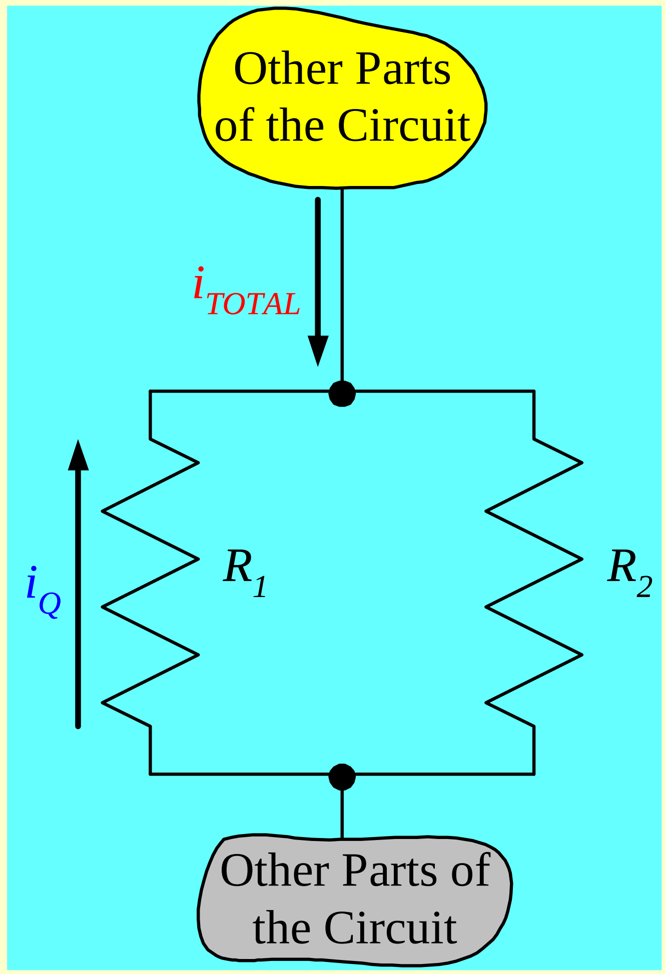
Go back to Overview slide.

Check for Signs in the Current Divider Rule

The rule for proper use of this tool, then, is to check the relative polarity of the current through the parallel resistors, and the current through one of the resistors.

$$i_Q = -i_{TOTAL} \frac{R_2}{R_1 + R_2}.$$





Do We Always Need to Worry About Signs?

- Unfortunately, the answer to this question is: YES! There is almost always a question of what the sign should be in a given circuits equation. The key is to learn how to get the sign right every time. As mentioned earlier, this is the key purpose in introducing reference polarities.

Solving circuit problems

1. Identify what the problem is asking you to find. Assign variables to the unknown quantities.
2. Make a large clear diagram of the circuit. Label all of the known resistances, currents, and voltages. Use the variables you defined to label the unknowns.
3. You may need to combine resistances to find the total circuit resistance. Use multiple steps to combine series and

Solving circuit problems

4. If you know the total resistance and current, use Ohm's law as $V = IR$ to calculate voltages or voltage drops. If you know the resistance and voltage, use Ohm's law as $I = V \div R$ to calculate the current.
5. An unknown resistance can be found using Ohm's law as $R = V \div I$, if you know the current and the voltage drop through the resistor.

Network circuits

- In many circuits, resistors are connected both in series and in parallel.
- Such a circuit is called a network circuit.
- There is no single formula for adding resistors in a network circuit.
- For very complex circuits, electrical engineers use computer programs that can rapidly solve equations for the circuit using Kirchhoff’s laws.

Kirchhoff's Laws in Detail

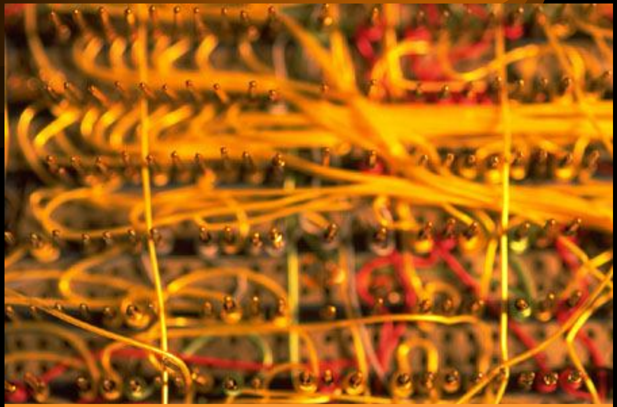
Overview of this Part

In this part of the module, we will cover the following topics:

- Some Basic Assumptions
- Kirchhoff's Current Law (KCL)
- Kirchhoff's Voltage Law (KVL)

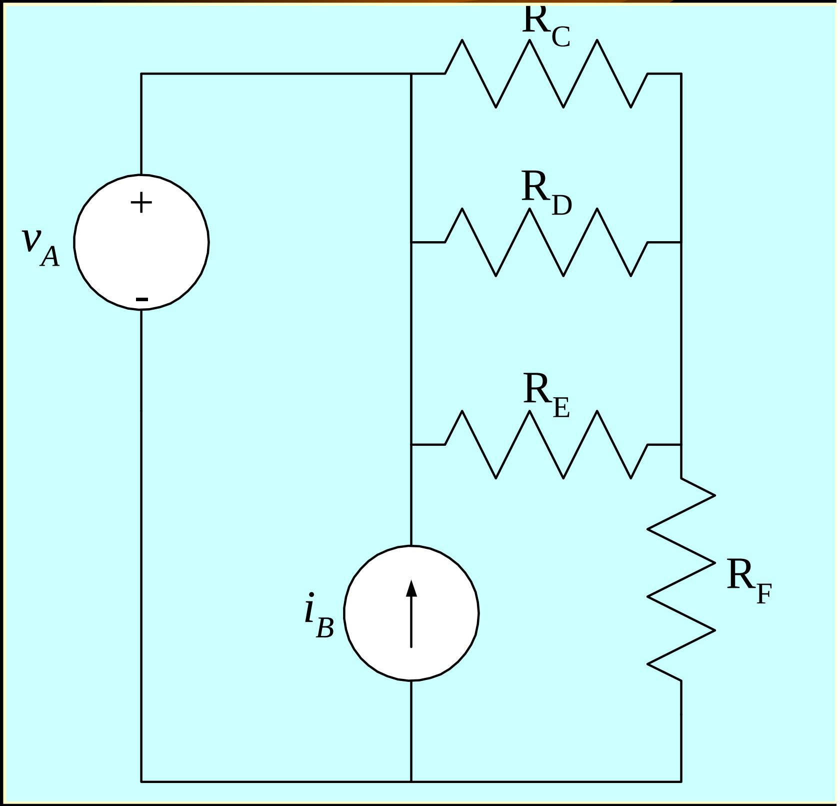
Some Fundamental Assumptions – Wires

- Although you may not have stated it, or thought about it, when you have drawn circuit schematics, you have connected components or devices with wires, and shown this with lines.
- Wires can be modeled pretty well as resistors. However, their resistance is usually negligibly small.
- We will think of wires as connections with zero resistance. Note that this is equivalent to having a zero-valued voltage source.


This picture shows wires used to connect electrical components. This particular way of connecting components is called wirewrapping, since the ends of the wires are wrapped around posts.

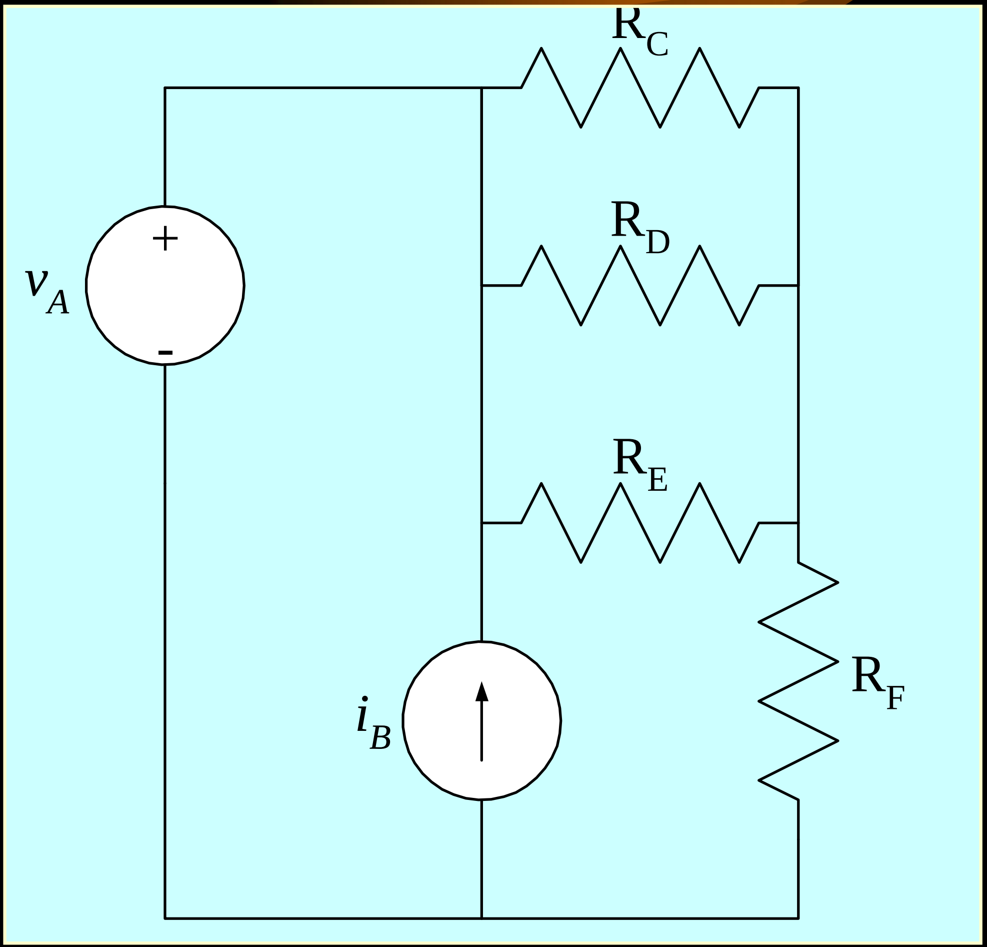
Some Fundamental Assumptions – Nodes

- A node is defined as a place where two or more components are connected.
- The key thing to remember is that we connect components with wires. It doesn't matter how many wires are being used; it only matters how many components are connected together.



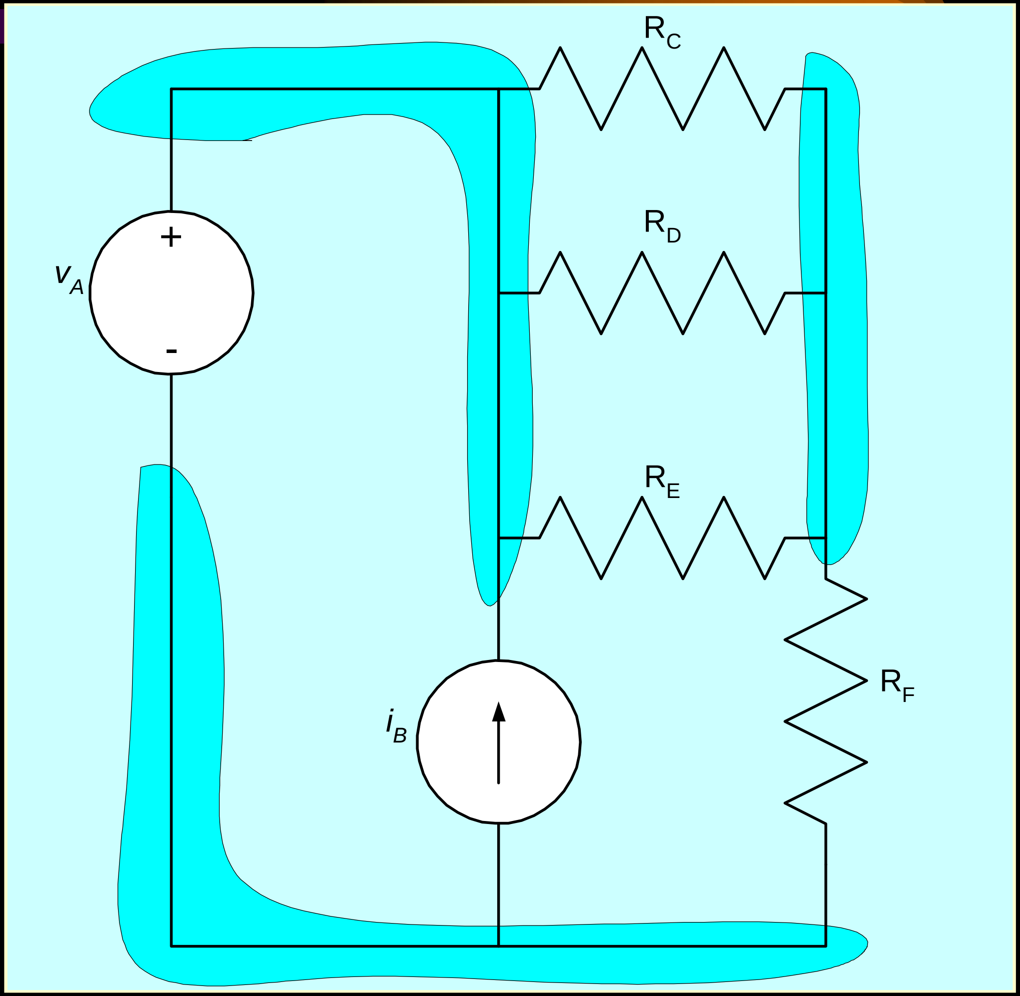
How Many Nodes?

- To test our understanding of nodes, let’s look at the example circuit schematic given here.
- How many nodes are there in this circuit?



How Many Nodes – Correct Answer

- In this schematic, there are three nodes. These nodes are shown in dark blue here.
- Some students count more than three nodes in a circuit like this. When they do, it is usually because they have considered two points connected by a wire to be two nodes.

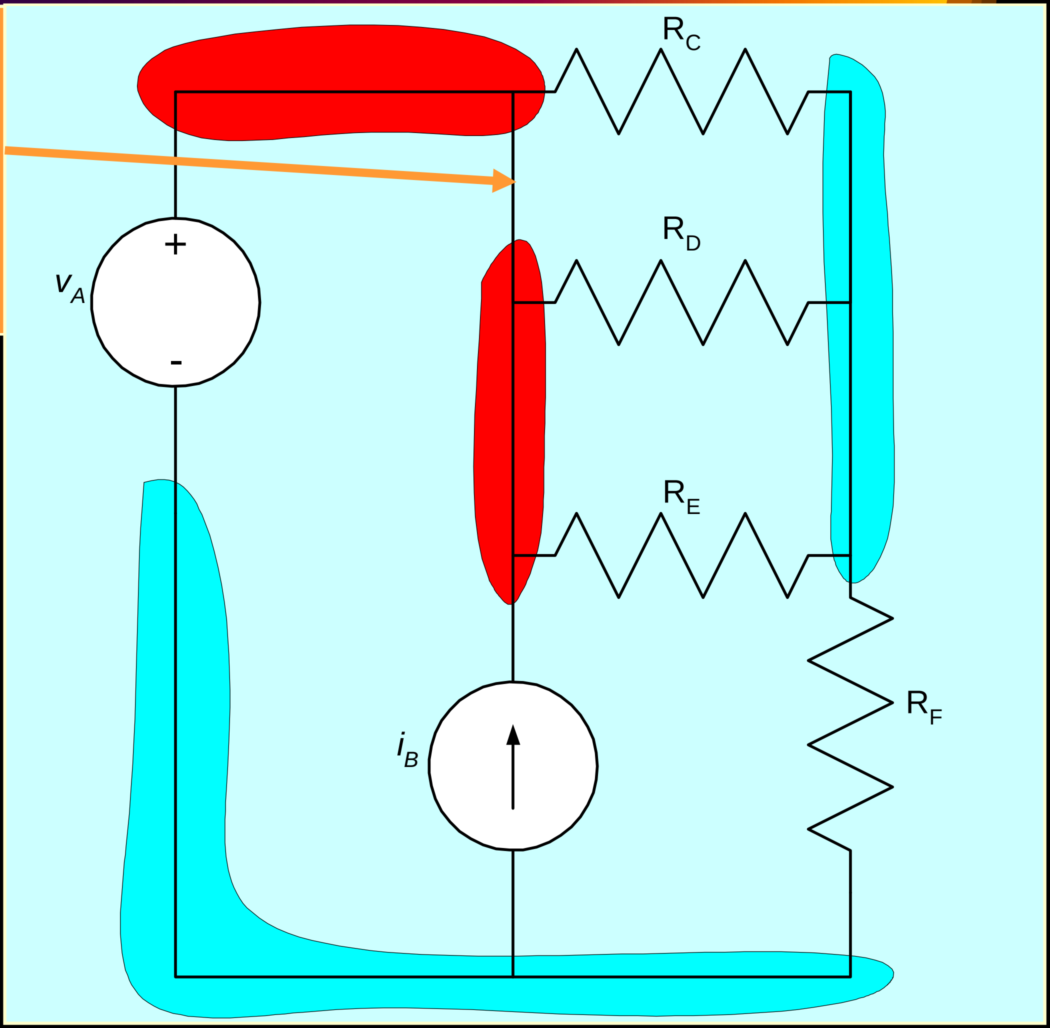


DCE,Gurgaon

How Many Nodes – Wrong Answer

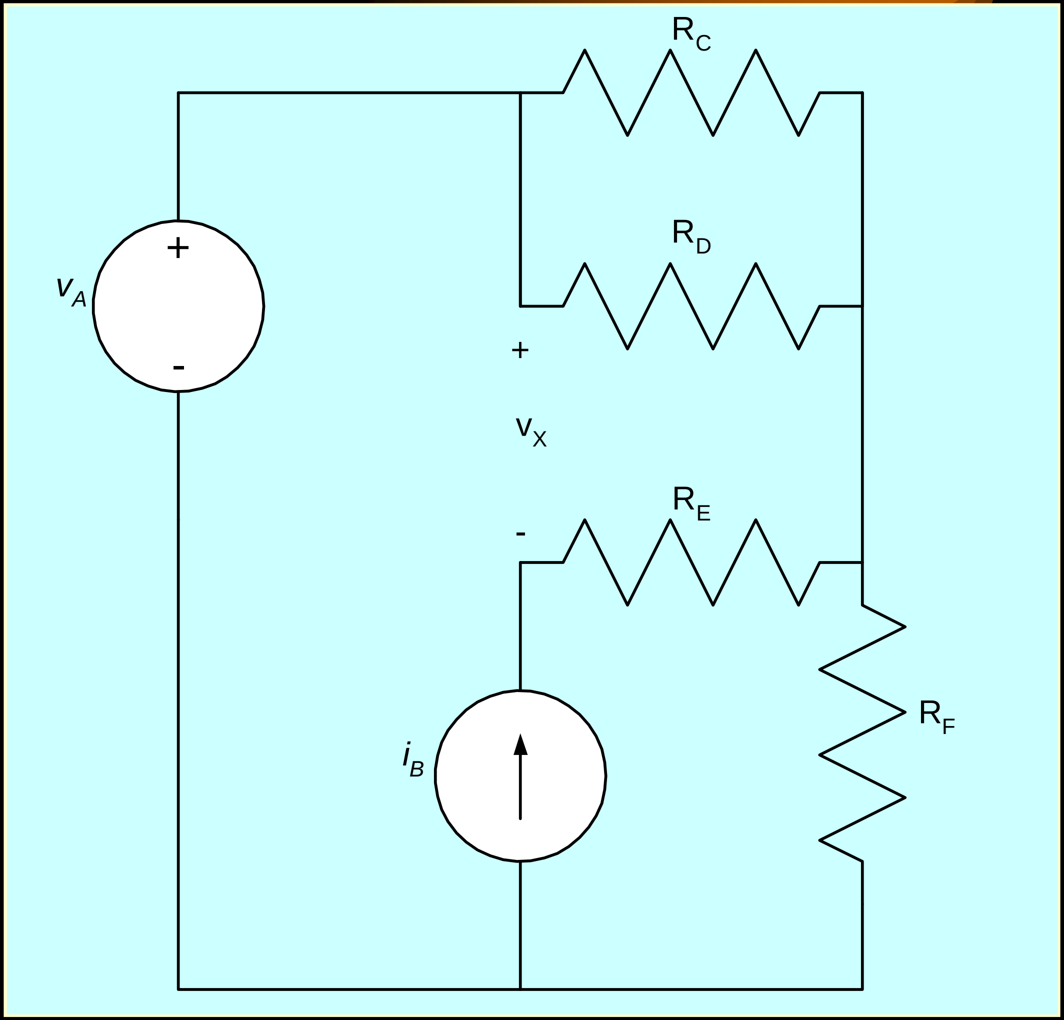
Wire connecting two nodes means that these are really a single node.

- In the example circuit schematic given here, the two red nodes are really the same node. There are not four nodes.
- Remember, two nodes connected by a wire were really only one node in the first place.



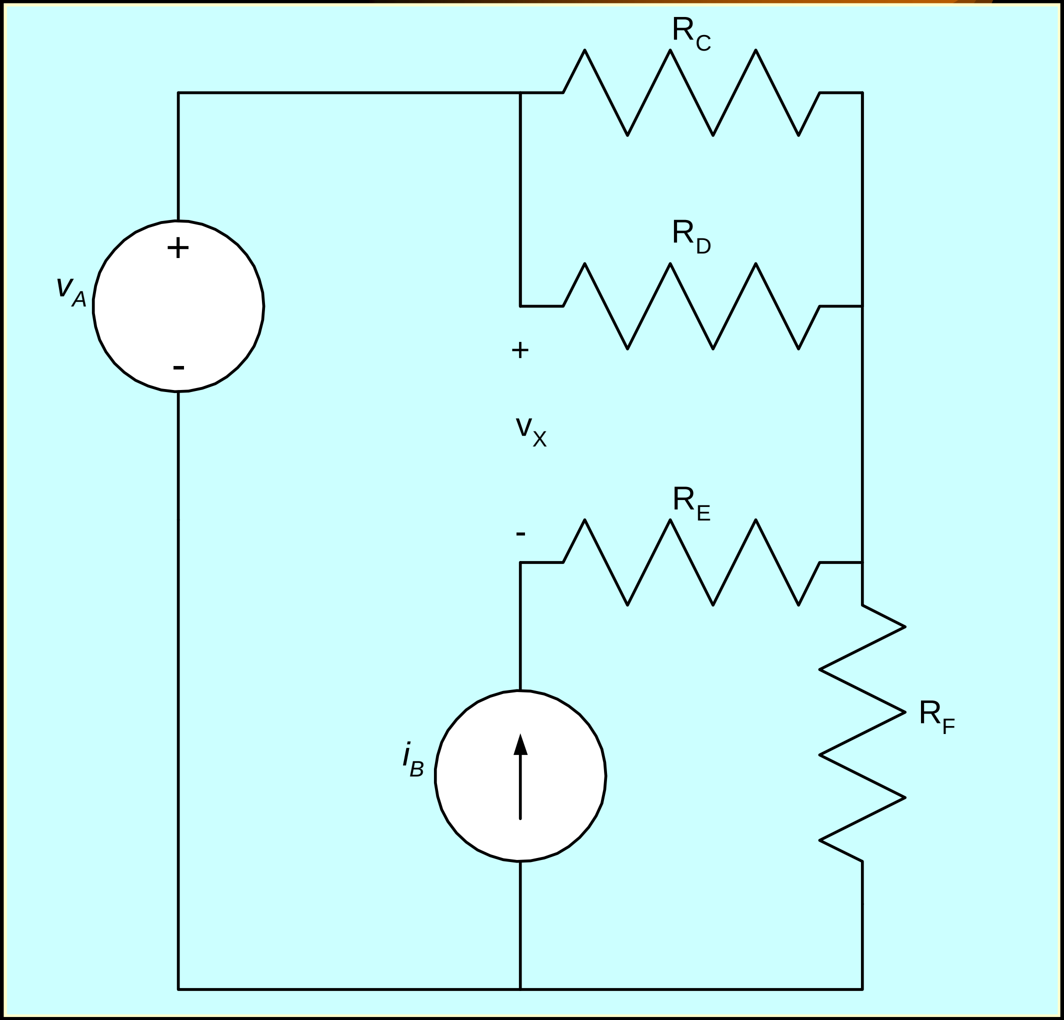
Some Fundamental Assumptions – Closed Loops

- A closed loop can be defined in this way: Start at any node and go in any direction and end up where you start. This is a closed loop.
- Note that this loop does not have to follow components. It can jump across open space. Most of the time we will follow components, but we will also have situations where we need to jump between nodes that have no connections.



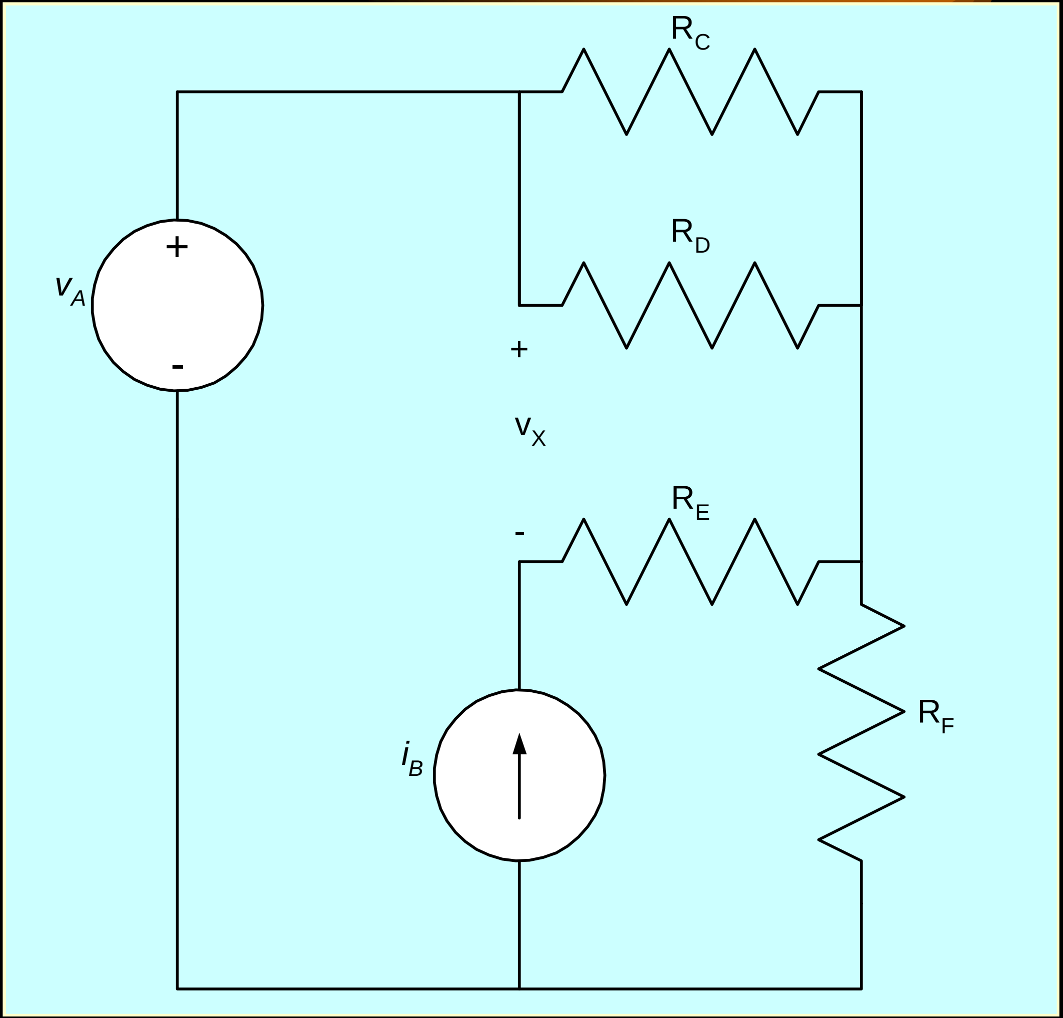
How Many Closed Loops

- To test our understanding of closed loops, let's look at the example circuit schematic given here.
- How many closed loops are there in this circuit?



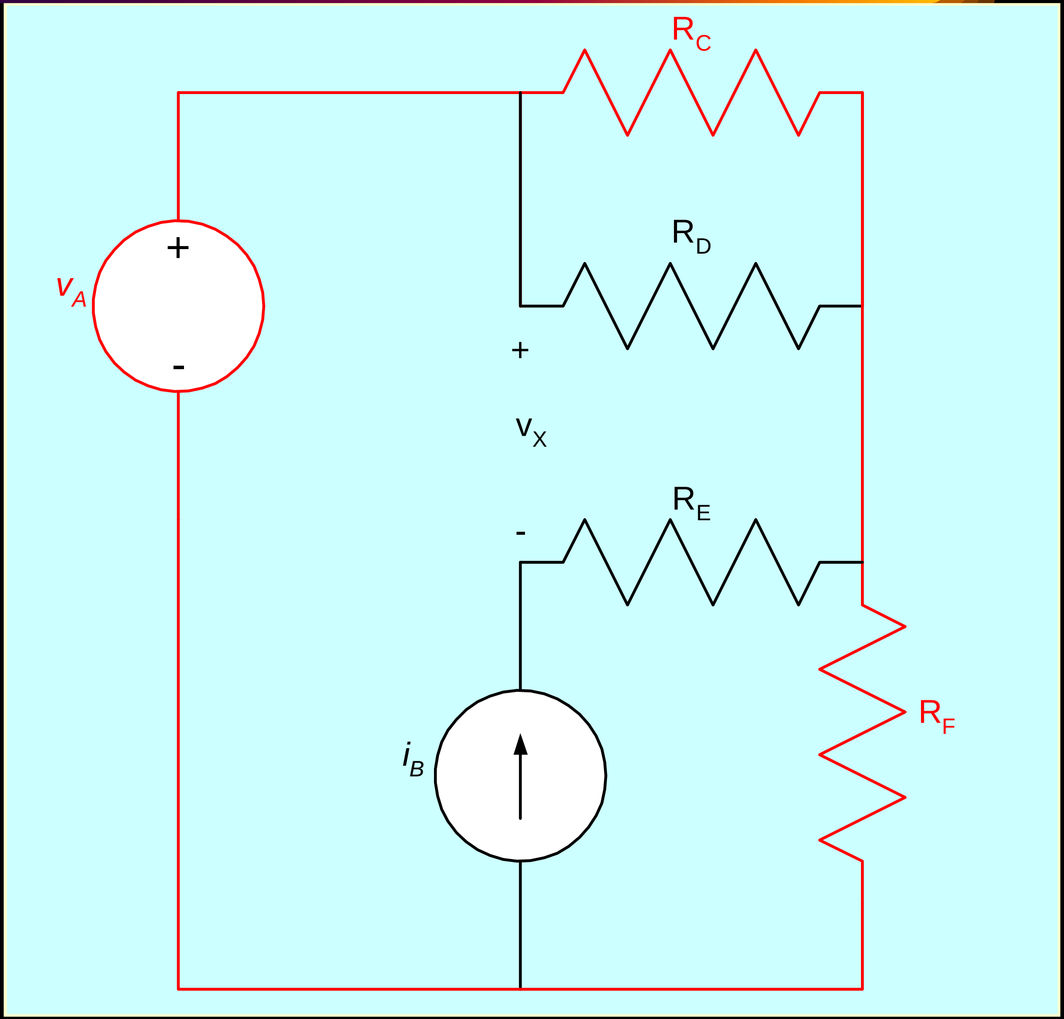
How Many Closed Loops – An Answer

- There are several closed loops that are possible here. We will show a few of them, and allow you to find the others.
- The total number of simple closed loops in this circuit is 13.
- Finding the number will not turn out to be important. What is important is to recognize closed loops when you see them.



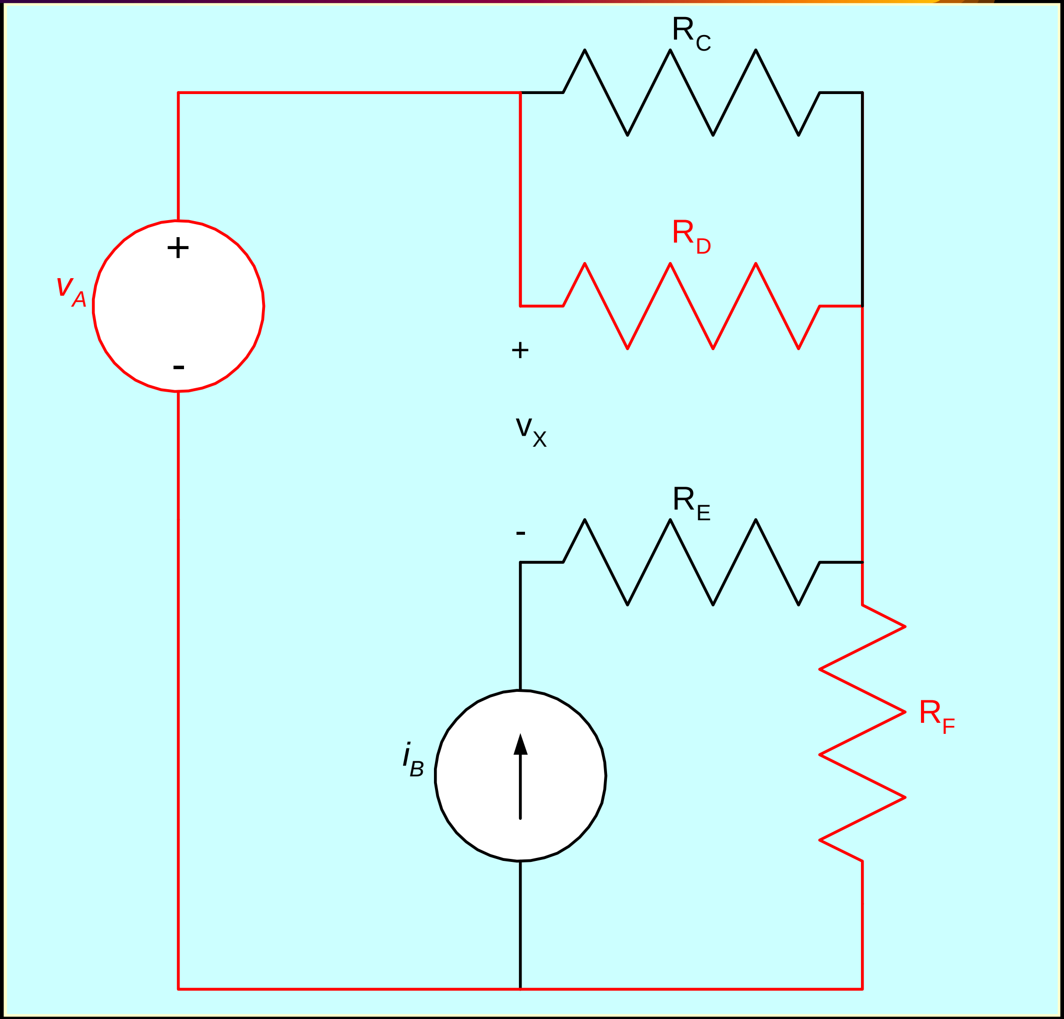
Closed Loops – Loop #1

- Here is a loop we will call Loop #1. The path is shown in red.



Closed Loops – Loop #2

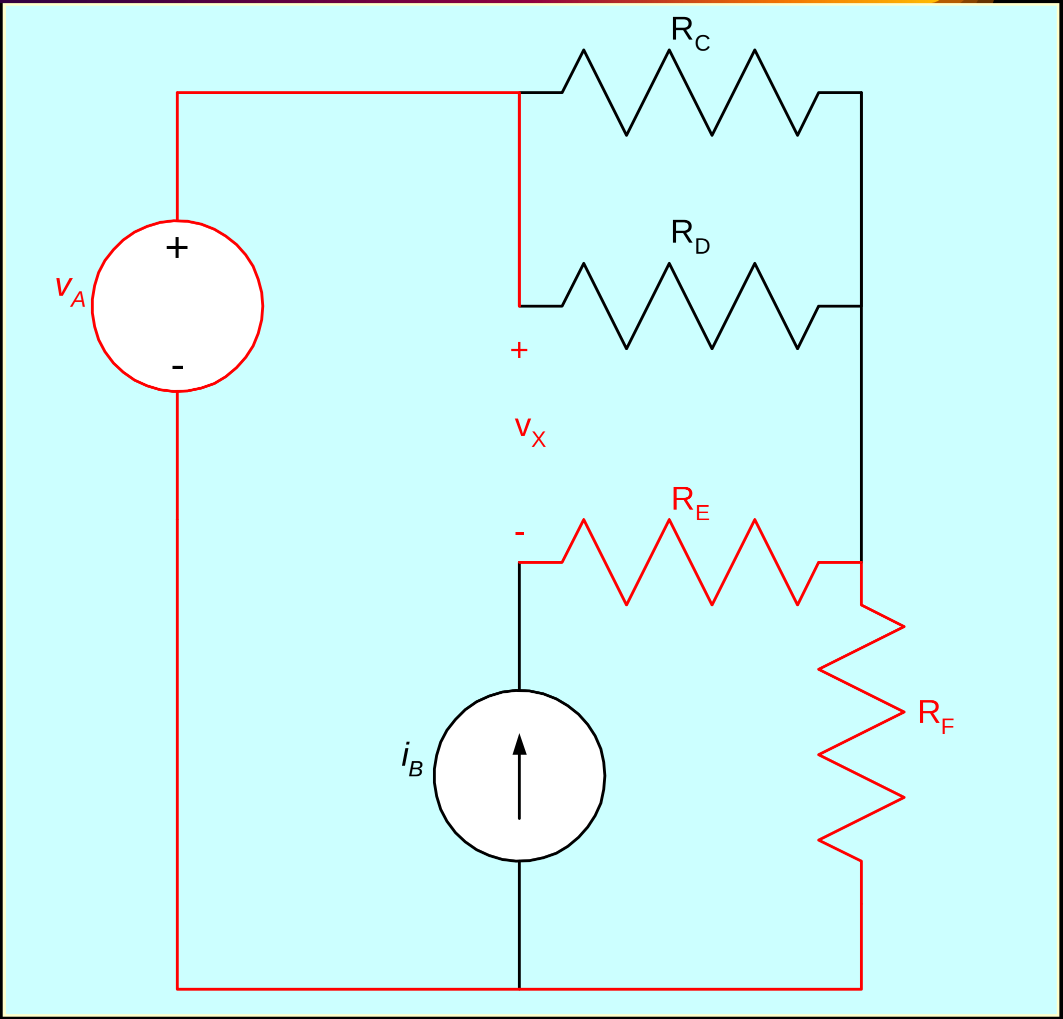
- Here is Loop #2. The path is shown in red.



Closed Loops – Loop #3

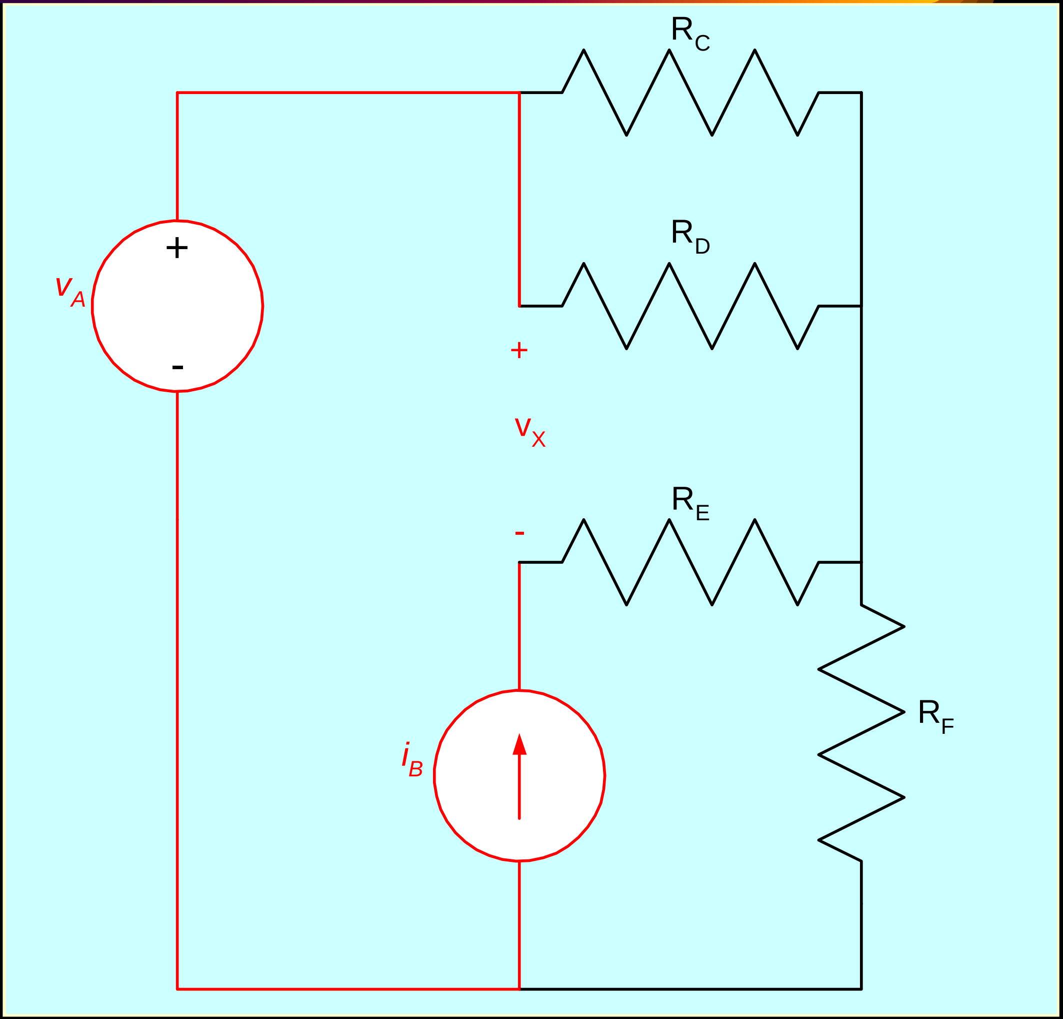
- Here is Loop #3. The path is shown in red.

- Note that this path is a closed loop that jumps across the voltage labeled v_X. This is still a closed loop.



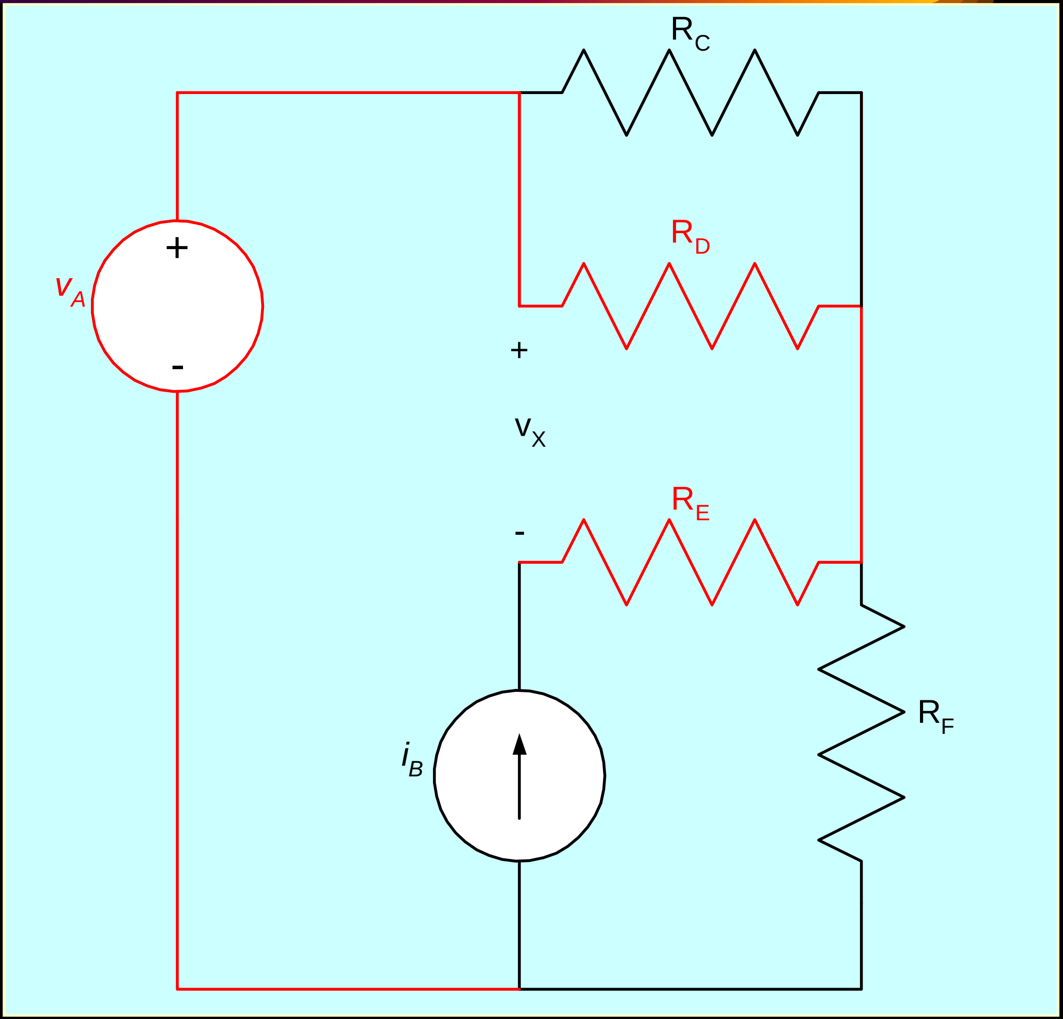
Closed Loops – Loop #4

- Here is Loop #4. The path is shown in red.
- Note that this path is a closed loop that jumps across the voltage labeled v_X. This is still a closed loop. The loop also crossed the current source. Remember that a current source can have a voltage across it.



A Not-Closed Loop

- The path is shown in red here is not closed.
- Note that this path does not end where it started.



Kirchhoff’s Current Law (KCL)

- With these definitions, we are prepared to state Kirchhoff’s Current Law:

The algebraic (or signed) summation of currents through a closed surface must equal zero.

DCE,Gurgaon

Kirchhoff's Current Law (KCL) – Some notes.

<u>The algebraic (or signed) summation of currents through any closed surface must equal zero.</u>

This definition essentially means that charge does not build up at a connection point, and that charge is conserved.

This definition is often stated as applying to nodes. It applies to any closed surface. For any closed surface, the charge that enters must leave somewhere else. A node is just a <u>small</u> closed surface. A node is the closed surface that we use most often. But, we can use any closed surface, and sometimes it is really necessary to use closed surfaces that are not nodes.

Current Polarities

Again, the issue of the sign, or polarity, or direction, of the current arises. When we write a Kirchhoff Current Law equation, we attach a sign to each reference current polarity, depending on whether the reference current is entering or leaving the closed surface. This can be done in different ways.

Kirchhoff's Current Law (KCL) – a Systematic Approach

The algebraic (or signed) summation of currents through any closed surface must equal zero.

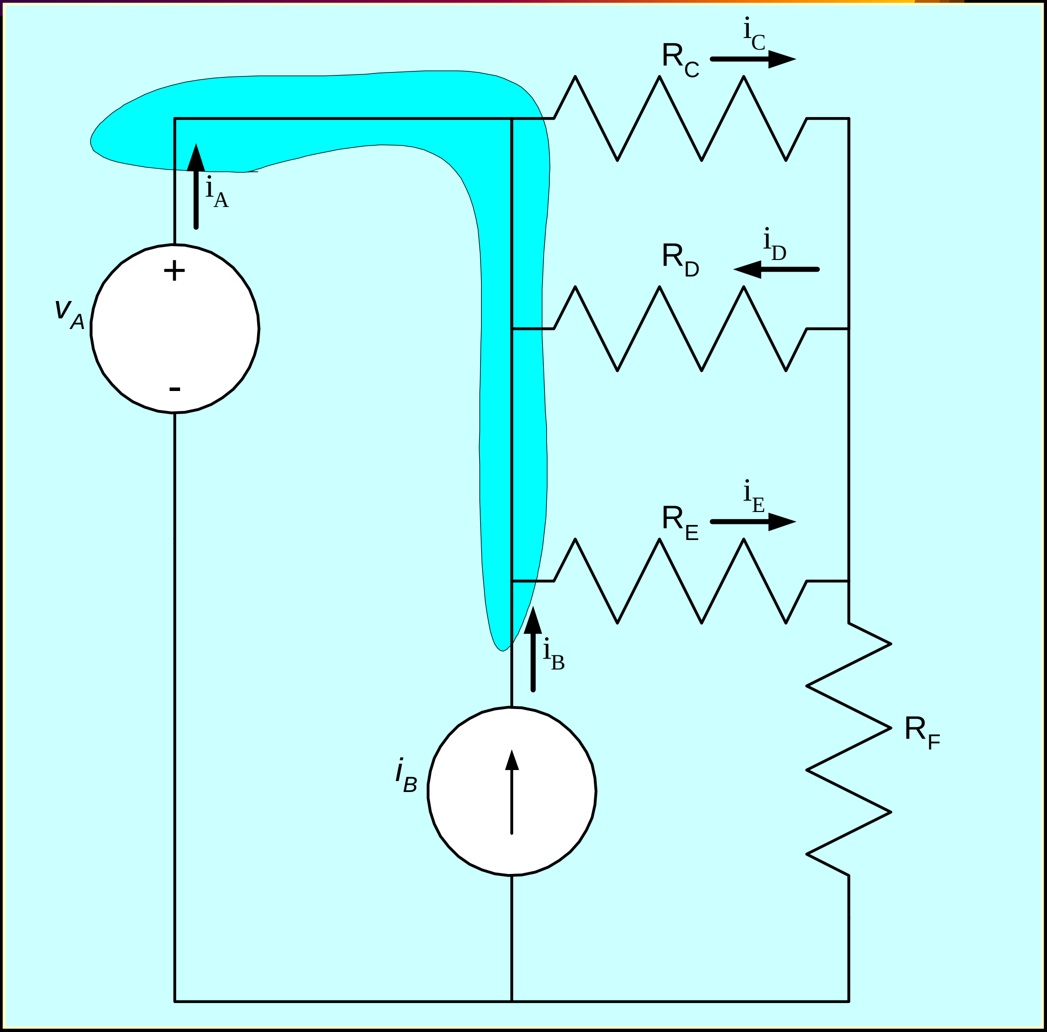
For most students, it is a good idea to choose one way to write KCL equations, and just do it that way every time. The idea is this: If you always do it the same way, you are less likely to get confused about which way you were doing it in a certain equation.

For this set of material, we will always assign a positive sign to a term that refers to a reference current that leaves a closed surface, and a negative sign to a term that refers to a reference current that enters a closed surface.

Kirchhoff's Current Law (KCL) – an Example

- For this set of material, we will always assign a positive sign to a term that refers to a current that leaves a closed surface, and a negative sign to a term that refers to a current that enters a closed surface.
- In this example, we have already assigned reference polarities for all of the currents for the nodes indicated in darker blue.
- For this circuit, and using my rule, we have the following equation:

$$-i_A + i_C - i_D + i_E - i_B = 0$$



DCE,Gurgaon

Kirchhoff's Current Law (KCL) – Example Done Another Way

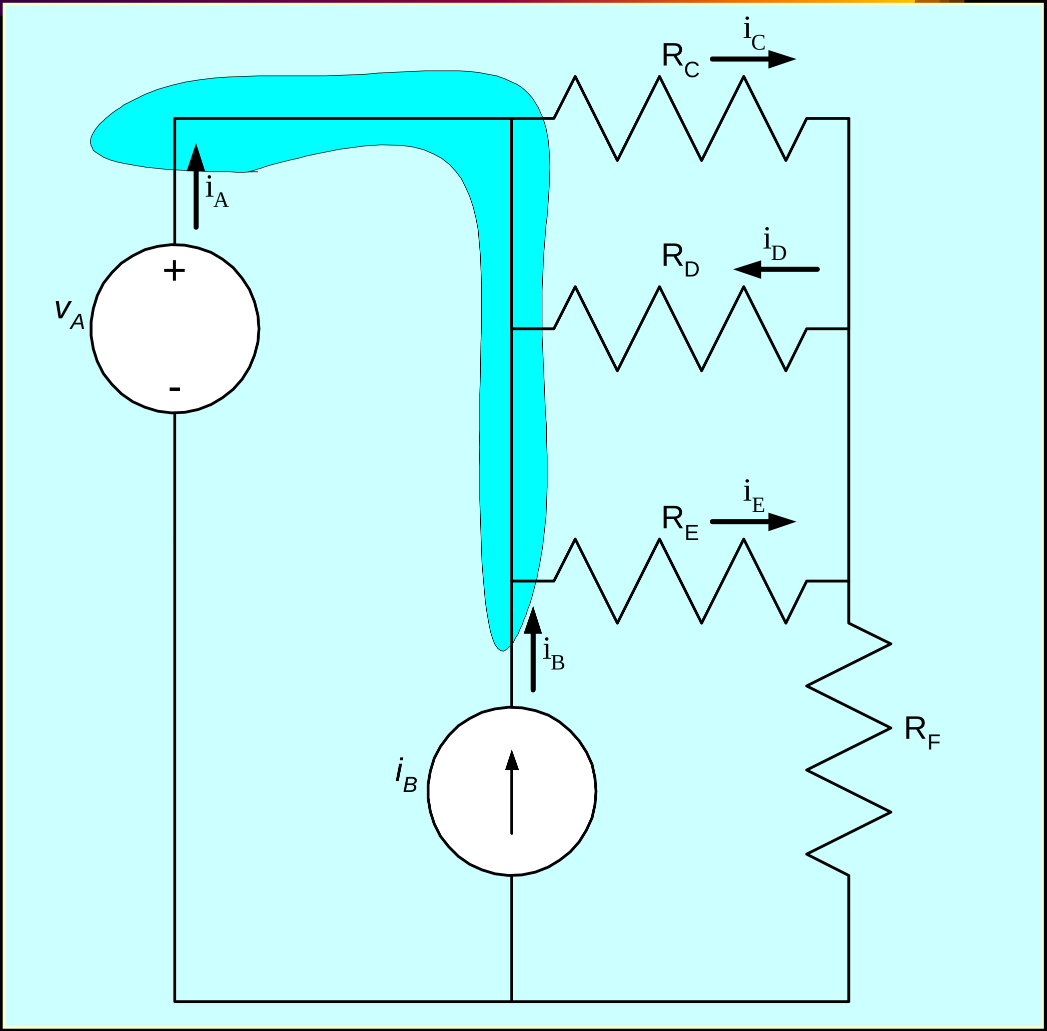
- Some prefer to write this same equation in a different way; they say that the current entering the closed surface must equal the current leaving the closed surface. Thus, they write :

$$i_A + i_D + i_B = i_C + i_E$$

- Compare this to the equation that we wrote in the last slide:

$$-i_A + i_C - i_D + i_E - i_B = 0$$

- These are the same equation. Use either method.



Kirchhoff's Voltage Law (KVL)

- Now, we are prepared to state Kirchhoff's Voltage Law:

The algebraic (or signed) summation of voltages around a closed loop must equal zero.

Kirchhoff's Voltage Law (KVL) – Some notes.

The algebraic (or signed) summation of voltages around a closed loop must equal zero.

This definition essentially means that energy is conserved. If we move around, wherever we move, if we end up in the place we started, we cannot have changed the potential at that point.

This applies to all closed loops. While we usually write equations for closed loops that follow components, we do not need to. The only thing that we need to do is end up where we started.

Voltage Polarities

Again, the issue of the sign, or polarity, or direction, of the voltage arises. When we write a Kirchhoff Voltage Law equation, we attach a sign to each reference voltage polarity, depending on whether the reference voltage is a rise or a drop. This can be done in different ways.

Kirchhoff's Voltage Law (KVL) – a Systematic Approach

The algebraic (or signed) summation of voltages around a closed loop must equal zero.

For most students, it is a good idea to choose one way to write KVL equations, and just do it that way every time. The idea is this: If you always do it the same way, you are less likely to get confused about which way you were doing it in a certain equation.

(At least we will do this for planar circuits. For nonplanar circuits, clockwise does not mean anything. If this is confusing, ignore it for now.)

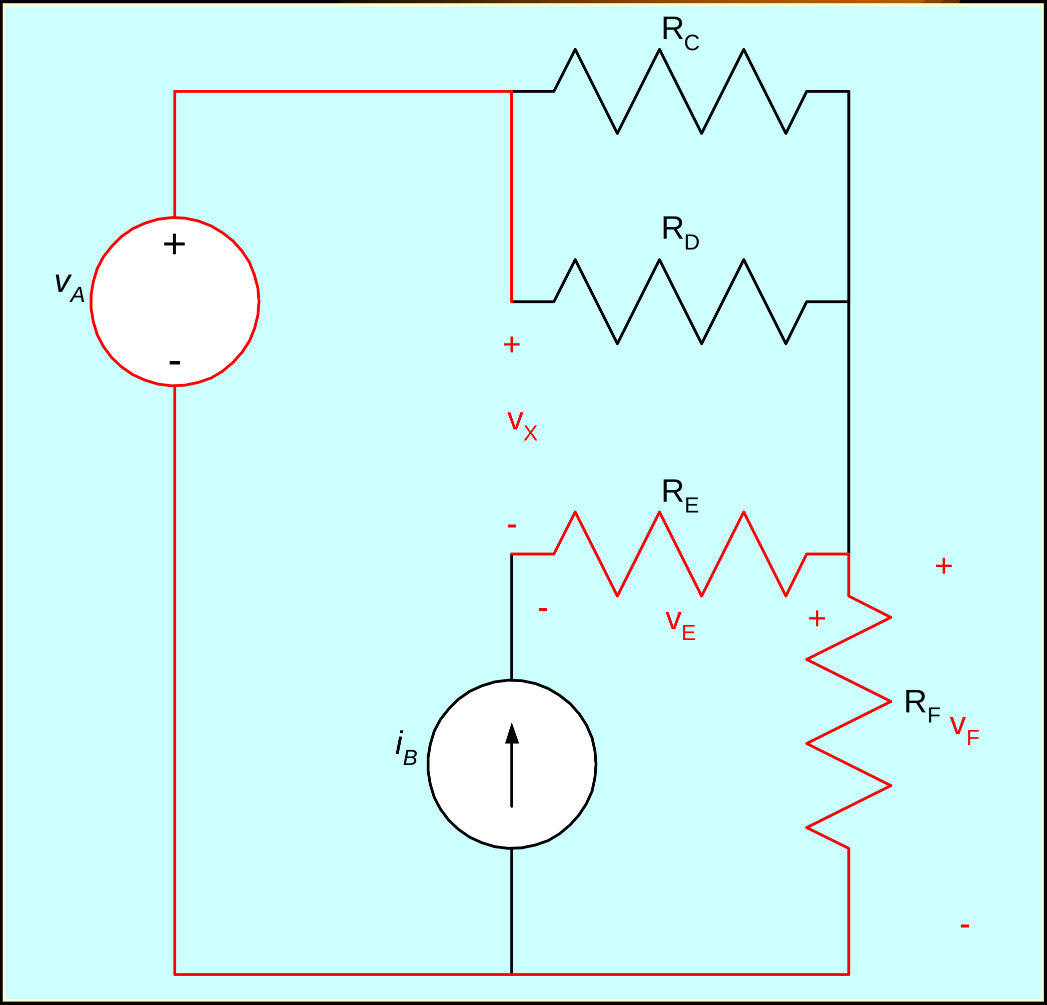
For this set of material, we will always go around loops clockwise. We will assign a positive sign to a term that refers to a reference voltage drop, and a negative sign to a term that refers to a reference voltage rise.

DCE,Gurgaon

Kirchhoff's Voltage Law (KVL) – an Example

- For this set of material, we will always go around loops clockwise. We will assign a positive sign to a term that refers to a voltage drop, and a negative sign to a term that refers to a voltage rise.
- In this example, we have already assigned reference polarities for all of the voltages for the loop indicated in red.
- For this circuit, and using our rule, starting at the bottom, we have the following equation:

$$-v_A + v_X - v_E + v_F = 0$$



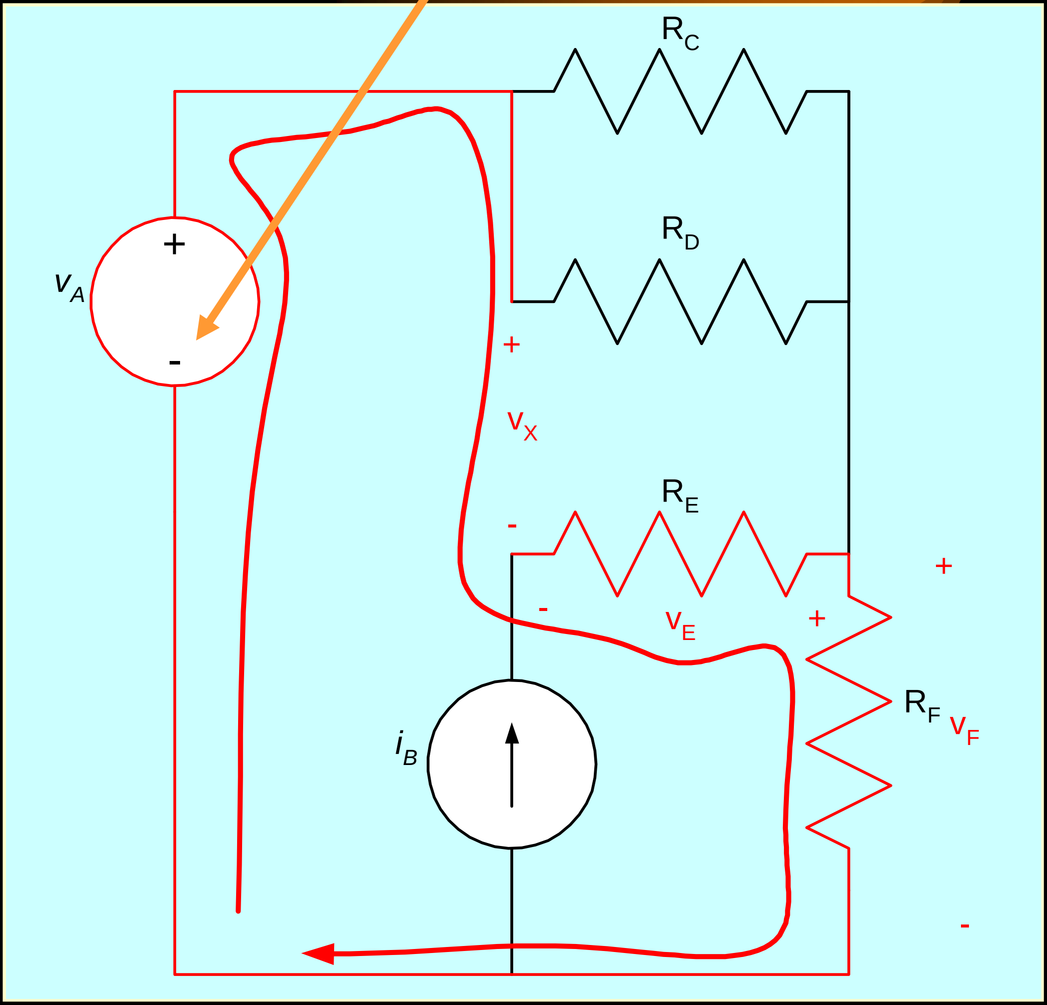
DCE,Gurgaon

Kirchhoff's Voltage Law (KVL) – Notes

As we go up through the voltage source, we enter the negative sign first. Thus, v_A has a negative sign in the equation.

- For this set of material, we will always go around loops clockwise. We will assign a positive sign to a term that refers to a voltage drop, and a negative sign to a term that refers to a voltage rise.
- Some students like to use the following handy mnemonic device: Use the sign of the voltage that is on the side of the voltage that you enter. This amounts to the same thing.

$$-v_A + v_X - v_E + v_F = 0$$



Kirchhoff's Voltage Law (KVL) – Example Done Another Way

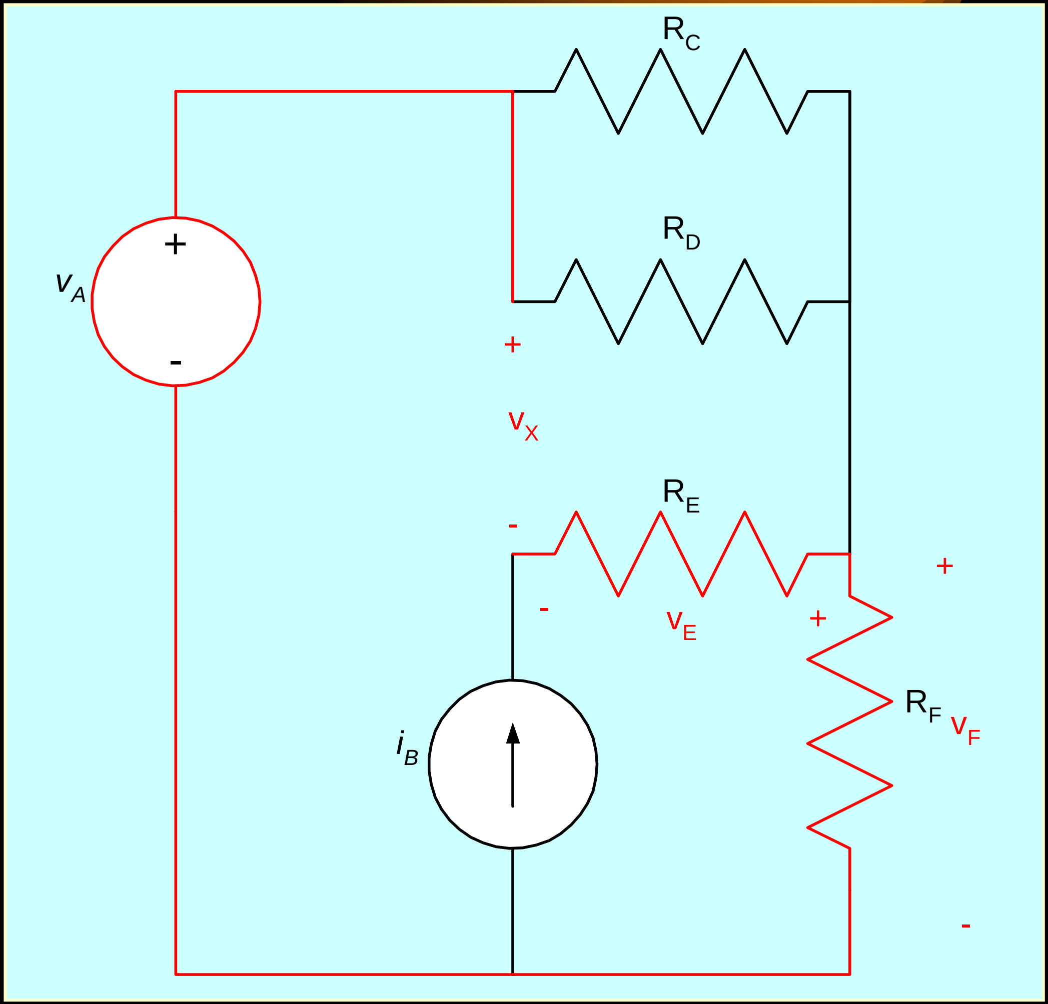
- Some textbooks, and some students, prefer to write this same equation in a different way; they say that the voltage drops must equal the voltage rises. Thus, they write the following equation:

$$v_X + v_F = v_A + v_E$$

Compare this to the equation that we wrote in the last slide:

$$-v_A + v_X - v_E + v_F = 0$$

These are the same equation. Use either method.



How many of these equations do I need to write?

- This is a very important question. In general, it boils down to the old rule that you need the same number of equations as you have unknowns.
- Speaking more carefully, we would say that to have a single solution, we need to have the same number of <u>independent equations</u> as we have variables.
- At this point, we are not going to introduce you to the way to know how many equations you will need, or which ones to write. It is assumed that you will be able to judge whether you have what you need because the circuits will be fairly simple. Later we will develop methods to answer this question specifically and efficiently.

How many more laws are we going to learn?

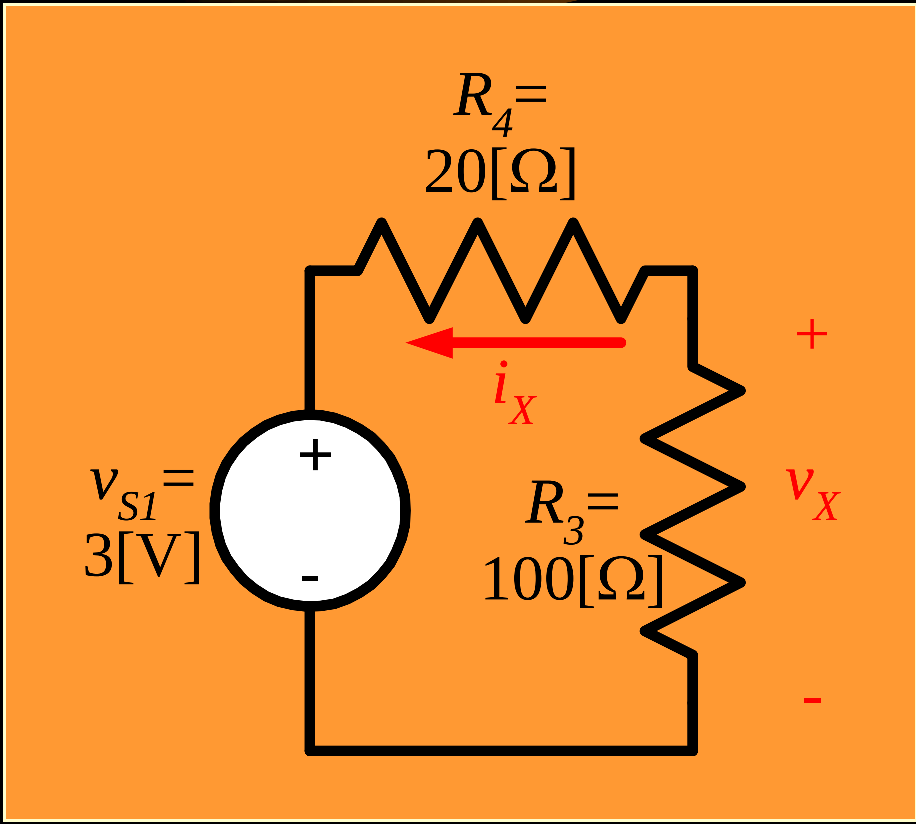
- This is another very important question. Until, we get to inductors and capacitors, the answer is, **<u>none</u>**.
- Speaking more carefully, we would say that most of the rules that follow until we introduce the other basic elements, can be derived from these laws.
- At this point, you have the tools to solve many, many circuits problems. Specifically, you have Ohm's Law, and Kirchhoff's Laws. However, we need to be able to use these laws efficiently and accurately.
  We will spend some time in ECE 2300 learning techniques, concepts and approaches that help us to do just that.

How many f's and h's are there in Kirchhoff?

- This is another **not**-important question. But, we might as well learn how to spell Kirchhoff. Our approach might be to double almost everything, but we might end up with something like Kirrcchhooff.
- We suspect that this is one reason why people typically abbreviate these laws as KCL and KVL. This is pretty safe, and seems like a pretty good idea to us.

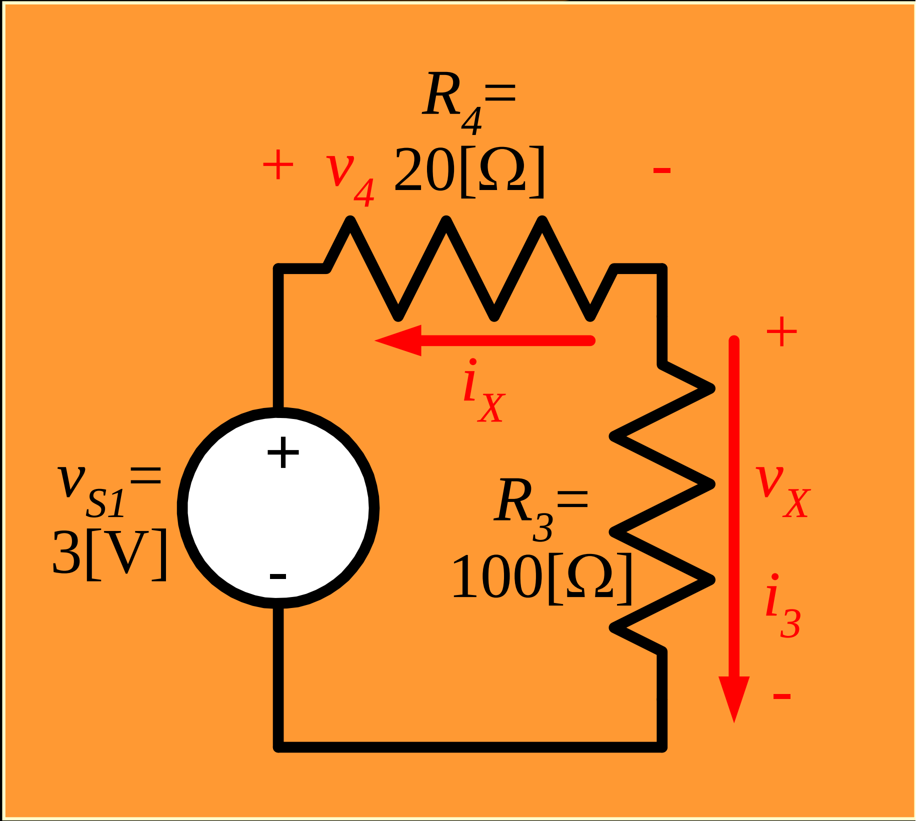
Example

- Let's do an example to test out our new found skills.
- In the circuit shown here, find the voltage v_X and the current i_X.



Example – Step 1

- The first step in solving is to define variables we need.
- In the circuit shown here, we will define v_4 and i_3.

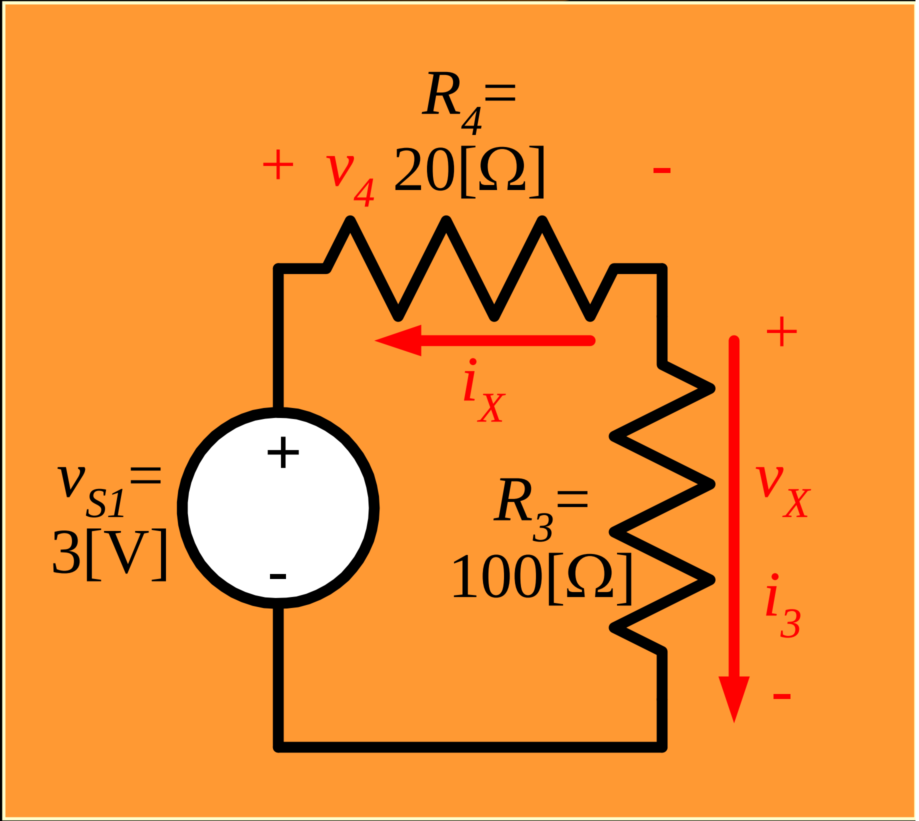


DCE,Gurgaon

Example – Step 2

- The second step in solving is to write some equations. Let’s start with KVL.

$$-v_{S1} + v_4 + v_X = 0, \text{ or}$$
$$-3[\text{V}] + v_4 + v_X = 0.$$

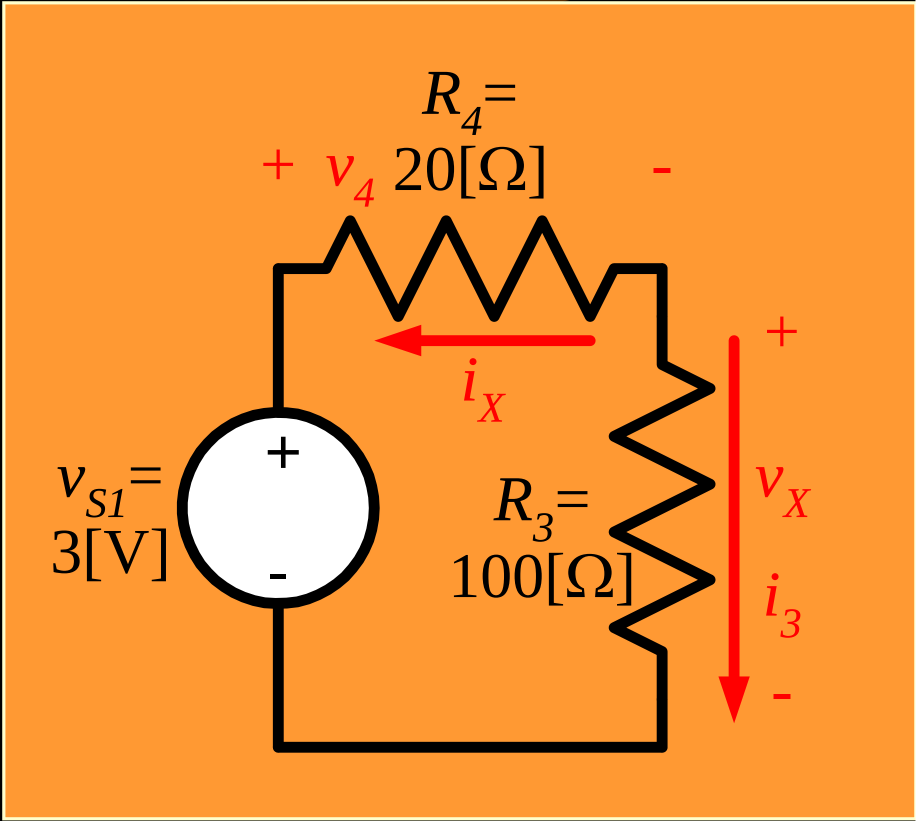


Example – Step 3

- Now let's write Ohm's Law for the resistors.

$$v_4 = -i_X R_4, \text{ and}$$
$$v_X = i_3 R_3.$$

Notice that there is a **sign** in Ohm's Law.

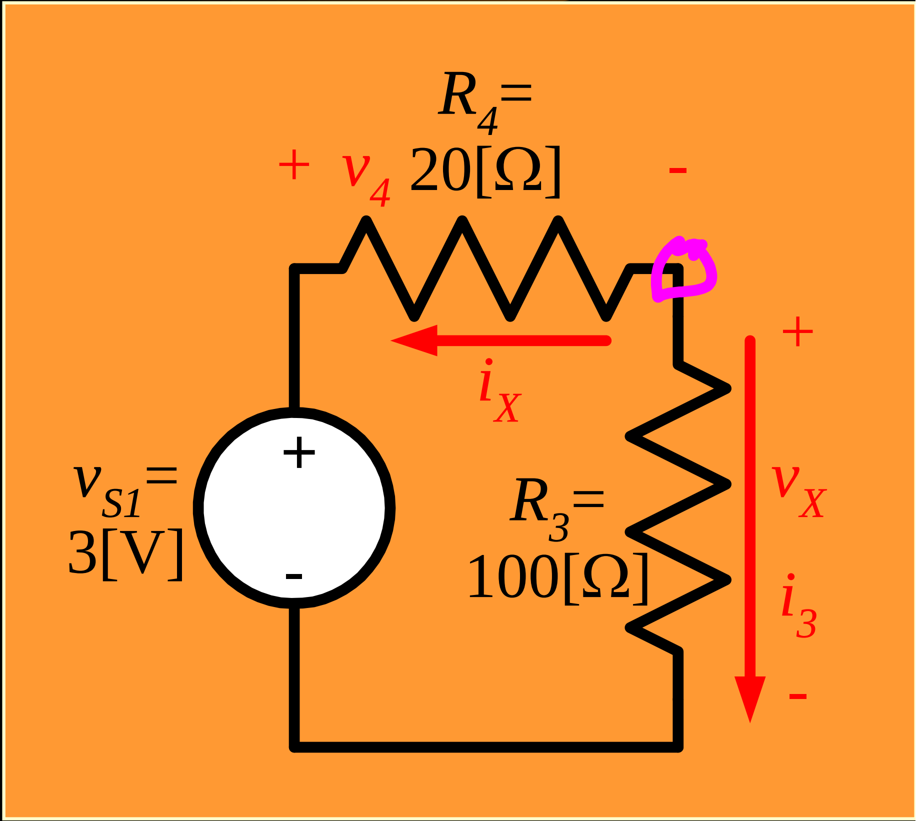


Example – Step 4

- Next, let's write KCL for the node marked in violet.

$$i_X + i_3 = 0,\ \text{or}$$
$$i_3 = -i_X\,.$$

Notice that we can write KCL for a node, or any other closed surface.



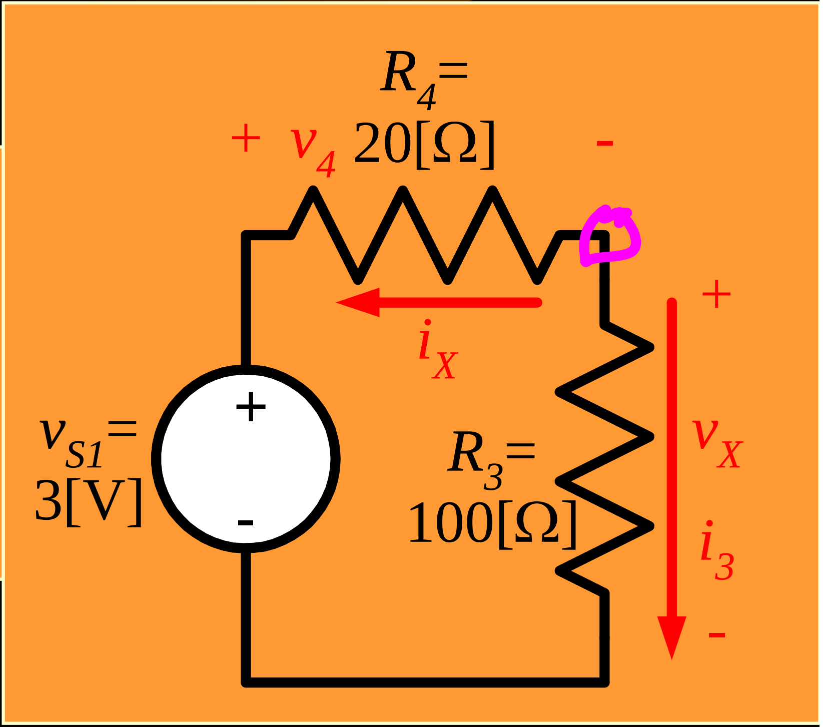
Example – Step 5

- We are ready to solve.

$$-3[\text{V}] - i_X\, 20[\Omega] - i_X\, 100[\Omega] = 0, \text{ or}$$

$$\boxed{i_X = \frac{-3[\text{V}]}{120[\Omega]} = -25[\text{mA}].}$$

We have substituted into our KVL equation from other equations.

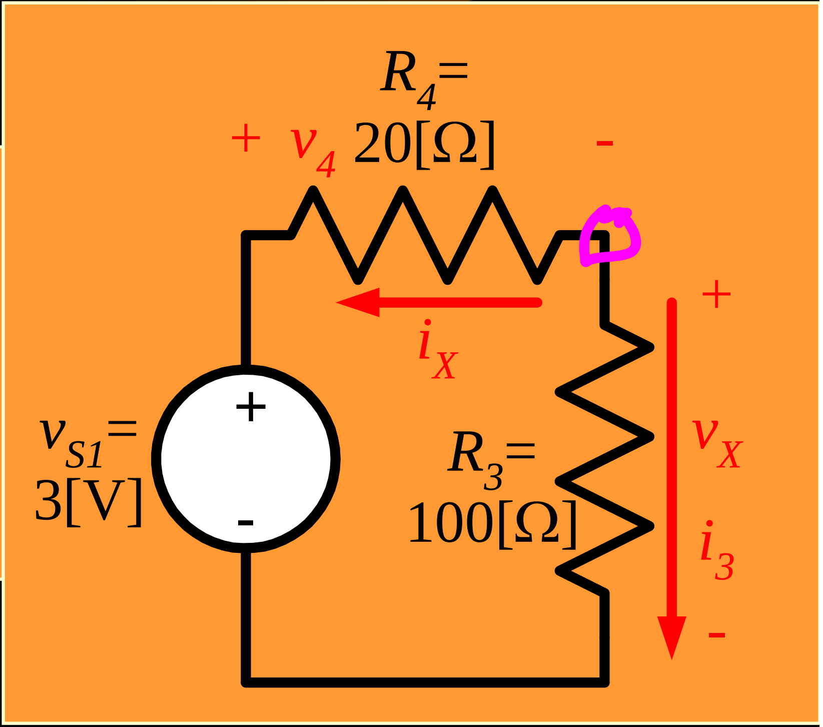


Example – Step 6

- Next, for the other requested solution.

$$v_X = i_3 R_3 = -i_X R_3, \text{ or}$$

$$\boxed{v_X = -(-25[\text{mA}])100[\Omega] = 2.5[\text{V}].}$$



We have substituted into Ohm’s Law, using our solution for i_X.

C Program to Illustrate Pass by Value

This C Program illustrates pass by value. This program is used to explain how pass by value function work Pass by Value: In this method, the value of each of the actual arguments in the calling function is copied int corresponding formal arguments of the called function. In pass by value, the changes made to formal argumen in the called function have no effect on the values of actual arguments in the calling function.

Here is source code of the C Program to illustrate pass by value. The C program is successfull compiled and run on a Linux system. The program output is also shown below.

```c
/*
 * C Program to Illustrate Pass by Value.
 */
#include <stdio.h>

void swap(int a, int b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}

int main()
{
    int num1 = 10, num2 = 20;

    printf("Before swapping num1 = %d num2 = %d\n", num1, num2);
    swap(num1, num2);
    printf("After swapping num1 = %d num2 = %d \n", num2, num1);
    return 0;
}
```

```
Output:
$ cc pgm43.c
$ a.out
Before swapping num1 = 10 num2 = 20
After swapping num1 = 20 num2 = 10
```

C Program to accept Sorted Array and do Search using Binary Search

This C Program accepts the sorted array and does search using Binary search. Binary search is a algorithm for locating the position of an item in a sorted array. A search of sorted data, in which th middle position is examined first. Search continues with either the left or the right portion of the data thus eliminating half of the remaining search space. In other words, a search which can be applied t an ordered linear list to progressively divide the possible scope of a search in half until the searc object is found.

Here is source code of the C program to accept the sorted array and do Search using Binary Searcl The C program is successfully compiled and run on a Linux system. The program output is also show below.

```c
/*
 * C program to accept N numbers sorted in ascending order
 * and to search for a given number using binary search.
 * Report success or failure.
 */
#include <stdio.h>

void main()
{
    int array[10];
    int i, j, num, temp, keynum;
    int low, mid, high;

    printf("Enter the value of num \n");
    scanf("%d", &num);
    printf("Enter the elements one by one \n");
    for (i = 0; i < num; i++)
    {
        scanf("%d", &array[i]);
    }
    printf("Input array elements \n");
    for (i = 0; i < num; i++)
    {
        printf("%d\n", array[i]);
    }
    /*  Bubble sorting begins */
    for (i = 0; i < num; i++)
    {
        for (j = 0; j < (num - i - 1); j++)
        {
            if (array[j] > array[j + 1])
            {
                temp = array[j];
                array[j] = array[j + 1];
                array[j + 1] = temp;
            }
    printf("Sorted array is...\n");
    for (i = 0; i < num; i++)
    {
        printf("%d\n", array[i]);
    }
    printf("Enter the element to be searched \n");
    scanf("%d", &keynum);
    /*  Binary searching begins */
    low = 1;
    high = num;
    do
    {
        mid = (low + high) / 2;
        if (keynum < array[mid])
            high = mid - 1;
        else if (keynum > array[mid])
            low = mid + 1;
    } while (keynum != array[mid] && low <= high);
    if (keynum == array[mid])
    {
        printf("SEARCH SUCCESSFUL \n");
    }
    else
    {
        printf("SEARCH FAILED \n");
    }
}
```

```
$ cc pgm22.c
$ a.out
Enter the value of num
5
Enter the elements one by one
23
90
56
15
58
Input array elements
23
90
56
15
58
Sorted array is...
15
23
56
58
90
Enter the element to be searched
```

```
$ a.out
Enter the value of num
4
Enter the elements one by one
1
98
65
45
Input array elements
1
98
65
45
Sorted array is...
1
45
65
98
Enter the element to be searched
6
SEARCH FAILED
```

C Program to Accepts two Strings & Compare them

This C Program accepts two strings & compares them. The program accepts two strings say string1 and string2. If both the strings are equal then it display both strings are equal. If string1 > string2 then display a appropriat message and so on.

Here is source code of the C program to accepts two strings & compare them. The C program i successfully compiled and run on a Linux system. The program output is also shown below.

```c
/*
 * C Program to accepts two strings and compare them. Display
 * the result whether both are equal, or first string is greater
 * than the second or the first string is less than the second string
 */
#include <stdio.h>

void main()
{
    int count1 = 0, count2 = 0, flag = 0, i;
    char string1[10], string2[10];

    printf("Enter a string:");
    gets(string1);
    printf("Enter another string:");
    gets(string2);
    /*  Count the number of characters in string1 */
    while (string1[count1] != '\0')
        count1++;
    /*  Count the number of characters in string2 */
    while (string2[count2] != '\0')
        count2++;
    i = 0;

    while ((i < count1) && (i < count2))
    {
        if (string1[i] == string2[i])
        {
            i++;
            continue;
        }
        if (string1[i] < string2[i])
        {
            flag = -1;
            break;
        }
        if (string1[i] > string2[i])
        {
            flag = 1;
            break;
        }
    }
```

```
        printf("Both strings are equal \n");
    if (flag == 1)
        printf("String1 is greater than string2 \n", string1, string2);
    if (flag == -1)
        printf("String1 is less than string2 \n", string1, string2);
}
```

```
$ cc pgm50.c
/$ a.out
Enter a string: hello
Enter another string: world
String1 is less than string2

$ a.out
Enter a string:object
Enter another string:class
String1 is greater than string2

$ a.out
Enter a string:object
Enter another string:object
Both strings are equal
```

C Program to Calculate the Simple Interest

This C Program calculates the simple interest given the principal amount, rate of interest and time. The formul to calculate the simple interest is: simple_interest = (p * t * r) / 100 where p is principal amount, t is time & r i rate of interest.

Here is source code of the C program to calculate the simple interest. The C program is successfull compiled and run on a Linux system. The program output is also shown below.

```
/*
 * C program to find the simple interest, given principal,
 * rate of interest and time.
 */
#include <stdio.h>

void main()
{
    float principal_amt, rate, simple_interest;
    int time;

    printf("Enter the values of principal_amt, rate and time \n");
    scanf("%f %f %d", &principal_amt, &rate, &time);
    simple_interest = (principal_amt * rate * time) / 100.0;
    printf("Amount = Rs. %5.2f\n", principal_amt);
    printf("Rate = Rs. %5.2f%\n", rate);
    printf("Time = %d years\n", time);
    printf("Simple interest = %5.2f\n", simple_interest);
}
```

```
$ cc pgm3.c
$ a.out
Enter the values of principal_amt, rate and time
12
10
5
Amount = Rs. 12.00
Rate = Rs. 10.00%
Time = 5 years
Simple interest =  6.00
```

C Program to Calculate the Sum & Difference of the Matrices

This C Program calculates the sum & difference of the matrices. The program first reads 2 matrices and the performs both addition and subtraction of matrices

Here is source code of the C program to calculates the sum & difference of the matrices. The (program is successfully compiled and run on a Linux system. The program output is also shown below

```c
/*
 * C program to accept two matrices and find the sum
 * and difference of the matrices
 */
#include <stdio.h>
#include <stdlib.h>

void readmatA();
void printmatA();
void readmatB();
void printmatB();
void sum();
void diff();

int a[10][10], b[10][10], sumarray[10][10], arraydiff[10][10];
int i, j, row1, column1, row2, column2;

void main()
{
    printf("Enter the order of the matrix A \n");
    scanf("%d %d", &row1, &column1);
    printf("Enter the order of the matrix B \n");
    scanf("%d %d", &row2, &column2);
    if (row1 != row2 && column1 != column2)
    {
        printf("Addition and subtraction are possible \n");
        exit(1);
    }
    else
    {
        printf("Enter the elements of matrix A \n");
        readmatA();
        printf("MATRIX A is \n");
        printmatA();
        printf("Enter the elements of matrix B \n");
        readmatB();
        printf("MATRIX B is \n");
        printmatB();
        sum();
        diff();
    }
}
/*  Function to read a matrix A */
```

```c
{
    for (i = 0; i < row1; i++)
    {
        for (j = 0; j < column1; j++)
        {
            scanf("%d", &a[i][j]);
        }
    }
    return;
}
/*  Function to read a matrix B */
void readmatB()
{
    for (i = 0; i < row2; i++)
    {
        for (j = 0; j < column2; j++)
        {
            scanf("%d", &b[i][j]);
        }
    }
}
/*  Function to print a matrix A */
void printmatA()
{
    for (i = 0; i < row1; i++)
    {
        for (j = 0; j < column1; j++)
        {
            printf("%3d", a[i][j]);
        }
        printf("\n");
    }
}
/*  Function to print a matrix B */
void printmatB()
{
    for (i = 0; i < row2; i++)
    {
        for (j = 0; j < column2; j++)
        {
            printf("%3d", b[i][j]);
        }
        printf("\n");
    }
}
/*  Function to do the sum of elements of matrix A and Matrix B */
void sum()
{
    for (i = 0; i < row1; i++)
    {
        for (j = 0; j < column2; j++)
           {
            sumarray[i][j] = a[i][j] + b[i][j];
    }
    printf("Sum matrix is \n");
    for (i = 0; i < row1; i++)
    {
        for (j = 0; j < column2; j++)
           {
            printf("%3d", sumarray[i][j]) ;
        }
        printf("\n");
    }
    return;
}
/*  Function to do the difference of elements of matrix A and Matrix B */
void diff()
{
    for (i = 0; i < row1; i++)
    {
        for (j = 0; j < column2; j++)
           {
            arraydiff[i][j] = a[i][j] - b[i][j];
        }
    }
    printf("Difference matrix is \n");
    for (i = 0; i < row1; i++)
    {
        for (j = 0; j < column2; j++)
        {
            printf("%3d", arraydiff[i][j]);
        }
        printf("\n");
    }
    return;
}
```

```
$ cc pgm56.c
$ a.out
Enter the order of the matrix A
3 3
Enter the order of the matrix B
3 3
Enter the elements of matrix A
1 4 5
6 7 8
4 8 9
MATRIX A is
  1  4  5
  6  7  8
  4  8  9
Enter the elements of matrix B
3 6 7
8 4 2
```

```
  3  6  7
  8  4  2
  1  5  3
Sum matrix is
  4 10 12
 14 11 10
  5 13 12
Difference matrix is
 -2 -2 -2
 -2  3  6
  3  3  6
```

C Program to Check if a given Integer is Odd or Even

This C Program checks if a given integer is odd or even. Here if a given number is divisible by 2 with th remainder 0 then the number is even number. If the number is not divisible by 2 then that number will be od number.

Here is source code of the C program which checks a given integer is odd or even. The C program i successfully compiled and run on a Linux system. The program output is also shown below.

```c
/*
 * C program to check whether a given integer is odd or even
 */
#include <stdio.h>

void main()
{
    int ival, remainder;

    printf("Enter an integer : ");
    scanf("%d", &ival);
    remainder = ival % 2;
    if (remainder == 0)
        printf("%d is an even integer\n", ival);
    else
        printf("%d is an odd integer\n", ival);
}
```

```
$ cc pgm4.c
$ a.out
Enter an integer : 100
100 is an even integer

$ a.out
Enter an integer : 105
105 is an odd integer
```

C Program to Check if a given Matrix is an Identity Matrix

This C Program checks a given Matrix is an Identity Matrix. Identity matrix is a square matrix with 1’s along th diagonal from upper left to lower right and 0’s in all other positions. If it satisfies the structure as explained befor then the matrix is called as identity matrix.

Here is source code of the C program to check a given Matrix is an Identity Matrix. The C program i successfully compiled and run on a Linux system. The program output is also shown below.

```
/*
 * C Program to check if a given matrix is an identity matrix
 */
#include <stdio.h>

void main()
{
    int a[10][10];
    int i, j, row, column, flag = 1;

    printf("Enter the order of the matrix A \n");
    scanf("%d %d", &row, &column);
    printf("Enter the elements of matrix A \n");
    for (i = 0; i < row; i++)
    {
        for (j = 0; j < column; j++)
        {
            scanf("%d", &a[i][j]);
        }
    }
    printf("MATRIX A is \n");
    for (i = 0; i < row; i++)
    {
        for (j = 0; j < column; j++)
        {
            printf("%3d", a[i][j]);
        }
        printf("\n");
    }
    /*  Check for unit (or identity) matrix */
    for (i = 0; i < row; i++)
    {
        for (j = 0; j < column; j++)
        {
            if (a[i][j] != 1 && a[j][i] != 0)
            {
                flag = 0;
                break;
            }
        }
    }
    if (flag == 1 )
```

```c
    else
        printf("It is not a identity matrix \n");
}
```

```
$ cc pgm58.c
$ a.out
Enter the order of the matrix A
3 3
Enter the elements of matrix A
1 2 3
5 1 8
6 4 1
MATRIX A is
  1  2  3
  5  1  8
  6  4  1
It is not a identity matrix

$ a.out
Enter the order of the matrix A
3 3
Enter the elements of matrix A
 1 0 0
 0 1 0
 0 0 1
MATRIX A is
  1  0  0
  0  1  0
  0  0  1
It is identity matrix
```

C Program to Check if a given Number is Prime number

This C Program checks if a given number is prime. If a given number that has no other factor than that of th given number itself and 1, then that number is called as prime number.

Here is source code of the C program to check if a given number is prime. The C program i successfully compiled and run on a Linux system. The program output is also shown below.

```c
/*
 * C program to check whether a given number is prime or not
 * and output the given number with suitable message.
 */
#include <stdio.h>
#include <stdlib.h>

void main()
{
    int num, j, flag;

    printf("Enter a number \n");
    scanf("%d", &num);

    if (num <= 1)
    {
        printf("%d is not a prime numbers \n", num);
        exit(1);
    }
    flag = 0;
    for (j = 2; j <= num / 2; j++)
    {
        if ((num % j) == 0)
        {
            flag = 1;
            break;
        }
    }
    if (flag == 0)
        printf("%d is a prime number \n", num);
     else
        printf("%d is not a prime number \n", num);
}
```

```
$ cc pgm16.c
$ a.out
Enter a number
23
23 is a prime number

$ a.out
Enter a number
```

```
15
15 is not a prime number
```

C Program to Check if a given String is Palindrome

This C Program checks if a given string is palindrome. This program first performs reverse of a string. Then checks whether the given string is equivalent to the reversed string. If they are equal then the string is called a palindrome.

Here is source code of the C program to check a given string is palindrome. The C program i successfully compiled and run on a Linux system. The program output is also shown below.

```
/*
 * C program to read a string and check if it's a palindrome, without
 * using library functions. Display the result.
 */
#include <stdio.h>
#include <string.h>

void main()
{
    char string[25], reverse_string[25] = {'\0'};
    int  i, length = 0, flag = 0;

    fflush(stdin);
    printf("Enter a string \n");
    gets(string);
    /*  keep going through each character of the string till its end */
    for (i = 0; string[i] != '\0'; i++)
    {
        length++;
    }
    for (i = length - 1; i >= 0; i--)
    {
       reverse_string[length - i - 1] = string[i];
    }
    /*
     * Compare the input string and its reverse. If both are equal
     * then the input string is palindrome.
     */
    for (i = 0; i < length; i++)
    {
        if (reverse_string[i] == string[i])
            flag = 1;
        else
            flag = 0;
    }
    if (flag == 1)
        printf("%s is a palindrome \n", string);
    else
        printf("%s is not a palindrome \n", string);
}
```

```
$ cc pgm28.c
$ a.out
Enter a string
sanfoundry
sanfoundry is not a palindrome

$ a.out
Enter a string
malayalam
malayalam is a palindrome
```

C Program to Check whether a given Number is Armstrong

This C Program checks whether a given number is armstrong number. An Armstrong number is an n-digit base number such that the sum of its (base b) digits raised to the power n is the number itself. Hence 153 becaus 1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153.

Here is source code of the C Program to check whether a given number is armstrong number. The C program is successfully compiled and run on a Linux system. The program output is also show below.

```c
/*
 * C Program to Check whether a given Number is Armstrong
 */
#include <stdio.h>
#include <math.h>

void main()
{
    int number, sum = 0, rem = 0, cube = 0, temp;

    printf ("enter a number");
    scanf("%d", &number);
    temp = number;
    while (number != 0)
    {
        rem = number % 10;
        cube = pow(rem, 3);
        sum = sum + cube;
        number = number / 10;
    }
    if (sum == temp)
        printf ("The given no is armstrong no");
    else
        printf ("The given no is not a armstrong no");
}
```

```
Output:
$ cc pgm41.c -lm
$ a.out
enter a number370
The given no is armstrong no

$ a.out
enter a number1500
The given no is not a armstrong no
```

C Program to Compute the Product of Two Matrices

This C Program computes the product of two matrices. This program accepts the 2 matrices and then find th product of 2 matrices.

Here is source code of the C program to compute the product of two matrices. The C program i successfully compiled and run on a Linux system. The program output is also shown below.

```c
/*
 * Develop functions to read a matrix, display a matrix and compute
 * product of two matrices.
 * Use these functions to read two MxN matrices and compute their
 * product & display the result
 */
#include <stdio.h>
#define MAXROWS 10
#define MAXCOLS 10

void readMatrix(int arr[][MAXCOLS], int m, int n);
void printMatrix(int arr[][MAXCOLS], int m, int n);
void productMatrix(int array1[][MAXCOLS], int array2[][MAXCOLS],
int array3[][MAXCOLS], int m, int n);

void main()
{
    int array1[MAXROWS][MAXCOLS], array2[MAXROWS][MAXCOLS],
    array3[MAXROWS][MAXCOLS];
    int m, n;

    printf("Enter the value of m and n \n");
    scanf("%d %d", &m, &n);
    printf("Enter Matrix array1 \n");
    readMatrix(array1, m, n);
    printf("Matrix array1 \n");
    printMatrix(array1, m, n);
    printf("Enter Matrix array2 \n");
    readMatrix(array2, m, n);
    printf("Matrix B \n");
    printMatrix(array2, m, n);
    productMatrix(array1, array2, array3, m, n);
    printf("The product matrix is \n");
    printMatrix(array3, m, n);
}
/*  Input Matrix array1 */
void readMatrix(int arr[][MAXCOLS], int m, int n)
{
    int i, j;
    for (i = 0; i < m; i++)
    {
        for (j = 0; j < n; j++)
        {
        }
    }
}
void printMatrix(int arr[][MAXCOLS], int m, int n)
{
    int i, j;
    for (i = 0; i < m; i++)
    {
        for (j = 0; j < n; j++)
        {
            printf("%3d", arr[i][j]);
        }
        printf("\n");
    }
}
/*  Multiplication of matrices */
void productMatrix(int array1[][MAXCOLS], int array2[][MAXCOLS],
int array3[][MAXCOLS], int m, int n)
{
    int i, j, k;
    for (i = 0; i < m; i++)
    {
        for (j = 0; j < n; j++)
        {
            array3[i][j] = 0;
            for (k = 0; k < n; k++)
            {
                array3[i][j] = array3[i][j] + array1[i][k] *
                array2[k][j];
            }
        }
    }
}
```

```
$ cc pgm34.c
$ a.out
Enter the value of m and n
3 3
Enter matrix array1
4 5 6
1 2 3
3 7 8
Matrix array1
  4  5  6
  1  2  3
  3  7  8
Enter matrix array2
5 6 9
8 5 3
2 9 1
Matrix array2
```

```
  2  9  1
The product matrix is
 72103 57
 27 43 18
 87125 56
```

C Program to Compute the Sum of Digits in a given Integer

This C Program computes the sum of digits in a given integer. This program m accepts integer. Then adds all th digits of a given integer, that becomes the sum of digits of integer.

Here is source code of the C program to compute the sum of digits in a given integer. The C prograr is successfully compiled and run on a Linux system. The program output is also shown below.

```c
/*
 * C program to accept an integer & find the sum of its digits
 */
#include <stdio.h>

void main()
{
    long num, temp, digit, sum = 0;

    printf("Enter the number \n");
    scanf("%ld", &num);
    temp = num;
    while (num > 0)
    {
        digit = num % 10;
        sum  = sum + digit;
        num /= 10;
    }
    printf("Given number = %ld\n", temp);
    printf("Sum of the digits %ld = %ld\n", temp, sum);
}
```

```
$ cc pgm81.c
$ a.out
Enter the number
300
Given number = 300
Sum of the digits 300 = 3

$ a.out
Enter the number
16789
Given number = 16789
Sum of the digits 16789 = 31
```

C Program to Find out the Roots of a Quadratic Equation

This C Program calculates the roots of a quadratic equation. First it finds discriminant using the formula : disc =
* b – 4 * a * c. There are 3 types of roots. They are complex, distinct & equal roots. We have to find the give
equation belongs to which type of root.

Here is source code of the C program to calculate the roots of a quadratic equation. The C program i
successfully compiled and run on a Linux system. The program output is also shown below.

```c
/*
 * C program to find out the roots of a quadratic equation
 * for non-zero coefficients. In case of errors the program
 * should report suitable error message.
 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

void main()
{
    float a, b, c, root1, root2;
    float realp, imagp, disc;

    printf("Enter the values of a, b and c \n");
    scanf("%f %f %f", &a, &b, &c);
    /* If a = 0, it is not a quadratic equation */
    if (a == 0 || b == 0 || c == 0)
    {
        printf("Error: Roots cannot be determined \n");
        exit(1);
    }
    else
    {
        disc = b * b - 4.0 * a * c;
        if (disc < 0)
        {
            printf("Imaginary Roots\n");
            realp = -b / (2.0 * a) ;
            imagp = sqrt(abs(disc)) / (2.0 * a);
            printf("Root1 = %f  +i %f\n", realp, imagp);
            printf("Root2 = %f  -i %f\n", realp, imagp);
        }
        else if (disc == 0)
        {
            printf("Roots are real and equal\n");
            root1 = -b / (2.0 * a);
            root2 = root1;
            printf("Root1 = %f\n", root1);
            printf("Root2 = %f\n", root2);
        }
        else if (disc > 0 )
```

```
            printf("Roots are real and distinct \n");
            root1 =(-b + sqrt(disc)) / (2.0 * a);
            root2 =(-b - sqrt(disc)) / (2.0 * a);
            printf("Root1 = %f  \n", root1);
            printf("Root2 = %f  \n", root2);
        }
    }
}
```

```
$ cc pgm7.c -lm
$ a.out
Enter the values of a, b and c
45 50 65
Imaginary Roots
Root1 = -0.555556  +i 1.065740
Root2 = -0.555556  -i 1.065740
```

C Program to find Reverse of a Number using Recursion

The following C program using recursion reverses the digits of the number and displays it on the output of th terminal.Eg: 123 becomes 321.

Here is the source code of the C program to find the reverse of a number. The C program i successfully compiled and run on a Linux system. The program output is also shown below.

```c
/*
 * C program to find the reverse of a number using recursion
 */
#include <stdio.h>
#include <math.h>

int rev(int, int);

int main()
{
    int num, result;
    int length = 0, temp;

    printf("Enter an integer number to reverse: ");
    scanf("%d", &num);
    temp = num;
    while (temp != 0)
    {
        length++;
        temp = temp / 10;
    }
    result = rev(num, length);
    printf("The reverse of %d is %d.\n", num, result);
    return 0;
}

int rev(int num, int len)
{
    if (len == 1)
    {
        return num;
    }
    else
    {
        return (((num % 10) * pow(10, len - 1)) + rev(num / 10, --len));
    }
}
```

```
$ cc pgm34.c
$ a.out
Enter an integer number to reverse: 1234
The reverse of 1234 is 4321.
```

C Program to Find Sum of Digits of a Number using Recursion

The following C program, using recursion, finds the sum of its digits.

Here is the source code of the C program to find an element in a linked list. The C Program i successfully compiled and run on a Linux system. The program output is also shown below.

```c
/*
 * C Program to find Sum of Digits of a Number using Recursion
 */
#include <stdio.h>

int sum (int a);

int main()
{
    int num, result;

    printf("Enter the number: ");
    scanf("%d", &num);
    result = sum(num);
    printf("Sum of digits in %d is %d\n", num, result);
    return 0;
}

int sum (int num)
{
    if (num != 0)
    {
        return (num % 10 + sum (num / 10));
    }
    else
    {
       return 0;
    }
}
```

```
$ cc pgm25.c
$ a.out
Enter the number: 2345
Sum of digits in 2345 is 14
```

C Program to Find the Biggest of 3 Numbers

This C Program calculates the biggest of 3 numbers.The program assumes 3 numbers as a, b, c. First compares any 2 numbers check which is bigger. After that it compares the biggest element with the remainin number. Now the number which is greater becomes your biggest of 3 numbers.

Here is source code of the C program to calculate the biggest of 3 numbers. The C program i successfully compiled and run on a Linux system. The program output is also shown below.

```c
/*
 * C program to find the biggest of three numbers
 */
#include <stdio.h>

void main()
{
    int num1, num2, num3;

    printf("Enter the values of num1, num2 and num3\n");
    scanf("%d %d %d", &num1, &num2, &num3);
    printf("num1 = %d\tnum2 = %d\tnum3 = %d\n", num1, num2, num3);
    if (num1 > num2)
    {
        if (num1 > num3)
        {
            printf("num1 is the greatest among three \n");
        }
        else
        {
            printf("num3 is the greatest among three \n");
        }
    }
    else if (num2 > num3)
        printf("num2 is the greatest among three \n");
    else
        printf("num3 is the greatest among three \n");
}
```

```
$ cc pgm6.c
$ a.out
Enter the values of num1, num2 and num3
6 8 10
num1 = 6  num2 = 8  num3 = 10
num3 is the greatest among three
```

C Program to Find the Largest Number in an Array

This C Program finds the largest number in an array

Here is source code of the C Program to find the largest number in an array. The C program i successfully compiled and run on a Linux system. The program output is also shown below.

```c
/*
 * C Program to Find the Largest Number in an Array
 */
#include <stdio.h>

int main()
{
    int array[50], size, i, largest;
    printf("\n Enter the size of the array: ");
    scanf("%d", &size);
    printf("\n Enter %d elements of  the array: ", size);
    for (i = 0; i < size; i++)
        scanf("%d", &array[i]);
    largest = array[0];
    for (i = 1; i < size; i++)
    {
        if (largest < array[i])
            largest = array[i];
    }
    printf("\n largest element present in the given array is : %d", largest);
    return 0;
}
```

```
$ cc pgm73.c
$ a.out

Enter the size of the array: 5

Enter 5 elements of  the array: 12
56
34
78
100

largest element present in the given array is : 100
```

C Program to Find the Nth Fibonacci Number using Recursion

This C Program prints the fibonacci of a given number using recursion. In fibonacci series, each number is th
sum of the two preceding numbers. Eg: 0, 1, 1, 2, 3, 5, 8, …
The following program returns the nth number entered by user residing in the fibonacci series.

Here is the source code of the C program to print the nth number of a fibonacci number. The
program is successfully compiled and run on a Linux system. The program output is also shown below

```c
/*
 * C Program to find the nth number in Fibonacci series using recursion
 */
#include <stdio.h>
int fibo(int);

int main()
{
    int num;
    int result;

    printf("Enter the nth number in fibonacci series: ");
    scanf("%d", &num);
    if (num < 0)
    {
        printf("Fibonacci of negative number is not possible.\n");
    }
    else
    {
        result = fibo(num);
        printf("The %d number in fibonacci series is %d\n", num, result);
    }
    return 0;
}
int fibo(int num)
{
    if (num == 0)
    {
        return 0;
    }
    else if (num == 1)
    {
        return 1;
    }
    else
    {
        return(fibo(num - 1) + fibo(num - 2));
    }
}
```

```
$ cc pgm9.c
```

```
Enter the nth number in fibonacci series: 8
The 8 number in fibonacci series is 21

$ a.out
Enter the nth number in fibonacci series: 12
The 12 number in fibonacci series is 144
```

C Program to Find the Transpose of a given Matrix

This C Program finds the transpose of a given matrix. The transpose of a given matrix is formed by interchangin the rows and columns of a matrix.

Here is source code of the C program to find the transpose of a given matrix .The C program i successfully compiled and run on a Linux system. The program output is also shown below.

```
/*
 * C program to accept a matrix of order MxN and find its transpose
 */
#include <stdio.h>

void main()
{
    static int array[10][10];
    int i, j, m, n;

    printf("Enter the order of the matrix \n");
    scanf("%d %d", &m, &n);
    printf("Enter the coefiicients of the matrix\n");
    for (i = 0; i < m; ++i)
    {
        for (j = 0; j < n; ++j)
        {
            scanf("%d", &array[i][j]);
        }
    }
    printf("The given matrix is \n");
    for (i = 0; i < m; ++i)
    {
        for (j = 0; j < n; ++j)
        {
            printf(" %d", array[i][j]);
        }
        printf("\n");
    }
    printf("Transpose of matrix is \n");
    for (j = 0; j < n; ++j)
    {
        for (i = 0; i < m; ++i)
        {
            printf(" %d", array[i][j]);
        }
        printf("\n");
    }
}
```

```
$ cc pgm85.c
$ a.out
```

```
3 3
Enter the coefiicients of the matrix
3 7 9
2 7 5
6 3 4
The given matrix is
 3 7 9
 2 7 5
 6 3 4
Transpose of matrix is
 3 2 6
 7 7 3
 9 5 4
```

C Program to Generate Fibonacci Series

This C Program generates fibonacci series. In fibonacci series the first two numbers in the Fibonacci sequenc are 0 and 1 and each subsequent number is the sum of the previous two. For example fibonacci series is 0, 1, 2, 3, 5, 8,13, 21…………

Here is source code of the C program to generate fibonacci series. The C program is successfull compiled and run on a Linux system. The program output is also shown below.

```c
/*
 * C program to generate Fibonacci Series. Fibonacci Series
 * is 0 1 1 2 3 5 8 13 21 ...
 */
#include <stdio.h>

void main()
{
    int  fib1 = 0, fib2 = 1, fib3, limit, count = 0;

    printf("Enter the limit to generate the Fibonacci Series \n");
    scanf("%d", &limit);
    printf("Fibonacci Series is ...\n");
    printf("%d\n", fib1);
    printf("%d\n", fib2);
    count = 2;
    while (count < limit)
    {
        fib3 = fib1 + fib2;
        count++;
        printf("%d\n", fib3);
        fib1 = fib2;
        fib2 = fib3;
    }
}
```

```
$ cc pgm40.c
$ a.out
Enter the limit to generate the Fibonacci Series
6
Fibonacci Series is ...
0
1
1
2
3
5
```

C Program to Illustrate Pass by Reference

This C Program illustrates pass by reference. This program is used to explain how pass by reference functio works.In pass by reference method, the function will operate on the original variable itself. It doesn’t work on copy of the argument but works on the argument itself.

Here is source code of the C Program to illustrate pass by reference. The C program is successfull compiled and run on a Linux system. The program output is also shown below.

```
/*
 * C Program to Illustrate Pass by Reference
 */
#include <stdio.h>

void cube( int *x);

int main()
{
    int num = 10;

    cube(&num);
    printf("the cube of the given number is %d", num);
    return 0;
}

void  cube(int *x)
{
    *x = (*x) * (*x) * (*x);
}
```

```
$ cc pgm49.c
$ a.out
Enter file name: pgm2.c
There are 43 lines in pgm2.c  in a file
```

C Program to Merge the Elements of 2 Sorted Array

This C Program merge the elements of 2 sorted array.

Here is source code of the C Program to merge the elements of 2 sorted array . The C program i successfully compiled and run on a Linux system. The program output is also shown below.

```c
/*
 * C Program to Merge the Elements of 2 Sorted Array
 */
#include <stdio.h>

void main()
{
    int array1[50], array2[50], array3[100], m, n, i, j, k = 0;

    printf("\n Enter size of array Array 1: ");
    scanf("%d", &m);
    printf("\n Enter sorted elements of array 1: \n");
    for (i = 0; i < m; i++)
    {
        scanf("%d", &array1[i]);
    }
    printf("\n Enter size of array 2: ");
    scanf("%d", &n);
    printf("\n Enter sorted elements of array 2: \n");
    for (i = 0; i < n; i++)
    {
        scanf("%d", &array2[i]);
    }
    i = 0;
    j = 0;
    while (i < m && j < n)
    {
        if (array1[i] < array2[j])
        {
            array3[k] = array1[i];
            i++;
        }
        else
        {
            array3[k] = array2[j];
            j++;
        }
        k++;
    }
    if (i >= m)
    {
        while (j < n)
        {
            array3[k] = array2[j];
```

```
                k++;
            }
        }
        if (j >= n)
        {
            while (i < m)
            {
                array3[k] = array1[i];
                i++;
                k++;
            }
        }
        printf("\n After merging: \n");
        for (i = 0; i < m + n; i++)
        {
            printf("\n%d", array3[i]);
        }
    }
```

```
$ cc pgm81.c
$ a.out

Enter size of array Array 1: 4

Enter sorted elements of array 1:
12
18
40
60

Enter size of array 2: 4

Enter sorted elements of array 2:
47
56
89
90

After merging:

12
18
40
47
56
60
89
90
```

C Program to Print Diamond Pattern

This C Program prints diamond pattern.

Here is source code of the C Program to print diamond pattern. The C program is successfull compiled and run on a Linux system. The program output is also shown below.

```c
/*
 * C Program to Print Diamond Pattern
 */
#include <stdio.h>

int main()
{
    int number, i, k, count = 1;

    printf("Enter number of rows\n");
    scanf("%d", &number);
    count = number - 1;
    for (k = 1; k <= number; k++)
    {
        for (i = 1; i <= count; i++)
            printf(" ");
        count--;
        for (i = 1; i <= 2 * k - 1; i++)
            printf("*");
        printf("\n");
     }
     count = 1;
     for (k = 1; k <= number - 1; k++)
     {
         for (i = 1; i <= count; i++)
             printf(" ");
         count++;
         for (i = 1 ; i <= 2 *(number - k)-  1; i++)
             printf("*");
         printf("\n");
      }
      return 0;
}
```

```
$ cc pgm60.c
$ a.out
Enter number of rows
5
    *
   ***
  *****
 *******
*********
```

```
  *****
   ***
    *
$ a.out
Enter number of rows
2
 *
***
 *
```

C Program to Read an Array and Search for an Element

This C Program reads an array and searches for an element.

Here is source code of the C program to read an array and search for an element. The C program i successfully compiled and run on a Linux system. The program output is also shown below.

```c
/*
 * C program accept an array of N elements and a key to search.
 * If the search is successful, it displays "SUCCESSFUL SEARCH".
 * Otherwise, a message "UNSUCCESSFUL SEARCH" is displayed.
 */
#include <stdio.h>

void main()
{
    int array[20];
    int i, low, mid, high, key, size;

    printf("Enter the size of an array\n");
    scanf("%d", &size);
    printf("Enter the array elements\n");
    for (i = 0; i < size; i++)
    {
        scanf("%d", &array[i]);
    }
    printf("Enter the key\n");
    scanf("%d", &key);
    /*  search begins */
    low = 0;
    high = (size - 1);
    while (low <= high)
    {
        mid = (low + high) / 2;
        if (key == array[mid])
        {
            printf("SUCCESSFUL SEARCH\n");
            return;
        }
        if (key < array[mid])
            high = mid - 1;
        else
            low = mid + 1;
    }
    printf("UNSUCCESSFUL SEARCH\n");
}
```

```
$ cc pgm97.c
$ a.out
Enter the size of an array
Enter the array elements
90
560
300
390
Enter the key
90
SUCCESSFUL SEARCH

$ a.out
Enter the size of an array
4
Enter the array elements
100
500
580
470
Enter the key
300
UNSUCCESSFUL SEARCH
```

C Program to Read Two Integers M and N & Swap their Values

This C Program reads two integers & swap their values.

Here is source code of the C program to read two integers & swap their values. The C program i
successfully compiled and run on a Linux system. The program output is also shown below.

```c
/*
 * C program to read two integers M and N and to swap their values.
 * Use a user-defined function for swapping. Output the values of M
 * and N before and after swapping.
 */
#include <stdio.h>
void swap(float *ptr1, float  *ptr2);

void main()
{
    float m, n;

    printf("Enter the values of M and N \n");
    scanf("%f %f", &m, &n);
    printf("Before Swapping:M = %5.2ftN = %5.2f\n", m, n);
    swap(&m, &n);
    printf("After Swapping:M  = %5.2ftN = %5.2f\n", m, n);
}
/*  Function swap - to interchanges the contents of two items */
void swap(float *ptr1, float *ptr2)
{
    float temp;

    temp = *ptr1;
    *ptr1 = *ptr2;
    *ptr2 = temp;
}
```

```
$ cc pgm36.c
$ a.out
Enter the values of M and N
2 3
Before Swapping:M =  2.00    N =  3.00
After Swapping:M  =  3.00    N =  2.00
```

C Program to read two Strings & Concatenate the Strings

This C Program reads the two strings & concatenate the strings without using string library functions. Th program first reads the 2 strings using scanf(), then joins the one string with another. Later it reads and prints as a single string.

Here is source code of the C program to read two strings & concatenate the strings. The C program i successfully compiled and run on a Linux system. The program output is also shown below.

```c
/*
 * C program to read two strings and concatenate them, without using
 * library functions. Display the concatenated string.
 */
#include <stdio.h>
#include <string.h>

void main()
{
    char string1[20], string2[20];
    int i, j, pos;

    /*  Initialize the string to NULL values */
    memset(string1, 0, 20);
    memset(string2, 0, 20);

    printf("Enter the first string : ");
    scanf("%s", string1);
    printf("Enter the second string: ");
    scanf("%s", string2);
    printf("First string  = %s\n", string1);
    printf("Second string = %s\n", string2);

    /*  Concate the second string to the end of the first string */
    for (i = 0; string1[i] != '\0'; i++)
    {
        /*  null statement: simply traversing the string1 */
        ;
    }
    pos = i;
    for (j = 0; string2[j] != '\0'; i++)
    {
        string1[i] = string2[j++];
    }
    /*  set the last character of string1 to NULL */
    string1[i] = '\0';
    printf("Concatenated string = %s\n", string1);
}
```

```
$ cc pgm29.c
$ a.out
```

```
Enter the second string: foundry
First string  = San
Second string = foundry
Concatenated string = Sanfoundry
```

C Program to Reverse a Given Number

This C Program reverses a given number by using modulo operation.

Here is source code of the C program to reverse a given number. The C program is successfull compiled and run on a Linux system. The program output is also shown below.

```c
/*
 * C program to accept an integer and reverse it
 */
#include <stdio.h>

void main()
{
    long  num, reverse = 0, temp, remainder;

    printf("Enter the number\n");
    scanf("%ld", &num);
    temp = num;
    while (num > 0)
    {
        remainder = num % 10;
        reverse = reverse * 10 + remainder;
        num /= 10;
    }
    printf("Given number = %ld\n", temp);
    printf("Its reverse is = %ld\n", reverse);
}
```

```
$ cc pgm42.c
$ a.out
Enter the number
567865
Given number   = 567865
Its reverse is = 568765
```

C Program to Reverse a Number & Check if it is a Palindrome

This C Program reverses a number & checks if it is a palindrome or not. First it reverses a number. Then checks if given number and reversed numbers are equal. If they are equal, then its a palindrome.

Here is source code of the C program to reverse a number & checks it is a palindrome or not. The program is successfully compiled and run on a Linux system. The program output is also shown below

```c
/*
 * C program to reverse a given integer number and check
 * whether it is a palindrome. Display the given number
 * with appropriate message
 */
#include <stdio.h>

void main()
{
    int num, temp, remainder, reverse = 0;

    printf("Enter an integer \n");
    scanf("%d", &num);
    /*  original number is stored at temp */
    temp = num;
    while (num > 0)
    {
        remainder = num % 10;
        reverse = reverse * 10 + remainder;
        num /= 10;
    }
    printf("Given number is = %d\n", temp);
    printf("Its reverse is  = %d\n", reverse);
    if (temp == reverse)
        printf("Number is a palindrome \n");
    else
        printf("Number is not a palindrome \n");
}
```

```
$ cc pgm13.c
$ a.out
Enter an integer
6789
Given number is = 6789
Its reverse is  = 9876
Number is not a palindrome

$ a.out
Enter an integer
58085
Given number is = 58085
Its reverse is  = 58085
```

C Program to Simulate a Simple Calculator

This C Program simulates a simple calculator. This program performs arithmatic operations like addtior subraction, multiplication & division. Assume that the 2 numbers a & b are given. For the given element we nee to perform addition, subtraction, multiplication & division.

Here is source code of the C program which simulates a simple calculator. The C program i successfully compiled and run on a Linux system. The program output is also shown below.

```c
/*
 * C program to simulate a simple calculator to perform arithmetic
 * operations like addition, subtraction, multiplication and division
 */
#include <stdio.h>

void main()
{
    char operator;
    float num1, num2, result;

    printf("Simulation of a Simple Calculator\n");
    printf("*********************************\n");
    printf("Enter two numbers \n");
    scanf("%f %f", &num1, &num2);
    fflush(stdin);
    printf("Enter the operator [+,-,*,/] \n");
    scanf("%s", &operator);
    switch(operator)
    {
    case '+': result = num1 + num2;
        break;
    case '-': result = num1 - num2;
        break;
    case '*': result = num1 * num2;
        break;
    case '/': result = num1 / num2;
        break;
    default : printf("Error in operationn");
        break;
    }
    printf("\n %5.2f %c %5.2f = %5.2f\n", num1, operator, num2, result);
}
```

```
$ cc pgm.c
$ a.out8
Simulation of a Simple Calculator
*********************************
Enter two numbers
2 3
Enter the operator [+,-,*,/]
```

```
2.00 +  3.00 =  5.00

$ a.out
Simulation of a Simple Calculator
*********************************
Enter two numbers
50 40
Enter the operator [+,-,*,/]
*

50.00 * 40.00 = 2000.00

$ a.out
Simulation of a Simple Calculator
*********************************
Enter two numbers
500 17
Enter the operator [+,-,*,/]
/

500.00 / 17.00 = 29.41

$ a.out
Simulation of a Simple Calculator
*********************************
Enter two numbers
65000 4700
Enter the operator [+,-,*,/]
-

65000.00 - 4700.00 = 60300.00
```

C Program to Sort the Array in an Ascending Order

This C Program sorts array in an ascending order.

Here is source code of the C program to sort the array in an ascending order. The C program i successfully compiled and run on a Linux system. The program output is also shown below.

```c
/*
 * C program to accept N numbers and arrange them in an ascending order
 */
#include <stdio.h>

void main()
{
    int i, j, a, n, number[30];

    printf("Enter the value of N \n");
    scanf("%d", &n);
    printf("Enter the numbers \n");
    for (i = 0; i < n; ++i)
        scanf("%d", &number[i]);
    for (i = 0; i < n; ++i)
    {
        for (j = i + 1; j < n; ++j)
        {
            if (number[i] > number[j])
            {
                a =  number[i];
                number[i] = number[j];
                number[j] = a;
            }
        }
    }
    printf("The numbers arranged in ascending order are given below \n");
    for (i = 0; i < n; ++i)
        printf("%d\n", number[i]);
}
```

```
$ cc pgm66.c
$ a.out
Enter the value of N
6
Enter the numbers
3
78
90
456
780
200
The numbers arranged in ascending order are given below
```

```
78
90
200
456
780
```

C Program to Sort the Array in Descending Order

This C Program sorts array in an descending order.

Here is source code of the C program to sort the array in an descending order. The C program i successfully compiled and run on a Linux system. The program output is also shown below.

```c
/*
 * C program to accept a set of numbers and arrange them
 * in a descending order
 */
#include <stdio.h>

void main ()
{
    int number[30];
    int i, j, a, n;

    printf("Enter the value of N\n");
    scanf("%d", &n);
    printf("Enter the numbers \n");
    for (i = 0; i < n; ++i)
    scanf("%d", &number[i]);
    /*  sorting begins ... */
    for (i = 0; i < n; ++i)
    {
        for (j = i + 1; j < n; ++j)
        {
            if (number[i] < number[j])
            {
                a = number[i];
                number[i] = number[j];
                number[j] = a;
            }
        }
    }
    printf("The numbers arranged in descending order are given below\n");
    for (i = 0; i < n; ++i)
    {
        printf("%d\n", number[i]);
    }
}
```

```
$ cc pgm67.c
$ a.out
Enter the value of N
5
Enter the numbers
234
780
```

```
56
90
The numbers arranged in descending order are given below
780
234
130
90
56
```

C Programming

en.wikibooks.org

November 24, 2013

Contents

1 Why learn C?

C[1] is the most commonly used programming language[2] for writing operating systems[3]. Unix[4] was the first operating system written in C. Later Microsoft Windows[5], Mac OS X[6], and GNU/Linux[7] were all written in C. Not only is C the language of operating systems, it is the precursor and inspiration for almost all of the most popular high-level languages available today. In fact, Perl[8], PHP[9], and Python[10] are all written in C.

By way of analogy, let's say that you were going to be learning Spanish, Italian, French, or Portuguese. Do you think knowing Latin would be helpful? Just as Latin was the basis of all of those languages, knowing C will enable you to understand and appreciate an entire family of programming languages built upon the traditions of C. Knowledge of C enables freedom.

1.0.1 Why C, and not assembly language?

While assembly language can provide speed and maximum control of the program, C provides portability.

Different processors are programmed using different Assembly languages and having to choose and learn only one of them is too arbitrary. In fact, one of the main strengths of C is that it combines universality and portability across various computer architectures while retaining most of the control of the hardware provided by assembly language.

For example, C programs can be compiled and run on the HP 50g calculator (ARM processor), the TI-89 calculator (68000 processor), Palm OS Cobalt smartphones (ARM processor), the original iMac (PowerPC), the Arduino (Atmel AVR), and the Intel iMac (Intel Core 2 Duo). Each of these devices has its own assembly language that is completely incompatible with the assembly language of any other.

Assembly[11], while extremely powerful, is simply too difficult to program large applications and hard to read or interpret in a logical way. C is a compiled language, which creates fast and efficient executable files. It is also a small "what you see is all you get" language: a

1 `http://en.wikipedia.org/wiki/C%20%28programming%20language%29`
2 `http://en.wikipedia.org/wiki/programming%20language`
3 `http://en.wikipedia.org/wiki/operating%20systems`
4 `http://en.wikipedia.org/wiki/Unix`
5 `http://en.wikipedia.org/wiki/Microsoft%20Windows`
6 `http://en.wikipedia.org/wiki/Mac%20OS%20X`
7 `http://en.wikipedia.org/wiki/Linux`
8 `http://en.wikipedia.org/wiki/Perl`
9 `http://en.wikipedia.org/wiki/PHP`
10 `http://en.wikipedia.org/wiki/Python%20%28programming%20language%29`
11 `http://en.wikipedia.org/wiki/Assembly%20language`

C statement corresponds to at most a handful of assembly statements, everything else is provided by library functions.

So is it any wonder that C is such a popular language?

Like toppling dominoes, the next generation of programs follows the trend of its ancestors. Operating systems designed in C always have system libraries designed in C. Those system libraries are in turn used to create higher-level libraries (like OpenGL[12], or GTK[13]), and the designers of those libraries often decide to use the language the system libraries used. Application developers use the higher-level libraries to design word processors, games, media players and the like. Many of them will choose to program in the language that the higher-level library uses. And the pattern continues on and on and on......

1.0.2 Why C, and not another language?

The primary design of C is to produce portable code while maintaining performance and minimizing footprint, as is the case for operating systems or other programs where a "high-level" interface would affect performance. It is a stable and mature language whose features are unlikely to disappear for a long time and has been ported to most, if not all, platforms.

For example, C programs can be compiled and run on the HP 50g calculator (ARM processor), the TI-89 calculator (68000 processor), Palm OS Cobalt smartphones (ARM processor), the original iMac (PowerPC), the Arduino (Atmel AVR), and the Intel iMac (Intel Core 2 Duo). While nearly all popular programming languages will run on at least one of these devices, C may be the only programming language that runs on more than 3 of these devices.

One powerful reason is memory allocation. Unlike most computer languages, C allows the programmer to write directly to memory. Key constructs in C such as structs, pointers and arrays are designed to structure, and manipulate memory in an efficient, machine-independent fashion. In particular, C gives control over the memory layout of data structures. Moreover dynamic memory allocation is under the control of the programmer, which inevitably means that memory deallocation is the burden of the programmer. Languages like Java[14] and Perl shield the programmer from having to worry about memory allocation and pointers. This is usually a good thing, since dealing with memory allocation when building a high-level program is a highly error-prone process. However, when dealing with low level code such as the part of the OS that controls a device, C provides a uniform, clean interface. These capabilities just do not exist in other languages such as Java.

While Perl, PHP, Python and Ruby may be powerful and support many features not provided by default in C, they are not normally implemented in their own language. Rather, most such languages initially relied on being written in C (or another high-performance programming language), and would require their implementation be ported to a new platform before they can be used.

12 `http://en.wikipedia.org/wiki/OpenGL`
13 `http://en.wikipedia.org/wiki/GTK`
14 `http://en.wikipedia.org/wiki/Java%20%28programming%20language%29`

As with all programming languages, whether you want to choose C over another high-level language is a matter of opinion and both technical and business requirements.

2 History

The field of computing as we know it today started in 1947 with three scientists at Bell Telephone Laboratories—William Shockley[1], Walter Brattain[2], and John Bardeen[3]—and their groundbreaking invention: the transistor[4]. In 1956, the first fully transistor-based computer, the TX-0[5], was completed at MIT. The first integrated circuit[6] was created in 1958 by Jack Kilby[7] at Texas Instruments, but the first high-level programming language existed even before then.

"The Fortran[8] project" was originally developed in 1954 by IBM. A shortening of "*The IBM Mathematical **For**mula **Tran**slating System*", the project had the purpose of creating and fostering development of a procedural, imperative programming language that was especially suited to numeric computation and scientific computing. It was a breakthrough in terms of productivity and programming ease (compared to assembly language) and speed (Fortran programs ran nearly as fast as, and in some cases, just as fast as, programs written in assembly). Furthermore, Fortran was written at a high-enough level (and thus was machine independent enough) to become the first widely adopted programming language. The Algorithmic Language (Algol 58[9]) was derived from Fortran in 1958 and evolved into Algol 60[10] in 1960. The Combined Programming Language (CPL)[11] was then created out of Algol 60 in 1963. In 1967, it evolved into Basic CPL[12], which was itself, the base for B[13] in 1969. Finally, B was the root of C, created in 1971.

B was the first language in C's direct lineage. B was created by Ken Thompson[14] at Bell Labs and was an interpreted language[15] used in early internal versions of the UNIX operating system. Thompson and Dennis Ritchie[16], also working at Bell Labs, improved B and called the result NB. Further extensions to NB created its logical successor, C, a compiled language[17]. Most of UNIX was rewritten in NB, and then C, which resulted in a more portable operating system.

1 `http://en.wikipedia.org/wiki/William%20Shockley`
2 `http://en.wikipedia.org/wiki/Walter%20Brattain`
3 `http://en.wikipedia.org/wiki/John%20Bardeen`
4 `http://en.wikipedia.org/wiki/transistor`
5 `http://en.wikipedia.org/wiki/TX-0`
6 `http://en.wikipedia.org/wiki/integrated%20circuit`
7 `http://en.wikipedia.org/wiki/Jack%20Kilby`
8 `http://en.wikipedia.org/wiki/Fortran`
9 `http://en.wikipedia.org/wiki/ALGOL%2058`
10 `http://en.wikipedia.org/wiki/ALGOL%2060`
11 `http://en.wikipedia.org/wiki/Combined%20Programming%20Language`
12 `http://en.wikipedia.org/wiki/BCPL`
13 `http://en.wikipedia.org/wiki/B%20%28programming%20language%29`
14 `http://en.wikipedia.org/wiki/Ken%20Thompson`
15 `http://en.wikipedia.org/wiki/interpreted%20language`
16 `http://en.wikipedia.org/wiki/Dennis%20Ritchie`
17 `http://en.wikipedia.org/wiki/compiled%20language`

The portability of UNIX was the main reason for the initial popularity of both UNIX and C. Rather than creating a new operating system for each new machine, system programmers could simply write the few system-dependent parts required for the machine, and then write a C compiler for the new system. Since most of the system utilities were thus written in C, it simply made sense to also write new utilities in C.

The American National Standards Institute began work on standardizing the C language in 1983, and completed the standard in 1989. The standard, ANSI X3.159-1989 "Programming Language C", served as the basis for all implementations of C compilers. The standards were later updated in 1990 and 1999, allowing for features that were either in common use, or were appearing in C++.

3 What you need before you can learn

3.1 Getting Started

The goal of this book is to introduce you to the C programming language. Basic computer literacy is assumed, but no special knowledge is needed.

Before you can start programming in C, you will need a **C compiler**[1]. A compiler is a program that converts C code into executable machine code[2].[3]

Popular C compilers Include:

Name	Website	Platform	License	Details
Microsoft Visual Studio Express[4]	Visual Studio[5]	Windows	Free Version	Powerful and student-friendly version of an industry standard compiler.
Tiny C Compiler (TCC)[6]	tinycc[7]	GNU/Linux, Windows	LGPL[8]	Small, fast and simple compiler.
Clang[9]	clang[10]	GNU/Linux, Windows, Unix, OS X	University of Illinois/NCSA License[11]	A front-end which compiles (Objective) C/C++ using a LLVM backend.
GNU C Compiler[12]	gcc[13]	GNU/Linux, MinGW(Windows)[14], Unix, OS X.	GPL[15]	The De facto standard. Ships with most Unix systems.

1 http://en.wikipedia.org/wiki/Compiler
2 http://en.wikipedia.org/wiki/machine%20code
3 Actually, GCC's(GNU C Compiler) **cc** (C Compiler) translates the input .c file to the target cpu's assembly, output is written to an .s file. Then **as** (assembler) generates a machine code file from the .s file. Pre-processing is done by another sub-program **cpp** (C PreProcessor), which is not to be confused with **c++** the compiler.
4 http://en.wikipedia.org/wiki/Microsoft%20Visual%20Studio%20Express
5 http://www.microsoft.com/visualstudio/en-us/products/2010-editions/visual-cpp-express
6 http://en.wikipedia.org/wiki/Tiny%20C%20Compiler
7 http://www.tinycc.org
8 http://en.wikipedia.org/wiki/GNU%20Lesser%20General%20Public%20License
9 http://en.wikipedia.org/wiki/Clang
10 http://clang.llvm.org
11 http://opensource.org/licenses/UoI-NCSA.php
12 http://en.wikipedia.org/wiki/GNU%20Compiler%20Collection
13 http://gcc.gnu.org
14 http://mingw.org
15 http://en.wikipedia.org/wiki/GNU%20General%20Public%20License

The minimum software requirements to program in C is a text editor[16], as opposed to a word processor[17]. A plain text Notepad Editor can be used but it does not offer any advanced capabilities such as code completion or debugging. There are many text editors (see List of Text Editors[18]), among the most popular are Notepad++[19] for Windows, Sublime Text[20], Vim[21] and Emacs[22] are also available cross-platform. These text editors come with syntax highlighting[23] and line numbers, which makes code easier to read at a glance, and to spot syntax errors.

Though not absolutely needed, many programmers prefer and recommend using an Integrated development environment[24] (**IDE**) instead of a text editor. An IDE is a suite of programs that developers need, combined into one convenient package, usually with a graphical user interface. These programs include a text editor, linker, project management and sometimes bundled with a compiler. They also typically include a debugger, a tool that will preserve your C source code after compilation and enable you to do such things as step through it manually, or alter data as an aid to finding and correcting programming errors.

For beginners it is recommended not to use an IDE, since it hides most of what is going on. Using the command line builds up familiarity with the toolchain. An IDE may be useful to somebody with programming experience but knows how the IDE works. So as a general guideline: Do not use an IDE unless you know what the IDE does!

Popular IDEs Include:

Name	**Website**	**Platform**	**License**	**Details**
Eclipse CDT[25]	Eclipse[26]	Windows, Mac OS X, Linux	Open source	Eclipse[27] IDE for C/C++ developement, a popular open source IDE.
Netbeans[28]	Netbeans[29]	Cross-platform	CDDL[30] and GPL[31] 2.0	A Good comparable matured IDE to Eclipse.

16 http://en.wikipedia.org/wiki/Text%20Editor
17 http://en.wikipedia.org/wiki/Word%20Processor
18 http://en.wikipedia.org/wiki/List%20of%20text%20editors
19 http://en.wikipedia.org/wiki/Notepad%2B%2B
20 http://en.wikipedia.org/wiki/Sublime%20Text
21 http://en.wikipedia.org/wiki/Vim%20%28text%20editor%29
22 http://en.wikipedia.org/wiki/Emacs
23 http://en.wikipedia.org/wiki/syntax%20highlighting
24 http://en.wikipedia.org/wiki/Integrated%20development%20environment
25 http://en.wikipedia.org/wiki/Eclipse_%28software%29
26 http://www.eclipse.org/downloads/packages/eclipse-ide-cc-developers/junor
27 http://en.wikipedia.org/wiki/Eclipse%20%28software%29
28 http://en.wikipedia.org/wiki/Netbeans
29 http://netbeans.org
30 http://en.wikipedia.org/wiki/Common%20Development%20and%20Distribution%20License
31 http://en.wikipedia.org/wiki/GNU%20General%20Public%20License

Name	**Website**	**Platform**	**License**	**Details**
Anjuta[32]	Anjuta[33]	Linux	GPL[34]	A GTK+2 IDE for the GNOME[35] desktop environment.
Geany[36]	geany[37]	Cross-platform	GPL[38]	A lightweight cross-platform GTK+ notepad based on Scintilla, with basic IDE features.
Little C Compiler (LCC)[39]	lcc[40]	Windows	Free for non-commercial use	Small open source compiler.
Xcode[41]	Xcode[42]	Mac OS X	Free	Available for free at Mac App Store[43].
Pelles C[44]	Pelles C[45]	Windows, Pocket PC	Free	A complete C development kit for Windows.
Dev C++[46]	Dev C++[47]	Windows	GPL[48]	Updated version of the formerly popular Bloodshed Dev-C++.

32 http://en.wikipedia.org/wiki/Anjuta
33 http://anjuta.org
34 http://en.wikipedia.org/wiki/GNU%20General%20Public%20License
35 http://en.wikipedia.org/wiki/GNOME
36 http://en.wikipedia.org/wiki/Geany
37 http://www.geany.org
38 http://en.wikipedia.org/wiki/GNU%20General%20Public%20License
39 http://en.wikipedia.org/wiki/LCC%20%28compiler%29
40 http://www.cs.virginia.edu/~lcc-win32
41 http://en.wikipedia.org/wiki/Xcode
42 https://developer.apple.com/xcode
43 https://itunes.apple.com/us/app/xcode/id497799835?ls=1&mt=12
44 http://en.wikipedia.org/wiki/Pelles%20C
45 http://smorgasbordet.com/pellesc
46 http://en.wikipedia.org/wiki/Dev%20C%2B%2B%20
47 http://sourceforge.net/projects/orwelldevcpp/
48 http://en.wikipedia.org/wiki/GNU%20General%20Public%20License

Name	Website	Platform	License	Details
Microsoft Visual Studio Express[49]	Visual C++[50]	Windows	Free	A powerful, user friendly version of an industry standard compiler.
CodeLite[51]	CodeLite[52]	Cross-platform	GPL[53] 2	Free IDE for C/C++ development.
Code::Blocks[54]	Code::Blocks[55]	Cross-platform	GPL[56] 3.0	Built to meet users' most demanding needs. Very extensible and fully configurable.

On **GNU/Linux**, GCC is almost always included automatically.

On **Microsoft Windows**, Dev-C++ is recommended for beginners because it is easy to use, free, and simple to install. However, the official release of Dev-C++ hasn't been updated since 22 February 2005.[57] An unofficial[58] version of Dev-C++ is being actively developed however.[59] An alternate option for those working only in the Windows environment is the official Microsoft Visual Studio Express which is free and has an excellent debugger.

On **Mac OS X**, the Xcode IDE provides the compilers needed to compile various source files. The newer versions do not not include the command line tools. They need to be downloaded via Xcode->Preferences->Downloads.

3.2 Footnotes

pl:C/Czego potrzebujesz[60]

49 http://en.wikipedia.org/wiki/Microsoft%20Visual%20Studio%20Express
50 http://www.microsoft.com/visualstudio/en-us/products/2010-editions/visual-cpp-express
51 http://en.wikipedia.org/wiki/CodeLite
52 http://codelite.org/
53 http://en.wikipedia.org/wiki/GNU%20General%20Public%20License
54 http://en.wikipedia.org/wiki/Code%3A%3ABlocks
55 http://codeblocks.org/
56 http://en.wikipedia.org/wiki/GNU%20General%20Public%20License
57 http://sourceforge.net/news/?group_id=10639
58 http://sourceforge.net/projects/orwelldevcpp/
59 http://orwelldevcpp.blogspot.com/
60 http://pl.wikibooks.org/wiki/C%2FCzego%20potrzebujesz

4 Using a Compiler

4.0.1 Dev-C++

Dev C++[1] is an Integrated Development Environment(IDE) for the C++ programming language, available from Bloodshed Software[2]. An updated version is available at Orwell Dev-C++[3].

C++ is a programming language which contains within itself, most of the C language, plus extensions. Most C++ compilers will compile C programs, sometimes with a few adjustments (like invoking them with a different name or command line switch). Therefore, you can use Dev C++ for C development.

However, Dev C++ is not the compiler. It is designed to use the MinGW[4] or Cygwin[5] versions of GCC[6] - both of which can be obtained as part of the Dev C++ package, although they are completely different projects.

Dev C++ simply provides an editor, syntax highlighting, some facilities for the visualisation of code (like class and package browsing) and a graphical interface to the chosen compiler. Because Dev C++ analyses the error messages produced by the compiler and attempts to distinguish the line numbers from the errors themselves, the use of other compiler software is discouraged since the format of their error messages is likely to be different.

The latest version of Dev-C++ is a beta[7] for version 5. However, it still has a significant number of bugs. All the features are there, and it is quite usable. It is considered one of the best free software C IDEs available for Windows.

A version of Dev C++ for Linux is in the pipeline. It is not quite usable yet, however. Linux users already have a wealth of IDEs available. (e.g. KDevelop[8] and Anjuta[9].) Most of the graphical text editors, and other common editors such as *emacs* and *vi(m)*, support syntax highlighting[10].

1 `http://en.wikipedia.org/wiki/Dev-C%20Plus%20Plus`
2 `http://www.bloodshed.net/`
3 `http://orwelldevcpp.blogspot.com/`
4 `http://en.wikipedia.org/wiki/MinGW`
5 `http://en.wikipedia.org/wiki/Cygwin`
6 `http://en.wikipedia.org/wiki/GCC`
7 `http://en.wikipedia.org/wiki/beta%20version`
8 `http://en.wikipedia.org/wiki/KDevelop`
9 `http://en.wikipedia.org/wiki/Anjuta`
10 `http://en.wikipedia.org/wiki/syntax%20highlighting`

4.0.2 GCC

The GNU Compiler Collection[11] (GCC) is a free[12] set of compilers developed by the Free Software Foundation[13].

Steps for Obtaining the GCC Compiler if You're on GNU/Linux

On **GNU/Linux,** Installing the GNU C Compiler can vary in method from distribution[14] to distribution. (Type in **cc -v** to see if it is installed already.)

- For Redhat[15], get a GCC RPM[16], e.g. using Rpmfind and then install (as root) using `rpm -ivh gcc-`*`version-release.arch`*`.rpm`
- For Fedora Core[17], install the GCC compiler (as root) by using `yum`[18] `install gcc`.
- For Mandrake[19], install the GCC compiler (as root) by using `urpmi`[20] `gcc`
- For Debian[21], install the GCC compiler (as root) by using `apt-get`[22] `install gcc`.
- For Ubuntu[23], install the GCC compiler (along with other necessary tools) by using `sudo apt-get`[24] `install build-essential`, or by using Synaptic. You do not need Universe enabled.
- For Slackware[25], the package is available on their website[26] - simply download, and type `installpkg gcc-xxxxx.tgz`
- For Gentoo[27], you should already have GCC installed as it will have been used when you first installed. To update it run (as root) `emerge -uav gcc`.
- For Arch Linux[28], install the GCC compiler (as root) by using `pacman -S gcc`.
- If you cannot become root, get the GCC tarball from ftp://ftp.gnu.org/ and follow the instructions in it to compile and install in your home directory. Be warned though, you need a C compiler to do that - yes, GCC itself is written in C.
- You can use some commercial C compiler/IDE.

Steps for Obtaining the GCC Compiler if You're on BSD Family Systems

11 `http://en.wikipedia.org/wiki/GNU%20Compiler%20Collection`
12 `http://en.wikipedia.org/wiki/free%20software`
13 `http://en.wikipedia.org/wiki/Free%20Software%20Foundation`
14 `http://en.wikipedia.org/wiki/Linux%20distribution`
15 `http://en.wikipedia.org/wiki/Redhat`
16 `http://en.wikipedia.org/wiki/RPM%20Package%20Manager`
17 `http://en.wikipedia.org/wiki/Fedora%20Core`
18 `http://en.wikipedia.org/wiki/yum`
19 `http://en.wikipedia.org/wiki/Mandrake`
20 `http://en.wikipedia.org/wiki/urpmi`
21 `http://en.wikipedia.org/wiki/Debian`
22 `http://en.wikipedia.org/wiki/Advanced%20Packaging%20Tool`
23 `http://en.wikipedia.org/wiki/Ubuntu`
24 `http://en.wikipedia.org/wiki/Advanced%20Packaging%20Tool`
25 `http://en.wikipedia.org/wiki/Slackware`
26 `http://www.slackware.com/pb/`
27 `http://en.wikipedia.org/wiki/Gentoo`
28 `http://en.wikipedia.org/wiki/Arch%20Linux`

- For Mac OS X[29], FreeBSD[30], NetBSD[31], OpenBSD[32], DragonFly BSD[33], Darwin[34] the port of GNU gcc is available in the base system, or it could be obtained using the ports collection or pkgsrc[35].

Steps for Obtaining the GCC Compiler if You're on Windows

There are two ways to use GCC on Windows: Cygwin and MinGW. Applications compiled with Cygwin will not run on any computer without Cygwin, so MinGW is recommended. MinGW is simpler to install, and takes less disk space.

To get MinGW, do this:

1. Go to `http://sourceforge.net/projects/mingw/` download and save this to your hard drive.
2. Once the download is finished, open it and follow the instructions. You can also choose to install additional compilers, or the tool Make, but these aren't necessary.
3. Now you need to set your PATH. Right-click on "My computer" and click "Properties". Go to the "Advanced" tab and click on "Environment variables". Go to the "System variables" section and scroll down until you see "Path". Click on it, then click "edit". Add ";C:\mingw\bin\" (without the quotes) to the end.
4. To test if GCC works, open a command prompt and type "gcc". You should get the message "gcc: no input files". If you get this message, GCC is installed correctly.

To get Cygwin, do this:

1. Go to `http://www.cygwin.com` and click on the "Install Cygwin Now" button in the upper right corner of the page.
2. Click "run" in the window that pops up, and click "next" several times, accepting all the default settings.
3. Choose any of the Download sites ("ftp.easynet.be", etc.) when that window comes up; press "next" and the Cygwin installer should start downloading.
4. When the "Select Packages" window appears, scroll down to the heading "Devel" and click on the "+" by it. In the list of packages that now displays, scroll down and find the "gcc-core" package; this is the compiler. Click once on the word "Skip", and it should change to some number like "3.4" etc. (the version number), and an "X" will appear next to "gcc-core" and several other related packages that will now be downloaded.
5. Click "next" and the compiler as well as the Cygwin tools should start downloading; this could take a while. While you're waiting for the installation to finish, download any text-editor designed for programming. While Cygwin does include some, you may prefer doing a web search to find other alternatives. While using a stock text editor is possible, it is not ideal.
6. Once the Cygwin downloads are finished and you have clicked "next", etc. to finish the installation, double-click the Cygwin icon on your desktop to begin the Cygwin

29 `http://en.wikipedia.org/wiki/Mac%20OS%20X`
30 `http://en.wikipedia.org/wiki/FreeBSD`
31 `http://en.wikipedia.org/wiki/NetBSD`
32 `http://en.wikipedia.org/wiki/OpenBSD`
33 `http://en.wikipedia.org/wiki/DragonFly%20BSD`
34 `http://en.wikipedia.org/wiki/Darwin`
35 `http://en.wikipedia.org/wiki/pkgsrc`

"command prompt". Your home directory will automatically be set up in the Cygwin folder, which now should be at "C:\cygwin" (the Cygwin folder is in some ways like a small unix/linux computer on your Windows machine -- not technically of course, but it may be helpful to think of it that way).
7. Type "gcc" at the Cygwin prompt and press "enter"; if "gcc: no input files" or something like it appears you have succeeded and now have the gcc compiler on your computer (and congratulations -- you have also just received your first error message!).

The current stable (usable) version of GCC is 5.1.6 published on 2009-10-02, which supports several platforms. In fact, GCC is not only a C compiler, but a family of compilers for several languages, such as C++, Ada[36], Java[37], and Fortran[38].

Once gcc is installed, it can be called with a list of c source files that have been written but not yet compiled. eg. there is a main.c file that includes a some functions described in myfun.h and implemented in myfun_a.c and myfun_b.c , then it is enough to write

```
gcc   main.c myfun_a.c myfun_b.c
```

myfun.h is included in main.c , but if is in a separate header files directory , then that directory can be listed after a "-I " switch.

In larger programs, Makefiles and gnu make program can compile c files into intermediate files ending with suffix .o which can be linked by gcc.

How to compile each object file is usually described in the Makefile with the object file as a label ending with a colon followed by two spaces (tabs are often bad characters) followed by a list of other files that are dependencies, eg. .c files and .o files compiled in another section, and on the next line, the invocation of gcc that is required. typing man make or info make often gives the information needed to jog the memory on how to use make, and the same goes for gcc, although gcc has a lot of option switches, the main ones being -g to generate debugging for gdb to allow it to show source code during stepping through of the machine code program. gdb has a 'h' command that shows what it can do, and is usually started with 'gdb a.out' if a.out is the anonymous executable machine code file that was compiled by gcc.

4.0.3 Embedded systems

- Most CPUs are microcontrollers in embedded systems, often programmed in C, but most of the compilers mentioned above (except GCC) do not support such CPUs. For specialized compilers that do support embedded systems, see Embedded Systems/C Programming[39].

36 http://en.wikibooks.org/wiki/Ada%20Programming
37 http://en.wikibooks.org/wiki/Java
38 http://en.wikibooks.org/wiki/Fortran
39 http://en.wikibooks.org/wiki/Embedded%20Systems%2FC%20Programming

pl:C/Używanie kompilatora[40]

40 http://pl.wikibooks.org/wiki/C%2FU%C5%BCywanie%20kompilatora

5 A taste of C

As with nearly every other programming language learning book, we use the *Hello world*[1] program to introduce you to C.

```
#include <stdio.h>

int main(void)
{
   puts("Hello, world!");
   return 0;
}
```

This program prints "Hello, world!" and then exits.

Enter this code into your text editor or IDE, and save it as "hello.c".

Then, presuming you are using GCC, type `gcc -o hello hello.c`. This tells gcc to compile your hello.c program into a form the machine can execute. The '-o hello' tells it to call the compiled program 'hello'.

If you have entered this correctly, you should now see a file called hello. This file is the binary version of your program, and when run should display "Hello, world!"

Here is an example of how compiling and running looks when using a terminal on a unix system. `ls` is a common unix command that will list the files in the current directory, which in this case is the directory `progs` inside the home directory (represented with the special tilde, ~, symbol). After running the `gcc` command, `ls` will list a new file, `hello` in green. Green is the standard color coding of `ls` for executable files.

```
~/progs$ ls
hello.c
~/progs$ gcc -o hello hello.c
~/progs$ ls
hello  hello.c
~/progs$ ./hello
Hello, world!
~/progs$
```

5.0.4 Part-by-part explanation

`#include <stdio.h>` tells the C compiler to find the standard header called *<stdio.h>*[2] and add it to this program. In C, you often have to pull in extra optional components when

1 http://en.wikipedia.org/wiki/Hello%20world%20program
2 http://en.wikipedia.org/wiki/stdio.h

you need them. *<stdio.h>* contains descriptions of standard input/output functions which you can use to send messages to a user, or to read input from a user.

`int main(void)` is something you'll find in every C program. Every program has a *main* function. Generally, the main function is where a program begins. However, one C program can be scattered across multiple files, so you won't always find a main function in every file. The *int* at the beginning means that main will return an integer to the operating system when it is finished.

`puts("Hello, world!");` is the statement that actually puts the message to the screen. *puts* is a string printing function that is declared in the file *stdio.h* (which is why you had to *#include* that at the start of the program) `puts` automatically prints a newline at the end of the string.

`return 0;` will return zero (which is the integer[3] referred to on line 3) to the operating system. When a program runs successfully its return value is zero (GCC4 complains if it doesn't when compiling). A non-zero value is returned to indicate a warning or error.

The empty line is there because it is (at least on UNIX) considered good practice to end a file with a new line. In gcc using the `-Wall -pedantic -ansi` options, if the file does not end with a new line this message is displayed: "warning: no newline at end of file". (The newline isn't shown on the example because MediaWiki automatically removes it)

3 `http://en.wikipedia.org/wiki/Integer%20%28computer%20science%29`

6 Intro exercise

6.1 Introductory Exercises

6.1.1 On GCC

If you are using a Unix(-like) system, such as GNU/Linux[1], Mac OS X[2], or Solaris[3], it will probably have GCC installed. Type the hello world program into a file called first.c and then compile it with gcc. Just type:

```
gcc first.c
```

Then run the program by typing:

```
./a.out
```

or, If you are using Cygwin.

```
a.exe
```

You should now see your very first C program.

There are a lot of options you can use with the gcc compiler. For example, if you want the output to have a name other than a.out, you can use the -o option. The following shows a few examples:

`-c`

indicates that the compiler is supposed to generate an *object file*, which can be later linked to other files to form a final program.

`-o`

indicates that the next parameter is the name of the resulting program (or library). If this option is not specified, the compiled program will, for historic reasons, end up in a file called "a.out" or "a.exe" (for cygwin users).

`-g3`

1 http://en.wikipedia.org/wiki/GNU%2FLinux
2 http://en.wikipedia.org/wiki/Mac%20OS%20X
3 http://en.wikipedia.org/wiki/Solaris%20Operating%20Environment

indicates that *debugging information* should be added to the results of compilation.

`-O2 -ffast-math`

indicates that the compilation should be optimized.

`-W -Wall -fno-common -Wcast-align -Wredundant-decls -Wbad-function-cast -Wwrite-strings -Waggregate-return -Wstrict-prototypes -Wmissing-prototypes`

indicates that gcc should warn about many types of suspicious code that are likely to be incorrect.

`-E`

indicates that gcc should only preprocess the code; this is useful when you are having trouble understanding what gcc is doing with #include and #define, among other things.

All the options are well documented in the manual page[4] for GCC.

the classical hello world program

The basic hello world program, from the K+R book on C, is often worth memorising, just for the structure of the main function which accepts switches, just like gcc is a program with a main function that accepts switches.

```
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char** argv) {
  printf("Hello WOrld! \n");
  return 0;
}
```

The commented program below is basically the same, with some variations that have the same effect. e.g. a " argv" , or "pointer to pointers" , is the same as *x[] or "array of pointers" ; and "exit(0)" does the same as "return 0" for the main function.

Note: It is a good chance to say that we usually return or exit a function with the 0 code when all of the commands executed successfully. e.g. in our Hello World program, all the commands before "return(0)", or "exit(0)" executed with no error. You will notice that this convention is very common, especially in the main function.

```
/* Hello World */

/*
this gives include statement, brings in the header file stdio.h ,
located often on unix systems at the directory /usr/include/,
and includes  the printf() function, as well as others, snprintf, scanf,
 getchar, getline.
In C, functions that are exported have their "signatures" - function name and
 parameter list -
```

4 http://gcc.gnu.org/onlinedocs/gcc-4.3.0/gcc/Debugging-Options.html

```
listed in header files for exporting, and then the same signatures are defined
 in another file,
often of the same name, to be compiled once into an object file, and on unix
 systems often reside
in /usr/lib/   with file names like stdlib.a  or stdlib.so , often as soft links
 to versioned files
e.g. stdlib.1.3.so ,
*/
#include <stdio.h>

/*
  this gives the standard library, which has functions
  such as rand() random number generation (e.g. for games)
  malloc() and free()  for dynamic heap memory allocation as opposed to stack
 memory allocation.
  stack memory can be allocated by declaring variables and arrays at the start
 of a function , including
  the main function, and will be destroyed when the function exits.
*/

#include <stdlib.h>

/*
 the next line is the standard expected function name "main" and argument list
 of the first function
 to be executed for this program when the compiled program is executed.
 the first argument is the number of arguments, and the second argument is an
 array of pointers to
 arrays of characters (strings) which contain arguments .e.g.  "-?" , "-v" ,
 "-c"
*/

int main(int n_args, char* args[]) {

  printf("Hello World!");  //  outputs a string without formatting.


  exit(0);  //  stdlib.h function to exit with a code, if executed from  say a
 bash shell script, 0 will be
            //  returned  , which can be used inside a shell conditional if
 statement.
}
```

6.1.2 On IDEs

If you are using a commercial IDE you may have to select console project, and to compile you just select build from the menu or the toolbar. The executable will appear inside the project folder, but you should have a menu button so you can just run the executable from the IDE.

One can also find opensource IDE's like Eclipse[5], Netbeans[6] or Qt Creator[7]. The process will be the same as a commercial IDE.

5 http://en.wikipedia.org/wiki/Eclipse_%28software%29
6 http://en.wikipedia.org/wiki/Netbeans
7 http://en.wikipedia.org/wiki/Qt_Creator

7 Beginning C

8 Preliminaries

8.1 Basic Concepts

Before one gets too deep into learning C syntax and programming constructs, it is beneficial to learn the meaning of a few key terms that are central to a thorough understanding of C.

8.2 Block Structure, Statements, Whitespace, and Scope

Now we **discuss the basic structure of a C program**. If you're familiar with PASCAL[1], you may have heard it referred to as a **block-structured** language. C does not have complete block structure (and you'll find out why when you go over functions in detail) but it is still very important to understand what blocks are and how to use them.

So what is in a **block**? Generally, a block consists of executable **statements**.

Before we say what a block is, what's a statement? One way to put it is that statements are text the compiler will attempt to turn into executable instructions, and the whitespace that surrounds them. An easier way to put it is that statements are bits of code that do things, like this:

```
int i = 6; /* this declares a variable 'i', and sets it to equal 6 */
```

You might have noticed the semicolon at the end of the statement. Statements in C always end with a semicolon (;) character. Leaving off the semicolon is a common mistake that a lot of people make, beginners and experts alike! So until it becomes second nature, be sure to double check your statements!

Since C is a "free-format" language, several statements can share a single line in the source file, like so:

```
/* this declares the variables 'i', 'test', 'foo', and 'bar'
    note that ONLY 'bar' is set to six! */
int i, test, foo, bar = 6;
```

There are several kinds of statements, and you've seen some of them. Assignment (i = 6;), conditional and flow-control. A substantial portion of this book deals with statement construction.

Now back to blocks. In C, blocks begin with an opening brace "{" and end with a closing brace "}". Blocks can contain other blocks which can contain their own blocks, and so on.

1 http://en.wikipedia.org/wiki/Pascal%20%28programming%20language%29

Let's show an example of blocks.

```
int main(void)
{
    /* this is a 'block' */
    int i = 5;

    {
        /* this is also a 'block,' separate from the last one */
        int i = 6;
    }

    return 0;
}
```

Blocks come in handy with readability and scope. You'll learn a little more about scope in a second.

Whitespace refers to the tab, space and newline/EOL (End Of Line) characters that separate the text characters that make up source code lines. Like many things in life, it's hard to appreciate whitespace until it's gone. To a C compiler, the source code

```
    printf("Hello world"); return 0;
```

is the same as

```
    printf("Hello world");
    return 0;
```

which is the same as

```
    printf (
    "Hello world") ;

    return 0;
```

The compiler simply skips over whitespace. However, it is common practice to use spaces (and tabs) to organize source code for human readability. You can use blocks without a conditional, loop, or other statement to organize your code.

In C, most of the time we do not want other functions or other programmer's routines[2] accessing data that we are currently manipulating. This is why it is important to understand the concept of scope.

Scope describes the level at which a piece of data or a function is visible. There are two kinds of scope in C, **local** and **global**. When we speak of something being **global**, we speak of something that can be seen or manipulated from anywhere in the program. When we speak of something being **local**, we speak of something that can be seen or manipulated only within the block it was declared.

2 http://en.wikipedia.org/wiki/Subroutine

Let's show some examples, to give a better picture of the idea of scope.

```
int i = 5; /* this is a 'global' variable, anywhere in the program can access it
 */

/* this is a function, all variables inside of it
    are "local" to the function. */
int main(void)
{
    int i = 6; /* 'i' now equals 6 */
    printf("%d\n", i); /* prints a '6' to the screen, instead of the global
 variable of 'i', which is 5 */

    return 0;
}
```

That shows a decent example of local and global, but what about different scopes *inside* of functions? (you'll learn more about functions later, for now, just focus on the "main" part.)

```
/* the main function */
int main(void)
{
    /* this is the beginning of a 'block', you read about those above */

    int i = 6; /* this is the first variable of this 'block', 'i' */

    {
        /* this is a new 'block', and because it's a different block, it has its
 own scope */

        /* this is also a variable called 'i', but in a different 'block',
            because it's in a different 'block' then the old 'i', it doesn't
 affect the old one! */
        int i = 5;
        printf("%d\n", i); /* prints a '5' onto the screen */
    }
    /* now we're back into the old block */

    printf("%d\n", i); /* prints a '6' onto the screen */

    return 0;
}
```

8.3 Basics of Using Functions

Functions are a big part of programming. A function is a special kind of block that performs a well-defined task. If a function is well-designed, it can enable a programmer to perform a task without knowing anything about how the function works. The act of requesting a function to perform its task is called a **function call**. Many functions require a caller to hand it certain pieces of data needed to perform its task; these are called **arguments**. Many functions also return a value to the caller when they're finished; this is called a **return value** (the return value in the above program is **0**).

The things you need to know before calling a function are:

- What the function does
- The data type (discussed later) of the arguments and what they mean
- The data type of the return value and what it means

All code other than global data definitions and declarations needs to be a part of a function.

Usually, you're free to call a function whatever you wish to. The only restriction is that every executable program needs to have one, and only one, **main** function, which is where the program begins executing.

We will discuss functions in more detail in a later chapter, C Programming/Procedures and functions[3].

8.4 The Standard Library

In 1983, when C was in the process of becoming standardized, the American National Standards Institute[4] (ANSI) formed a committee to establish a standard specification of C known as "ANSI C". That standard specification created a basic set of functions common to each implementation of C, which is referred to as the Standard Library[5]. The Standard Library provides functions for tasks such as input/output, string manipulation, mathematics, files, and memory allocation. The Standard Library does not provide functions that are dependent on specific hardware or operating systems, like graphics, sound, or networking. In the "Hello, World", program, a Standard Library function is used `printf` which outputs lines of text to the standard output[6] stream.

pl:C/Podstawy[7]

3 Chapter 17 on page 97
4 `http://en.wikipedia.org/wiki/American%20National%20Standards%20Institute`
5 `http://en.wikipedia.org/wiki/C%20standard%20library`
6 `http://en.wikipedia.org/wiki/standard%20output`
7 `http://pl.wikibooks.org/wiki/C%2FPodstawy`

9 Compiling

Having covered the basic concepts of C programming, we can now briefly discuss the process of *compilation*.

Like any programming language, C by itself is completely incomprehensible to a microprocessor[1]. Its purpose is to provide an intuitive way for humans to provide instructions that can be easily converted into machine code that *is* comprehensible to a microprocessor. The ***compiler*** is what takes this code, and translates it into the machine code.

To those new to programming, this seems fairly simple. A naive compiler might read in every source file, translate everything into machine code, and write out an executable. This could work, but has two serious problems. First, for a large project, the computer may not have enough memory to read all of the source code at once. Second, if you make a change to a single source file, you would rather not have to recompile the *entire* application.

To deal with these problems, compilers break their job down into steps; for each source file (each `.c` file), the compiler reads the file, reads the files it references with `#include`, and translates it to machine code. The result of this is an "object file" (`.o`). Once every object file is made, a "linker" collects all of the object files and writes the actual program. This way, if you change one source file, only that file needs to be recompiled and then the application needs to be re-linked.

Without going into the painful details, it can be beneficial to have a superficial understanding of the compilation process.

9.1 Preprocessor

The preprocessor provides the ability for the inclusion of header files, macro expansions, conditional compilation, and line control. Many times you will need to give special instructions to your compiler. This is done by inserting preprocessor directives[2] into your code. When you begin compiling your code, a special program called the preprocessor scans the source code and performs simple substitution of tokenized strings for others according to predefined rules. The preprocessor is not a part of the C language.

In C language, all preprocessor directives begin with the pound character (#). You can see one preprocessor directive in the Hello world program[3] introduced in A taste of C[4]:

Example:

1 `http://en.wikipedia.org/wiki/microprocessor`
2 `http://en.wikipedia.org/wiki/Preprocessor%20directives`
3 `http://en.wikibooks.org/wiki/Hello%20world%20program`
4 Chapter 5 on page 19

```
#include <stdio.h>
```

This directive causes the header to be included into your program. Other directives such as `#pragma` control compiler settings and macros. The result of the preprocessing stage is a text string. You can think of the preprocessor as a non-interactive text editor that prepares your code for the compilation step. The language of preprocessor directives is agnostic to the grammar of C, so the C preprocessor can also be used independently to process other kinds of text files.

9.2 Syntax Checking

This step ensures that the code is valid and will sequence into an executable program. Under most compilers, you may get messages or warnings indicating potential issues with your program (such as a statement always being true or false, etc.)

When an error is detected in the program, the compiler will normally report the file name and line that is preventing compilation.

9.3 Object Code

The compiler produces a machine code equivalent of the source code that can then be linked into the final program. The code itself can't be executed yet, as it has to complete the linking stage.

It's important to note after discussing the basics that compilation is a "one way street". That is, compiling a C source file into machine code is easy, but "decompiling" (turning machine code into the C source that creates it) is not. Decompilers for C do exist, but they rarely create useful code.

9.4 Linking

Linking combines the separate object codes into one complete program by integrating libraries and the code and producing either an executable program[5] or a library[6]. Linking is performed by a linker, which is often part of a compiler.

Common errors during this stage are either missing functions, or duplicate functions.

5 `http://en.wikipedia.org/wiki/Executable`
6 `http://en.wikipedia.org/wiki/Library%20%28computing%29`

9.5 Automation

For large C projects, many programmers choose to automate compilation, both in order to reduce user interaction requirements and to speed up the process by only recompiling modified files.

Most integrated development environments have some kind of project management, which makes such automation very easy. On UNIX-like systems, make[7] and Makefiles are often used to accomplish the same.

de:C-Programmierung: Kompilierung[8] es:Programación_en_C/Compilar_un_programa[9] et:Programmeerimiskeel C/Kompileerimine[10] fr:Programmation C-C%2B%2B/Modularité et compilation[11] it:C/Compilatore e precompilatore/Compilatore[12] pt:Programar em C/Utilizando um compilador[13]

7 `http://en.wikibooks.org/wiki/make`
8 `http://de.wikibooks.org/wiki/C-Programmierung%3A%20Kompilierung`
9 `http://es.wikibooks.org/wiki/Programaci%C3%B3n_en_C%2FCompilar_un_programa`
10 `http://et.wikibooks.org/wiki/Programmeerimiskeel%20C%2FKompileerimine`
11 `http://fr.wikibooks.org/wiki/Programmation%20C-C%252B%252B%2FModularit%C3%A9%20et%20compilation`
12 `http://it.wikibooks.org/wiki/C%2FCompilatore%20e%20precompilatore%2FCompilatore`
13 `http://pt.wikibooks.org/wiki/Programar%20em%20C%2FUtilizando%20um%20compilador`

10 Structure and style

10.1 C Structure and Style

This is a basic introduction to good code style in the C Programming Language. It is designed to provide information on how to effectively use indentation, comments, and other elements that will make your C code more readable. It is not a tutorial on actually programming in C.

As a beginning programmer, the point of creating structure in the programs' code might not be clear, as the compiler doesn't care about the difference. However, as programs become complex, chances are that writing the program has become a joint effort. (Or others might want to see how it was accomplished.) Therefore, the code is no longer designed purely for a compiler to read.

In the following sections, we will attempt to explain good programming practices that will in turn make your programs clearer and more effective.

10.2 Introduction

In C, programs are composed of statements. These statements are terminated with a semicolon, and are collected in sections known as functions. By convention, a statement should be kept on its own line, as shown in the example below:

```
#include <stdio.h>

int main(void)
{
        printf("Hello, World!\n");
        return 0;
}
```

The following block of code is essentially the same: while it contains exactly the same code, and will compile and execute with the same result, the removal of spacing causes an essential difference making it harder to read:

```
#include <stdio.h>
int main(void) {printf("Hello, World!\n");return 0;}
```

The simple use of indents and line breaks can greatly improve the readability of the code; without making any impact whatsoever on how the code performs. By having readable code, it is much easier to see where functions and procedures end, and which lines are part of which loops and procedures.

This book is going to focus on the above piece of code, and how to improve it. Please note that during the course of the tutorial, there will be many (apparently) redundant pieces of code added. These are only added to provide examples of techniques that we will be explaining, without breaking the overall flow of code that the program achieves.

10.3 Line Breaks and Indentation

The addition of white space inside your code is arguably the most important part of good code structure. Effective use of white space can create a visual scale of how your code flows, which can be very important when returning to your code when you want to maintain it.

10.3.1 Line Breaks

⚠ **Warning**

Note that we have used line numbers here; they are not a part of the actual code. They are only there for reference in this book.

With minimal line breaks, code is barely readable by humans, and may be hard to debug or understand:

<source lang="c" line>

1. include <stdio.h>

int main(void){ int i=0; printf("Hello, World!"); for (i=0; i<1; i++){ printf("\n"); break; } return 0; } </source>

Rather than putting everything on one line, it is much more readable to break up long lines so that each statement and declaration goes on its own line. After inserting line breaks, the code will look like this:

<source lang="c" line>

1. include <stdio.h>

int main(void) { int i=0; printf("Hello, World!"); for (i=0; i<1; i++) { printf("\n"); break; } return 0; } </source>

10.3.2 Blank Lines

Blank lines should be used to offset the main components of your code. Use them

- After precompiler declarations.
- After new variables are declared.

Based on these two rules, there should now be two line breaks added.

- After line 1, because line 1 has a preprocessor directive

- After line 4, because line 4 contains a variable declaration

This will make the code much more readable than it was before:

The following lines of code have line breaks between functions, but without indentation.

<source lang="c" line>

1. include <stdio.h>

int main(void) { int i=0;

printf("Hello, World!"); for (i=0; i<1; i++) { printf("\n"); break; } return 0; }

</source>

But this still isn't as readable as it can be.

10.3.3 Indentation

Note:
Many text editors automatically indent appropriately when you hit the enter/return key.

Although adding simple line breaks between key blocks of code can make code easier to read, it provides no information about the block structure of the program. Using the tab key can be very helpful now: indentation visually separates paths of execution by moving their starting points to a new column in the line. This simple practice will make it much easier to read and understand code. Indentation follows a fairly simple rule:

- All code inside a new block should be indented by one tab[1]

[2] more than the code in the previous path.

Based on the code from the previous section, there are two blocks requiring indentation:

- Lines 5 to 13
- Lines 10 and 11

<source lang="c" line>

1. include <stdio.h>

int main(void) {

1
2
Several programmers recommend "use spaces for indentation. Do not use tabs in your code. You should set your editor to emit spaces when you hit the tab key." `http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml` `http://www.jwz.org/doc/tabs-vs-spaces.html`
Other programmers disagree. `http://web.archive.org/20080118165124/diagrammes-modernes.blogspot.com/2006/04/tab-versus-spaces.html` `http://www.derkarl.org/why_to_tabs.html`
Regardless of whether you prefer spaces or tabs, make sure you keep it consistent with projects you are working on, because mixing tabs and spaces can cause code to become unreadable.

```
    int i=0;
```

```
    printf("Hello, World!");
    for (i=0; i<1; i++)
    {
        printf("\n");
        break;
    }
    return 0;
```

}

</source>

It is now fairly obvious as to which parts of the program fit inside which blocks. You can tell which parts of the program the coder has intended to loop, and which ones he has not. Although it might not be immediately noticeable, once many nested loops and paths get added to the structure of the program, the use of indentation can be very important. This indentation makes the structure of your program clear.

Indentation was originally one tab character, or the equivalent of 8 spaces. Research since the original indent size has shown that indents between 2 to 4 characters are easier to read[3], resulting in such tab sizes being used as default in modern IDEs. However, an indent of 8 characters may still be in use for some systems[4].

10.4 Comments

Comments in code can be useful for a variety of purposes. They provide the easiest way to set off specific parts of code (and their purpose); as well as providing a visual "split" between various parts of your code. Having good comments throughout your code will make it much easier to remember what specific parts of your code do.

Comments in modern flavors of C (and many other languages) can come in two forms:

```
//Single Line Comments  (added by C99 standard, famously known as c++ style of
 comments)
```

and

```
/*Multi-Line
Comments*/ (only form of comments supported by C89 standard)
```

Note that Single line comments are a fairly recent addition to C, so some compilers may not support them. A recent version of GCC[5] will have no problems supporting them.

This section is going to focus on the various uses of each form of commentary.

3 http://www.oualline.com/vim-cook.html#drawing
4 [http://lxr.linux.no/#linux+v2.6.31/Documentation/CodingStyle Linux Kernel coding standard
5 http://en.wikipedia.org/wiki/GNU%20Compiler%20Collection

10.4.1 Single-line Comments

Single-line comments are most useful for simple 'side' notes that explain what certain parts of the code do. The best places to put these comments are next to variable declarations, and next to pieces of code that may need explanation.

Based on our previous program, there are two good places to place comments

- Line 5, to explain what 'int i' is going to do
- Line 11, to explain why there is a 'break' keyword.

This will make our program look something like

```
#include <stdio.h>

int main(void)
{
    int i=0;                // loop variable.

    printf("Hello, World!");

    for (i=0; i<1; i++) {
        printf("\n");
        break;              //Exits 'for' loop.
    }

    return 0;
}
```

10.4.2 Multi-line Comments

Note:
Single-line comments are a new feature, so many C programmers only use multi-line comments.

Multi-line comments are most useful for long explanations of code. They can be used as copyright/licensing notices, and they can also be used to explain the purpose of a block of code. This can be useful for two reasons: They make your functions easier to understand, and they make it easier to spot errors in code. If you know what a block is *supposed* to do, then it is much easier to find the piece of code that is responsible if an error occurs.

As an example, suppose we had a program that was designed to print "Hello, World! " a certain number of lines, a specified number of times. There would be many for loops in this program. For this example, we shall call the number of lines *i*, and the number of strings per line as *j*.

A good example of a multi-line comment that describes 'for' loop ***is purpose would be:***

```
 /* For Loop (int i)
    Loops the following procedure i times (for number of lines).  Performs 'for'
 loop j on each loop,
    and prints a new line at end of each loop.
 */
```

This provides a good explanation of what ***is purpose is, whilst not going into detail of what*** j***does. By going into detail over what the specific path does (and not ones inside it), it will be easier to troubleshoot the path.***

Similarly, you should always include a multi-line comment before each function, to explain the role, preconditions and postconditions of each function. Always leave the technical details to the individual blocks inside your program - this makes it easier to troubleshoot.

A function descriptor should look something like:

```
/* Function : int hworld (int i,int j)
   Input    : int i (Number of lines), int j (Number of instances per line)
   Output   : 0 (on success)
   Procedure: Prints "Hello, World!" j times, and a new line to standard output
over i lines.
*/
```

This system allows for an at-a-glance explanation of what the function should do. You can then go into detail over how each aspect of the program is achieved later on in the program.

Finally, if you like to have aesthetically-pleasing source code, the multi-line comment system allows for the easy addition of comment boxes. These make the comments stand out much more than they would without otherwise. They look like this.

```
/***************************************
 *  This is a multi line comment
 *  That is nearly surrounded by a
 *  Cool, starry border!
 ***************************************/
```

Applied to our original program, we can now include a much more descriptive and readable source code:

```
#include <stdio.h>

int main(void)
{
    /****
********************************************************************************
     * Function: int main(void)
     * Input   : none
     * Output  : Returns 0 on success
     * Procedure: Prints "Hello, World!" and a new line to standard output then
exits.
     *****
********************************************************************************/
    int i=0;                //Temporary variable used for 'for' loop.

    printf("Hello, World!");

    /* FOR LOOP (int i)
       Prints a new line to standard output, and exits */
    for (i=0; i<1; i++)
    {
        printf("\n");
        break;               //Exits 'for' loop.
    }

    return 0;
}
```

This will allow any outside users of the program an easy way to comprehend what the code functions are and how they operate. It also inhibits uncertainty with other like-named functions.

A few programmers add a column of stars on the right side of a block comment:

```
/***************************************
 *  This is a multi line comment       *
 *  that is completely surrounded by a *
 *  cool, starry border!               *
 ***************************************/
```

But most programmers don't put any stars on the right side of a block comment. They feel that aligning the right side is a waste of time.

Comments written in source files can be used for documenting source code automatically by using popular tools like Doxygen[6][7]

10.5 Links

- Aladdin's C coding guidelines[8] - A more definitive C coding guideline.
- C/C++ Programming Styles[9] GNU Coding styles & Linux Kernel Coding style
- C Programming Tutorial[10] C Programming Tutorial

et:Programmeerimiskeel C/Stiil[11]

6 "Coding Conventions for C++ and Java" ^{http://www.macadamian.com/index.php?option=com_content&task=view&id=34&Itemid=37} "all the block comments illustrated in this document have no pretty stars on the right side of the block comment. This deliberate choice was made because aligning those pretty stars is a large waste of time and discourages the maintenance of in-line comments.",

7 wiki:BigBlocksOfAsterisks ^{http://en.wikibooks.org/wiki/wiki%3ABigBlocksOfAsterisks} , "Code craft" ^{http://books.google.com/books?id=i4zCzpkrt4sC&pg=PA82&lpg=PA82&dq=programming+comment+block+waste+time+lining+up&source=bl&ots=TUpTMIHBnh&sig=NeZm23WPmvnw2aKMnIRUeQoHmJg&hl=en&ei=pri3SevGIYGyNMn9jd4K&sa=X&oi=book_result&resnum=8&ct=result} by Pete Goodliffe page 82, Falvotech "C Programming Style Guide" ^{http://www.falvotech.com/content/publications/conventions/c/} , Fedora Directory Server Coding Style ^{http://directory.fedoraproject.org/wiki/Coding_Style}

8 http://www.cs.wisc.edu/~ghost/doc/AFPL/6.01/C-style.htm

9 http://www.mycplus.com/c.asp?ID=12

10 http://www.studiesinn.com/learn/Programming-Languages/C-Language.html

11 http://et.wikibooks.org/wiki/Programmeerimiskeel%20C%2FStiil

11 Error handling

C does not provide direct support for error handling (also known as exception handling). By convention, the programmer is expected to prevent errors from occurring in the first place, and test return values from functions. For example, -1 and NULL are used in several functions such as socket() (Unix socket programming) or malloc() respectively to indicate problems that the programmer should be aware about. In a worst case scenario where there is an unavoidable error and no way to recover from it, a C programmer usually tries to log the error and "gracefully" terminate the program.

There is an external variable called "errno", accessible by the programs after including <errno.h> - that file comes from the definition of the possible errors that can occur in some Operating Systems (e.g. Linux - in this case, the definition is in include/asm-generic/errno.h) when programs ask for resources. Such variable indexes error descriptions accessible by the function 'strerror(errno)'.

The following code tests the return value from the library function malloc to see if dynamic memory allocation completed properly:

```
#include <stdio.h>        /* fprintf */
#include <errno.h>        /* errno */
#include <stdlib.h>       /* malloc, free, exit */
#include <string.h>       /* strerror */

extern int errno;

int main( void )
{

    /* pointer to char, requesting dynamic allocation of 2,000,000,000
     * storage elements (declared as an integer constant of type
     * unsigned long int). (If your system has less than 2GB of memory
     * available, then this call to malloc will fail)
     */
    char *ptr = malloc( 2000000000UL );

    if ( ptr == NULL ){
        puts("malloc failed");
        puts(strerror(errno));
    }
    else
    {
        /* the rest of the code hereafter can assume that 2,000,000,000
         * chars were successfully allocated...
         */
        free( ptr );
    }

    exit(EXIT_SUCCESS); /* exiting program */
}
```

The code snippet above shows the use of the return value of the library function malloc to check for errors. Many library functions have return values that flag errors, and thus should be checked by the astute programmer. In the snippet above, a NULL pointer returned from malloc signals an error in allocation, so the program exits. In more complicated implementations, the program might try to handle the error and try to recover from the failed memory allocation.

11.1 Preventing divide by zero errors

A common pitfall made by C programmers is not checking if a divisor is zero before a division command. The following code will produce a runtime error and in most cases, exit.

```
int dividend = 50;
int divisor = 0;
int quotient;

quotient = (dividend/divisor); /* This will produce a runtime error! */
```

For reasons beyond the scope of this document, you must check or make sure that a divisor is never zero. Alternatively, for *nix processes, you can stop the OS from terminating your process by blocking the SIGFPE signal.

The code below fixes this by checking if the divisor is zero before dividing.

```
#include <stdio.h> /* for fprintf and stderr */
#include <stdlib.h> /* for exit */
int main( void )
{
    int dividend = 50;
    int divisor = 0;
    int quotient;

    if (divisor == 0) {
        /* Example handling of this error. Writing a message to stderr, and
         * exiting with failure.
         */
        fprintf(stderr, "Division by zero! Aborting...\n");
        exit(EXIT_FAILURE); /* indicate failure.*/
    }

    quotient = dividend / divisor;
    exit(EXIT_SUCCESS); /* indicate success.*/
}
```

11.2 Signals

In some cases, the environment may respond to a programming error in C by raising a signal. Signals are events raised by the host environment or operating system to indicate that a specific error or critical event has occurred (e.g. a division by zero, interrupt, and so on.) However, these signals are not meant to be used as a means of error catching; they usually indicate a critical event that will interfere with normal program flow.

To handle signals, a program needs to use the `signal.h` header file. A signal handler will need to be defined, and the signal() function is then called to allow the given signal to be handled. Some signals that are raised to an exception within your code (e.g. a division by zero) are unlikely to allow your program to recover. These signal handlers will be required to instead ensure that some resources are properly cleaned up before the program terminates.

This example creates a signal handler and raises the signal:

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

static void catch_function(int signal) {
    puts("Interactive attention signal caught.");
}

int main(void) {
    if (signal(SIGINT, catch_function) == SIG_ERR) {
        fputs("An error occurred while setting a signal handler.\n", stderr);
        return EXIT_FAILURE;
    }
    puts("Raising the interactive attention signal.");
    if (raise(SIGINT) != 0) {
        fputs("Error raising the signal.\n", stderr);
        return EXIT_FAILURE;
    }
    puts("Exiting.");
    return 0;
}
```

11.3 setjmp

The setjmp[1] function can be used to emulate the exception handling feature of other programming languages. The first call to setjmp provides a reference point to returning to a given function, and is valid as long as the function containing setjmp() doesn't return or exit. A call to longjmp causes the execution to return to the point of the associated setjmp call.

```
#include <stdio.h>
#include <setjmp.h>

jmp_buf test1;

void tryjump()
{
    longjmp(test1, 3);
}

int main (void)
{
    if (setjmp(test1)==0) {
        printf ("setjmp() returned 0.");
        tryjump();
    } else {
        printf ("setjmp returned from a longjmp function call.");
```

1 http://en.wikibooks.org/wiki/C%20Programming%2FCoroutines%23setjmp

```
    }
}
```

The values of non-volatile variables may be corrupted when setjmp returns from a longjmp call.

While setjmp() and longjmp() may be used for error handling, it is generally preferred to use the return value of a function to indicate an error, if possible.

12 Variables

Like most programming languages, C is able to use and process named variables and their contents. **Variables** are simply names used to refer to some location in memory – a location that holds a value with which we are working.

It may help to think of variables as a placeholder for a value. You can think of a variable as being equivalent to its assigned value. So, if you have a variable *i* that is **initialized** (set equal) to 4, then it follows that *i+1* will equal *5*.

Since C is a relatively low-level programming language, before a C program can utilize memory to store a variable it must claim the memory needed to store the values for a variable. This is done by **declaring** variables. Declaring variables is the way in which a C program shows the number of variables it needs, what they are going to be named, and how much memory they will need.

Within the C programming language, when managing and working with variables, it is important to know the *type* of variables and the *size* of these types. Since C is a fairly low-level programming language, these aspects of its working can be hardware specific – that is, how the language is made to work on one type of machine can be different from how it is made to work on another.

All variables in C are **typed**. That is, every variable declared must be assigned as a certain type of variable.

12.1 Declaring, Initializing, and Assigning Variables

Here is an example of declaring an integer, which we've called `some_number`. (Note the semicolon at the end of the line; that is how your compiler separates one program *statement* from another.)

```
int some_number;
```

This statement means we're declaring some space for a variable called some_number, which will be used to store `int`eger data. Note that we must specify the type of data that a variable will store. There are specific keywords to do this – we'll look at them in the next section.

Multiple variables can be declared with one statement, like this:

```
int anumber, anothernumber, yetanothernumber;
```

We can also declare *and* assign some content to a variable at the same time.

```
int some_number=3;
```

This is called *initialization.*

In early versions of C, variables had to be declared at the beginning of a block. In C99 it is allowed to mix declarations and statements arbitrarily – but doing so is not usual, because it is rarely necessary, some compilers still don't support C99 (portability), and it may, because it is uncommon yet, irritate fellow programmers (maintainability).

After declaring variables, you can assign a value to a variable later on using a statement like this:

```
some_number=3;
```

You can also assign a variable the value of another variable, like so:

```
anumber = anothernumber;
```

Or assign multiple variables the same value with one statement:

```
anumber = anothernumber = yetanothernumber = 3;
```

This is because the assignment `x = y` returns the value of the assignment. `x = y = z` is really shorthand for `x = (y = z)`.

12.1.1 Naming Variables

Variable names in C are made up of letters (upper and lower case) and digits. The underscore character ("_") is also permitted. Names must not begin with a digit. Unlike some languages (such as Perl[1] and some BASIC[2] dialects), C does not use any special prefix characters on variable names.

Some examples of valid (but not very descriptive) C variable names:

```
foo
Bar
BAZ
foo_bar
_foo42
_
QuUx
```

Some examples of invalid C variable names:

```
2foo    (must not begin with a digit)
my foo  (spaces not allowed in names)
$foo    ($ not allowed -- only letters, digits, and _)
while   (language keywords cannot be used as names)
```

As the last example suggests, certain words are reserved as keywords in the language, and these cannot be used as variable names.

1 http://en.wikipedia.org/wiki/Perl
2 http://en.wikipedia.org/wiki/BASIC%20programming%20language

In addition there are certain sets of names that, while not language keywords, are reserved for one reason or another. For example, a C compiler might use certain names "behind the scenes", and this might cause problems for a program that attempts to use them. Also, some names are reserved for possible future use in the C standard library. The rules for determining exactly what names are reserved (and in what contexts they are reserved) are too complicated to describe here, and as a beginner you don't need to worry about them much anyway. For now, just avoid using names that begin with an underscore character.

The naming rules for C variables also apply to naming other language constructs such as function names, struct tags, and macros, all of which will be covered later.

12.2 Literals

Anytime within a program in which you specify a value explicitly instead of referring to a variable or some other form of data, that value is referred to as a **literal**. In the initialization example above, 3 is a literal. Literals can either take a form defined by their type (more on that soon), or one can use hexadecimal (hex) notation to directly insert data into a variable regardless of its type. Hex numbers are always preceded with *0x*. For now, though, you probably shouldn't be too concerned with hex.

12.3 The Four Basic Data Types

In Standard C there are four basic data types. They are **`int`**, **`char`**, **`float`**, and **`double`**.

We will briefly describe them here, then go into more detail in C Programming/Types[3].

12.3.1 The `int` type

The `int` type stores integers in the form of "whole numbers". An integer is typically the size of one machine word, which on most modern home PCs is 32 bits (4 octets). Examples of literals are whole numbers (integers) such as 1,2,3, 10, 100... When `int` is 32 bits (4 octets), it can store any whole number (integer) between -2147483648 and 2147483647. A 32 bit word (number) has the possibility of representing any one number out of 4294967296 possibilities (2 to the power of 32).

If you want to declare a new int variable, use the `int` keyword. For example:

```
int numberOfStudents, i, j=5;
```

In this declaration we declare 3 variables, numberOfStudents, i and j, j here is assigned the literal 5.

3 http://en.wikibooks.org/wiki/C%20Programming%2FTypes

12.3.2 The `char` type

The `char` type is capable of holding any member of the execution character set. It stores the same kind of data as an `int` (i.e. integers), but typically has a size of one byte. The size of a byte is specified by the macro `CHAR_BIT` which specifies the number of bits in a char (byte). In standard C it never can be less than 8 bits. A variable of type `char` is most often used to store character data, hence its name. Most implementations use the ASCII[4] character set as the execution character set, but it's best not to know or care about that unless the actual values are important.

Examples of character literals are 'a', 'b', '1', etc., as well as some special characters such as `'\0'` (the null character) and `'\n'` (newline, recall "Hello, World"). Note that the `char` value must be enclosed within single quotations.

When we initialize a character variable, we can do it two ways. One is preferred, the other way is ***bad*** programming practice.

The first way is to write

```
char letter1 = 'a';
```

This is *good* programming practice in that it allows a person reading your code to understand that letter1 is being initialized with the letter 'a' to start off with.

The second way, which should *not* be used when you are coding letter characters, is to write

```
char letter2 = 97; /* in ASCII, 97 = 'a' */
```

This is considered by some to be extremely ***bad*** practice, if we are using it to store a character, not a small number, in that if someone reads your code, most readers are forced to look up what character corresponds with the number 97 in the encoding scheme. In the end, `letter1` and `letter2` store both the same thing – the letter 'a', but the first method is clearer, easier to debug, and much more straightforward.

One important thing to mention is that characters for numerals are represented differently from their corresponding number, i.e. '1' is not equal to 1. In short, any single entry that is enclosed within 'single quotes'.

There is one more kind of literal that needs to be explained in connection with chars: the **string literal**. A string is a series of characters, usually intended to be displayed. They are surrounded by double quotations (" ", not ' '). An example of a string literal is the "Hello, World!\n" in the "Hello, World" example.

The string literal is assigned to a character **array**, arrays are described later. Example:

```
const char MY_CONSTANT_PEDANTIC_ITCH[] = "learn the usage context.\n";
printf("Square brackets after a variable name means it is a pointer to a string
 of memory blocks the size of the type of the array element.\n");
```

4 http://en.wikipedia.org/wiki/ASCII

12.3.3 The `float` type

`float` is short for **floating point**. It stores real numbers also, but is only one machine word in size. Therefore, it is used when less precision than a double provides is required. `float` literals must be suffixed with F or f, otherwise they will be interpreted as doubles. Examples are: 3.1415926f, 4.0f, 6.022e+23f. `float` variables can be declared using the `float` keyword.

12.3.4 The `double` type

The `double` and `float` types are very similar. The `float` type allows you to store single-precision floating point numbers, while the `double` keyword allows you to store double-precision floating point numbers – real numbers, in other words, both integer and non-integer values. Its size is typically two machine words, or 8 bytes on most machines. Examples of `double` literals are 3.1415926535897932, 4.0, 6.022e+23 (scientific notation[5]). If you use 4 instead of 4.0, the 4 will be interpreted as an `int`.

The distinction between floats and doubles was made because of the differing sizes of the two types. When C was first used, space was at a minimum and so the judicious use of a float instead of a double saved some memory. Nowadays, with memory more freely available, you do not really need to conserve memory like this – it may be better to use doubles consistently. Indeed, some C implementations use doubles instead of floats when you declare a float variable.

If you want to use a double variable, use the `double` keyword.

12.4 `sizeof`

If you have any doubts as to the amount of memory actually used by any variable (and this goes for types we'll discuss later, also), you can use the `sizeof` operator to find out for sure. (For completeness, it is important to mention that `sizeof` is a unary operator[6], not a function.) Its syntax is:

```
sizeof object
sizeof(type)
```

The two expressions above return the size of the object and type specified, in bytes. The return type is `size_t` (defined in the header <`stddef.h`>) which is an unsigned value. Here's an example usage:

```
size_t size;
int i;
size = sizeof(i);
```

`size` will be set to 4, assuming `CHAR_BIT` is defined as 8, and an integer is 32 bits wide. The value of `sizeof`'s result is the number of bytes.

5 `http://en.wikipedia.org/wiki/Scientific%20notation`
6 `http://en.wikipedia.org/wiki/Unary%20operation`

Note that when `sizeof` is applied to a `char`, the result is 1; that is:

```
sizeof(char)
```

always returns 1.

12.5 Data type modifiers

One can alter the data storage of any data type by preceding it with certain modifiers.

`long` and `short` are modifiers that make it possible for a data type to use either more or less memory. The `int` keyword need not follow the `short` and `long` keywords. This is most commonly the case. A `short` can be used where the values fall within a lesser range than that of an `int`, typically -32768 to 32767. A `long` can be used to contain an extended range of values. It is not guaranteed that a `short` uses less memory than an `int`, nor is it guaranteed that a `long` takes up more memory than an `int`. It is only guaranteed that sizeof(short) <= sizeof(int) <= sizeof(long). Typically a `short` is 2 bytes, an `int` is 4 bytes, and a `long` either 4 or 8 bytes. Modern C compilers also provide `long long` which is typically an 8 byte integer.

In all of the types described above, one bit is used to indicate the sign (positive or negative) of a value. If you decide that a variable will never hold a negative value, you may use the `unsigned` modifier to use that one bit for storing other data, effectively doubling the range of values while mandating that those values be positive. The `unsigned` specifier also may be used without a trailing `int`, in which case the size defaults to that of an `int`. There is also a `signed` modifier which is the opposite, but it is not necessary, except for certain uses of `char`, and seldom used since all types (except `char`) are signed by default.

To use a modifier, just declare a variable with the data type and relevant modifiers:

```
unsigned short int usi;  /* fully qualified -- unsigned short int */
short si;                /* short int */
unsigned long uli;       /* unsigned long int */
```

12.6 `const` qualifier

When the `const` qualifier is used, the declared variable must be initialized at declaration. It is then not allowed to be changed.

While the idea of a variable that never changes may not seem useful, there are good reasons to use `const`. For one thing, many compilers can perform some small optimizations on data when it knows that data will never change. For example, if you need the value of π in your calculations, you can declare a const variable of `pi`, so a program or another function written by someone else cannot change the value of `pi`.

Note that a Standard conforming compiler must issue a warning if an attempt is made to change a `const` variable - but after doing so the compiler is free to ignore the `const` qualifier.

12.7 Magic numbers

When you write C programs, you may be tempted to write code that will depend on certain numbers. For example, you may be writing a program for a grocery store. This complex program has thousands upon thousands of lines of code. The programmer decides to represent the cost of a can of corn, currently 99 cents, as a literal throughout the code. Now, assume the cost of a can of corn changes to 89 cents. The programmer must now go in and manually change each entry of 99 cents to 89. While this is not that big of a problem, considering the "global find-replace" function of many text editors, consider another problem: the cost of a can of green beans is also initially 99 cents. To reliably change the price, you have to look at every occurrence of the number 99.

C possesses certain functionality to avoid this. This functionality is approximately equivalent, though one method can be useful in one circumstance, over another.

12.7.1 Using the `const` keyword

The `const` keyword helps eradicate **magic numbers**. By declaring a variable `const corn` at the beginning of a block, a programmer can simply change that const and not have to worry about setting the value elsewhere.

There is also another method for avoiding magic numbers. It is much more flexible than `const`, and also much more problematic in many ways. It also involves the preprocessor, as opposed to the compiler. Behold...

12.7.2 `#define`

When you write programs, you can create what is known as a *macro*, so when the computer is reading your code, it will replace all instances of a word with the specified expression.

Here's an example. If you write

```
#define PRICE_OF_CORN 0.99
```

when you want to, for example, print the price of corn, you use the word `PRICE_OF_CORN` instead of the number 0.99 – the preprocessor will replace all instances of `PRICE_OF_CORN` with 0.99, which the compiler will interpret as the literal `double` 0.99. The preprocessor performs substitution, that is, `PRICE_OF_CORN` is replaced by 0.99 so this means there is no need for a semicolon.

It is important to note that `#define` has basically the same functionality as the "find-and-replace" function in a lot of text editors/word processors.

For some purposes, `#define` can be harmfully used, and it is usually preferable to use `const` if `#define` is unnecessary. It is possible, for instance, to `#define`, say, a macro `DOG` as the number 3, but if you try to print the macro, thinking that `DOG` represents a string that you can show on the screen, the program will have an error. `#define` also has no regard for type. It disregards the structure of your program, replacing the text *everywhere* (in effect, disregarding scope), which could be advantageous in some circumstances, but can be the source of problematic bugs.

You will see further instances of the `#define` directive later in the text. It is good convention to write `#define`d words in all capitals, so a programmer will know that this is not a variable that you have declared but a `#define`d macro. It is not necessary to end a preprocessor directive such as `#define` with a semicolon; in fact, some compilers may warn you about unnecessary tokens in your code if you do.

12.8 Scope

In the Basic Concepts section, the concept of scope was introduced. It is important to revisit the distinction between local types and global types, and how to declare variables of each. To declare a local variable, you place the declaration at the beginning (i.e. before any non-declarative statements) of the block to which the variable is intended to be local. To declare a global variable, declare the variable outside of any block. If a variable is global, it can be read, and written, from anywhere in your program.

Global variables are not considered good programming practice, and should be avoided whenever possible. They inhibit code readability, create naming conflicts, waste memory, and can create difficult-to-trace bugs. Excessive usage of globals is usually a sign of laziness and/or poor design. However, if there is a situation where local variables may create more obtuse and unreadable code, there's no shame in using globals.

12.9 Other Modifiers

Included here, for completeness, are more of the modifiers that standard C provides. For the beginning programmer, *static* and *extern* may be useful. *volatile* is more of interest to advanced programmers. *register* and *auto* are largely deprecated and are generally not of interest to either beginning or advanced programmers.

12.9.1 static

`static` is sometimes a useful keyword. It is a common misbelief that the only purpose is to make a variable stay in memory.

When you declare a function or global variable as *static* it will become internal. You cannot access the function or variable through the extern (see below) keyword from other files in your project.

When you declare a local variable as *static*, it is created just like any other variable. However, when the variable goes out of scope (i.e. the block it was local to is finished) the variable stays in memory, retaining its value. The variable stays in memory until the program ends. While this behaviour resembles that of global variables, static variables still obey scope rules and therefore cannot be accessed outside of their scope.

Variables declared static are initialized to zero (or for pointers, NULL) by default.

You can use static in (at least) two different ways. Consider this code, and imagine it is in a file called jfile.c:

```
#include <stdio.h>

static int j = 0;

void up(void)
{
   /* k is set to 0 when the program starts. The line is then "ignored"
    * for the rest of the program (i.e. k is not set to 0 every time up()
    * is called)
    */
   static int k = 0;
   j++;
   k++;
   printf("up() called.   k= %2d, j= %2d\n", k , j);
}

void down(void)
{
   static int k = 0;
   j--;
   k--;
   printf("down() called. k= %2d, j= %2d\n", k , j);
}

int main(void)
{
   int i;

   /* call the up function 3 times, then the down function 2 times */
   for (i= 0; i < 3; i++)
      up();
   for (i= 0; i < 2; i++)
      down();

   return 0;
}
```

The j var is accessible by both up and down and retains its value. The k vars also retain their value, but they are two different variables, one in each of their scopes. Static vars are a good way to implement encapsulation, a term from the object-oriented way of thinking that effectively means not allowing changes to be made to a variable except through function calls.

Running the program above will produce the following output:

```
up() called.   k=  1, j=  1
up() called.   k=  2, j=  2
up() called.   k=  3, j=  3
down() called. k= -1, j=  2
down() called. k= -2, j=  1
```

Features of `static` variables :

```
    1. Keyword used        - static
    2. Storage             - Memory
    3. Default value       - Zero
    4. Scope               - Local to the block in which it is declared
    5. Lifetime            - Value persists between different function calls
    6. Keyword optionality - Mandatory to use the keyword
```

12.9.2 extern

`extern` is used when a file needs to access a variable in another file that it may not have `#include`d directly. Therefore, *extern* does not actually carve out space for a new variable, it just provides the compiler with sufficient information to access the remote variable.

Features of `external` variable :

```
    1. Keyword used        - extern
    2. Storage             - Memory
    3. Default value       - Zero
    4. Scope               - Global (all over the program)
    5. Lifetime            - Value persists till the program's execution comes
to an end
    6. Keyword optionality - Optional if declared outside all the functions
```

12.9.3 volatile

`volatile` is a special type of modifier which informs the compiler that the value of the variable may be changed by external entities other than the program itself. This is necessary for certain programs compiled with optimizations – if a variable were not defined `volatile` then the compiler may assume that certain operations involving the variable are safe to optimize away when in fact they aren't. *volatile* is particularly relevant when working with embedded systems (where a program may not have complete control of a variable) and multi-threaded applications.

12.9.4 auto

`auto` is a modifier which specifies an "automatic" variable that is automatically created when in scope and destroyed when out of scope. If you think this sounds like pretty much what you've been doing all along when you declare a variable, you're right: all declared items within a block are implicitly "automatic". For this reason, the *auto* keyword is more like the answer to a trivia question than a useful modifier, and there are lots of very competent programmers that are unaware of its existence.

Features of `automatic` variables :

```
    1. Keyword used        - auto
    2. Storage             - Memory
    3. Default value       - Garbage value (random value)
    4. Scope              - Local to the block in which it is defined
    5. Lifetime           - Value persists while the control remains within
the block
    6. Keyword optionality - Optional
```

12.9.5 register

`register` is a hint to the compiler to attempt to optimize the storage of the given variable by storing it in a register of the computer's CPU when the program is run. Most optimizing compilers do this anyway, so use of this keyword is often unnecessary. In fact, ANSI C states that a compiler can ignore this keyword if it so desires – and many do. Microsoft Visual C++ is an example of an implementation that completely ignores the *register* keyword.

Features of `register` variables :

```
    1. Keyword used        - register
    2. Storage             - CPU registers (values can be retrieved faster than
from memory)
    3. Default value       - Garbage value
    4. Scope               - Local to the block in which it is defined
    5. Lifetime            - Value persists while the control remains within
the block
    6. Keyword optionality - Mandatory to use the keyword
```

12.9.6 Concepts

- Variables[7]
- Types[8]
- Data Structures[9]
- Arrays[10]

12.9.7 In this section

- C variables[11]
  - C types[12]
  - C arrays[13]

7 `http://en.wikibooks.org/wiki/Computer%20Programming%2FVariables`
8 `http://en.wikibooks.org/wiki/Computer%20Programming%2FTypes`
9 `http://en.wikibooks.org/wiki/Data%20Structures`
10 `http://en.wikibooks.org/wiki/Data%20Structures%2FArrays`
11 Chapter 12 on page 47
12 `http://en.wikibooks.org/wiki/C%20Programming%2FTypes`
13 Chapter 24 on page 181

et:Programmeerimiskeel C/Muutujad[14] it:C/Variabili, operatori e costanti/Variabili[15] pl:C/Zmienne[16] fi:C/Muuttujat[17]

14 http://et.wikibooks.org/wiki/Programmeerimiskeel%20C%2FMuutujad
15 http://it.wikibooks.org/wiki/C%2FVariabili%2C%20operatori%20e%20costanti%2FVariabili
16 http://pl.wikibooks.org/wiki/C%2FZmienne
17 http://fi.wikibooks.org/wiki/C%2FMuuttujat

13 Simple Input and Output

When you take time to consider it, a computer would be pretty useless without some way to talk to the people who use it. Just like we need information in order to accomplish tasks, so do computers. And just as we supply information to others so that *they* can do tasks, so do computers.

These supplies and returns of information to a computer are called **input** and **output**. 'Input' is information supplied to a computer or program. 'Output' is information provided by a computer or program. Frequently, computer programmers will lump the discussion in the more general term *input/output* or simply, **I/O**.

In C, there are many different ways for a program to communicate with the user. Amazingly, the most simple methods usually taught to beginning programmers may also be the most powerful. In the "Hello, World" example[1] at the beginning of this text, we were introduced to a Standard Library file stdio.h, and one of its functions, printf(). Here we discuss more of the functions that stdio.h gives us.

13.1 Output using printf()

Recall from the beginning of this text the demonstration program duplicated below:

```
#include <stdio.h>

int main(void)
{
   printf("Hello, world!\n");
   return 0;
}
```

If you compile and run this program, you will see the sentence below show up on your screen:

```
Hello, world!
```

This amazing accomplishment was achieved by using the *function* `printf()`. A function is like a "black box" that does something for you without exposing the internals inside. We can write functions ourselves in C, but we will cover that later.

You have seen that to use `printf()` one puts text, surrounded by quotes, in between the parentheses. We call the text surrounded by quotes a *literal string* (or just a *string*), and we call that string an *argument* to printf.

1 http://en.wikibooks.org/wiki/Programming%3AC%23A%20taste%20of%20C

As a note of explanation, it is sometimes convenient to include the open and closing parentheses after a function name to remind us that it is, indeed, a function. However usually when the name of the function we are talking about is understood, it is not necessary.

As you can see in the example above, using `printf()` can be as simple as typing in some text, surrounded by double quotes (note that these are double quotes and not two single quotes). So, for example, you can print any string by placing it as an argument to the `printf()` function:

```
printf("This sentence will print out exactly as you see it...");
```

And once it is contained in a proper `main()` function, it will show:

```
This sentence will print out exactly as you see it...
```

13.1.1 Printing numbers and escape sequences

Placeholder codes

The `printf` function is a powerful function, and is probably the most-used function in C programs.

For example, let us look at a problem. Say we don't know what 19 + 31 is. Let's use C to get the answer.

We start writing

```
#include "stdio.h" // this is important, since printf
                   // can't be used without this line
```

```
int main(void)
{
   printf("19+31 is");
```

but here we are stuck! `printf` only prints strings! Thankfully, printf has methods for printing numbers. What we do is put a *placeholder* format code in the string. We write:

```
   printf("19+31 is %d", 19+31);
```

The placeholder %d literally "holds the place" for the actual number that is the result of adding 19 to 31.

These placeholders are called **format specifiers**. Many other format specifiers work with `printf`. If we have a floating-point number, we can use `%f` to print out a floating-point number, decimal point and all. Other format specifiers are:

- %d - int (same as %i)
- %ld - long int (same as %li)
- %f - float

- %lf - double
- %c - char
- %s - string
- %x - hexadecimal

Tabs and newlines

What if, we want to achieve some output that will look like:

```
 1905
312 +
-----
```

`printf` will not put line breaks in at the end of each statement: we must do this ourselves. But how?

What we can do is use the newline *escape character*. An escape character is a special character that we can write but will do something special onscreen, such as make a beep, write a tab, and so on. To write a newline we write `\n`. All escape characters start with a backslash.

So to achieve the output above, we write

```
printf(" 1905\n312 +\n-----\n");
```

or to be a bit more clear, we can break this long printf statement over several lines. So our program will be

```
#include <stdio.h>

int main(void)
{
   printf(" 1905\n");
   printf("312 +\n");
   printf("-----\n");
   printf("%d", 1905+312);
   return 0;
}
```

There are other escape characters we can use. Another common one is to use `\t` to write a tab. You can use `\a` to ring the computer's bell, but you should not use this very much in your programs, as excessive use of sound is not very friendly to the user.

13.2 Other output methods

13.2.1 puts()

The puts() function is a very simple way to send a string to the screen when you have no placeholders to be concerned about. It works very much like the printf() function we saw in the "Hello, World!" example:

```
puts("Print this string.");
```

will print to the screen:

```
Print this string.
```

followed by the newline character (as discussed above). (The `puts` function appends a newline character to its output.)

```
#include<stdio.h>
f(int i,int j,int k)
{
  printf("%d%d%d",i,j,k);
}
main()
{
  int x=1,y=2,z=3;
  f(x+y,y=x+z,z=x+y);
}
```

13.3 Input using scanf()

The scanf() function is the input method equivalent to the printf() output function - simple yet powerful. In its simplest invocation, the scanf *format string* holds a single *placeholder* representing the type of value that will be entered by the user. These placeholders are exactly the same as the printf() function - %d for ints, %f for floats, and %lf for doubles.

There is, however, one variation to scanf() as compared to printf(). The scanf() function requires the memory address of the variable to which you want to save the input value. While *pointers* are possible here, this is a concept that won't be approached until later in the text. Instead, the simple technique is to use the *address-of* operator, **&**. For now it may be best to consider this "magic" before we discuss pointers.

A typical application might be like this:

```
#include "stdio.h"

int main(void)
{
   int a;

   printf("Please input an integer value: ");
   scanf("%d", &a);
   printf("You entered: %d\n", a);

   return 0;
}
```

If you were to describe the effect of the scanf() function call above, it might read as: "Read in an integer from the user and store it at the address of variable *a* ".

If you are trying to input a *string* using *scanf*, you should **not** include the & operator. The code below will not compile.

```
    scanf("%s", &a);
```

The correct usage would be:

```
    scanf("%s", a);
```

This is because, whenever you use a format specifier for a string (%s), the variable that you use to store the value will be an array and, the array names (in this case - a) themselves point out to their base address and hence, the **address of** operator is not required.

(Although, this is vulnerable to Buffer overflow[2]. fgets() is preferred to scanf()).

Note on inputs: When data is typed at a keyboard, the information does not go straight to the program that is running. It is first stored in what is known as a **buffer** - a small amount of memory reserved for the input source. Sometimes there will be data left in the buffer when the program wants to read from the input source, and the scanf() function will read this data instead of waiting for the user to type something. Some may suggest you use the function fflush(stdin), which may work as desired on some computers, but isn't considered good practice, as you will see later. Doing this has the downfall that if you take your code to a different computer with a different compiler, your code may not work properly.

13.4 Links

Back to contents: Beginning C[3]

et:Programmeerimiskeel C/IO[4] pl:C/Podstawowe procedury wejścia i wyjścia[5] pt:Programar em C/Entrada e saída simples[6] [7]

2 `http://en.wikipedia.org/wiki/Buffer%20overflow`
3 `http://en.wikibooks.org/wiki/C%20Programming%23Beginning%20C`
4 `http://et.wikibooks.org/wiki/Programmeerimiskeel%20C%2FIO`
5 `http://pl.wikibooks.org/wiki/C%2FPodstawowe%20procedury%20wej%C5%9Bcia%20i%20wyj%C5%9Bcia`
6 `http://pt.wikibooks.org/wiki/Programar%20em%20C%2FEntrada%20e%20sa%C3%ADda%20simples`
7 `http://en.wikibooks.org/wiki/Category%3AC%20Programming`

14 Simple math

14.1 Operators and Assignments

C has a wide range of operators that make simple math easy to handle. The list of operators grouped into precedence levels is as follows:

14.1.1 Primary expressions

An identifier is a primary expression, provided that it has been declared as designating an object (in which case it is an lvalue [a value that can be used as the left side of an assignment expression]) or a function (in which case it is a function designator).

A constant is a primary expression. Its type depends on its form and value.

A string literal is a primary expression.

A parenthesized expression is a primary expression. Its type and value are those of the unparenthesized expression.

14.1.2 Postfix operators

First, a primary expression is also a postfix expression. The following expressions are also postfix expressions:

A postfix expression followed by a left square bracket (`[`), an expression, and a right square bracket (`]`) constitutes an invocation of the array subscript operator. One of the expressions shall have type "pointer to object *type*" and the other shall have an integer type; the result type is *type*. Successive array subscript operators designate an element of a multidimensional array.

A postfix expression followed by parentheses or an optional parenthesized argument list indicates an invocation of the function call operator.

A postfix expression followed by a dot (`.`) followed by an identifier selects a member from a structure or union; a postfix expression followed by an arrow (`->`) followed by an identifier selects a member from a structure or union who is pointed to by the pointer on the left-hand side of the expression.

A postfix expression followed by the increment or decrement operators (`++` or `--`) indicates that the variable is to be incremented or decremented as a side effect. The value of the expression is the value of the postfix expression *before* the increment or decrement.

14.1.3 Unary expressions

First, a unary expression is a postfix expression. The following expressions are all postfix expressions:

The increment or decrement operators followed by a unary expression is a unary expression. The value of the expression is the value of the unary expression *after* the increment or decrement.

The following operators followed by a cast expression are unary expressions:

```
Operator    Meaning
========    =======
   &        Address-of; value is the location of the operand
   *        Contents-of; value is what is stored at the location
   -        Negation
   +        Value-of operator
   !        Logical negation ( (!E) is equivalent to (0==E) )
   ~        Bit-wise complement
```

The keyword `sizeof` followed by a unary expression is a unary expression. The value is the size of the type of the expression in bytes. The expression is not evaluated.

The keyword `sizeof` followed by a parenthesized type name is a unary expression. The value is the size of the type in bytes.

14.1.4 Cast operators

A cast expression is a unary expression.

A parenthesized type name followed by a cast expression is a cast expression. The parenthesized type name has the effect of forcing the cast expression into the type specified by the type name in parentheses. For arithmetic types, this either does not change the value of the expression, or truncates the value of the expression if the expression is an integer and the new type is smaller than the previous type.

An example of casting a float as an int:

```
float pi = 3.141592;
int truncated_pi = (int)pi; // truncated_pi == 3
```

An example of casting a char as an int:

```
char my_char = 'A';
int my_int = (int)my_char; // my_int == 65, which is the ASCII value of 'A'
```

14.1.5 Multiplicative and additive operators

In C, simple math is very easy to handle. The following operators exist: + (addition), - (subtraction), * (multiplication), / (division), and % (modulus); You likely know all of them from your math classes - except, perhaps, modulus. It returns the **remainder** of a division (e.g. 5 % 2 = 1).

Care must be taken with the modulus, because it's not the equivalent of the mathematical modulus: (-5) % 2 is not 1, but -1. Division of integers will return an integer, and the division of a negative integer by a positive integer will round towards zero instead of rounding down (e.g. (-5) / 3 = -1 instead of -2).

There is no inline operator to do the power (e.g. 5 ^ 2 is **not** 25, and 5 ** 2 is an error), but there is a power function[1].

The mathematical order of operations does apply. For example (2 + 3) * 2 = 10 while 2 + 3 * 2 = 8. Multiplicative operators have precedence over additive operators.

```
#include <stdio.h>

int main()
{
int i = 0, j = 0;

    /* while i is less than 5 AND j is less than 5, loop */
    while( (i < 5) && (j < 5) )
    {
        /* postfix increment, i++
         *     the value of i is read and then incremented
         */
        printf("i: %d\t", i++);

        /*
         * prefix increment, ++j
         *     the value of j is incremented and then read
         */
        printf("j: %d\n", ++j);
    }

    printf("At the end they have both equal values:\ni: %d\tj: %d\n", i, j);

    return 0;
}
```

will display the following:

```
i: 0    j: 1
i: 1    j: 2
i: 2    j: 3
i: 3    j: 4
i: 4    j: 5
At the end they have both equal values:
i: 5    j: 5
```

14.1.6 shift and rotate

Shift functions are often used in low-level I/O hardware interfacing. Shift and rotate functions are heavily used in cryptography and software floating point emulation. Other than that, shifts can be used in place of division or multiplication by a power of two. Many processors have dedicated function blocks to make these operations fast -- see Microprocessor Design/Shift and Rotate Blocks[2]. On processors which have such blocks, most C

1 Chapter 15.4 on page 77
2 http://en.wikibooks.org/wiki/Microprocessor%20Design%2FShift%20and%20Rotate%20Blocks

compilers compile shift and rotate operators to a single assembly-language instruction -- see X86 Assembly/Shift and Rotate[3].

shift left

The << operator shifts the binary representation to the left, dropping the most significant bits and appending it with zero bits. The result is equivalent to multiplying the integer by a power of two.

unsigned shift right

The unsigned shift right operator, also sometimes called the logical right shift operator. It shifts the binary representation to the right, dropping the least significant bits and prepending it with zeros. The >> operator is equivalent to division by a power of two for unsigned integers.

signed shift right

The signed shift right operator, also sometimes called the arithmetic right shift operator. It shifts the binary representation to the right, dropping the least significant bit, but prepending it with copies of the original sign bit. The >> operator is not equivalent to division for signed integers.

In C, the behavior of the >> operator depends on the data type it acts on. Therefore, a signed and an unsigned right shift looks exactly the same, but produces a different result in some cases.

rotate right

Contrary to popular belief, it is possible to write C code that compiles down to the "rotate" assembly language instruction (on CPUs that have such an instruction).

Most compilers recognize this idiom:

```
unsigned int x;
unsigned int y;
/* ... */
y = (x >> shift) | (x << (32 - shift));
```

and compile it to a single 32 bit rotate instruction. [4] [5]

3 http://en.wikibooks.org/wiki/X86%20Assembly%2FShift%20and%20Rotate

4 GCC: "Optimize common rotate constructs" ^{http://gcc.gnu.org/ml/gcc-patches/2007-11/msg01112.html}

5 "Cleanups in ROTL/ROTR DAG combiner code" ^{http://www.mail-archive.com/llvm-commits@cs.uiuc.edu/msg17216.html} mentions that this code supports the "rotate" instruction in the CellSPU

On some systems, this may be "#define"ed as a macro or defined as an inline function called something like "rightrotate32" or "rotr32" or "ror32" in a standard header file like "bitops.h". [6]

rotate left

Most compilers recognize this idiom:

```
unsigned int x;
unsigned int y;
/* ... */
y = (x << shift) | (x >> (32 - shift));
```

and compile it to a single 32 bit rotate instruction.

On some systems, this may be "#define"ed as a macro or defined as an inline function called something like "leftrotate32" or "rotl32" in a header file like "bitops.h".

14.1.7 Relational and equality operators

The relational binary operators < (less than), > (greater than), <= (less than or equal), and >= (greater than or equal) operators return a value of 1 if the result of the operation is true, 0 if false.

The equality binary operators `==` (equals) and `!=` (not equals) operators are similar to the relational operators except that their precedence is lower.

14.1.8 Bitwise operators

The bitwise operators are `&` (and), `^` (exclusive or) and `|` (inclusive or). The `&` operator has higher precedence than `^`, which has higher precedence than `|`.

14.1.9 Logical operators

The logical operators are `&&` (and), and `||` (or). Both of these operators produce 1 if the relationship is true and 0 for false. Both of these operators short-circuit; if the result of the expression can be determined from the first operand, the second is ignored.

`&&` is used to evaluate expressions left to right, and returns a 1 if *both* statements are true.

```
int x = 7;
int y = 5;
if(x == 7 && y == 5) {
    ...
}
```

6 "replace private copy of bit rotation routines" ^{`http://kerneltrap.org/mailarchive/linux-kernel/2008/4/15/1440064`} -- recommends includeing "bitops.h" and using its rol32 and ror32 rather than copy-and-paste into a new program.

Here, the && operator checks the left-most expression, then the expression to it's right. Since both statements return true, the && operator returns true, and the code block is executed.

```
    if(x == 5 && y == 5) {
        ...
    }
```

The && operator checks in the same way as before, and finds that the first expression is false. The && operator stops evaluating as soon as it finds a statement to be false, and returns a false.

`||` is used to evaluate expressions left to right, and returns a 1 if *either* of the expressions are true.

```
    /* Use the same variables as before. */
    if(x == 2 || y == 5) { // the || statement checks both expressions, finds
 that the latter is true, and returns true
        ...
    }
```

The `||` operator here checks the left-most expression, finds it false, but continues to evaluate the next expression. It finds that the next expression returns true, stops, and returns a 1. Much how the `&&` operator ceases when it finds an expression that returns false, the `||` operator ceases when it finds an expression that returns true.

It is worth noting that C does not have Boolean values (true and false) commonly found in other languages. It instead interprets a 0 as false, and any nonzero value as true.

14.1.10 Conditional operators

The ternary `?:` operator is the conditional operator. The expression `(x ? y : z)` has the value of `y` if `x` is nonzero, `z` otherwise.

Example:

```
int x = 0;
int y;
y = (x ? 10:6);
```

The expression `x` evaluates to 0. The ternary operator then looks for the "if-false" value, which in this case, is 6. It returns that, so `y` is equal to six. Had `x` been a non-zero, then the expression would have returned a 10.

14.1.11 Assignment operators

The assignment operators are `=`, `*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `&=`, `^=`, and `|=` . The `=` operator stores the value of the right operand into the location determined by the left operand, which must be an lvalue[7]

For the others, `x op= y` is shorthand for `x = x op (y)` . Hence, the following expressions are the same :

```
    1. x += y     -     x = x+y
    2. x -= y     -     x = x-y
    3. x *= y     -     x = x*y
    4. x /= y     -     x = x/y
    5. x %= y     -     x = x%y
```

14.1.12 Comma operator

The operator with the least precedence is the comma operator. The value of the expression `x, y` will evaluate both `x` and `y`, but provides the value of `y`.

This operator is useful for including multiple actions in one statement (e.g. within a for loop conditional).

Here are some small examples of the comma operator:

```
int i, x;      /* declares two ints, i and x, in one statement */

/* this loop initializes x and i to 0, then runs the loop */
for(x = 0, i = 0; i <= 6; i++) {
    printf("x = %d, and i = %d\n", x, i);
}
```

pl:C/Operatory[8]

7 http://en.wikibooks.org/wiki/lvalue
8 http://pl.wikibooks.org/wiki/C%2FOperatory

15 Further math

w:math.h[1]

The `<math.h>` header contains prototypes for several functions that deal with mathematics. In the 1990 version of the ISO standard, only the `double` versions of the functions were specified; the 1999 version added the `float` and `long double` versions. To use these math functions, you must link your program with the math library. For some compilers (including GCC), you must specify the additional parameter `-lm`.

The functions can be grouped into the following categories:

15.1 Trigonometric functions

15.1.1 The `acos` and `asin` functions

The `acos` functions return the arccosine of their arguments in radians, and the `asin` functions return the arcsine of their arguments in radians. All functions expect the argument in the range [-1,+1]. The arccosine returns a value in the range [0,π]; the arcsine returns a value in the range [-π/2,+π/2].

```
#include <math.h>
float asinf(float x); /* C99 */
float acosf(float x); /* C99 */
double asin(double x);
double acos(double x);
long double asinl(long double x); /* C99 */
long double acosl(long double x); /* C99 */
```

15.1.2 The `atan` and `atan2` functions

The `atan` functions return the arctangent of their arguments in radians, and the `atan2` function return the arctangent of `y/x` in radians. The `atan` functions return a value in the range [-π/2,+π/2] (the reason why ±π/2 are included in the range is because the floating-point value may represent infinity, and atan(±∞) = ±π/2); the `atan2` functions return a value in the range [-π/2,+π/2]. For `atan2`, a domain error may occur if both arguments are zero.

```
#include <math.h>
float atanf(float x); /* C99 */
float atan2f(float y, float x); /* C99 */
```

1 http://en.wikipedia.org/wiki/math.h

```
double atan(double x);
double atan2(double y, double x);
long double atanl(long double x); /* C99 */
long double atan2l(long double y, long double x); /* C99 */
```

15.1.3 The `cos`, `sin`, and `tan` functions

The `cos`, `sin`, and `tan` functions return the cosine, sine, and tangent of the argument, expressed in radians.

```
#include <math.h>
float cosf(float x); /* C99 */
float sinf(float x); /* C99 */
float tanf(float x); /* C99 */
double cos(double x);
double sin(double x);
double tan(double x);
long double cosl(long double x); /* C99 */
long double sinl(long double x); /* C99 */
long double tanl(long double x); /* C99 */
```

15.2 Hyperbolic functions

The `cosh`, `sinh` and `tanh` functions compute the hyperbolic cosine, the hyperbolic sine, and the hyperbolic tangent of the argument respectively. For the hyperbolic sine and cosine functions, a range error occurs if the magnitude of the argument is too large.

The `acosh` functions compute the inverse hyperbolic cosine of the argument. A domain error occurs for arguments less than 1.

The `asinh` functions compute the inverse hyperbolic sine of the argument.

The `atanh` functions compute the inverse hyperbolic tangent of the argument. A domain error occurs if the argument is not in the interval [-1, +1]. A range error may occur if the argument equals -1 or +1.

```
#include <math.h>
float coshf(float x); /* C99 */
float sinhf(float x); /* C99 */
float tanhf(float x); /* C99 */
double cosh(double x);
double sinh(double x);
double tanh(double x);
long double coshl(long double x); /* C99 */
long double sinhl(long double x); /* C99 */
long double tanhl(long double x); /* C99 */
float acoshf(float x); /* C99 */
float asinhf(float x); /* C99 */
float atanhf(float x); /* C99 */
double acosh(double x); /* C99 */
double asinh(double x); /* C99 */
double atanh(double x); /* C99 */
long double acoshl(long double x); /* C99 */
long double asinhl(long double x); /* C99 */
long double atanhl(long double x); /* C99 */
```

15.3 Exponential and logarithmic functions

15.3.1 The exp, exp2, and expm1 functions

The `exp` functions compute the base-e exponential function of `x` (e^x). A range error occurs if the magnitude of `x` is too large.

The `exp2` functions compute the base-2 exponential function of `x` (2^x). A range error occurs if the magnitude of `x` is too large.

The `expm1` functions compute the base-e exponential function of the argument, minus 1. A range error occurs in the magnitude of `x` is too large.

```
#include <math.h>
float expf(float x); /* C99 */
double exp(double x);
long double expl(long double x); /* C99 */
float exp2f(float x); /* C99 */
double exp2(double x); /* C99 */
long double exp2l(long double x); /* C99 */
float expm1f(float x); /* C99 */
double expm1(double x); /* C99 */
long double expm1l(long double x); /* C99 */
```

15.3.2 The frexp, ldexp, modf, scalbn, and scalbln functions

These functions are heavily used in software floating-point emulators, but are otherwise rarely directly called.

Inside the computer, each floating point number is represented by two parts:

- The significand is either in the range [1/2, 1), or it equals zero.
- The exponent is an integer.

The value of a floating point number v is $v = \text{significand} \times 2^{\text{exponent}}$.

The `frexp` functions break the argument floating point number `value` into those two parts, the exponent and significand. After breaking it apart, it stores the exponent in the `int` object pointed to by `ex`, and returns the significand. In other words, the value returned is a copy of the given floating point number but with an exponent replaced by 0. If `value` is zero, both parts of the result are zero.

The `ldexp` functions multiply a floating-point number by a integral power of 2 and return the result. In other words, it returns copy of the given floating point number with the exponent increased by ex. A range error may occur.

The `modf` functions break the argument `value` into integer and fraction parts, each of which has the same sign as the argument. They store the integer part in the object pointed to by `*iptr` and return the fraction part. The `*iptr` is a floating-point type, rather than an "int" type, because it might be used to store an integer like 1 000 000 000 000 000 000 000 which is too big to fit in an int.

The `scalbn` and `scalbln` compute `x` $\times$ `FLT_RADIX`$^{\mathtt{n}}$. `FLT_RADIX` is the base of the floating-point system; if it is 2, the functions are equivalent to `ldexp`.

```
#include <math.h>
float frexpf(float value, int *ex); /* C99 */
double frexp(double value, int *ex);
long double frexpl(long double value, int *ex); /* C99 */
float ldexpf(float x, int ex); /* C99 */
double ldexp(double x, int ex);
long double ldexpl(long double x, int ex); /* C99 */
float modff(float value, float *iptr); /* C99 */
double modf(double value, double *iptr);
long double modfl(long double value, long double *iptr); /* C99 */
float scalbnf(float x, int ex); /* C99 */
double scalbn(double x, int ex); /* C99 */
long double scalbnl(long double x, int ex); /* C99 */
float scalblnf(float x, long int ex); /* C99 */
double scalbln(double x, long int ex); /* C99 */
long double scalblnl(long double x, long int ex); /* C99 */
```

Most C floating point libraries also implement the IEEE754-recommended nextafter(), nextUp( ), and nextDown( ) functions. `http://www.opengroup.org/onlinepubs/009695399/functions/nextafter.html`

15.3.3 The `log`, `log2`, `log1p`, and `log10` functions

The `log` functions compute the base-*e* natural (**not** common) logarithm of the argument and return the result. A domain error occurs if the argument is negative. A range error may occur if the argument is zero.

The `log1p` functions compute the base-*e* natural (**not** common) logarithm of one plus the argument and return the result. A domain error occurs if the argument is less than -1. A range error may occur if the argument is -1.

The `log10` functions compute the common (base-10) logarithm of the argument and return the result. A domain error occurs if the argument is negative. A range error may occur if the argument is zero.

The `log2` functions compute the base-2 logarithm of the argument and return the result. A domain error occurs if the argument is negative. A range error may occur if the argument is zero.

```
#include <math.h>
float logf(float x); /* C99 */
double log(double x);
long double logl(long double x); /* C99 */
float log1pf(float x); /* C99 */
double log1p(double x); /* C99 */
long double log1pl(long double x); /* C99 */
float log10f(float x); /* C99 */
double log10(double x);
long double log10l(long double x); /* C99 */
float log2f(float x); /* C99 */
double log2(double x); /* C99 */
long double log2l(long double x); /* C99 */
```

15.3.4 The `ilogb` and `logb` functions

The `ilogb` functions extract the exponent of `x` as a signed int value. If `x` is zero, they return the value `FP_ILOGB0`; if `x` is infinite, they return the value `INT_MAX`; if `x` is not-a-number they return the value `FP_ILOGBNAN`; otherwise, they are equivalent to calling the corresponding `logb` function and casting the returned value to type `int`. A range error may occur if `x` is zero. `FP_ILOGB0` and `FP_ILOGBNAN` are macros defined in `math.h`; `INT_MAX` is a macro defined in `limits.h`.

The `logb` functions extract the exponent of `x` as a signed integer value in floating-point format. If `x` is subnormal, it is treated as if it were normalized; thus, for positive finite `x`, 1 ≤ `x` × `FLT_RADIX`$^{\texttt{-logb(x)}}$ < `FLT_RADIX` . `FLT_RADIX` is the radix for floating-point numbers, defined in the `float.h` header.

```
#include <math.h>
int ilogbf(float x); /* C99 */
int ilogb(double x); /* C99 */
int double ilogbl(long double x); /* C99 */
float logbf(float x); /* C99 */
double logb(double x); /* C99 */
long double logbl(long double x); /* C99 */
```

15.4 Power functions

15.4.1 The `pow` functions

The `pow` functions compute `x` raised to the power `y` and return the result. A domain error occurs if `x` is negative and `y` is not an integral value. A domain error occurs if the result cannot be represented when `x` is zero and `y` is less than or equal to zero. A range error may occur.

```
#include <math.h>
float powf(float x, float y); /* C99 */
double pow(double x, double y);
long double powl(long double x, long double y); /* C99 */
```

15.4.2 The `sqrt` functions

The `sqrt` functions compute the positive square root of `x` and return the result. A domain error occurs if the argument is negative.

```
#include <math.h>
float sqrtf(float x); /* C99 */
double sqrt(double x);
long double sqrtl(long double x); /* C99 */
```

15.4.3 The `cbrt` functions

The `cbrt` functions compute the cube root of `x` and return the result.

```
#include <math.h>
float cbrtf(float x); /* C99 */
double cbrt(double x); /* C99 */
long double cbrtl(long double x); /* C99 */
```

15.4.4 The `hypot` functions

The `hypot` functions compute the square root of the sums of the squares of `x` and `y`, without overflow or underflow, and return the result.

```
#include <math.h>
float hypotf(float x, float y); /* C99 */
double hypot(double x, double y); /* C99 */
long double hypotl(long double x, long double y); /* C99 */
```

15.5 Nearest integer, absolute value, and remainder functions

15.5.1 The `ceil` and `floor` functions

The `ceil` functions compute the smallest integral value not less than `x` and return the result; the `floor` functions compute the largest integral value not greater than `x` and return the result.

```
#include <math.h>
float ceilf(float x); /* C99 */
double ceil(double x);
long double ceill(long double x); /* C99 */
float floorf(float x); /* C99 */
double floor(double x);
long double floorl(long double x); /* C99 */
```

15.5.2 The `fabs` functions

The `fabs` functions compute the absolute value of a floating-point number `x` and return the result.

```
#include <math.h>
float fabsf(float x); /* C99 */
double fabs(double x);
long double fabsl(long double x); /* C99 */
```

15.5.3 The `fmod` functions

The `fmod` functions compute the floating-point remainder of `x/y` and return the value `x -` *i* * `y`, for some integer *i* such that, if `y` is nonzero, the result has the same sign as `x` and magnitude less than the magnitude of `y`. If `y` is zero, whether a domain error occurs or the `fmod` functions return zero is implementation-defined.

```
#include <math.h>
float fmodf(float x, float y); /* C99 */
double fmod(double x, double y);
long double fmodl(long double x, long double y); /* C99 */
```

15.5.4 The `nearbyint`, `rint`, `lrint`, and `llrint` functions

The `nearbyint` functions round their argument to an integer value in floating-point format, using the current rounding direction and without raising the "inexact" floating-point exception.

The `rint` functions are similar to the `nearbyint` functions except that they can raise the "inexact" floating-point exception if the result differs in value from the argument.

The `lrint` and `llrint` functions round their arguments to the nearest integer value according to the current rounding direction. If the result is outside the range of values of the return type, the numeric result is undefined and a range error may occur if the magnitude of the argument is too large.

```
#include <math.h>
float nearbyintf(float x); /* C99 */
double nearbyint(double x); /* C99 */
long double nearbyintl(long double x); /* C99 */
float rintf(float x); /* C99 */
double rint(double x); /* C99 */
long double rintl(long double x); /* C99 */
long int lrintf(float x); /* C99 */
long int lrint(double x); /* C99 */
long int lrintl(long double x); /* C99 */
long long int llrintf(float x); /* C99 */
long long int llrint(double x); /* C99 */
long long int llrintl(long double x); /* C99 */
```

15.5.5 The `round`, `lround`, and `llround` functions

The `round` functions round the argument to the nearest integer value in floating-point format, rounding halfway cases away from zero, regardless of the current rounding direction.

The `lround` and `llround` functions round the argument to the nearest integer value, rounding halfway cases away from zero, regardless of the current rounding direction. If the result is outside the range of values of the return type, the numeric result is undefined and a range error may occur if the magnitude of the argument is too large.

```
#include <math.h>
float roundf(float x); /* C99 */
double round(double x); /* C99 */
long double roundl(long double x); /* C99 */
long int lroundf(float x); /* C99 */
long int lround(double x); /* C99 */
long int lroundl(long double x); /* C99 */
long long int llroundf(float x); /* C99 */
long long int llround(double x); /* C99 */
long long int llroundl(long double x); /* C99 */
```

15.5.6 The `trunc` functions

The `trunc` functions round their argument to the integer value in floating-point format that is nearest but no larger in magnitude than the argument.

```
#include <math.h>
float truncf(float x); /* C99 */
double trunc(double x); /* C99 */
long double truncl(long double x); /* C99 */
```

15.5.7 The `remainder` functions

The `remainder` functions compute the remainder `x` REM `y` as defined by IEC 60559. The definition reads, "When $y \neq 0$, the remainder $r = x$ REM y is defined regardless of the rounding mode by the mathematical reduction $r = x - ny$, where n is the integer nearest the exact value of x/y; whenever $|n - x/y| = \frac{1}{2}$, then n is even. Thus, the remainder is always exact. If $r = 0$, its sign shall be that of x." This definition is applicable for all implementations.

```
#include <math.h>
float remainderf(float x, float y); /* C99 */
double remainder(double x, double y); /* C99 */
long double remainderl(long double x, long double y); /* C99 */
```

15.5.8 The `remquo` functions

The `remquo` functions return the same remainder as the `remainder` functions. In the object pointed to by `quo`, they store a value whose sign is the sign of `x/y` and whose magnitude is congruent modulo 2^n to the magnitude of the integral quotient of `x/y`, where n is an implementation-defined integer greater than or equal to 3.

```
#include <math.h>
float remquof(float x, float y, int *quo); /* C99 */
double remquo(double x, double y, int *quo); /* C99 */
long double remquol(long double x, long double y, int *quo); /* C99 */
```

15.6 Error and gamma functions

The `erf` functions compute the error function of the argument $(2/(\pi^{\frac{1}{2}})\ \int_0^x e^{-t^2}\ dt)$; the `erfc` functions compute the complimentary error function of the argument (that is, 1 - erf x). For the `erfc` functions, a range error may occur if the argument is too large.

The `lgamma` functions compute the natural logarithm of the absolute value of the gamma of the argument (that is, $\log_e|\Gamma(x)|$). A range error may occur if the argument is a negative integer or zero.

The `tgamma` functions compute the gamma of the argument (that is, $\Gamma(x)$). A domain error occurs if the argument is a negative integer or if the result cannot be represented when the argument is zero. A range error may occur.

```
#include <math.h>
float erff(float x); /* C99 */
double erf(double x); /* C99 */
long double erfl(long double x); /* C99 */
float erfcf(float x); /* C99 */
double erfc(double x); /* C99 */
long double erfcl(long double x); /* C99 */
float lgammaf(float x); /* C99 */
double lgamma(double x); /* C99 */
long double lgammal(long double x); /* C99 */
float tgammaf(float x); /* C99 */
double tgamma(double x); /* C99 */
long double tgammal(long double x); /* C99 */
```

15.7 Further reading

w:circular shift[2]

pl:C/Zaawansowane operacje matematyczne[3]

2 http://en.wikipedia.org/wiki/circular%20shift
3 http://pl.wikibooks.org/wiki/C%2FZaawansowane%20operacje%20matematyczne

16 Control

Very few programs follow exactly one control path and have each instruction stated explicitly. In order to program effectively, it is necessary to understand how one can alter the steps taken by a program due to user input or other conditions, how some steps can be executed many times with few lines of code, and how programs can appear to demonstrate a rudimentary grasp of logic. C constructs known as conditionals and loops grant this power.

From this point forward, it is necessary to understand what is usually meant by the word *block*. A block is a group of code statements that are associated and intended to be executed as a unit. In C, the beginning of a block of code is denoted with { (left curly), and the end of a block is denoted with }. It is not necessary to place a semicolon after the end of a block. Blocks can be empty, as in {}. Blocks can also be nested; i.e. there can be blocks of code within larger blocks.

16.1 Conditionals

There is likely no meaningful program written in which a computer does not demonstrate basic decision-making skills. It can actually be argued that there is no meaningful human activity in which some sort of decision-making, instinctual or otherwise, does not take place. For example, when driving a car and approaching a traffic light, one does not think, "I will continue driving through the intersection." Rather, one thinks, "I will stop if the light is red, go if the light is green, and if yellow go only if I am traveling at a certain speed a certain distance from the intersection." These kinds of processes can be simulated in C using conditionals.

A conditional is a statement that instructs the computer to execute a certain block of code or alter certain data only if a specific condition has been met. The most common conditional is the If-Else statement, with conditional expressions and Switch-Case statements typically used as more shorthanded methods.

Before one can understand conditional statements, it is first necessary to understand how C expresses logical relations. C treats logic as being arithmetic. The value 0 (zero) represents false, and ***all other values*** represent true. If you chose some particular value to represent true and then compare values against it, sooner or later your code will fail when your assumed value (often 1) turns out to be incorrect. Code written by people uncomfortable with the C language can often be identified by the usage of #define to make a "TRUE" value. [1]

1 C FAQ ^{`http://www.c-faq.com/bool/bool2.html`}

Because logic is arithmetic in C, arithmetic operators and logical operators are one and the same. Nevertheless, there are a number of operators that are typically associated with logic:

16.1.1 Relational and Equivalence Expressions:

a < b

1 if **a** is less than **b**, 0 otherwise.

a > b

1 if **a** is greater than **b**, 0 otherwise.

a <= b

1 if **a** is less than or equal to **b**, 0 otherwise.

a >= b

1 if **a** is greater than or equal to **b**, 0 otherwise.

a == b

1 if **a** is equal to **b**, 0 otherwise.

a != b

1 if **a** is not equal to **b**, 0 otherwise

New programmers should take special note of the fact that the "equal to" operator is ==, not =. This is the cause of numerous coding mistakes and is often a difficult-to-find bug, as the expression `(a = b)` sets `a` equal to `b` and subsequently evaluates to `b`; but the expression `(a == b)`, which is usually intended, checks if `a` is equal to `b`. It needs to be pointed out that, if you confuse = with ==, your mistake will often not be brought to your attention by the compiler. A statement such as `if ( c = 20) {}` is considered perfectly valid by the language, but will always assign 20 to `c` and evaluate as true. A simple technique to avoid this kind of bug (in many, not all cases) is to put the constant first. This will cause the compiler to issue an error, if == got misspelled with =.

Note that C does not have a dedicated boolean type as many other languages do. 0 means false and anything else true. So the following are equivalent:

```
if (foo()) {
  //do something
}
```

and

```
if (foo() != 0) {
  //do something
}
```

Often `#define TRUE 1` and `#define FALSE 0` are used to work around the lack of a boolean type. This is bad practice, since it makes assumptions that do not hold. It is a better idea to indicate what you are actually expecting as a result from a function call, as there are many different ways of indicating error conditions, depending on the situation.

```
if (strstr("foo", bar) >= 0) {
  //bar contains "foo"
}
```

Here, `strstr` returns the index where the substring foo is found and -1 if it was not found. Note that this would fail with the `TRUE` definition mentioned in the previous paragraph. It would also not produce the expected results if we omitted the `>= 0`.

One other thing to note is that the relational expressions do not evaluate as they would in mathematical texts. That is, an expression `myMin < value < myMax` does not evaluate as you probably think it might. Mathematically, this would test whether or not *value* is between *myMin* and *myMax*. But in C, what happens is that *value* is first compared with *myMin*. This produces either a 0 or a 1. It is this value that is compared against myMax. Example:

```
int value = 20;
/* ... */
if ( 0 < value < 10) { // don't do this! it always evaluates to "true"!
   /* do some stuff */
}
```

Because *value* is greater than 0, the first comparison produces a value of 1. Now 1 is compared to be less than 10, which is true, so the statements in the if are executed. This probably is not what the programmer expected. The appropriate code would be

```
int value = 20;
/* ... */
if ( 0 < value && value < 10) {   // the && means "and"
 /* do some stuff */
}
```

16.1.2 Logical Expressions

a || b

when EITHER **a** or **b** is true (or both), the result is 1, otherwise the result is 0.

a && b

when BOTH **a** and **b** are true, the result is 1, otherwise the result is 0.

!a

when **a** is true, the result is 0, when **a** is 0, the result is 1.

Here's an example of a larger logical expression. In the statement:

```
    e = ((a && b) || (c > d));
```

e is set equal to 1 if a and b are non-zero, or if c is greater than d. In all other cases, e is set to 0.

C uses short circuit evaluation of logical expressions. That is to say, once it is able to determine the truth of a logical expression, it does no further evaluation. This is often useful as in the following:

```
int myArray[12];
....
if ( i < 12 && myArray[i] > 3) {
....
```

In the snippet of code, the comparison of i with 12 is done first. If it evaluates to 0 (false), **i** would be out of bounds as an index to **myArray**. In this case, the program never attempts to access **myArray[i]** since the truth of the expression is known to be false. Hence we need not worry here about trying to access an out-of-bounds array element if it is already known that i is greater than or equal to zero. A similar thing happens with expressions involving the or || operator.

```
while( doThis() || doThat()) ...
```

doThat() is never called if doThis() returns a non-zero (true) value.

16.1.3 Bitwise Boolean Expressions

The bitwise operators work bit by bit on the operands. The operands must be of integral type (one of the types used for integers). The six bitwise operators are & (AND), | (OR), ^ (exclusive OR, commonly called XOR), ~ (NOT, which changes 1 to 0 and 0 to 1), << (shift left), and >> (shift right). The negation operator is a unary operator which precedes the operand. The others are binary operators which lie between the two operands. The precedence of these operators is lower than that of the relational and equivalence operators; it is often required to parenthesize expressions involving bitwise operators.

For this section, recall that a number starting with **0x** is hexadecimal, or hex for short. Unlike the normal decimal system using powers of 10 and digits 0123456789, hex uses powers of 16 and digits 0123456789abcdef. Hexadecimal is commonly used in C programs because a programmer can quickly convert it to or from binary (powers of 2 and digits 01). C does not directly support binary notation, which would be really verbose anyway.

a & b

bitwise boolean and of **a** and **b**

0xc & 0xa produces the value 0x8 (in binary, 1100 & 1010 produces 1000)

a | b

bitwise boolean or of **a** and **b**

0xc | 0xa produces the value 0xe (in binary, 1100 | 1010 produces 1110)

a ^ b

bitwise xor of **a** and **b**

0xc ^ 0xa produces the value 0x6 (in binary, 1100 ^ 1010 produces 0110)

~a

bitwise complement of **a**.

~0xc produces the value -1-0xc (in binary, ~1100 produces ...11110011 where "..." may be many more 1 bits)

a << b

shift **a** left by **b** (multiply a by 2^b)

0xc << 1 produces the value 0x18 (in binary, 1100 << 1 produces the value 11000)

a >> b

shift **a** right by **b** (divide a by 2^b)

0xc >> 1 produces the value 0x6 (in binary, 1100 >> 1 produces the value 110)

16.1.4 The If-Else statement

If-Else provides a way to instruct the computer to execute a block of code only if certain conditions have been met. The syntax of an If-Else construct is:

```
if (/* condition goes here */) {
   /* if the condition is non-zero (true), this code will execute */
} else {
   /* if the condition is 0 (false), this code will execute */
}
```

The first block of code executes if the condition in parentheses directly after the *if* evaluates to non-zero (true); otherwise, the second block executes.

The *else* and following block of code are completely optional. If there is no need to execute code if a condition is not true, leave it out.

Also, keep in mind that an *if* can directly follow an *else* statement. While this can occasionally be useful, chaining more than two or three if-elses in this fashion is considered bad programming practice. We can get around this with the Switch-Case construct described later.

Two other general syntax notes need to be made that you will also see in other control constructs: First, note that there is no semicolon after *if* or *else*. There could be, but the block (code enclosed in { and }) takes the place of that. Second, if you only intend to execute one statement as a result of an *if* or *else*, curly braces are not needed. However, many programmers believe that inserting curly braces anyway in this case is good coding practice.

The following code sets a variable c equal to the greater of two variables a and b, or 0 if a and b are equal.

```
if(a > b) {
   c = a;
} else if(b > a) {
    c = b;
 } else {
    c = 0;
 }
```

Consider this question: why can't you just forget about *else* and write the code like:

```
if(a > b) {
   c = a;
}

if(a < b) {
  c = b;
}

if(a == b) {
  c = 0;
}
```

There are several answers to this. Most importantly, if your conditionals are not mutually exclusive, *two* cases could execute instead of only one. If the code was different and the value of a or b changes somehow (e.g.: you reset the lesser of a and b to 0 after the comparison) during one of the blocks? You could end up with multiple *if* statements being invoked, which is not your intent. Also, evaluating *if* conditionals takes processor time. If you use *else* to handle these situations, in the case above assuming (a > b) is non-zero (true), the program is spared the expense of evaluating additional *if* statements. The bottom line is that it is usually best to insert an *else* clause for all cases in which a conditional will not evaluate to non-zero (true).

The conditional expression

A conditional expression is a way to set values conditionally in a more shorthand fashion than If-Else. The syntax is:

```
(/* logical expression goes here */) ? (/* if non-zero (true) */) : (/* if 0
(false) */)
```

The logical expression is evaluated. If it is non-zero (true), the overall conditional expression evaluates to the expression placed between the ? and :, otherwise, it evaluates to the expression after the :. Therefore, the above example (changing its function slightly such that c is set to b when a and b are equal) becomes:

```
c = (a > b) ? a : b;
```

Conditional expressions can sometimes clarify the intent of the code. Nesting the conditional operator should usually be avoided. It's best to use conditional expressions only when the expressions for a and b are simple. Also, contrary to a common beginner belief, conditional expressions do not make for faster code. As tempting as it is to assume that fewer lines of code result in faster execution times, there is no such correlation.

16.1.5 The Switch-Case statement

Say you write a program where the user inputs a number 1-5 (corresponding to student grades, A(represented as 1)-D(4) and F(5)), stores it in a variable **grade** and the program responds by printing to the screen the associated letter grade. If you implemented this using If-Else, your code would look something like this:

```
if(grade == 1) {
   printf("A\n");
} else if(grade == 2) {
   printf("B\n");
} else if /* etc. etc. */
```

Having a long chain of if-else-if-else-if-else can be a pain, both for the programmer and anyone reading the code. Fortunately, there's a solution: the Switch-Case construct, of which the basic syntax is:

```
switch(/* integer or enum goes here */) {
  case /* potential value of the aforementioned int or enum */:
     /* code */
  case /* a different potential value */:
     /* different code */
  /* insert additional cases as needed */
  default:
     /* more code */
}
```

The Switch-Case construct takes a variable, usually an int or an enum, placed after *switch*, and compares it to the value following the *case* keyword. If the variable is equal to the value specified after *case*, the construct "activates", or begins executing the code after the case statement. Once the construct has "activated", there will be no further evaluation of *case*s.

Switch-Case is syntactically "weird" in that no braces are required for code associated with a *case*.

Very important: Typically, the last statement for each case is a break statement. This causes program execution to jump to the statement following the closing bracket of the switch statement, which is what one would normally want to happen. However if the break statement is omitted, program execution continues with the first line of the next case, if any. This is called a *fall-through*. When a programmer desires this action, a comment should be placed at the end of the block of statements indicating the desire to fall through. Otherwise another programmer maintaining the code could consider the omission of the 'break' to be an error, and inadvertently 'correct' the problem. Here's an example:

```
switch ( someVariable ) {
case 1:
   printf("This code handles case 1\n");
   break;
case 2:
   printf("This prints when someVariable is 2, along with...\n");
   /* FALL THROUGH */
case 3:
   printf("This prints when someVariable is either 2 or 3.\n" );
   break;
}
```

If a *default* case is specified, the associated statements are executed if none of the other cases match. A *default* case is optional. Here's a switch statement that corresponds to the sequence of if - else if statements above.

Back to our example above. Here's what it would look like as Switch-Case:

```
switch (grade) {
case 1:
   printf("A\n");
   break;
case 2:
   printf("B\n");
   break;
case 3:
   printf("C\n");
   break;
case 4:
   printf("D\n");
   break;
default:
   printf("F\n");
   break;
}
```

A set of statements to execute can be grouped with more than one value of the variable as in the following example. (the fall-through comment is not necessary here because the intended behavior is obvious)

```
switch (something) {
case 2:
case 3:
case 4:
   /* some statements to execute for 2, 3 or 4 */
   break;
case 1:
default:
   /* some statements to execute for 1 or other than 2,3,and 4 */
   break;
}
```

Switch-Case constructs are particularly useful when used in conjunction with user defined *enum* data types. Some compilers are capable of warning about an unhandled enum value, which may be helpful for avoiding bugs.

16.2 Loops

Often in computer programming, it is necessary to perform a certain action a certain number of times or until a certain condition is met. It is impractical and tedious to simply type a certain statement or group of statements a large number of times, not to mention that this approach is too inflexible and unintuitive to be counted on to stop when a certain event has happened. As a real-world analogy, someone asks a dishwasher at a restaurant what he did all night. He will respond, "I washed dishes all night long." He is not likely to respond, "I washed a dish, then washed a dish, then washed a dish, then...". The constructs that enable computers to perform certain repetitive tasks are called loops.

16.2.1 While loops

A while loop is the most basic type of loop. It will run as long as the condition is non-zero (true). For example, if you try the following, the program will appear to lock up and you will have to manually close the program down. A situation where the conditions for exiting the loop will never become true is called an infinite loop.

```
int a=1;
while(42) {
   a = a*2;
}
```

Here is another example of a while loop. It prints out all the powers of two less than 100.

```
int a=1;
while(a<100) {
   printf("a is %d \n",a);
   a = a*2;
}
```

The flow of all loops can also be controlled by **break** and **continue** statements. A break statement will immediately exit the enclosing loop. A continue statement will skip the remainder of the block and start at the controlling conditional statement again. For example:

```
int a=1;
while (42) { // loops until the break statement in the loop is executed
   printf("a is %d ",a);
   a = a*2;
   if(a>100) {
       break;
   } else if(a==64) {
       continue;  // Immediately restarts at while, skips next step
   }
   printf("a is not 64\n");
}
```

In this example, the computer prints the value of a as usual, and prints a notice that a is not 64 (unless it was skipped by the continue statement).

Similar to If above, braces for the block of code associated with a While loop can be omitted if the code consists of only one statement, for example:

```
int a=1;
while(a < 100) a = a*2;
```

This will merely increase a until a is not less than 100.

When the computer reaches the end of the while loop, it always goes back to the while statement at the top of the loop, where it re-evaluates the controlling condition. If that condition is "true" at that instant -- even if it was temporarily 0 for a few statements inside the loop -- then the computer begins executing the statements inside the loop again; otherwise the computer exits the loop. The computer does not "continuously check" the controlling condition of a while loop during the execution of that loop. It only "peeks" at the controlling condition each time it reaches the `while` at the top of the loop.

It is very important to note, once the controlling condition of a While loop becomes 0 (false), the loop will not terminate until the block of code is finished and it is time to reevaluate the conditional. If you need to terminate a While loop immediately upon reaching a certain condition, consider using **break**.

A common idiom is to write:

```
int i = 5;
while(i--) {
   printf("java and c# can't do this\n");
}
```

This executes the code in the while loop 5 times, with i having values that range from 4 down to 0 (inside the loop). Conveniently, these are the values needed to access every item of an array containing 5 elements.

16.2.2 For loops

For loops generally look something like this:

```
for(initialization; test; increment) {
   /* code */
}
```

The *initialization* statement is executed exactly once - before the first evaluation of the *test* condition. Typically, it is used to assign an initial value to some variable, although this is not strictly necessary. The *initialization* statement can also be used to declare and initialize variables used in the loop.

The *test* expression is evaluated each time before the code in the *for* loop executes. If this expression evaluates as 0 (false) when it is checked (i.e. if the expression is not true), the loop is not (re)entered and execution continues normally at the code immediately following the FOR-loop. If the expression is non-zero (true), the code within the braces of the loop is executed.

After each iteration of the loop, the *increment* statement is executed. This often is used to increment the loop index for the loop, the variable initialized in the initialization expression and tested in the test expression. Following this statement execution, control returns to the top of the loop, where the *test* action occurs. If a *continue* statement is executed within the *for* loop, the increment statement would be the next one executed.

Each of these parts of the for statement is optional and may be omitted. Because of the free-form nature of the for statement, some fairly fancy things can be done with it. Often a for loop is used to loop through items in an array, processing each item at a time.

```
int  myArray[12];
int ix;
for (ix = 0; ix<12; ix++) {
   myArray[ix] = 5 * ix + 3;
}
```

The above for loop initializes each of the 12 elements of myArray. The loop index can start from any value. In the following case it starts from 1.

```
for(ix = 1; ix <= 10; ix++) {
   printf("%d ", ix);
}
```

which will print

```
1 2 3 4 5 6 7 8 9 10
```

You will most often use loop indexes that start from 0, since arrays are indexed at zero, but you will sometimes use other values to initialize a loop index as well.

The *increment* action can do other things, such as *decrement.* So this kind of loop is common:

```
for (i = 5; i > 0; i--) {
   printf("%d ",i);
}
```

which yields

```
5 4 3 2 1
```

Here's an example where the test condition is simply a variable. If the variable has a value of 0 or NULL, the loop exits, otherwise the statements in the body of the loop are executed.

```
for (t = list_head; t; t = NextItem(t) ) {
  /*body of loop */
}
```

A WHILE loop can be used to do the same thing as a FOR loop, however a FOR loop is a more condensed way to perform a set number of repetitions since all of the necessary information is in a one line statement.

A FOR loop can also be given no conditions, for example:

```
for(;;) {
  /* block of statements */
}
```

This is called an infinite loop since it will loop forever unless there is a break statement within the statements of the for loop. The empty test condition effectively evaluates as true.

It is also common to use the comma operator in for loops to execute multiple statements.

```
int i, j, n = 10;
for(i = 0, j = 0; i <= n; i++,j+=2) {
     printf("i = %d , j = %d \n",i,j);
}
```

Special care should be taken when designing or refactoring the conditional part, especially whether using < or <= , whether start and stop should be corrected by 1, and in case of prefix- and postfix-notations. (On a 100 yards promenade with a tree every 10 yards there are 11 trees.)

```
int i, n = 10;
for(i = 0; i < n; i++) printf("%d ",i); // processed n times => 0 1 2 3 ...
(n-1)
printf("\n");
for(i = 0; i <= n; i++) printf("%d ",i); // processed (n+1) times => 0 1 2 3
... n
printf("\n");
for(i = n; i--;) printf("%d ",i); // processed n times => (n-1) ...3 2 1 0
printf("\n");
for(i = n; --i;) printf("%d ",i); // processed (n-1) times => (n-1) ...4 3 2 1
printf("\n");
```

16.2.3 Do-While loops

A DO-WHILE loop is a post-check while loop, which means that it checks the condition after each run. As a result, even if the condition is zero (false), it will run at least once. It follows the form of:

```
do {
   /* do stuff */
} while (condition);
```

Note the terminating semicolon. This is required for correct syntax. Since this is also a type of while loop, **break** and **continue** statements within the loop function accordingly. A **continue** statement causes a jump to the test of the condition and a *break* statement exits the loop.

It is worth noting that Do-While and While are functionally almost identical, with one important difference: Do-While loops are always guaranteed to execute at least once, but While loops will not execute at all if their condition is 0 (false) on the first evaluation.

16.3 One last thing: goto

goto is a very simple and traditional control mechanism. It is a statement used to immediately and unconditionally jump to another line of code. To use goto, you must place a label at a point in your program. A label consists of a name followed by a colon (:) on a line by itself. Then, you can type "goto *label*;" at the desired point in your program. The code will then continue executing beginning with *label*. This looks like:

```
MyLabel:
   /* some code */
goto MyLabel;
```

The ability to transfer the flow of control enabled by gotos is so powerful that, in addition to the simple if, all other control constructs can be written using gotos instead. Here, we can let "S" and "T" be any arbitrary statements:

```
if (''cond'') {
  S;
} else {
  T;
  }
  /* ... */
```

The same statement could be accomplished using two gotos and two labels:

```
if (''cond'') goto Label1;
  T;
  goto Label2;
Label1:
  S;
Label2:
  /* ... */
```

Here, the first goto is conditional on the value of "cond". The second goto is unconditional. We can perform the same translation on a loop:

```
  while (''cond1'') {
     S;
     if (''cond2'') break;
     T;
  }
  /* ... */
```

Which can be written as:

```
Start:
  if (!''cond1'') goto End;
  S;
  if (''cond2'') goto End;
  T;
  goto Start;
End:
  /* ... */
```

As these cases demonstrate, often the structure of what your program is doing can usually be expressed without using gotos. Undisciplined use of gotos can create unreadable, unmaintainable code when more idiomatic alternatives (such as if-elses, or for loops) can better express your structure. Theoretically, the goto construct does not ever *have* to be used, but there are cases when it can increase readability, avoid code duplication, or make control variables unnecessary. You should consider first mastering the idiomatic solutions, and use goto only when necessary. Keep in mind that many, if not most, C style guidelines *strictly forbid* use of **goto**, with the only common exceptions being the following examples.

One use of goto is to break out of a deeply nested loop. Since **break** will not work (it can only escape one loop), **goto** can be used to jump completely outside the loop. Breaking outside of deeply nested loops without the use of the goto is always possible, but often involves the creation and testing of extra variables that may make the resulting code far less readable than it would be with **goto**. The use of **goto** makes it easy to undo actions in an orderly fashion, typically to avoid failing to free memory that had been allocated.

Another accepted use is the creation of a state machine. This is a fairly advanced topic though, and not commonly needed.

16.4 Examples

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
        int years;

        printf("Enter your age in years : ");
        fflush(stdout);
        errno = 0;
        if(scanf("%d", &years) != 1 || errno)
                return EXIT_FAILURE;
        printf("Your age in days is %d\n", years * 365);
        return 0;
}
```

16.5 Further reading

de:C-Programmierung: Kontrollstrukturen[2] et:Programmeerimiskeel C/Keelestruktuurid[3] pl:C/Instrukcje sterujące[4] pt:Programar em C/Controle de fluxo[5] fi:C/Ohjausrakenteet[6]

2 http://de.wikibooks.org/wiki/C-Programmierung%3A%20Kontrollstrukturen
3 http://et.wikibooks.org/wiki/Programmeerimiskeel%20C%2FKeelestruktuurid
4 http://pl.wikibooks.org/wiki/C%2FInstrukcje%20steruj%C4%85ce
5 http://pt.wikibooks.org/wiki/Programar%20em%20C%2FControle%20de%20fluxo
6 http://fi.wikibooks.org/wiki/C%2FOhjausrakenteet

17 Procedures and functions

In C programming, all executable code resides within a **function**. A function is a named block of code that performs a task and then returns control to a caller. Note that other programming languages may distinguish between a "function", "subroutine", "subprogram", "procedure", or "method" -- in C, these are all functions.

A function is often executed (called) several times, from several different places, during a single execution of the program. After finishing a subroutine, the program will branch back (return) to the point after the call.

Functions are a powerful programming tool.

As a basic example, suppose you are writing code to print out the first 5 squares of numbers, do some intermediate processing, then print the first 5 squares again. We could write it like this:

```
#include <stdio.h>

int main(void)
{
  int i;
  for(i=1; i <= 5; i++)
  {
     printf("%d ", i*i);
  }
  for(i=1; i <= 5; i++)
  {
     printf("%d ", i*i);
  }
  return 0;
}
```

We have to write the same loop twice. We may want to somehow put this code in a separate place and simply jump to this code when we want to use it. This would look like:

```
#include <stdio.h>

void Print_Squares(void)
{
  int i;
  for(i=1; i <=5; i++)
  {
    printf("%d ", i*i);
  }
}

int main(void)
{
  Print_Squares();
  Print_Squares();
  return 0;
}
```

This is precisely what functions are for.

17.1 More on functions

A function is like a black box. It takes in input, does something with it, then spits out an answer.

Note that a function may not take any inputs at all, or it may not return anything at all. In the above example, if we were to make a function of that loop, we may not need any inputs, and we aren't returning anything at all (Text output doesn't count - when we speak of *returning* we mean to say meaningful data that the program can use).

We have some terminology to refer to functions:

- A function, call it *f*, that uses another function *g*, is said to *call g.* For example, *f* calls *g* to print the squares of ten numbers.
- A function's inputs are known as its *arguments*
- A function *g* that gives some kind of answer back to *f* is said to *return* that answer. For example, *g* returns the sum of its arguments.

17.2 Writing functions in C

It's always good to learn by example. Let's write a function that will return the square of a number.

```
int square(int x)
{
   int square_of_x;
   square_of_x = x * x;
   return square_of_x;
}
```

To understand how to write such a function like this, it may help to look at what this function does as a whole. It takes in an `int`, x, and squares it, storing it in the variable square_of_x. Now this value is returned.

The first int at the beginning of the function declaration is the type of data that the function returns. In this case when we square an integer we get an integer, and we are returning this integer, and so we write `int` as the return type.

Next is the name of the function. It is good practice to use meaningful and descriptive names for functions you may write. It may help to name the function after what it is written to do. In this case we name the function "square", because that's what it does - it squares a number.

Next is the function's first and only argument, an `int`, which will be referred to in the function as x. This is the function's *input.*

In between the braces is the actual guts of the function. It declares an integer variable called square_of_x that will be used to hold the value of the square of x. Note that the variable square_of_x can **only** be used within this function, and not outside. We'll learn more about this sort of thing later, and we will see that this property is very useful.

We then assign x multiplied by x, or x squared, to the variable square_of_x, which is what this function is all about. Following this is a `return` statement. We want to return the value of the square of x, so we must say that this function returns the contents of the variable square_of_x.

Our brace to close, and we have finished the declaration.

Written in a more concise manner, this code performs exactly the same function as the above:

```
int square(int x)
{
   return x * x;
}
```

Note this should look familiar - you have been writing functions already, in fact - main is a function that is always written.

17.2.1 In general

In general, if we want to declare a function, we write

```
type name(type1 arg1, type2 arg2, ...)
{
  /* code */
}
```

We've previously said that a function can take no arguments, or can return nothing, or both. What do we write if we want the function to return nothing? We use C's `void` keyword. `void` basically means "nothing" - so if we want to write a function that returns nothing, for example, we write

```
void sayhello(int number_of_times)
{
  int i;
  for(i=1; i <= number_of_times; i++) {
     printf("Hello!\n'''");
 }
}
```

Notice that there is no `return` statement in the function above. Since there's none, we write `void` as the return type. (Actually, one can use the `return` keyword in a procedure to return to the caller before the end of the procedure, but one cannot return a value as if it were a function.)

What about a function that takes no arguments? If we want to do this, we can write for example

```
float calculate_number(void)
{
  float to_return=1;
  int i;
  for(i=0; i < 100; i++) {
     to_return += 1;
     to_return = 1/to_return;
  }
  return to_return;
}
```

Notice this function doesn't take any inputs, but merely returns a number calculated by this function.

Naturally, you can combine both void return and void in arguments together to get a valid function, also.

17.2.2 Recursion

Here's a simple function that does an infinite loop. It prints a line and calls itself, which again prints a line and calls itself again, and this continues until the stack overflows and the program crashes. A function calling itself is called recursion, and normally you will have a conditional that would stop the recursion after a small, finite number of steps.

```
      // don't run this!
void infinite_recursion()
{
    printf("Infinite loop!\n");
    infinite_recursion();
}
```

A simple check can be done like this. Note that ++depth is used so the increment will take place before the value is passed into the function. Alternatively you can increment on a separate line before the recursion call. If you say print_me(3,0); the function will print the line Recursion 3 times.

```
void print_me(int j, int depth)
{
   if(depth < j) {
       printf("Recursion! depth = %d j = %d\n",depth,j); //j keeps its value
       print_me(j, ++depth);
   }
}
```

Recursion is most often used for jobs such as directory tree scans, seeking for the end of a linked list, parsing a tree structure in a database and factorising numbers (and finding primes) among other things.

17.2.3 Static functions

If a function is to be called only from within the file in which it is declared, it is appropriate to declare it as a static function. When a function is declared static, the compiler will now compile to an object file in a way that prevents the function from being called from code in other files. Example:

```
static int compare( int a, int b )
{
    return (a+4 < b)? a : b;
}
```

17.3 Using C functions

We can now *write* functions, but how do we use them? When we write main, we place the function outside the braces that encompass main.

When we want to use that function, say, using our `calculate_number` function above, we can write something like

```
float f;
f = calculate_number();
```

If a function takes in arguments, we can write something like

```
int square_of_10;
square_of_10 = square(10);
```

If a function doesn't return anything, we can just say

```
say_hello();
```

since we don't need a variable to catch its return value.

17.4 Functions from the C Standard Library

While the C language doesn't itself contain functions, it is usually linked with the C Standard Library. To use this library, you need to add an #include directive at the top of the C file, which may be one of the following:

- `<assert.h>`[1]
- `<ctype.h>`[2]
- `<errno.h>`[3]
- `<float.h>`[4]
- `<limits.h>`[5]
- `<locale.h>`[6]
- `<math.h>`[7]
- `<setjmp.h>`[8]
- `<signal.h>`[9]
- `<stdarg.h>`[10]
- `<stddef.h>`[11]
- `<stdio.h>`[12]
- `<stdlib.h>`[13]
- `<string.h>`[14]
- `<time.h>`[15]
- `<complex.h>`[16]

The functions available are:

<assert.h>	**<limits.h>**	**<signal.h>**	**<stdlib.h>**

1 http://en.wikipedia.org/wiki/Assert.h
2 http://en.wikipedia.org/wiki/Ctype.h
3 http://en.wikipedia.org/wiki/Errno.h
4 http://en.wikipedia.org/wiki/Float.h
5 http://en.wikipedia.org/wiki/Limits.h
6 http://en.wikipedia.org/wiki/Locale.h
7 http://en.wikipedia.org/wiki/Math.h
8 http://en.wikipedia.org/wiki/Setjmp.h
9 http://en.wikipedia.org/wiki/Signal.h
10 http://en.wikipedia.org/wiki/Stdarg.h
11 http://en.wikipedia.org/wiki/Stddef.h
12 http://en.wikipedia.org/wiki/Stdio.h
13 http://en.wikipedia.org/wiki/Stdlib.h
14 http://en.wikipedia.org/wiki/String.h
15 http://en.wikipedia.org/wiki/Time.h
16 http://en.wikipedia.org/wiki/Complex.h

`<assert.h>`	`<limits.h>`	`<signal.h>`	`<stdlib.h>`
• assert(int)	• (constants only)	• int raise(int sig). This • void* signal(int sig, void (*func)(int))	• atof(char*), atoi(char*), atol(char*) • strtod(char * str, char ** endptr), strtol(char *str, char **endptr), strtoul(char *str, char **endptr) • rand(), srand() • malloc(size_t), calloc (size_t elements, size_t elementSize), realloc(void*, int) • free (void*) • exit(int), abort() • atexit(void (*func)()) • getenv • system • qsort(void *, size_t number, size_t size, int (*sortfunc)(void*, void*)) • abs, labs • div, ldiv
`<ctype.h>`	`<locale.h>`	`<stdarg.h>`	`<string.h>`

`<assert.h>`	`<limits.h>`	`<signal.h>`	`<stdlib.h>`
• isalnum, isalpha, isblank • iscntrl, isdigit, isgraph • islower, isprint, ispunct • isspace, isupper, isxdigit • tolower, toupper	• struct lconv* localeconv(void); • char* setlocale(int, const char*);	• va_start (va_list, ap) • va_arg (ap, (type)) • va_end (ap) • va_copy (va_list, va_list)	• memcpy, memmove • memchr, memcmp, memset • strcat, strncat, strchr, strrchr • strcmp, strncmp, strccoll • strcpy, strncpy • strerror • strlen • strspn, strcspn • strpbrk • strstr • strtok • strxfrm
errno.h	**math.h**	**stddef.h**	**time.h**

`<assert.h>`	`<limits.h>`	`<signal.h>`	`<stdlib.h>`
• (errno)	• sin, cos, tan • asin, acos, atan, atan2 • sinh, cosh, tanh • ceil • exp • fabs • floor • fmod • frexp • ldexp • log, log10 • modf • pow • sqrt	• offsetof macro	• asctime (struct tm* tmptr) • clock_t clock() • char* ctime(const time_t* timer) • double diff-time(time_t timer2, time_t timer1) • struct tm* gmtime(const time_t* timer) • struct tm* gm-time_r(const time_t* timer, struct tm* re-sult) • struct tm* lo-caltime(const time_t* timer) • time_t mk-time(struct tm* ptm) • time_t time(time_t* timer) • char * strp-time(const char* buf, const char* format, struct tm* tptr) • time_t timegm(struct tm *broken-time)
float.h	**setjmp.h**	**stdio.h**	

`<assert.h>`	`<limits.h>`	`<signal.h>`	`<stdlib.h>`
• (constants)	• int setjmp(jmp_buf env) • void longjmp(jmp_buf env, int value)	• fclose • fopen, freopen • remove • rename • rewind • tmpfile • clearerr • feof, ferror • fflush • fgetpos, fsetpos • fgetc, fputc • fgets, fputs • ftell, fseek	• fread, fwrite • getc, putc • getchar, putchar, fputchar • gets, puts • printf, vprintf • fprintf, vfprintf • sprintf, snprintf, vsprintf, vsnprintf • perror • scanf, vscanf • fscanf, vfscanf • sscanf, vsscanf • setbuf, setvbuf • tmpnam • ungetc

- /printf/[17]
- full list[18]

17.5 Variable-length argument lists

Functions with variable-length argument lists are functions that can take a varying number of arguments. An example in the C standard library is the `printf` function, which can take any number of arguments depending on how the programmer wants to use it.

C programmers rarely find the need to write new functions with variable-length arguments. If they want to pass a bunch of things to a function, they typically define a structure to hold all those things -- perhaps a linked list, or an array -- and call that function with the data in the arguments.

However, you may occasionally find the need to write a new function that supports a variable-length argument list. To create a function that can accept a variable-length argument list, you must first include the standard library header `stdarg.h`. Next, declare the function as you would normally. Next, add as the last argument an ellipsis ("..."). This indicates to the compiler that a variable list of arguments is to follow. For example, the following function declaration is for a function that returns the average of a list of numbers:

```
float average (int n_args, ...);
```

17 http://en.wikibooks.org/wiki/%2Fprintf%2F
18 http://www.utas.edu.au/infosys/info/documentation/C/CStdLib.html#ctype.h

Note that because of the way variable-length arguments work, we must somehow, in the arguments, specify the number of elements in the variable-length part of the arguments. In the `average` function here, it's done through an argument called `n_args.` In the `printf` function, it's done with the format codes that you specify in that first string in the arguments you provide.

Now that the function has been declared as using variable-length arguments, we must next write the code that does the actual work in the function. To access the numbers stored in the variable-length argument list for our `average` function, we must first declare a variable for the list itself:

```
  va_list myList;
```

The `va_list` type is a type declared in the `stdarg.h` header that basically allows you to keep track of your list. To start actually using `myList`, however, we must first assign it a value. After all, simply declaring it by itself wouldn't do anything. To do this, we must call `va_start`, which is actually a macro defined in `stdarg.h.` In the arguments to `va_start`, you must provide the `va_list` variable you plan on using, as well as the name of the last variable appearing before the ellipsis in your function declaration:

```
#include <stdarg.h>
float average (int n_args, ...)
{
    va_list myList;
    va_start (myList, n_args);
    va_end (myList);
}
```

Now that `myList` has been prepped for usage, we can finally start accessing the variables stored in it. To do so, use the `va_arg` macro, which pops off the next argument on the list. In the arguments to `va_arg`, provide the `va_list` variable you're using, as well as the primitive data type (e.g. `int`, `char`) that the variable you're accessing should be:

```
#include <stdarg.h>
float average (int n_args, ...)
{
    va_list myList;
    va_start (myList, n_args);

    int myNumber = va_arg (myList, int);
    va_end (myList);
}
```

By popping `n_args` integers off of the variable-length argument list, we can manage to find the average of the numbers:

```
#include <stdarg.h>
float average (int n_args, ...)
{
    va_list myList;
    va_start (myList, n_args);

    int numbersAdded = 0;
    int sum = 0;

    while (numbersAdded < n_args) {
        int number = va_arg (myList, int); // Get next number from list
        sum += number;
        numbersAdded += 1;
    }
    va_end (myList);

    float avg = (float)(sum) / (float)(numbersAdded); // Find the average
    return avg;
}
```

By calling `average (2, 10, 20)`, we get the average of `10` and `20`, which is `15.`

it:C/Blocchi e funzioni/Funzioni[19] pl:C/Funkcje[20]

19 `http://it.wikibooks.org/wiki/C%2FBlocchi%20e%20funzioni%2FFunzioni`

20 `http://pl.wikibooks.org/wiki/C%2FFunkcje`

18 Preprocessor

Preprocessors are a way of making text processing with your C program before they are actually compiled. Before the actual compilation of every C program it is passed through a Preprocessor. The Preprocessor looks through the program trying to find out specific instructions called Preprocessor directives that it can understand. All Preprocessor directives begin with the # (hash) symbol. C++ compilers use the same C preprocessor.[1]

The preprocessor[2] is a part of the compiler which performs preliminary operations (conditionally compiling code, including files etc...) to your code before the compiler sees it. These transformations are lexical, meaning that the output of the preprocessor is still text.

> NOTE: Technically the output of the preprocessing phase for C consists of a sequence of tokens, rather than source text, but it is simple to output source text which is equivalent to the given token sequence, and that is commonly supported by compilers via a `-E` or `/E` option -- although command line options to C compilers aren't completely standard, many follow similar rules.

18.1 Directives

Directives are special instructions directed to the preprocessor (preprocessor directive) or to the compiler[3] (compiler directive) on how it should process part or all of your source code or set some flags on the final object and are used to make writing source code easier (more portable for instance) and to make the source code more understandable. Directives are handled by the preprocessor, which is either a separate program invoked by the compiler or part of the compiler itself.

18.1.1 #include

C has some features as part of the language and some others as part of a **standard library**, which is a repository of code that is available alongside every standard-conformant C compiler. When the C compiler compiles your program it usually also links it with the standard C library. For example, on encountering a `#include <stdio.h>` directive, it replaces the directive with the contents of the `stdio.h` header file.

1 Understanding C++/C Preprocessor ^{`http://en.wikibooks.org/wiki/Understanding%20C%2B%2B%2FC%20Preprocessor`}
2 `http://en.wikipedia.org/wiki/Preprocessor`
3 `http://en.wikipedia.org/wiki/compiler`

When you use features from the library, C requires you to *declare* what you would be using. The first line in the program is a **preprocessing directive** which should look like this:

```
#include <stdio.h>
```

The above line causes the C declarations which are in the `stdio.h` header[4] to be included for use in your program. Usually this is implemented by just inserting into your program the contents of a **header file** called `stdio.h`, located in a system-dependent location. The location of such files may be described in your compiler's documentation. A list of standard C header files is listed below in the Headers table.

The `stdio.h` header contains various declarations for input/output (I/O) using an abstraction of I/O mechanisms called **streams**. For example there is an output stream object called `stdout` which is used to output text to the standard output, which usually displays the text on the computer screen.

If using angle brackets like the example above, the preprocessor is instructed to search for the include file along the development environment path for the standard includes.

```
#include "other.h"
```

If you use quotation marks (`" "`), the preprocessor is expected to search in some additional, usually user-defined, locations for the header file, and to fall back to the standard include paths only if it is not found in those additional locations. It is common for this form to include searching in the same directory as the file containing the `#include` directive.

> NOTE: You should check the documentation of the development environment you are using for any vendor specific implementations of the `#include` directive.

Headers

The C90 standard headers list:

4 http://en.wikipedia.org/wiki/Header%20file

* `<assert.h>`[5]*	* `<locale.h>`[10]*	* `<stddef.h>`[15]*
`<ctype.h>`[6]*	`<math.h>`[11]*	`<stdio.h>`[16]*
`<errno.h>`[7]*	`<setjmp.h>`[12]*	`<stdlib.h>`[17]*
`<float.h>`[8]*	`<signal.h>`[13]*	`<string.h>`[18]*
`<limits.h>`[9]	`<stdarg.h>`[14]	`<time.h>`[19]

Headers added since C90:

* `<complex.h>`[20]*	* `<iso646.h>`[23]*	* `<tgmath.h>`[26]*
`<fenv.h>`[21]*	`<stdbool.h>`[24]*	`<wchar.h>`[27]*
`<inttypes.h>`[22]	`<stdint.h>`[25]	`<wctype.h>`[28]

18.1.2 #pragma

The **pragma** (pragmatic information) directive is part of the standard, but the meaning of any pragma depends on the software implementation of the standard that is used. The #pragma directive provides a way to request special behavior from the compiler. This directive is most useful for programs that are unusually large or that need to take advantage of the capabilities of a particular compiler.

Pragmas are used within the source program.

```
#pragma token(s)
```

5 http://en.wikipedia.org/wiki/Assert.h
6 http://en.wikipedia.org/wiki/Ctype.h
7 http://en.wikipedia.org/wiki/Errno.h
8 http://en.wikipedia.org/wiki/Float.h
9 http://en.wikipedia.org/wiki/Limits.h
10 http://en.wikipedia.org/wiki/Locale.h
11 http://en.wikipedia.org/wiki/Math.h
12 http://en.wikipedia.org/wiki/Setjmp.h
13 http://en.wikipedia.org/wiki/Signal.h
14 http://en.wikipedia.org/wiki/Stdarg.h
15 http://en.wikipedia.org/wiki/Stddef.h
16 http://en.wikipedia.org/wiki/Stdio.h
17 http://en.wikipedia.org/wiki/Stdlib.h
18 http://en.wikipedia.org/wiki/String.h
19 http://en.wikipedia.org/wiki/Time.h
20 http://en.wikipedia.org/wiki/Complex.h
21 http://en.wikipedia.org/wiki/Fenv.h
22 http://en.wikipedia.org/wiki/Inttypes.h
23 http://en.wikipedia.org/wiki/Iso646.h
24 http://en.wikipedia.org/wiki/Stdbool.h
25 http://en.wikipedia.org/wiki/Stdint.h
26 http://en.wikipedia.org/wiki/Tgmath.h
27 http://en.wikipedia.org/wiki/Wchar.h
28 http://en.wikipedia.org/wiki/Wctype.h

1. pragma is usually followed by a single token, which represents a command for the compiler to obey. You should check the software implementation of the C standard you intend on using for a list of the supported tokens. Not surprisingly, the set of commands that can appear in #pragma directives is different for each compiler; you'll have to consult the documentation for your compiler to see which commands it allows and what those commands do.

For instance one of the most implemented preprocessor directives, `#pragma once` when placed at the beginning of a header file, indicates that the file where it resides will be skipped if included several times by the preprocessor.

NOTE: Other methods exist to do this action that is commonly referred as using **include guards**.

18.1.3 `#define`

WARNING: Preprocessor macros, although tempting, can produce quite unexpected results if not done right. Always keep in mind that macros are textual substitutions done to your source code before anything is compiled. The compiler does not know anything about the macros and never gets to see them. This can produce obscure errors, amongst other negative effects. Prefer to use language features, if there are equivalent (In example use `const int` or `enum` instead of `#define`d constants).That said, there are cases, where macros are very useful (see the `debug` macro below for an example).

The `#define` directive is used to define values or macros that are used by the preprocessor to manipulate the program source code before it is compiled. Because preprocessor definitions are substituted before the compiler acts on the source code, any errors that are introduced by `#define` are difficult to trace.

By convention, values defined using `#define` are named in uppercase. Although doing so is not a requirement, it is considered very bad practice to do otherwise. This allows the values to be easily identified when reading the source code.

Today, `#define` is primarily used to handle compiler and platform differences. E.g., a define might hold a constant which is the appropriate error code for a system call. The use of `#define` should thus be limited unless absolutely necessary; `typedef` statements and constant variables can often perform the same functions more safely.

Another feature of the `#define` command is that it can take arguments, making it rather useful as a pseudo-function creator. Consider the following code:

```
#define ABSOLUTE_VALUE( x ) ( ((x) < 0) ? -(x) : (x) )
...
int x = -1;
```

```
while( ABSOLUTE_VALUE( x ) ) {
...
}
```

It's generally a good idea to use extra parentheses when using complex macros. Notice that in the above example, the variable "x" is always within its own set of parentheses. This way, it will be evaluated in whole, before being compared to 0 or multiplied by -1. Also, the entire macro is surrounded by parentheses, to prevent it from being contaminated by other code. If you're not careful, you run the risk of having the compiler misinterpret your code.

Because of side-effects it is considered a very bad idea to use macro functions as described above.

```
int x = -10;
int y = ABSOLUTE_VALUE( x++ );
```

If ABSOLUTE_VALUE() were a real function 'x' would now have the value of '-9', but because it was an argument in a macro it was expanded twice and thus has a value of -8.

Example:To illustrate the dangers of macros, consider this naive macro #define MAX(a,b) a>b?a:band the code i = MAX(2,3)+5; j = MAX(3,2)+5;Take a look at this and consider what the value after execution might be. The statements are turned into int i = 2>3?2:3+5; int j = 3>2?3:2+5;Thus, after execution `i=8` and `j=3` instead of the expected result of `i=j=8`! This is why you were cautioned to use an extra set of parenthesis above, but even with these, the road is fraught with dangers. The alert reader might quickly realize that if `a` or `b` contains expressions, the definition must parenthesize every use of `a,b` in the macro definition, like this: #define MAX(a,b) ((a)>(b)?(a):(b))This works, provided `a,b` have no side effects. Indeed, i = 2; j = 3; k = MAX(i++, j++);would result in `k=4`, `i=3` and `j=5`. This would be highly surprising to anyone expecting `MAX()` to behave like a function.So what is the correct solution? The solution is not to use macro at all. A global, inline function, like this inline int max(int a, int b) { return a>b?a:b }has none of the pitfalls above, but will not work with all types.

NOTE: The explicit `inline` declaration is not really necessary unless the definition is in a header file, since your compiler can inline functions for you (with gcc this can be done with `-finline-functions` or `-O3`). The compiler is often better than the programmer at predicting which functions are worth inlining. Also, function calls are not really expensive (they used to be). The compiler is actually free to ignore the `inline` keyword. It is only a hint (except that `inline` is necessary in order to allow a function to be defined in a header file without generating an error message due to the function being defined in more than one translation unit).

(#, ##)

The # and ## operators are used with the `#define` macro. Using # causes the first argument after the # to be returned as a string in quotes. For example, the command

```
#define as_string( s ) # s
```

will make the compiler turn this command

```
puts( as_string( Hello World! ) ) ;
```

into

```
puts( "Hello World!" );
```

Using ## concatenates what's before the ## with what's after it. For example, the command

```
#define concatenate( x, y ) x ## y
...
int xy = 10;
...
```

will make the compiler turn

```
printf( "%d", concatenate( x, y ));
```

into

```
printf( "%d", xy);
```

which will, of course, display `10` to standard output.

It is possible to concatenate a macro argument with a constant prefix or suffix to obtain a valid identifier as in

```
#define make_function( name ) int my_ ## name (int foo) {}
make_function( bar )
```

which will define a function called `my_bar()`. But it isn't possible to integrate a macro argument into a constant string using the concatenation operator. In order to obtain such an effect, one can use the ANSI C property that two or more consecutive string constants are considered equivalent to a single string constant when encountered. Using this property, one can write

```
#define eat( what ) puts( "I'm eating " #what " today." )
eat( fruit )
```

which the macro-processor will turn into

```
puts( "I'm eating " "fruit" " today." )
```

which in turn will be interpreted by the C parser as a single string constant.

The following trick can be used to turn a numeric constants into string literals

```
#define num2str(x) str(x)
#define str(x) #x
#define CONST 23

puts(num2str(CONST));
```

This is a bit tricky, since it is expanded in 2 steps. First `num2str(CONST)` is replaced with `str(23)`, which in turn is replaced with `"23"`. This can be useful in the following example:

```
#ifdef DEBUG
#define debug(msg) fputs(__FILE__ ":" num2str(__LINE__) " - " msg, stderr)
#else
#define debug(msg)
#endif
```

This will give you a nice debug message including the file and the line where the message was issued. If DEBUG is not defined however the debugging message will completely vanish from your code. Be careful not to use this sort of construct with anything that has side effects, since this can lead to bugs, that appear and disappear depending on the compilation parameters.

18.1.4 macros

Macros aren't type-checked and so they do not evaluate arguments. Also, they do not obey scope properly, but simply take the string passed to them and replace each occurrence of the macro argument in the text of the macro with the actual string for that parameter (the code is literally copied into the location it was called from).

An example on how to use a macro:

```
#include <stdio.h>

#define SLICES 8
#define ADD(x) ( (x) / SLICES )

int main()
{
  int a = 0, b = 10, c = 6;

  a = ADD(b + c);
  printf("%d\n", a);
  return 0;
}
```

-- the result of "a" should be "2" (b + c = 16 -> passed to ADD -> 16 / SLICES -> result is "2")

> NOTE:
> It is usually bad practice to define macros in headers.A macro should be defined only when it is not possible to achieve the same result with a function or some other mechanism. Some compilers are able to optimize code to where calls to small functions are replaced with inline code, negating any possible speed advantage.Using typedefs, enums, and `inline` (in C99) is often a better option.

One of the few situations where inline functions won't work -- so you are pretty much forced to use function-like macros instead -- is to initialize compile time constants (static initialization of structs). This happens when the arguments to the macro are literals that the compiler can optimize to another literal. [29]

18.1.5 #error

The **#error** directive halts compilation. When one is encountered the standard specifies that the compiler should emit a diagnostic containing the remaining tokens in the directive. This is mostly used for debugging purposes.

```
#error message
```

18.1.6 #warning

Many compilers support a **#warning** directive. When one is encountered, the compiler emits a diagnostic containing the remaining tokens in the directive.

1. warning message

18.1.7 #undef

The **#undef** directive undefines a macro. The identifier need not have been previously defined.

18.1.8 #if,#else,#elif,#endif (conditionals)

The **#if** command checks whether a controlling conditional expression evaluates to zero or nonzero, and excludes or includes a block of code respectively. For example:

```
#if 1
```

29 David Hart, Jon Reid. "9 Code Smells of Preprocessor Use" ^{`http://qualitycoding.org/preprocessor/`} . 2012.

```
/* This block will be included */
#endif
#if 0
/* This block will not be included */
#endif
```

The conditional expression could contain any C operator except for the assignment operators, the increment and decrement operators, the address-of operator, and the sizeof operator.

One unique operator used in preprocessing and nowhere else is the **defined** operator. It returns 1 if the macro name, optionally enclosed in parentheses, is currently defined; 0 if not.

The **#endif** command ends a block started by `#if`, `#ifdef`, or `#ifndef`.

The **#elif** command is similar to `#if`, except that it is used to extract one from a series of blocks of code. E.g.:

```
#if /* some expression */
  :
  :
  :
#elif /* another expression */
  :
/* imagine many more #elifs here ... */
  :
#else
/* The optional #else block is selected if none of the previous #if or
   #elif blocks are selected */
  :
  :
#endif /* The end of the #if block */
```

18.1.9 #ifdef,#ifndef

The **#ifdef** command is similar to `#if`, except that the code block following it is selected if a macro name is defined. In this respect,

```
#ifdef NAME
```

is equivalent to

```
#if defined NAME
```

The **#ifndef** command is similar to **#ifdef**, except that the test is reversed:

```
#ifndef NAME
```

is equivalent to

```
#if !defined NAME
```

18.2 Useful Preprocessor Macros for Debugging

ANSI C defines some useful preprocessor macros and variables,[30][31] also called "magic constants", include:

___FILE___ => The name of the current file, as a string literal

___LINE___ => Current line of the source file, as a numeric literal

___DATE___ => Current system date, as a string

___TIME___ => Current system time, as a string

___TIMESTAMP___ => Date and time (non-standard)

___cplusplus => undefined when your C code is being compiled by a C compiler; 199711L when your C code is being compiled by a C++ compiler compliant with 1998 C++ standard.

___func___ => Current function name of the source file, as a string (part of C99)

___PRETTY_FUNCTION___ => "decorated" Current function name of the source file, as a string (in GCC; non-standard)

Compile-time assertions

Some people[32] define a preprocessor macro to allow compile-time assertions, something like:

```
#define COMPILE_TIME_ASSERT(pred) switch(0){case 0:case pred:;}

COMPILE_TIME_ASSERT( BOOLEAN CONDITION );
```

30 HP C Compiler Reference Manual ^{http://docs.hp.com/en/B3901-90003/ch07s04.html}
31 C++ reference: Predefined preprocessor variables ^{http://www.cppreference.com/wiki/preprocessor/preprocessor_vars}
32 "Compile Time Assertions in C" ^{http://www.jaggersoft.com/pubs/CVu11_3.html} by Jon Jagger 1999

The `static_assert.hpp` Boost library[33] defines a similar macro. Some compilers define a `static_assert` keyword used in the same way.[34]

Such compile-time assertions can help you debug faster than using only run-time assert() statements, because the compile-time assertions are all tested at compile time, while it is possible that a test run of a program may fail to exercise some run-time assert() statements.

X-Macros

One little-known usage pattern of the C preprocessor is known as "X-Macros".[35][36][37][38] An X-Macro is a header file[39] or macro. Commonly these use the extension ".def" instead of the traditional ".h". This file contains a list of similar macro calls, which can be referred to as "component macros". The include file is then referenced repeatedly in the following pattern. Here, the include file is "xmacro.def" and it contains a list of component macros of the style "foo(x, y, z)".

```
#define foo(x, y, z) doSomethingWith(x, y, z);
#include "xmacro.def"
#undef foo

#define foo(x, y, z) doSomethingElseWith(x, y, z);
#include "xmacro.def"
#undef foo

(etc...)
```

The most common usage of X-Macros is to establish a list of C objects and then automatically generate code for each of them. Some implementations also perform any `#undef`s they need inside the X-Macro, as opposed to expecting the caller to undefine them.

33 `http://en.wikibooks.org/wiki/C%2B%2B%20Programming%2FLibraries%2FBoost%20`

34 Wikipedia: C++0x#Static assertions ^{`http://en.wikipedia.org/wiki/%20C%2B%2B0x%23Static%20assertions`}

35 Wirzenius, Lars. C Preprocessor Trick For Implementing Similar Data Types ^{`http://liw.iki.fi/liw/texts/cpp-trick.html`} Retrieved January 9, 2011.

36 Randy Meyers . The New C: X Macros The New C: X Macros ^{`www.ddj.com/cpp/184401387`} . *Dr. Dobb's Journal* , May 2001

37 Beal

```
| first = Stephan
| month = August
| year = 2004
| title = Supermacros
| url = http://wanderinghorse.net/computing/papers/#supermacros
|  accessdate = 27 October 2008
```

. . ,

38 Keith Schwarz. "Advanced Preprocessor Techniques" ^{`http://www.keithschwarz.com/cs106l/spring2009/handouts/080_Preprocessor_2.pdf`} . 2009. Includes "Practical Applications of the Preprocessor II: The X Macro Trick".

39 `http://en.wikibooks.org/wiki/C%2B%2B%20Programming%E2%80%8E%2FProgramming%20Languages%E2%80%8E%2FC%2B%2B%E2%80%8E%2FCode%2FFile%20Organization%23.h`

Common sets of objects are a set of global configuration settings, a set of members of a struct[40], a list of possible XML[41] tags for converting an XML file to a quickly-traversable tree, or the body of an enum[42] declaration; other lists are possible.

Once the X-Macro has been processed to create the list of objects, the component macros can be redefined to generate, for instance, accessor and/or mutator[43] functions. Structure serializing and deserializing[44] are also commonly done.

Here is an example of an X-Macro that establishes a struct and automatically creates serialize/deserialize functions. For simplicity, this example doesn't account for endianness or buffer overflows.

File **star.def**:

```
EXPAND_EXPAND_STAR_MEMBER(x, int)
EXPAND_EXPAND_STAR_MEMBER(y, int)
EXPAND_EXPAND_STAR_MEMBER(z, int)
EXPAND_EXPAND_STAR_MEMBER(radius, double)
#undef EXPAND_EXPAND_STAR_MEMBER
```

File **star_table.c**:

```
typedef struct {
  #define EXPAND_EXPAND_STAR_MEMBER(member, type) type member;
  #include "star.def"
  } starStruct;

void serialize_star(const starStruct *const star, unsigned char *buffer) {
  #define EXPAND_EXPAND_STAR_MEMBER(member, type) \
    memcpy(buffer, &(star->member), sizeof(star->member)); \
    buffer += sizeof(star->member);
  #include "star.def"
  }

void deserialize_star(starStruct *const star, const unsigned char *buffer) {
  #define EXPAND_EXPAND_STAR_MEMBER(member, type) \
    memcpy(&(star->member), buffer, sizeof(star->member)); \
    buffer += sizeof(star->member);
  #include "star.def"
  }
```

Handlers for individual data types may be created and accessed using token concatenation ("##") and quoting ("#") operators. For example, the following might be added to the above code:

```
#define print_int(val)    printf("%d", val)
#define print_double(val) printf("%g", val)

void print_star(const starStruct *const star) {
  /* print_##type will be replaced with print_int or print_double */
  #define EXPAND_EXPAND_STAR_MEMBER(member, type) \
    printf("%s: ", #member); \
    print_##type(star->member); \
    printf("\n");
```

40 http://en.wikibooks.org/wiki/struct%20%28C%20programming%20language%29
41 http://en.wikibooks.org/wiki/XML
42 http://en.wikibooks.org/wiki/enumerated%20type
43 http://en.wikibooks.org/wiki/mutator%20method
44 http://en.wikibooks.org/wiki/serialization

```
  #include "star.def"
  }
```

Note that in this example you can also avoid the creation of separate handler functions for each datatype in this example by defining the print format for each supported type, with the additional benefit of reducing the expansion code produced by this header file:

```
#define FORMAT_(type) FORMAT_##type
#define FORMAT_int    "%d"
#define FORMAT_double "%g"

void print_star(const starStruct *const star) {
  /* FORMAT_(type) will be replaced with FORMAT_int or FORMAT_double */
  #define EXPAND_EXPAND_STAR_MEMBER(member, type) \
    printf("%s: " FORMAT_(type) "\n", #member, star->member);
  #include "star.def"
  }
```

The creation of a separate header file can be avoided by creating a single macro containing what would be the contents of the file. For instance, the above file "star.def" could be replaced with this macro at the beginning of:

File **star_table.c**:

```
#define EXPAND_STAR \
  EXPAND_STAR_MEMBER(x, int) \
  EXPAND_STAR_MEMBER(y, int) \
  EXPAND_STAR_MEMBER(z, int) \
  EXPAND_STAR_MEMBER(radius, double)
```

and then all calls to `#include "star.def"` could be replaced with a simple `EXPAND_STAR` statement. The rest of the above file would become:

```
typedef struct {
  #define EXPAND_STAR_MEMBER(member, type) type member;
  EXPAND_STAR
  #undef  EXPAND_STAR_MEMBER
  } starStruct;

void serialize_star(const starStruct *const star, unsigned char *buffer) {
  #define EXPAND_STAR_MEMBER(member, type) \
    memcpy(buffer, &(star->member), sizeof(star->member)); \
    buffer += sizeof(star->member);
  EXPAND_STAR
  #undef  EXPAND_STAR_MEMBER
  }

void deserialize_star(starStruct *const star, const unsigned char *buffer) {
  #define EXPAND_STAR_MEMBER(member, type) \
    memcpy(&(star->member), buffer, sizeof(star->member)); \
    buffer += sizeof(star->member);
  EXPAND_STAR
  #undef  EXPAND_STAR_MEMBER
  }
```

and the print handler could be added as well as:

```
#define print_int(val)    printf("%d", val)
#define print_double(val) printf("%g", val)

void print_star(const starStruct *const star) {
  /* print_##type will be replaced with print_int or print_double */
  #define EXPAND_STAR_MEMBER(member, type) \
    printf("%s: ", #member); \
    print_##type(star->member); \
    printf("\n");
  EXPAND_STAR
  #undef EXPAND_STAR_MEMBER
}
```

or as:

```
#define FORMAT_(type) FORMAT_##type
#define FORMAT_int    "%d"
#define FORMAT_double "%g"

void print_star(const starStruct *const star) {
  /* FORMAT_(type) will be replaced with FORMAT_int or FORMAT_double */
  #define EXPAND_STAR_MEMBER(member, type) \
    printf("%s: " FORMAT_(type) "\n", #member, star->member);
  EXPAND_STAR
  #undef EXPAND_STAR_MEMBER
  }
```

A variant which avoids needing to know the members of any expanded sub-macros is to accept the operators as an argument to the list macro:

File **star_table.c**:

```
/*
 Generic
 */
#define STRUCT_MEMBER(member, type, dummy) type member;

#define SERIALIZE_MEMBER(member, type, obj, buffer) \
  memcpy(buffer, &(obj->member), sizeof(obj->member)); \
  buffer += sizeof(obj->member);

#define DESERIALIZE_MEMBER(member, type, obj, buffer) \
  memcpy(&(obj->member), buffer, sizeof(obj->member)); \
  buffer += sizeof(obj->member);

#define FORMAT_(type) FORMAT_##type
#define FORMAT_int    "%d"
#define FORMAT_double "%g"

/* FORMAT_(type) will be replaced with FORMAT_int or FORMAT_double */
#define PRINT_MEMBER(member, type, obj) \
  printf("%s: " FORMAT_(type) "\n", #member, obj->member);

/*
 starStruct
 */

#define EXPAND_STAR(_, ...) \
  _(x, int, __VA_ARGS__) \
  _(y, int, __VA_ARGS__) \
  _(z, int, __VA_ARGS__) \
  _(radius, double, __VA_ARGS__)

typedef struct {
  EXPAND_STAR(STRUCT_MEMBER, )
  } starStruct;

void serialize_star(const starStruct *const star, unsigned char *buffer) {
  EXPAND_STAR(SERIALIZE_MEMBER, star, buffer)
  }
```

```
void deserialize_star(starStruct *const star, const unsigned char *buffer) {
  EXPAND_STAR(DESERIALIZE_MEMBER, star, buffer)
  }

void print_star(const starStruct *const star) {
  EXPAND_STAR(PRINT_MEMBER, star)
  }
```

This approach can be dangerous in that the entire macro set is always interpreted as if it was on a single source line, which could encounter compiler limits with complex component macros and/or long member lists.

This technique was reported by Lars Wirzenius[45] in a web page dated January 17, 2000, in which he gives credit to Kenneth Oksanen for "refining and developing" the technique prior to 1997. The other references describe it as a method from at least a decade before the turn of the century.

w:C preprocessor[46]

de:C-Programmierung: Präprozessor[47] fr:Programmation C/Préprocesseur[48] it:C/Compilatore e precompilatore/Direttive[49] pl:C/Preprocesor[50]

45 Wirzenius, Lars. C Preprocessor Trick For Implementing Similar Data Types ^{http://liw.iki.fi/liw/texts/cpp-trick.html} Retrieved January 9, 2011.
46 http://en.wikipedia.org/wiki/C%20preprocessor
47 http://de.wikibooks.org/wiki/C-Programmierung%3A%20Pr%C3%A4prozessor
48 http://fr.wikibooks.org/wiki/Programmation%20C%2FPr%C3%A9processeur
49 http://it.wikibooks.org/wiki/C%2FCompilatore%20e%20precompilatore%2FDirettive
50 http://pl.wikibooks.org/wiki/C%2FPreprocesor

19 Libraries

A *library* in C is a group of functions and declarations, exposed for use by other programs. The library therefore consists of an *interface* expressed in a `.h` file (named the "header") and an *implementation* expressed in a `.c` file. This `.c` file might be precompiled or otherwise inaccessible, or it might be available to the programmer. (Note: Libraries may call functions in other libraries such as the Standard C or math libraries to do various tasks.)

The format of a library varies with the operating system and compiler one is using. For example, in the Unix and Linux operating systems, a library consists of one or more *object files*, which consist of object code that is usually the output of a compiler (if the source language is C or something similar) or an assembler (if the source language is assembly language). These object files are then turned into a library in the form of an archive by the *ar* archiver (a program that takes files and stores them in a bigger file without regard to compression). The filename for the library usually starts with "lib" and ends with ".a"; e.g. the *libc.a* file contains the Standard C library and the "libm.a" the mathematics routines, which the linker would then link in. Other operating systems such as Microsoft Windows use a ".lib" extension for libraries and an ".obj" extension for object files.

We're going to use as an example a function to parse[1] arguments from the command line. Arguments on the command line could be by themselves:

```
     -i
```

have an optional argument that is concatenated[2] to the letter:

```
     -ioptarg
```

or have the argument in a separate argv-element:

```
     -i optarg
```

In order to parse all these types of arguments, we have written the following "getopt.c" file:

```
#include <stdio.h>              /* for fprintf() and EOF */
#include <string.h>             /* for strchr() */
#include "getopt.h"             /* consistency check */

/* variables */
int opterr = 1;                 /* getopt prints errors if this is on */
```

1 http://en.wikipedia.org/wiki/Parsing
2 http://en.wikipedia.org/wiki/Concatenate

```
int optind = 1;                  /* token pointer */
int optopt;                      /* option character passed back to user */
char *optarg;                    /* flag argument (or value) */

/* function */
/* return option character, EOF if no more or ? if problem.
        The arguments to the function:
        argc, argv - the arguments to the main() function. An argument of "--"
        stops the processing.
        opts - a string containing the valid option characters.
        an option character followed by a colon (:) indicates that
        the option has a required argument.
*/
int
getopt (int argc, char **argv, char *opts)
{
        static int sp = 1;              /* character index into current token */
        register char *cp;              /* pointer into current token */

        if (sp == 1)
        {
                /* check for more flag-like tokens */
                if (optind >= argc || argv[optind][0] != '-' || argv[optind][1] == '\0')
                        return EOF;
                else if (strcmp (argv[optind], "--") == 0)
                {
                        optind++;
                        return EOF;
                }
        }

        optopt = argv[optind][sp];

        if (optopt == ':' || (cp = strchr (opts, optopt)) == NULL)
        {
                if (opterr)
                        fprintf (stderr, "%s: invalid option -- '%c'\n", argv[0], optopt);

                /* if no characters left in this token, move to next token */
                if (argv[optind][++sp] == '\0')
                {
                        optind++;
                        sp = 1;
                }

                return '?';
        }

        if (*++cp == ':')
        {
                /* if a value is expected, get it */
                if (argv[optind][sp + 1] != '\0')
                        /* flag value is rest of current token */
                        optarg = argv[optind++] + (sp + 1);
                else if (++optind >= argc)
                {
                        if (opterr)
                                fprintf (stderr, "%s: option requires an argument -- '%c'\n",
                                                        argv[0], optopt);
                        sp = 1;
                        return '?';
                }
                else
                /* flag value is next token */
                optarg = argv[optind++];
                sp = 1;
        }
        else
        {
                /* set up to look at next char in token, next time */
                if (argv[optind][++sp] == '\0')
                {
                        /* no more in current token, so setup next token */
                        sp = 1;
                        optind++;
                }
                optarg = 0;
        }
        return optopt;
}
/* END OF FILE */
```

The interface would be the following "getopt.h" file:

```
#ifndef GETOPT_H
        #define GETOPT_H

        /* exported variables */
        extern int opterr, optind, optopt;
        extern char *optarg;

        /* exported function */
        int getopt(int, char **, char *);
#endif

/* END OF FILE */
```

At a minimum, a programmer has the interface file to figure out how to use a library, although, in general, the library programmer also wrote documentation on how to use the library. In the above case, the documentation should say that the provided arguments `**argv` and `*opts` both shouldn't be null pointers (or why would you be using the `getopt` function anyway?). Specifically, it typically states what each parameter is for and what return values can be expected in which conditions. Programmers that use a library, are normally not interested in the implementation of the library -- unless the implementation has a bug, in which case he would want to complain somehow.

Both the implementation of the getopts library, and programs that use the library should state `#include "getopt.h"`, in order to refer to the corresponding interface. Now the library is "linked" to the program -- the one that contains the main() function. The program may refer to dozens of interfaces.

In some cases, just placing `#include "getopt.h"` may appear correct but will still fail to link properly. This indicates that the library is not installed correctly, or there may be some additional configuration required. You will have to check either the compiler's documentation or library's documentation on how to resolve this issue.

19.1 What to put in header files

As a general rule, headers contain anything that should be exported, or "seen" by the other modules in a program. This includes macro definitions (preprocessor `#defines`); structure, union, and enumeration declarations; typedef declarations; external function declarations; and global variable declarations. In the above `getopt.h` example file, one function declaration (`getopt`) and four global variables (`optind`, `optopt`, `optarg`, and `opterr`) are defined.

The `#ifndef GETOPT_H/#define GETOPT_H` trick is colloquially called **include guards**. This is used so that if the `getopt.h` file were included more than once in a translation unit, the unit would only see the contents once.

19.2 Further reading

- C FAQ: "I'm wondering what to put in .c files and what to put in .h files. (What does ".h" mean, anyway?)"[3]
- PIClist thread: "Global variables in projects with many C files."[4]

pl:C/Biblioteki[5]

3 `http://c-faq.com/cpp/hfiles.html`
4 `http://www.piclist.com/techref/postbot.asp?by=time&id=piclist\T1\textbackslash{}2007\T1\textbackslash{}10\T1\textbackslash{}25\T1\textbackslash{}073430a&tgt=post`
5 `http://pl.wikibooks.org/wiki/C%2FBiblioteki`

20 Standard libraries

The **C standard library** is a standardized collection of header files and library routines used to implement common operations, such as input/output and character string handling. Unlike other languages (such as COBOL, Fortran, and PL/I) C does not include builtin keywords for these tasks, so nearly all C programs rely on the standard library to function.

20.1 History

The C programming language previously did not provide any elementary functionalities, such as I/O operations. Over time, user communities of C shared ideas and implementations to provide that functionality. These ideas became common, and were eventually incorporated into the definition of the standardized C programming language. These are now called the **C standard libraries**.

Both Unix and C were created at AT&T's Bell Laboratories in the late 1960s and early 1970s. During the 1970s the C programming language became increasingly popular, with many universities and organizations beginning to create their own variations of the language for their own projects. By the start of the 1980s compatibility problems between the various C implementations became apparent. In 1983 the American National Standards Institute (ANSI) formed a committee to establish a standard specification of C known as "ANSI C". This work culminated in the creation of the so-called **C89** standard in 1989. Part of the resulting standard was a set of software libraries called the **ANSI C standard library**.

Later revisions of the C standard have added several new required header files to the library. Support for these new extensions varies between implementations.

The headers **<iso646.h>**, **<wchar.h>**, and **<wctype.h>** were added with Normative Addendum 1 (hereafter abbreviated as **NA1**), an addition to the C Standard ratified in 1995.

The headers **<complex.h>**, **<fenv.h>**, **<inttypes.h>**, **<stdbool.h>**, **<stdint.h>**, and **<tgmath.h>** were added with **C99**, a revision to the C Standard published in 1999.

Note:
The C++[a] programming language includes the functionality of the ANSI C 89 standard library, but has made several modifications, such as placing all identifiers into the `std` namespace and changing the names of the header files from `<xxx.h>` to `<cxxx>` (however, the C-style names are still available, although deprecated).

a `http://en.wikibooks.org/wiki/C%2B%2B`

20.2 Design

The declaration of each function is kept in a header file, while the actual implementation of functions are separated into a library file. The naming and scope of headers have become common but the organization of libraries still remains diverse. The standard library is usually shipped along with a compiler. Since C compilers often provide extra functionalities that are not specified in ANSI C, a standard library with a particular compiler is mostly incompatible with standard libraries of other compilers.

Much of the C standard library has been shown to have been well-designed. A few parts, with the benefit of hindsight, are regarded as mistakes. The string input functions `gets()` (and the use of `scanf()` to read string input) are the source of many buffer overflows, and most programming guides recommend avoiding this usage. Another oddity is `strtok()`, a function that is designed as a primitive lexical analyser[1] but is highly "fragile" and difficult to use.

20.3 ANSI Standard

The ANSI C standard library consists of 24 C header files which can be included into a programmer's project with a single directive. Each header file contains one or more function declarations, data type definitions and macros. The contents of these header files follows.

In comparison to some other languages (for example Java) the standard library is minuscule. The library provides a basic set of mathematical functions, string manipulation, type conversions, and file and console-based I/O. It does not include a standard set of "container types" like the C++ Standard Template Library, let alone the complete graphical user interface (GUI) toolkits, networking tools, and profusion of other functionality that Java provides as standard. The main advantage of the small standard library is that providing a working ANSI C environment is much easier than it is with other languages, and consequently porting C to a new platform is relatively easy.

Many other libraries have been developed to supply equivalent functionality to that provided by other languages in their standard library. For instance, the GNOME desktop environment project has developed the GTK+ graphics toolkit and GLib, a library of container data structures, and there are many other well-known examples. The variety of libraries available has meant that some superior toolkits have proven themselves through history. The considerable downside is that they often do not work particularly well together, programmers are often familiar with different sets of libraries, and a different set of them may be available on any particular platform.

20.3.1 ANSI C library header files

<assert.h>[2]	Contains the assert macro, used to assist with detecting logical errors and other types of bug in debugging versions of a program.

1 `http://en.wikipedia.org/wiki/lexical%20analysis`
2 `http://en.wikipedia.org/wiki/Assert.h`

<complex.h>[3]	A set of functions for manipulating complex numbers. (New with **C99**)
<ctype.h>[4]	This header file contains functions used to classify characters by their types or to convert between upper and lower case in a way that is independent of the used character set (typically ASCII or one of its extensions, although implementations utilizing EBCDIC are also known).
<errno.h>[5]	For testing error codes reported by library functions.
<fenv.h>[6]	For controlling floating-point environment. (New with **C99**)
<float.h>[7]	Contains defined constants specifying the implementation-specific properties of the floating-point library, such as the minimum difference between two different floating-point numbers (_EPSILON), the maximum number of digits of accuracy (_DIG) and the range of numbers which can be represented (_MIN, _MAX).
<inttypes.h>[8]	For precise conversion between integer types. (New with **C99**)
<iso646.h>[9]	For programming in ISO 646 variant character sets. (New with **NA1**)
<limits.h>[10]	Contains defined constants specifying the implementation-specific properties of the integer types, such as the range of numbers which can be represented (_MIN, _MAX).
<locale.h>[11]	For setlocale() and related constants. This is used to choose an appropriate locale.
<math.h>[12]	For computing common mathematical functions-- see ../Further math/[13] or C++ Programming/Code/Standard C Library/Math[14] for details.
<setjmp.h>[15]	setjmp and longjmp, which are used for non-local exits
<signal.h>[16]	For controlling various exceptional conditions
<stdarg.h>[17]	For accessing a varying number of arguments passed to functions.
<stdbool.h>[18]	For a boolean data type. (New with **C99**)

3 `http://en.wikipedia.org/wiki/Complex.h`
4 `http://en.wikipedia.org/wiki/Ctype.h`
5 `http://en.wikipedia.org/wiki/Errno.h`
6 `http://en.wikipedia.org/wiki/Fenv.h`
7 `http://en.wikipedia.org/wiki/Float.h`
8 `http://en.wikipedia.org/wiki/Inttypes.h`
9 `http://en.wikipedia.org/wiki/Iso646.h`
10 `http://en.wikipedia.org/wiki/Limits.h`
11 `http://en.wikipedia.org/wiki/Locale.h`
12 `http://en.wikipedia.org/wiki/Math.h`
13 Chapter 15 on page 73
14 `http://en.wikibooks.org/wiki/C%2B%2B%20Programming%2FCode%2FStandard%20C%20Library%2FMath`
15 `http://en.wikipedia.org/wiki/Setjmp.h`
16 `http://en.wikipedia.org/wiki/Signal.h`
17 `http://en.wikipedia.org/wiki/Stdarg.h`
18 `http://en.wikipedia.org/wiki/Stdbool.h`

<stdint.h>[19]	For defining various integer types. (New with **C99**)
<std-def.h>[20]	For defining several useful types and macros.
<stdio.h>[21]	Provides the core input and output capabilities of the C language. This file includes the venerable `printf` function.
<stdlib.h>[22]	For performing a variety of operations, including conversion, pseudo-random numbers, memory allocation, process control, environment, signalling, searching, and sorting.
<string.h>[23]	For manipulating several kinds of strings.
<tg-math.h>[24]	For type-generic mathematical functions. (New with **C99**)
<time.h>[25]	For converting between various time and date formats.
<wchar.h>[26]	For manipulating wide streams and several kinds of strings using wide characters - key to supporting a range of languages. (New with **NA1**)
<wc-type.h>[27]	For classifying wide characters. (New with **NA1**)

20.4 Common support libraries

While not standardized, C programs may depend on a runtime library of routines which contain code the compiler uses at runtime. The code that initializes the process for the operating system, for example, before calling `main()`, is implemented in the C Run-Time Library for a given vendor's compiler. The Run-Time Library code might help with other language feature implementations, like handling uncaught exceptions or implementing floating point code.

The C standard library only documents that the specific routines mentioned in this article are available, and how they behave. Because the compiler implementation might depend on these additional implementation-level functions to be available, it is likely the vendor-specific routines are packaged with the C Standard Library in the same module, because they're both likely to be needed by any program built with their toolset.

Though often confused with the C Standard Library because of this packaging, the C Runtime Library is not a standardized part of the language and is vendor-specific.

19 `http://en.wikipedia.org/wiki/Stdint.h`
20 `http://en.wikipedia.org/wiki/Stddef.h`
21 `http://en.wikipedia.org/wiki/Stdio.h`
22 `http://en.wikipedia.org/wiki/Stdlib.h`
23 `http://en.wikipedia.org/wiki/String.h`
24 `http://en.wikipedia.org/wiki/Tgmath.h`
25 `http://en.wikipedia.org/wiki/Time.h`
26 `http://en.wikipedia.org/wiki/Wchar.h`
27 `http://en.wikipedia.org/wiki/Wctype.h`

20.5 Compiler built-in functions

Some compilers (for example, GCC[28]) provide built-in versions of many of the functions in the C standard library; that is, the implementations of the functions are written into the compiled object file, and the program calls the built-in versions instead of the functions in the C library shared object file. This reduces function call overhead, especially if function calls are replaced with inline variants, and allows other forms of optimization (as the compiler knows the control-flow characteristics of the built-in variants), but may cause confusion when debugging (for example, the built-in versions cannot be replaced with instrumented variants).

20.6 POSIX standard library

POSIX, (along with the Single Unix Specification), specifies a number of routines that should be available over and above those in the C standard library proper; these are often implemented alongside the C standard library functionality, with varying degrees of closeness. For example, glibc implements functions such as fork within libc.so, but before NPTL was merged into glibc it constituted a separate library with its own linker flag. Often, this POSIX-specified functionality will be regarded as part of the library; the C library proper may be identified as the ANSI or ISO C library.

pl:C/Biblioteka standardowa[29]

28 `http://en.wikipedia.org/wiki/GCC`
29 `http://pl.wikibooks.org/wiki/C%2FBiblioteka%20standardowa`

21 File IO

21.1 Introduction

The `stdio.h` header declares a broad assortment of functions that perform input and output to files and devices such as the console. It was one of the earliest headers to appear in the C library. It declares more functions than any other standard header and also requires more explanation because of the complex machinery that underlies the functions.

The device-independent model of input and output has seen dramatic improvement over the years and has received little recognition for its success. FORTRAN II was touted as a machine-independent language in the 1960s, yet it was essentially impossible to move a FORTRAN program between architectures without some change. In FORTRAN II, you named the device you were talking to right in the FORTRAN statement in the middle of your FORTRAN code. So, you said `READ INPUT TAPE 5` on a tape-oriented IBM 7090 but `READ CARD` to read a card image on other machines. FORTRAN IV had more generic `READ` and `WRITE` statements, specifying a *logical unit number* (LUN) instead of the device name. The era of device-independent I/O had dawned.

Peripheral devices such as printers still had fairly strong notions about what they were asked to do. And then, *peripheral interchange* utilities were invented to handle bizarre devices. When cathode-ray tubes came onto the scene, each manufacturer of consoles solved problems such as console cursor movement in an independent manner, causing further headaches.

It was into this atmosphere that Unix was born. Ken Thompson and Dennis Ritchie, the developers of Unix, deserve credit for packing any number of bright ideas into the operating system. Their approach to device independence was one of the brightest.

The ANSI C `<stdio.h>` library is based on the original Unix file I/O primitives but casts a wider net to accommodate the least-common denominator across varied systems.

21.2 Streams

Input and output, whether to or from physical devices such as terminals and tape drives, or whether to or from files supported on structured storage devices, are mapped into logical data streams, whose properties are more uniform than their various inputs and outputs. Two forms of mapping are supported: text streams and binary streams.

A text stream consists of one or more lines. A line in a text stream consists of zero or more characters plus a terminating new-line character. (The only exception is that in some implementations the last line of a file does not require a terminating new-line character.) Unix adopted a standard internal format for all text streams. Each line of text is terminated by a new-line character. That's what any program expects when it reads text, and that's what any program produces when it writes text. (This is the most basic convention, and if it doesn't meet the needs of a text-oriented peripheral attached to a Unix machine, then the fix-up occurs out at the edges of the system. Nothing in between needs to change.) The string of characters that go into, or come out of a text stream may have to be modified to conform to specific conventions. This results in a possible difference between the data that go into a text stream and the data that come out. For instance, in some implementations when a space-character precedes a new-line character in the input, the space character gets removed out of the output. In general, when the data only consists of printable characters and control characters like horizontal tab and new-line, the input and output of a text stream are equal.

Compared to a text stream, a binary stream is pretty straight forward. A binary stream is an ordered sequence of characters that can transparently record internal data. Data written to a binary stream shall always equal the data that gets read out under the same implementation. Binary streams, however, may have an implementation-defined number of null characters appended to the end of the stream. There are no further conventions which need to be considered.

Nothing in Unix prevents the program from writing arbitrary 8-bit binary codes to any open file, or reading them back unchanged from an adequate repository. Thus, Unix obliterated the long-standing distinction between text streams and binary streams.

21.3 Standard Streams

When a C program starts its execution the program automatically opens three standard streams named `stdin`, `stdout`, and `stderr`. These are attached for every C program.

The first standard stream is used for input buffering and the other two are used for output. These streams are sequences of bytes.

Consider the following program:

```
/* An example program. */
int main()
{
    int var;
    scanf ("%d", &var); /* use stdin for scanning an integer from keyboard. */
    printf ("%d", var); /* use stdout for printing a character. */
    return 0;
}
/* end program. */
```

By default `stdin` points to the keyboard and `stdout` and `stderr` point to the screen. It is possible under Unix and may be possible under other operating systems to redirect input from or output to a file or both.

21.4 FILE pointers

The `<stdio.h>` header contains a definition for a type `FILE` (usually via a `typedef`) which is capable of processing all the information needed to exercise control over a stream, including its file position indicator, a pointer to the associated buffer (if any), an error indicator that records whether a read/write error has occurred, and an end-of-file indicator that records whether the end of the file has been reached.

It is considered bad manners to access the contents of `FILE` directly unless the programmer is writing an implementation of `<stdio.h>` and its contents. Better access to the contents of `FILE` is provided via the functions in `<stdio.h>`. It can be said that the `FILE` type is an early example of object-oriented programming[1].

21.5 Opening and Closing Files

To open and close files, the `<stdio.h>` library has three functions: `fopen`, `freopen`, and `fclose`.

21.5.1 Opening Files

```
#include <stdio.h>
FILE *fopen(const char *filename, const char *mode);
FILE *freopen(const char *filename, const char *mode, FILE *stream);
```

`fopen` and `freopen` opens the file whose name is in the string pointed to by `filename` and associates a stream with it. Both return a pointer to the object controlling the stream, or if the open operation fails a null pointer. The error and end-of-file indicators are cleared, and if the open operation fails error is set. `freopen` differs from `fopen` in that the file pointed to by `stream` is closed first when already open and any close errors are ignored.

`mode` for both functions points to a string consisting of one of the following sequences:

r	open a text file for reading
w	truncate to zero length or create a text file for writing
a	append; open or create text file for writing at end-of-file
rb	open binary file for reading
wb	truncate to zero length or create a binary file for writing
ab	append; open or create binary file for writing at end-of-file
r+	open text file for update (reading and writing)
w+	truncate to zero length or create a text file for update
a+	append; open or create text file for update
r+b or rb+	open binary file for update (reading and writing)
w+b or wb+	truncate to zero length or create a binary file for update
a+b or ab+	append; open or create binary file for update

Opening a file with read mode (`'r'` as the first character in the `mode` argument) fails if the file does not exist or cannot be read.

1 http://en.wikipedia.org/wiki/Object-oriented%20programming

Opening a file with append mode ('a' as the first character in the mode argument) causes all subsequent writes to the file to be forced to the then-current end-of-file, regardless of intervening calls to the fseek function. In some implementations, opening a binary file with append mode ('b' as the second or third character in the above list of mode arguments) may initially position the file position indicator for the stream beyond the last data written, because of null character padding.

When a file is opened with update mode ('+' as the second or third character in the above list of mode argument values), both input and output may be performed on the associated stream. However, output may not be directly followed by input without an intervening call to the fflush function or to a file positioning function (fseek, fsetpos, or rewind), and input may not be directly followed by output without an intervening call to a file positioning function, unless the input operation encounters end-of-file. Opening (or creating) a text file with update mode may instead open (or create) a binary stream in some implementations.

When opened, a stream is fully buffered if and only if it can be determined not to refer to an interactive device.

21.5.2 Closing Files

```
#include <stdio.h>
int fclose(FILE *stream);
```

The fclose function causes the stream pointed to by stream to be flushed and the associated file to be closed. Any unwritten buffered data for the stream are delivered to the host environment to be written to the file; any unread buffered data are discarded. The stream is disassociated from the file. If the associated buffer was automatically allocated, it is deallocated. The function returns zero if the stream was successfully closed or EOF if any errors were detected.

21.6 Other file access functions

21.6.1 The fflush function

```
#include <stdio.h>
int fflush(FILE *stream);
```

If stream points to an output stream or an update stream in which the most recent operation was not input, the fflush function causes any unwritten data for that stream to be deferred to the host environment to be written to the file. The behavior of fflush is undefined for input stream.

If stream is a null pointer, the fflush function performs this flushing action on all streams for which the behavior is defined above.

The fflush functions returns EOF if a write error occurs, otherwise zero.

The reason for having a fflush function is because streams in C can have buffered input/output; that is, functions that write to a file actually write to a buffer inside the FILE structure. If the buffer is filled to capacity, the write functions will call `fflush` to actually "write" the data that is in the buffer to the file. Because `fflush` is only called every once in a while, calls to the operating system to do a raw write are minimized.

21.6.2 The `setbuf` function

```
#include <stdio.h>
void setbuf(FILE *stream, char *buf);
```

Except that it returns no value, the `setbuf` function is equivalent to the `setvbuf` function invoked with the values `_IOFBF` for `mode` and `BUFSIZ` for `size`, or (if `buf` is a null pointer) with the value `_IONBF` for `mode`.

21.6.3 The `setvbuf` function

```
#include <stdio.h>
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

The `setvbuf` function may be used only after the stream pointed to by `stream` has been associated with an open file and before any other operation is performed on the stream. The argument `mode` determines how the stream will be buffered, as follows: `_IOFBF` causes input/output to be fully buffered; `_IOLBF` causes input/output to be line buffered; `_IONBF` causes input/output to be unbuffered. If `buf` is not a null pointer, the array it points to may be used instead of a buffer associated by the `setvbuf` function. (The buffer must have a lifetime at least as great as the open stream, so the stream should be closed before a buffer that has automatic storage duration is deallocated upon block exit.) The argument `size` specifies the size of the array. The contents of the array at any time are indeterminate.

The `setvbuf` function returns zero on success, or nonzero if an invalid value is given for `mode` or if the request cannot be honored.

21.7 Functions that Modify the File Position Indicator

The `stdio.h` library has five functions that affect the file position indicator besides those that do reading or writing: `fgetpos`, `fseek`, `fsetpos`, `ftell`, and `rewind`.

The `fseek` and `ftell` functions are older than `fgetpos` and `fsetpos`.

21.7.1 The `fgetpos` and `fsetpos` functions

```
#include <stdio.h>
int fgetpos(FILE *stream, fpos_t *pos);
int fsetpos(FILE *stream, const fpos_t *pos);
```

The `fgetpos` function stores the current value of the file position indicator for the stream pointed to by `stream` in the object pointed to by `pos`. The value stored contains unspecified information usable by the `fsetpos` function for repositioning the stream to its position at the time of the call to the `fgetpos` function.

If successful, the `fgetpos` function returns zero; on failure, the `fgetpos` function returns nonzero and stores an implementation-defined positive value in `errno`.

The `fsetpos` function sets the file position indicator for the stream pointed to by `stream` according to the value of the object pointed to by `pos`, which shall be a value obtained from an earlier call to the `fgetpos` function on the same stream.

A successful call to the `fsetpos` function clears the end-of-file indicator for the stream and undoes any effects of the `ungetc` function on the same stream. After an `fsetpos` call, the next operation on an update stream may be either input or output.

If successful, the `fsetpos` function returns zero; on failure, the `fsetpos` function returns nonzero and stores an implementation-defined positive value in `errno`.

21.7.2 The `fseek` and `ftell` functions

```
#include <stdio.h>
int fseek(FILE *stream, long int offset, int whence);
long int ftell(FILE *stream);
```

The `fseek` function sets the file position indicator for the stream pointed to by `stream`.

For a binary stream, the new position, measured in characters from the beginning of the file, is obtained by adding `offset` to the position specified by `whence`. Three macros in `stdio.h` called `SEEK_SET`, `SEEK_CUR`, and `SEEK_END` expand to unique values. If the position specified by `whence` is `SEEK_SET`, the specified position is the beginning of the file; if `whence` is `SEEK_END`, the specified position is the end of the file; and if `whence` is `SEEK_CUR`, the specified position is the current file position. A binary stream need not meaningfully support `fseek` calls with a `whence` value of `SEEK_END`.

For a text stream, either `offset` shall be zero, or `offset` shall be a value returned by an earlier call to the `ftell` function on the same stream and `whence` shall be `SEEK_SET`.

The `fseek` function returns nonzero only for a request that cannot be satisfied.

The `ftell` function obtains the current value of the file position indicator for the stream pointed to by `stream`. For a binary stream, the value is the number of characters from the beginning of the file; for a text stream, its file position indicator contains unspecified information, usable by the `fseek` function for returning the file position indicator for the stream to its position at the time of the `ftell` call; the difference between two such return values is not necessarily a meaningful measure of the number of characters written or read.

If successful, the `ftell` function returns the current value of the file position indicator for the stream. On failure, the `ftell` function returns `-1L` and stores an implementation-defined positive value in `errno`.

21.7.3 The `rewind` function

```
#include <stdio.h>
void rewind(FILE *stream);
```

The `rewind` function sets the file position indicator for the stream pointed to by `stream` to the beginning of the file. It is equivalent to

```
(void)fseek(stream, 0L, SEEK_SET)
```

except that the error indicator for the stream is also cleared.

21.8 Error Handling Functions

21.8.1 The `clearerr` function

```
#include <stdio.h>
void clearerr(FILE *stream);
```

The `clearerr` function clears the end-of-file and error indicators for the stream pointed to by `stream`.

21.8.2 The `feof` function

```
#include <stdio.h>
int feof(FILE *stream);
```

The `feof` function tests the end-of-file indicator for the stream pointed to by `stream` and returns nonzero if and only if the end-of-file indicator is set for `stream`, otherwise it returns zero.

21.8.3 The `ferror` function

```
#include <stdio.h>
int ferror(FILE *stream);
```

The `ferror` function tests the error indicator for the stream pointed to by `stream` and returns nonzero if and only if the error indicator is set for `stream`, otherwise it returns zero.

21.8.4 The `perror` function

```
#include <stdio.h>
void perror(const char *s);
```

The `perror` function maps the error number in the integer expression `errno` to an error message. It writes a sequence of characters to the standard error stream thus: first, if `s` is not a null pointer and the character pointed to by `s` is not the null character, the string pointed to by `s` followed by a colon (`:`) and a space; then an appropriate error message string followed by a new-line character. The contents of the error message are the same as those returned by the `strerror` function with the argument `errno`, which are implementation-defined.

21.9 Other Operations on Files

The `stdio.h` library has a variety of functions that do some operation on files besides reading and writing.

21.9.1 The `remove` function

```
#include <stdio.h>
int remove(const char *filename);
```

The `remove` function causes the file whose name is the string pointed to by `filename` to be no longer accessible by that name. A subsequent attempt to open that file using that name will fail, unless it is created anew. If the file is open, the behavior of the `remove` function is implementation-defined.

The `remove` function returns zero if the operation succeeds, nonzero if it fails.

21.9.2 The `rename` function

```
#include <stdio.h>
int rename(const char *old_filename, const char *new_filename);
```

The `rename` function causes the file whose name is the string pointed to by `old_filename` to be henceforth known by the name given by the string pointed to by `new_filename`. The file named `old_filename` is no longer accessible by that name. If a file named by the string pointed to by `new_filename` exists prior to the call to the `rename` function, the behavior is implementation-defined.

The `rename` function returns zero if the operation succeeds, nonzero if it fails, in which case if the file existed previously it is still known by its original name.

21.9.3 The `tmpfile` function

```
#include <stdio.h>
FILE *tmpfile(void);
```

The `tmpfile` function creates a temporary binary file that will automatically be removed when it is closed or at program termination. If the program terminates abnormally, whether an open temporary file is removed is implementation-defined. The file is opened for update with `"wb+"` mode.

The `tmpfile` function returns a pointer to the stream of the file that it created. If the file cannot be created, the `tmpfile` function returns a null pointer.

21.9.4 The `tmpnam` function

```
#include <stdio.h>
char *tmpnam(char *s);
```

The `tmpnam` function generates a string that is a valid file name and that is not the name of an existing file.

The `tmpnam` function generates a different string each time it is called, up to `TMP_MAX` times. (`TMP_MAX` is a macro defined in `stdio.h`.) If it is called more than `TMP_MAX` times, the behavior is implementation-defined.

The implementation shall behave as if no library function calls the `tmpnam` function.

If the argument is a null pointer, the `tmpnam` function leaves its result in an internal static object and returns a pointer to that object. Subsequent calls to the `tmpnam` function may modify the same object. If the argument is not a null pointer, it is assumed to point to an array of at least `L_tmpnam` characters (`L_tmpnam` is another macro in `stdio.h`); the `tmpnam` function writes its result in that array and returns the argument as its value.

The value of the macro `TMP_MAX` must be at least 25.

21.10 Reading from Files

21.10.1 Character Input Functions

The `fgetc` function

```
#include <stdio.h>
int fgetc(FILE *stream);
```

The `fgetc` function obtains the next character (if present) as an `unsigned char` converted to an `int`, from the input stream pointed to by `stream`, and advances the associated file position indicator for the stream (if defined).

The `fgetc` function returns the next character from the input stream pointed to by `stream`. If the stream is at end-of-file, the end-of-file indicator for the stream is set and `fgetc` returns `EOF` (`EOF` is a negative value defined in `<stdio.h>`, usually `(-1)`). If a read error occurs, the error indicator for the stream is set and `fgetc` returns `EOF`.

The `fgets` function

```
#include <stdio.h>
char *fgets(char *s, int n, FILE *stream);
```

The `fgets` function reads at most one less than the number of characters specified by `n` from the stream pointed to by `stream` into the array pointed to by `s`. No additional characters are read after a new-line character (which is retained) or after end-of-file. A null character is written immediately after the last character read into the array.

The `fgets` function returns `s` if successful. If end-of-file is encountered and no characters have been read into the array, the contents of the array remain unchanged and a null pointer is returned. If a read error occurs during the operation, the array contents are indeterminate and a null pointer is returned.

Warning: Different operating systems may use different character sequences to represent the end-of-line sequence. For example, some filesystems use the terminator `\r\n` in text files; `fgets` may read those lines, removing the `\n` but keeping the `\r` as the last character of `s`. This expurious character should be removed in the string `s` before the string is used for anything (unless the programmer doesn't care about it). Unixes typically use `\n` as its end-of-line sequence, MS-DOS and Windows uses `\r\n`, and Mac OSes used `\r` before OS X.

```
/* A example program that reads from stdin and writes to stdout */
#include <stdio.h>

#define BUFFER_SIZE 100

int main(void)
{
    char buffer[BUFFER_SIZE]; /* a read buffer */
    while( fgets (buffer, BUFFER_SIZE, stdin) != NULL)
    {
        printf("%s",buffer);
    }
    return 0;
}
/* end program. */
```

The `getc` function

```
#include <stdio.h>
int getc(FILE *stream);
```

The `getc` function is equivalent to `fgetc`, except that it may be implemented as a macro. If it is implemented as a macro, the `stream` argument may be evaluated more than once, so the argument should never be an expression with side effects (i.e. have an assignment, increment, or decrement operators, or be a function call).

The `getc` function returns the next character from the input stream pointed to by `stream`. If the stream is at end-of-file, the end-of-file indicator for the stream is set and `getc` returns `EOF` (`EOF` is a negative value defined in `<stdio.h>`, usually `(-1)`). If a read error occurs, the error indicator for the stream is set and `getc` returns `EOF`.

The `getchar` function

```
#include <stdio.h>
int getchar(void);
```

The `getchar` function is equivalent to `getc` with the argument `stdin`.

The `getchar` function returns the next character from the input stream pointed to by `stdin`. If `stdin` is at end-of-file, the end-of-file indicator for `stdin` is set and `getchar` returns `EOF` (`EOF` is a negative value defined in `<stdio.h>`, usually `(-1)`). If a read error occurs, the error indicator for `stdin` is set and `getchar` returns `EOF`.

The `gets` function

```
#include <stdio.h>
char *gets(char *s);
```

The `gets` function reads characters from the input stream pointed to by `stdin` into the array pointed to by `s` until an end-of-file is encountered or a new-line character is read. Any new-line character is discarded, and a null character is written immediately after the last character read into the array.

The `gets` function returns `s` if successful. If the end-of-file is encountered and no characters have been read into the array, the contents of the array remain unchanged and a null pointer is returned. If a read error occurs during the operation, the array contents are indeterminate and a null pointer is returned.

This function and description is only included here for completeness. Most C programmers nowadays shy away from using `gets`, as there is no way for the function to know how big the buffer is that the programmer wants to read into. Commandment #5 of Henry Spencer[2]'s *The Ten Commandments for C Programmers (Annotated Edition)* reads, "Thou shalt check the array bounds of all strings (indeed, all arrays), for surely where thou typest *foo* someone someday shall type *supercalifragilisticexpialidocious*." It mentions `gets` in the annotation: "As demonstrated by the deeds of the Great Worm, a consequence of this commandment is that robust production software should never make use of `gets()`, for it is truly a tool of the Devil. Thy interfaces should always inform thy servants of the bounds of thy arrays, and servants who spurn such advice or quietly fail to follow it should be dispatched forthwith to the Land Of Rm, where they can do no further harm to thee."

The `ungetc` function

```
#include <stdio.h>
int ungetc(int c, FILE *stream);
```

The `ungetc` function pushes the character specified by `c` (converted to an `unsigned char`) back onto the input stream pointed to by stream. The pushed-back characters will be

2 `http://en.wikipedia.org/wiki/Henry%20Spencer%20`

returned by subsequent reads on that stream in the reverse order of their pushing. A successful intervening call (with the stream pointed to by `stream`) to a file-positioning function (`fseek`, `fsetpos`, or `rewind`) discards any pushed-back characters for the stream. The external storage corresponding to the stream is unchanged.

One character of pushback is guaranteed. If the `ungetc` function is called too many times on the same stream without an intervening read or file positioning operation on that stream, the operation may fail.

If the value of `c` equals that of the macro `EOF`, the operation fails and the input stream is unchanged.

A successful call to the `ungetc` function clears the end-of-file indicator for the stream. The value of the file position indicator for the stream after reading or discarding all pushed-back characters shall be the same as it was before the characters were pushed back. For a text stream, the value of its file-position indicator after a successful call to the `ungetc` function is unspecified until all pushed-back characters are read or discarded. For a binary stream, its file position indicator is decremented by each successful call to the `ungetc` function; if its value was zero before a call, it is indeterminate after the call.

The `ungetc` function returns the character pushed back after conversion, or `EOF` if the operation fails.

21.10.2 EOF pitfall

A mistake when using `fgetc`, `getc`, or `getchar` is to assign the result to a variable of type `char` *before* comparing it to `EOF`. The following code fragments exhibit this mistake, and then show the correct approach (using type int):

Mistake

```
char c;
while ((c = getchar()) != EOF)
    putchar(c);
```

Correction

```
int c;
while ((c = getchar()) != EOF)
    putchar(c);
```

Consider a system in which the type `char` is 8 bits wide, representing 256 different values. `getchar` may return any of the 256 possible characters, and it also may return `EOF` to indicate end-of-file[3], for a total of 257 different possible return values.

When `getchar`'s result is assigned to a `char`, which can represent only 256 different values, there is necessarily some loss of information—when packing 257 items into 256 slots, there must be a collision[4]. The `EOF` value, when converted to `char`, becomes indistinguishable from whichever one of the 256 characters shares its numerical value. If that character is

3 `http://en.wikibooks.org/wiki/end-of-file`
4 `http://en.wikibooks.org/wiki/Pigeonhole%20principle`

found in the file, the above example may mistake it for an end-of-file indicator; or, just as bad, if type `char` is unsigned, then because `EOF` is negative, it can never be equal to any unsigned `char`, so the above example will not terminate at end-of-file. It will loop forever, repeatedly printing the character which results from converting `EOF` to `char`.

However, this looping failure mode does not occur if the char definition is signed (C makes the signedness of the default char type implementation-dependent),[5] assuming the commonly used `EOF` value of -1[6]. However, the fundamental issue remains that if the `EOF` value is defined outside of the range of the `char` type, when assigned to a `char` that value is sliced and will no longer match the full `EOF` value necessary to exit the loop. On the other hand, if `EOF` is within range of `char`, this guarantees a collision between `EOF` and a char value. Thus, regardless of how system types are defined, never use `char` types when testing against `EOF`.

On systems where `int` and `char` are the same size (i.e., systems incompatible with minimally the POSIX and C99 standards), even the "good" example will suffer from the indistinguishability of `EOF` and some character's value. The proper way to handle this situation is to check `feof`[7] and `ferror`[8] after `getchar` returns `EOF`. If `feof` indicates that end-of-file has not been reached, and `ferror` indicates that no errors have occurred, then the `EOF` returned by `getchar` can be assumed to represent an actual character. These extra checks are rarely done, because most programmers assume that their code will never need to run on one of these "big `char`" systems. Another way is to use a compile-time assertion to make sure that `UINT_MAX > UCHAR_MAX`, which at least prevents a program with such an assumption from compiling in such a system.

21.10.3 Direct input function: the `fread` function

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

The `fread` function reads, into the array pointed to by `ptr`, up to `nmemb` elements whose size is specified by `size`, from the stream pointed to by `stream`. The file position indicator for the stream (if defined) is advanced by the number of characters successfully read. If an error occurs, the resulting value of the file position indicator for the stream is indeterminate. If a partial element is read, its value is indeterminate.

The `fread` function returns the number of elements successfully read, which may be less than `nmemb` if a read error or end-of-file is encountered. If `size` or `nmemb` is zero, `fread` returns zero and the contents of the array and the state of the stream remain unchanged.

21.10.4 Formatted input functions: the `scanf` family of functions

```
#include <stdio.h>
int fscanf(FILE *stream, const char *format, ...);
```

5 C99 §6.2.5/15
6 `http://en.wikibooks.org/wiki/End-of-file`
7 `http://en.wikibooks.org/wiki/feof`
8 `http://en.wikibooks.org/wiki/ferror`

```
int scanf(const char *format, ...);
int sscanf(const char *s, const char *format, ...);
```

The `fscanf` function reads input from the stream pointed to by `stream`, under control of the string pointed to by `format` that specifies the admissible sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive converted input. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored.

The format shall be a multibyte character sequence, beginning and ending in its initial shift state. The format is composed of zero or more directives: one or more white-space characters; an ordinary multibyte character (neither `%` or a white-space character); or a conversion specification. Each conversion specification is introduced by the character `%`. After the `%`, the following appear in sequence:

- An optional assignment-suppressing character `*`.
- An optional nonzero decimal integer that specifies the maximum field width.
- An optional `h`, `l` (ell) or `L` indicating the size of the receiving object. The conversion specifiers `d`, `i`, and `n` shall be preceded by `h` if the corresponding argument is a pointer to `short int` rather than a pointer to `int`, or by `l` if it is a pointer to `long int`. Similarly, the conversion specifiers `o`, `u`, and `x` shall be preceded by `h` if the corresponding argument is a pointer to `unsigned short int` rather than `unsigned int`, or by `l` if it is a pointer to `unsigned long int`. Finally, the conversion specifiers `e`, `f`, and `g` shall be preceded by `l` if the corresponding argument is a pointer to `double` rather than a pointer to `float`, or by `L` if it is a pointer to `long double`. If an `h`, `l`, or `L` appears with any other format specifier, the behavior is undefined.
- A character that specifies the type of conversion to be applied. The valid conversion specifiers are described below.

The `fscanf` function executes each directive of the format in turn. If a directive fails, as detailed below, the `fscanf` function returns. Failures are described as input failures (due to the unavailability of input characters) or matching failures (due to inappropriate input).

A directive composed of white-space character(s) is executed by reading input up to the first non-white-space character (which remains unread) or until no more characters remain unread.

A directive that is an ordinary multibyte character is executed by reading the next characters of the stream. If one of the characters differs from one comprising the directive, the directive fails, and the differing and subsequent characters remain unread.

A directive that is a conversion specification defines a set of matching input sequences, as described below for each specifier. A conversion specification is executed in the following steps:

Input white-space characters (as specified by the `isspace` function) are skipped, unless the specification includes a `[`, `c`, or `n` specifier. (The white-space characters are not counted against the specified field width.)

An input item is read from the stream, unless the specification includes an `n` specifier. An input item is defined as the longest matching sequences of input characters, unless that exceeds a specified field width, in which case it is the initial subsequence of that length in the sequence. The first character, if any, after the input item remains unread. If the length of the input item is zero, the execution of the directive fails; this condition is a matching failure, unless an error prevented input from the stream, in which case it is an input failure.

Except in the case of a `%` specifier, the input item (or, in the case of a `%n` directive, the count of input characters) is converted to a type appropriate to the conversion specifier. If the input item is not a matching sequence, the execution of the directive fails; this condition is a matching failure. Unless assignment suppression was indicated by a `*`, the result of the conversion is placed in the object pointed to by the first argument following the `format` argument that has not already received a conversion result. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the space provided, the behavior is undefined.

The following conversion specifiers are valid:

`d`

Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the `strtol` function with the value 10 for the `base` argument. The corresponding argument shall be a pointer to integer.

`i`

Matches an optionally signed integer, whose format is the same as expected for the subject sequence of the `strtol` function with the value 0 for the `base` argument. The corresponding argument shall be a pointer to integer.

`o`

Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of the `strtoul` function with the value 8 for the `base` argument. The corresponding argument shall be a pointer to unsigned integer.

`u`

Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the `strtoul` function with the value 10 for the `base` argument. The corresponding argument shall be a pointer to unsigned integer.

`x`

Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of the `strtoul` function with the value 16 for the `base` argument. The corresponding argument shall be a pointer to unsigned integer.

`e`, `f`, `g`

Matches an optionally signed floating-point number, whose format is the same as expected for the subject string of the `strtod` function. The corresponding argument will be a pointer to floating.

`s`

Matches a sequence of non-white-space characters. (No special provisions are made for multibyte characters.) The corresponding argument shall be a pointer to the initial character of an array large enough to accept the sequence and a terminating null character, which will be added automatically.

`[`

Matches a nonempty sequence of characters (no special provisions are made for multibyte characters) from a set of expected characters (the *scanset*). The corresponding argument shall be a pointer to the initial character of an array large enough to accept the sequence and a terminating null character, which will be added automatically. The conversion specifier includes all subsequent characters in the `format` string, up to and including the matching right bracket (`]`). The characters between the brackets (the *scanlist*) comprise the scanset, unless the character after the left bracket is a circumflex (`^`), in which case the scanset contains all the characters that do not appear in the scanlist between the circumflex and the right bracket. If the conversion specifier begins with `[]` or `[^]`, the right-bracket character is in the scanlist and the next right bracket character is the matching right bracket that ends the specification; otherwise, the first right bracket character is the one that ends the specification. If a `-` character is in the scanlist and is not the first, nor the second where the first character is a `^`, nor the last character, the behavior is implementation-defined.

`c`

Matches a sequence of characters (no special provisions are made for multibyte characters) of the number specified by the field width (1 if no field width is present in the directive). The corresponding argument shall be a pointer to the initial character of an array large enough to accept the sequence. No null character is added.

`p`

Matches an implementation-defined set of sequences, which should be the same as the set of sequences that may be produced by the `%p` conversion of the `fprintf` function. The corresponding argument shall be a pointer to `void`. The interpretation of the input then is implementation-defined. If the input item is a value converted earlier during the same program execution, the pointer that results shall compare equal to that value; otherwise the behavior of the `%p` conversion is undefined.

`n`

No input is consumed. The corresponding argument shall be a pointer to integer into which is to be written the number of characters read from the input stream so far by this call to the `fscanf` function. Execution of a `%n` directive does not increment the assignment count returned at the completion of execution of the `fscanf` function.

`%`

Matches a single `%`; no conversion or assignment occurs. The complete conversion specification shall be `%%`.

If a conversion specification is invalid, the behavior is undefined.

The conversion specifiers `E`, `G`, and `X` are also valid and behave the same as, respectively, `e`, `g`, and `x`.

If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before any characters matching the current directive have been read (other than leading white space, where permitted), execution of the current directive terminates with an input failure; otherwise, unless execution of the current directive is terminated with a matching failure, execution of the following directive (if any) is terminated with an input failure.

If conversion terminates on a conflicting input character, the offending input character is left unread in the input stream. Trailing white space (including new-line characters) is left unread unless matched by a directive. The success of literal matches and suppressed assignments is not directly determinable other than via the `%n` directive.

The `fscanf` function returns the value of the macro `EOF` if an input failure occurs before any conversion. Otherwise, the `fscanf` function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

The `scanf` function is equivalent to `fscanf` with the argument `stdin` interposed before the arguments to `scanf`. Its return value is similar to that of `fscanf`.

The `sscanf` function is equivalent to `fscanf`, except that the argument `s` specifies a string from which the input is to be obtained, rather than from a stream. Reaching the end of the string is equivalent to encountering the end-of-file for the `fscanf` function. If copying takes place between objects that overlap, the behavior is undefined.

21.11 Writing to Files

21.11.1 Character Output Functions

The `fputc` function

```
#include <stdio.h>
int fputc(int c, FILE *stream);
```

The `fputc` function writes the character specified by `c` (converted to an `unsigned char`) to the stream pointed to by `stream` at the position indicated by the associated file position indicator (if defined), and advances the indicator appropriately. If the file cannot support positioning requests, or if the stream is opened with append mode, the character is appended to the output stream. The function returns the character written, unless a write error occurs, in which case the error indicator for the stream is set and `fputc` returns `EOF`.

The `fputs` function

```
#include <stdio.h>
int fputs(const char *s, FILE *stream);
```

The `fputs` function writes the string pointed to by `s` to the stream pointed to by `stream`. The terminating null character is not written. The function returns `EOF` if a write error occurs, otherwise it returns a nonnegative value.

The `putc` function

```
#include <stdio.h>
int putc(int c, FILE *stream);
```

The `putc` function is equivalent to `fputc`, except that if it is implemented as a macro, it may evaluate `stream` more than once, so the argument should never be an expression with side effects. The function returns the character written, unless a write error occurs, in which case the error indicator for the stream is set and the function returns `EOF`.

The `putchar` function

```
#include <stdio.h>
int putchar(int c);
```

The `putchar` function is equivalent to `putc` with the second argument `stdout`. It returns the character written, unless a write error occurs, in which case the error indicator for `stdout` is set and the function returns `EOF`.

The `puts` function

```
#include <stdio.h>
int puts(const char *s);
```

The `puts` function writes the string pointed to by `s` to the stream pointed to by `stdout`, and appends a new-line character to the output. The terminating null character is not written. The function returns `EOF` if a write error occurs; otherwise, it returns a nonnegative value.

21.11.2 Direct output function: the `fwrite` function

```
#include <stdio.h>
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

The `fwrite` function writes, from the array pointed to by `ptr`, up to `nmemb` elements whose size is specified by `size` to the stream pointed to by `stream`. The file position indicator for the stream (if defined) is advanced by the number of characters successfully written. If an error occurs, the resulting value of the file position indicator for the stream is indeterminate. The function returns the number of elements successfully written, which will be less than `nmemb` only if a write error is encountered.

21.11.3 Formatted output functions: the `printf` family of functions

```
#include <stdarg.h>
#include <stdio.h>
int fprintf(FILE *stream, const char *format, ...);
int printf(const char *format, ...);
int sprintf(char *s, const char *format, ...);
int vfprintf(FILE *stream, const char *format, va_list arg);
int vprintf(const char *format, va_list arg);
int vsprintf(char *s, const char *format, va_list arg);
```

Note: Some length specifiers and format specifiers are new in C99. These may not be available in older compilers and versions of the stdio library, which adhere to the C89/C90 standard. Wherever possible, the new ones will be marked with (C99).

The `fprintf` function writes output to the stream pointed to by `stream` under control of the string pointed to by `format` that specifies how subsequent arguments are converted for output. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored. The `fprintf` function returns when the end of the format string is encountered.

The format shall be a multibyte character sequence, beginning and ending in its initial shift state. The format is composed of zero or more directives: ordinary multibyte characters (not `%`), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments, converting them, if applicable, according to the corresponding conversion specifier, and then writing the result to the output stream.

Each conversion specification is introduced by the character `%`. After the `%`, the following appear in sequence:

- Zero or more flags (in any order) that modify the meaning of the conversion specification.
- An optional minimum field width. If the converted value has fewer characters than the field width, it is padded with spaces (by default) on the left (or right, if the left adjustment flag, described later, has been given) to the field width. The field width takes the form of an asterisk `*` (described later) or a decimal integer. (Note that 0 is taken as a flag, not as the beginning of a field width.)
- An optional precision that gives the minimum number of digits to appear for the `d`, `i`, `o`, `u`, `x`, and `X` conversions, the number of digits to appear after the decimal-point character for `a`, `A`, `e`, `E`, `f`, and `F` conversions, the maximum number of significant digits for the `g` and `G` conversions, or the maximum number of characters to be written from a string in `s` conversions. The precision takes the form of a period (`.`) followed either by an asterisk `*` (described later) or by an optional decimal integer; if only the period is specified, the precision is taken as zero. If a precision appears with any other conversion specifier, the behavior is undefined. Floating-point numbers are *rounded* to fit the precision; i.e. `printf("%1.1f\n", 1.19);` produces `1.2`.
- An optional length modifier that specifies the size of the argument.
- A conversion specifier character that specifies the type of conversion to be applied.

As noted above, a field width, or precision, or both, may be indicated by an asterisk. In this case, an `int` argument supplies the field width or precision. The arguments specifying field width, or precision, or both, shall appear (in that order) before the argument (if any) to be converted. A negative field width argument is taken as a `-` flag followed by a positive field width. A negative precision argument is taken as if the precision were omitted.

The flag characters and their meanings are:

`-`

The result of the conversion is left-justified within the field. (It is right-justified if this flag is not specified.)

`+`

The result of a signed conversion always begins with a plus or minus sign. (It begins with a sign only when a negative value is converted if this flag is not specified. The results of all floating conversions of a negative zero, and of negative values that round to zero, include a minus sign.)

space

If the first character of a signed conversion is not a sign, or if a signed conversion results in no characters, a space is prefixed to the result. If the space and `+` flags both appear, the space flag is ignored.

`#`

The result is converted to an "alternative form". For `o` conversion, it increases the precision, if and only if necessary, to force the first digit of the result to be a zero (if the value and precision are both 0, a single 0 is printed). For `x` (or `X`) conversion, a nonzero result has `0x` (or `0X`) prefixed to it. For `a`, `A`, `e`, `E`, `f`, `F`, `g`, and `G` conversions, the result always contains a decimal-point character, even if no digits follow it. (Normally, a decimal-point character appears in the result of these conversions only if a digit follows it.) For `g` and `G` conversions, trailing zeros are not removed from the result. For other conversions, the behavior is undefined.

`0`

For `d`, `i`, `o`, `u`, `x`, `X`, `a`, `A`, `e`, `E`, `f`, `F`, `g`, and `G` conversions, leading zeros (following any indication of sign or base) are used to pad to the field width; no space padding is performed. If the `0` and `-` flags both appear, the `0` flag is ignored. For `d`, `i`, `o`, `u`, `x`, and `X` conversions, if a precision is specified, the `0` flag is ignored. For other conversions, the behavior is undefined.

The length modifiers and their meanings are:

`hh`

(C99) Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `signed char` or `unsigned char` argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to `signed char` or `unsigned char` before printing); or that a following `n` conversion specifier applies to a pointer to a `signed char` argument.

`h`

Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `short int` or `unsigned short int` argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to `short int` or `unsigned short` `int` before printing); or that a following `n` conversion specifier applies to a pointer to a `short int` argument.

`l` (ell)

Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `long int` or `unsigned long int` argument; that a following `n` conversion specifier applies to a pointer to a `long int` argument; (C99) that a following `c` conversion specifier applies to a `wint_t` argument; (C99) that a following `s` conversion specifier applies to a pointer to a `wchar_t` argument; or has no effect on a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier.

`ll` (ell-ell)

(C99) Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `long long int` or `unsigned long long int` argument; or that a following `n` conversion specifier applies to a pointer to a `long long int` argument.

`j`

(C99) Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to an `intmax_t` or `uintmax_t` argument; or that a following `n` conversion specifier applies to a pointer to an `intmax_t` argument.

`z`

(C99) Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `size_t` or the corresponding signed integer type argument; or that a following `n` conversion specifier applies to a pointer to a signed integer type corresponding to `size_t` argument.

`t`

(C99) Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a `ptrdiff_t` or the corresponding unsigned integer type argument; or that a following `n` conversion specifier applies to a pointer to a `ptrdiff_t` argument.

`L`

Specifies that a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to a `long double` argument.

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

The conversion specifiers and their meanings are:

`d, i`

The `int` argument is converted to signed decimal in the style *[−]dddd*. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

`o, u, x, X`

The `unsigned int` argument is converted to unsigned octal (`o`), unsigned decimal (`u`), or unsigned hexadecimal notation (`x` or `X`) in the style *dddd*; the letters **`abcdef`** are used for `x` conversion and the letters **`ABCDEF`** for `X` conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

`f, F`

A `double` argument representing a (finite) floating-point number is converted to decimal notation in the style *[−]ddd.ddd*, where the number of digits after the decimal-point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is zero and the `#` flag is not specified, no decimal-point character appears. If a decimal-point character appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.
(C99) A `double` argument representing an infinity is converted in one of the styles *[-]*`inf` or *[-]*`infinity` — which style is implementation-defined. A double argument representing a NaN is converted in one of the styles *[-]*`nan` or *[-]*`nan(`*n-char-sequence*`)` — which style, and the meaning of any *n-char-sequence*, is implementation-defined. The `F` conversion specifier produces `INF`, `INFINITY`, or `NAN` instead of `inf`, `infinity`, or `nan`, respectively. (When applied to infinite and NaN values, the `-`, `+`, and *space* flags have their usual meaning; the `#` and `0` flags have no effect.)

`e, E`

A `double` argument representing a (finite) floating-point number is converted in the style *[−]d.ddd*e±*dd*, where there is one digit (which is nonzero if the argument is nonzero) before the decimal-point character and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is zero and the `#` flag is not specified, no decimal-point character appears. The value is rounded to the appropriate number of digits. The `E` conversion specifier produces a number with `E` instead of `e` introducing the exponent. The exponent always contains at least two digits, and only as many more digits as necessary to represent the exponent. If the value is zero, the exponent is zero.
(C99) A `double` argument representing an infinity or NaN is converted in the style of an `f` or `F` conversion specifier.

`g, G`

A `double` argument representing a (finite) floating-point number is converted in style `f` or `e` (or in style `F` or `E` in the case of a `G` conversion specifier), with the precision specifying the number of significant digits. If the precision is zero, it is taken as 1. The style used depends on the value converted; style `e` (or `E`) is used only if the exponent resulting from such a conversion is less than −4 or greater than or equal to the precision. Trailing zeros are removed from the fractional portion of the result unless the `#` flag is specified; a decimal-point character appears only if it is followed by a digit.
(C99) A `double` argument representing an infinity or NaN is converted in the style of an `f` or `F` conversion specifier.

`a, A`

(C99) A double argument representing a (finite) floating-point number is converted in the style *[−]*`0x`*h.hhhh*`p`±*d*, where there is one hexadecimal digit (which is nonzero if the argument is a normalized floating-point number and is otherwise unspecified) before the decimal-point character (Binary implementations can choose the hexadecimal digit to the

left of the decimal-point character so that subsequent digits align to nibble [4-bit] boundaries.) and the number of hexadecimal digits after it is equal to the precision; if the precision is missing and `FLT_RADIX` is a power of 2, then the precision is sufficient for an exact representation of the value; if the precision is missing and `FLT_RADIX` is not a power of 2, then the precision is sufficient to distinguish (The precision p is sufficient to distinguish values of the source type if $16^{p-1} > b^n$ where b is `FLT_RADIX` and n is the number of base-b digits in the significand of the source type. A smaller p might suffice depending on the implementation's scheme for determining the digit to the left of the decimal-point character.) values of type `double`, except that trailing zeros may be omitted; if the precision is zero and the `#` flag is not specified, no decimal-point character appears. The letters `abcdef` are used for `a` conversion and the letters `ABCDEF` for `A` conversion. The `A` conversion specifier produces a number with `X` and `P` instead of `x` and `p`. The exponent always contains at least one digit, and only as many more digits as necessary to represent the decimal exponent of 2. If the value is zero, the exponent is zero.
A `double` argument representing an infinity or NaN is converted in the style of an `f` or `F` conversion specifier.

`c`

If no `l` length modifier is present, the `int` argument is converted to an `unsigned char`, and the resulting character is written.
(C99) If an `l` length modifier is present, the `wint_t` argument is converted as if by an `ls` conversion specification with no precision and an argument that points to the initial element of a two-element array of `wchar_t`, the first element containing the `wint_t` argument to the `lc` conversion specification and the second a null wide character.

`s`

If no `l` length modifier is present, the argument shall be a pointer to the initial element of an array of character type. (No special provisions are made for multibyte characters.) Characters from the array are written up to (but not including) the terminating null character. If the precision is specified, no more than that many characters are written. If the precision is not specified or is greater than the size of the array, the array shall contain a null character.
(C99) If an `l` length modifier is present, the argument shall be a pointer to the initial element of an array of `wchar_t` type. Wide characters from the array are converted to multibyte characters (each as if by a call to the `wcrtomb` function, with the conversion state described by an `mbstate_t` object initialized to zero before the first wide character is converted) up to and including a terminating null wide character. The resulting multibyte characters are written up to (but not including) the terminating null character (byte). If no precision is specified, the array shall contain a null wide character. If a precision is specified, no more than that many characters (bytes) are written (including shift sequences, if any), and the array shall contain a null wide character if, to equal the multibyte character sequence length given by the precision, the function would need to access a wide character one past the end of the array. In no case is a partial multibyte character written. (Redundant shift sequences may result if multibyte characters have a state-dependent encoding.)

`p`

The argument shall be a pointer to `void`. The value of the pointer is converted to a sequence of printable characters, in an implementation-defined manner.

`n`

The argument shall be a pointer to signed integer into which is written the number of characters written to the output stream so far by this call to `fprintf`. No argument is converted, but one is consumed. If the conversion specification includes any flags, a field width, or a precision, the behavior is undefined.

`%`

A `%` character is written. No argument is converted. The complete conversion specification shall be `%%`.

If a conversion specification is invalid, the behavior is undefined. If any argument is not the correct type for the corresponding coversion specification, the behavior is undefined.

In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

For `a` and `A` conversions, if `FLT_RADIX` is a power of 2, the value is correctly rounded to a hexadecimal floating number with the given precision.

It is recommended practice that if `FLT_RADIX` is not a power of 2, the result should be one of the two adjacent numbers in hexadecimal floating style with the given precision, with the extra stipulation that the error should have a correct sign for the current rounding direction.

It is recommended practice that for `e`, `E`, `f`, `F`, `g`, and `G` conversions, if the number of significant decimal digits is at most `DECIMAL_DIG`, then the result should be correctly rounded. (For binary-to-decimal conversion, the result format's values are the numbers representable with the given format specifier. The number of significant digits is determined by the format specifier, and in the case of fixed-point conversion by the source value as well.) If the number of significant decimal digits is more than `DECIMAL_DIG` but the source value is exactly representable with `DECIMAL_DIG` digits, then the result should be an exact representation with trailing zeros. Otherwise, the source value is bounded by two adjacent decimal strings $L < U$, both having `DECIMAL_DIG` significant digits; the value of the resultant decimal string D should satisfy $L \leq D \leq U$, with the extra stipulation that the error should have a correct sign for the current rounding direction.

The `fprintf` function returns the number of characters transmitted, or a negative value if an output or encoding error occurred.

The `printf` function is equivalent to `fprintf` with the argument `stdout` interposed before the arguments to `printf`. It returns the number of characters transmitted, or a negative value if an output error occurred.

The `sprintf` function is equivalent to `fprintf`, except that the argument `s` specifies an array into which the generated input is to be written, rather than to a stream. A null character is written at the end of the characters written; it is not counted as part of the returned sum. If copying takes place between objects that overlap, the behavior is undefined. The function returns the number of characters written in the array, not counting the terminating null character.

The `vfprintf` function is equivalent to `fprintf`, with the variable argument list replaced by `arg`, which shall have been initialized by the `va_start` macro (and possibly subsequent `va_arg` calls). The `vfprintf` function does not invoke the `va_end` macro. The function returns the number of characters transmitted, or a negative value if an output error occurred.

The `vprintf` function is equivalent to `printf`, with the variable argument list replaced by `arg`, which shall have been initialized by the `va_start` macro (and possibly subsequent `va_arg` calls). The `vprintf` function does not invoke the `va_end` macro. The function returns the number of characters transmitted, or a negative value if an output error occurred.

The `vsprintf` function is equivalent to `sprintf`, with the variable argument list replaced by `arg`, which shall have been initialized by the `va_start` macro (and possibly subsequent `va_arg` calls). The `vsprintf` function does not invoke the `va_end` macro. If copying takes place between objects that overlap, the behavior is undefined. The function returns the number of characters written into the array, not counting the terminating null character.

21.12 References

pl:C/Czytanie i pisanie do plików[9]

9 `http://pl.wikibooks.org/wiki/C%2FCzytanie%20i%20pisanie%20do%20plik%C3%B3w`

22 Beginning exercises

22.1 Variables

22.1.1 Naming

1. Can a variable name start with a number?
2. Can a variable name start with a typographical symbol(e.g. #, *, _)?
3. Give an example of a C variable name that would *not* work. Why doesn't it work?

Solution

1. No, the name of a variable must begin with a letter (lowercase or uppercase), or an underscore.
2. Only the underscore can be used.
3. for example, **#nm*rt** is not allowed because # and * are not the valid characters for the name of a variable.

```
#include<stdio.h>
main()
{
    int a,b,c,max;
    clrscr();
    printf("\nenter three numbers ");
    scanf("%d %d %d",&a,&b,&c);
    max=a;
    if(max<b)
        max=b;
    if(max<c)
        max=c;
    printf("\nlargest=%d \n",max);
    getch();
}
```

22.1.2 Data Types

1. List at least three data types in C
   a) On your computer, how much memory does each require?
   b) Which ones can be used in place of another? Why?
      i. Are there any limitations on these uses?
      ii. If so, what are they?
      iii. Is it necessary to do anything special to use the alternative?
2. Can the name we use for a data type (e.g. 'int', 'float') be used as a variable?

Solution

- 3 data types : **long int**, **short int**,**float**.

- On my computer :
  - long int : 4 byte
  - short int : 2 bytes
  - float : 4 bytes
- we can not use 'int' or 'float' as a variable's name.

22.1.3 Assignment

1. How would you assign the value 3.14 to a variable called pi?
2. Is it possible to assign an *int* to a *double*?
   a) Is the reverse possible?

Solution

- The standard way of assigning 3.14 to pi is:

```
double pi;
pi=3.14;
```

  - Since pi is a constant, good programming convention dictates to make it unchangeable during runtime. Extra credit if you use one of the following two lines:

```
const float pi = 3.14;
#define pi 3.14
```

- Yes, for example :

```
int a=67;
double b;
b=a;
```

- Yes, but a cast is necessary and the double is truncated:

```
double a=89;
int b;
b=(int) a;
```

22.1.4 Referencing

1. A common mistake for new students is reversing the assignment statement. Suppose you want to assign the value stored in the variable "pi" to another variable, say "pi2":
   a) What is the correct statement?
   b) What is the reverse? Is this a valid C statement (even if it gives incorrect results)?
   c) What if you wanted to assign a constant value (like 3.1415) to "pi2":
      a. What would the correct statement look like?
      b. Would the reverse be a valid or invalid C statement?

Solution

1. `pi2 = pi;`
2. The reverse, `pi = pi2;` is a valid C statement if `pi` is not a constant.
3. **a**. `pi2 = 3.1415;`
   b. The reverse: `3.1415 = pi2;` is not valid since it is impossible to assign a value to a literal.

22.2 Simple I/O

22.2.1 Input

1. scanf() is a very powerful function. Describe some features that make it so versatile.
2. Write the scanf() function call that will read into the variable "var":
   a) a float
   b) an int
   c) a double

Solution

```
scanf("%f",&var);    //read float into var
scanf("%d",&var);    //read int into var
scanf("%lf", &var);  //read double into var
```

22.2.2 String manipulation

1. Write a program that prompts the user for a string, and prints its reverse. **Solution** One possible solution could be:

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    char s[81]; // A string of upto 80 chars + '\0'
    int i;

    puts("Please write a string: ");
    fgets(s, 81, stdin);

    puts("Your sentence in reverse: ");
    for (i= strlen(s)-1; i >= 0; i--)
    {
        if (s[i] == '\n')
            continue; // don't write newline
        else
            putchar(s[i]);
    }
    putchar('\n');
    return 0;
}
```

2. Write a program that prompts the user for a sentence, and prints each word on its own line. **Solution** One possible solution could be:

```
#include <stdio.h>

int main(void)
{
    char s[81], word[81];
    int n= 0, idx= 0;

    puts("Please write a sentence:");
    fgets(s, 81, stdin);

    /* %s matches a sequence of non-whitespace character, which is a
```

```
     *    fair definition of "word" in this context.
     * %n matches nothing, but stores the number of characters that have
     *    been processed. i.e. if s is "Hello, World!", then word and n
     *    will be "Hello," and 6 respectively in the first iteration. In
     *    the second iteration they will be "World!" and 7 (6 chars +
     *    the space in front of the word).
     */
    while ( sscanf(&s[idx], "%s%n", word, &n) > 0 )
    {
        idx += n;
        puts(word);
    }
    return 0;
}
```

22.2.3 Loops

1. Write a function that outputs a right isosceles triangle of height and width n, so $n = 3$ would look like

```
*
**
***
```

Solution One possible solution:

```
void isosceles(int n)
{
    int x,y;
    for (y= 0; y < n; y++)
    {
        for (x= 0; x <= y; x++)
            putchar('*');
        putchar('\n');
    }
}
```

2. Write a function that outputs a sideways triangle of height *2n-1* and width n, so the output for $n = 4$ would be:

```
*
**
***
****
***
**
*
```

Solution One possible solution:

```
void sideways(int n)
{
    int x,y;
    for (y= 0; y < n; y++)
    {
        for (x= 0; x <= y; x++)
            putchar('*');
        putchar('\n');
    }
    for (y= n-1; y > 0; y--)
    {
        for (x= 0; x < y; x++)
            putchar('*');
        putchar('\n');
    }
}
```

or like this (all math)

```
void sideways(int n)
{
    int i=0,j=0;
    for(i=1;i<2*n;i++){
        for(j=1;j<=(n-(abs(n-i)));j++){
            printf("*");
        }
        printf("\n");
    }
}
```

3. Write a function that outputs a right-side-up triangle of height *n* and width *2n-1*; the output for *n = 6* would be:

```
     *
    ***
   *****
  *******
 *********
***********
```

Solution One possible solution:

```
void right_side_up(int n)
{
    int x,y;
    for (y= 1; y <= n; y++)
    {
        for (x= 0; x < n-y; x++)
            putchar(' ');
        for (x= (n-y); x < (n-y)+(2*y-1); x++)
            putchar('*');
        putchar('\n');
    }
}
```

22.3 Program Flow

1. Build a program where control passes from main to four different functions with 4 calls.

2. Now make a while loop in main with the function calls inside it. Ask for input at the beginning of the loop. End the while loop if the user hits Q

3. Next add conditionals to call the functions when the user enters numbers, so 1 goes to function1, 2 goes to function 2, etc.

4. Have function 1 call function a, which calls function b, which calls function c

5. Draw out a diagram of program flow, with arrows to indicate where control goes

22.4 Functions

1. Write a function to check if an integer is negative; the declaration should look like bool is_positive(int i);

2. Write a function to raise a floating point number to an integer power, so for example to when you use it

float a = raise_to_power(2, 3); //a gets 8

float b = raise_to_power(9, 2); //b gets 81

float raise_to_power(float f, int power); //make this your declaration

22.5 Math

1. Write a function to calculate if a number is prime. Return 1 if it is prime and 0 if it is not a prime. **Solution** One possible solution using a naïve primality test[1]:

```
// to compile: gcc -Wall prime.c -lm -o prime

#include <math.h>    // for the square root function sqrt()
#include <stdio.h>

int is_prime(int n);

int main()
{
  printf("Write an integer: ");
  int var;
  scanf("%d", &var);
  if (is_prime(var)==1) {
    printf("A prime\n");
  } else {
    printf("Not a prime\n");
  }
  return 1;
}

int is_prime(int n)
{
  int x;
  int sq= sqrt(n)+1;

  // Checking the trivial cases first
  if ( n < 2 )
    return 0;
  if ( n == 2 || n == 3 )
    return 1;

  // Checking if n is divisible by 2 or odd numbers between 3 and the
  // square root of n.
  if ( n % 2 == 0 )
    return 0;
  for (x= 3; x <= sq; x += 2)
    {
      if ( n % x == 0 )
```

1 http://en.wikipedia.org/wiki/primality%20test

```
        return 0;
    }

  return 1;
}
```

2. Write a function to determine the number of prime numbers below n.
3. Write a function to find the square root by using Newton's method.
4. Write functions to evaluate the trigonometric functions:
5. Try to write a random number generator.
6. Write a function to determine the prime number between 2 and 100:

22.6 Recursion

Merge sort

1. Write a C program to generate a random integer array with a given length n , and sort it recursively using the Merge sort algorithm.

- The merge sort algorithm is a recursive algorithm .

- sorting a one element array is easy.

- sorting two one-element arrays, requires the merge operation. The merge operation looks at two sorted arrays as lists, and compares the head of the list , and which ever head is smaller, this element is put on the sorted list and the head of that list is ticked off, so the next element becomes the head of that list. This is done until one of the lists is exhausted, and the other list is then copied onto the end of the sorted list.

- the recursion occurs, because merging two one-element arrays produces one two-element sorted array, which can be merged with another two-element sorted array produced the same way. This produces a sorted 4 element array, and the same applies for another 4 element sorted array.

- so the basic merge sort, is to check the size of list to be sorted, and if it is greater than one, divide the array into two, and call merge sort again on the two halves. After wards, merge the two halves in a temporary space of equal size, and then copy back the final sorted array onto the original array.

Solution One possible solution , after reading online descriptions of recursive merge sort, e.g. Dasgupta :

```
// to compile: gcc -Wall rmergesort.c -lm -o rmergesort

/*
 ============================================================================
 Name        : rmergesort.c
 Author      : Anon
 Version     : 0.1
 Copyright   : (C)2013 under CC-By-SA 3.0 License
 Description : Recursive Merge Sort, Ansi-style
 =========================================================================
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

//const int MAX = 200;
const int MAX = 20000000;

int *b;

int printOff = 0;

// this debugging print out of the array helps to show
// what is going on.
void printArray(char* label, int* a, int sz) {
        int h = sz/ 2;
        int i;

        if (printOff) return;

        printf("\n%s:\n", label);

        for (i = 0; i < h; ++i ) {

                printf("%d%c", a[i],
                                        // add in a newline every 20 numbers
                                        ( ( i != 0 && i % 20 == 0 )? '\n': ' ' ) );
        }

        printf(" | ");
        for (;i < sz; ++i) {
                printf("%d%c", a[i],
                                        ( ( i != 0 && i % 20 == 0 )? '\n': ' ' ) );
        }

        putchar('\n');


}

void mergesort(int* a, int m ) {

        printArray("BEFORE", a, m);

        if (m > 2) {
                // if greater than 2 elements, then recursive
                mergesort(a, m / 2);
                mergesort(a + m / 2, m - m / 2);


        } else if (m == 2 && a[0] > a[1]) {
                // if exactly 2 elements and need swapping, swap
                int t = a[1];
                a[1] = a[0];
                a[0] = t;
                goto end;

        }

        // 1 or greater than 2 elements which have been recursively sorted

        // divide the array into a left and right array
        // merge into the array b with index l.

        int n = m/2;
        int o = m - n;
        int i = 0, j = n;
        int l = 0;
        // i is left, j is right ;
        // l should equal m the size of the array
        while (i < n) {
                if ( j >= m) {
                        // the right array has finished, so copy the remaining left array
                        for(; i < n; ++i) {
                                b[l++] = a[i];
                        }

                // the merge operation is to copy the smaller current element and
                // increment the index of the parent sub-array.
                } else if(  a[i] < a[j] ) {
                        b[l++] = a[i++];
                } else {
                        b[l++] = a[j++];
                }
        }

        while ( j < m) {
                // copy the remaining right array , if any
                b[l++] = a[j++];
        }

        memcpy(a, b, sizeof(int) * l );

end:
        printArray("AFTER", a, m);

        return;

}

void rand_init(int* a, int n) {
        int i;
        for (i = 0; i < n; ++i ) {

                a[i] = rand() % MAX;

        }
}

int main(void) {
        puts("!!!Hello World!!!"); /* prints !!!Hello World!!! */

//          int N = 20;
//      int N = 1000;
//      int N = 1000000;
        int N = 100000000;  // still can't make a stack overflow on ubuntu,4GB,
 phenom
        printOff = 1;

        int *a;

        b = calloc( N, sizeof(int));

        a = calloc( N, sizeof(int));

        rand_init(a, N);

        mergesort(a, N);

        printOff = 0;

        printArray("LAST", a, N);

        free(a);
        free(b);

        return EXIT_SUCCESS;
}


/* Having failed to translate my concept of non-recursive merge sort,
 * I tackled the easier case of recursive merge sort.
 * The next task is to translate the recursive version to a non-recursive
 * version. This could be done by replacing calls to mergesort, with
 * pushes onto a stack of
 * tuples of ( <array start address>, <number of elements to process> )
 */

/* The central idea of merging, is that two sorted lists can be
 * merged into one sorted list, by comparing the top of each list and
 * moving the lowest valued element onto the end of the new list.
 *  The other list which has the higher valued element keeps its top
 *  element unchanged. When a list is exhausted, copy the remaining other list
 *  onto the end of the new list.
 */

/* The recursive part, is to defer any work in sorting an unsorted list,
 * by dividing it into two lists until there is only 1 or two elements,
 * and if there are two elements, sort them directly by swapping if
 * the first element is larger than the second element.
 *
 * After returning from a recursive call, merge the lists, which will
 * begin with one or two element sorted lists. The result is a sorted list
 * which will be returned to the parent of the recursive call, and can
 * be used for merging.
 */

/* The following is an imaginary discussion about what a programmer
 * might be thinking about when programming:
 *
 * Visualising recursion in terms of a Z80 assembly language, which
 * is similiar to most assembly languages, there is a data stack (DS) and
 * a call stack (CS) pointer, and each recursive call to mergesort
 * pushes the return address , which is the program address of the instruction
 * after the call , onto the stack pointed to by CS and CS is incremented,
 * and the address of the array start and integer which is the subarray length
 * onto the data stack pointed to by DS, which will be incremented twice.
 *
 * If the number of recursive , active calls exceed the allowable space for
either the call stack
 * or the data stack, then the program will crash , or a process space
protection
 * violation interrupt signal will be sent by the CPU, and the interrupt vector
 * for that signal will jump the processor's current instruction pointer to the
 * interrupt handling routine.
 */
```

Binary heaps

2. **Binary heaps** :

- A binary max-heap or min-heap, is an ordered structure where some nodes are guaranteed greater than other nodes, e.g. the parent vs two children. A binary heap can be stored in an array , where ,

- given a position **i** (the parent) , **i*2** is the left child, and **i*2+1** is the right child.

- (C arrays begin at position 0, but 0 * 2 = 0, and 0 *2 + 1= 1, which is incorrect , so start the heap at position 1, or add 1 for parent-to-child calculations, and subtract 1 for child-to-parent calculations).

- try to model this using with a pencil and paper, using 10 random unsorted numbers, and inserting each of them into a "heapsort" array of 10 elements.
- To insert into a heap, **end-add** and **swap-parent** if higher, until parent higher.
- To delete the top of a heap, move **end-to-top**, and **defer-higher-child** or **sift-down** , until no child is higher.
- try it on a pen and paper the numbers 10, 4, 6 ,3 ,5 , 11.

pen-and-paper-solution

- 10, 4, 6, 3, 5, 11 -> 10
- 4, 6,3, 5, 11 -> 10, 4 : 4 is end-added, no swap-parent because 4 < 10.
- 6, 3, 5, 11 -> 10, 4, 6 : 6 is end-added, no swap-parent because 6 < 10.
- 3, 5, 11 -> 10, 4, 6, 3 : 3 is end-added, 3 is position 4 , divide by 2 = 2, 4 at position 2, no swap-parent because 4 > 3.
- 5 , 11 -> 10, 4, 6, 3 , 5 : 5 is end-added , 5 is position 5, divided by 2 = 2, 4 at position 2, swap-parent as 4 < 5; 5 at position 2, no swap-parent because 5 < 10 at position 1.
  - 10 , 5, 6, 3, 4
- 11 -> 10, 5, 6, 3, 4, 11 : 11 is end-added, 11 is position 6, divide by 2 = 3, swap 6 with 11, 11 is position 3, swap 11 with 10, stop as no parent.
  - 11, 5, 10, 3, 4, 6
  - 11 has children 5, 10 ; 5 has children 3 and 4 ; 10 has child 6. Parent always > child.

- the answer was 11, 5, 10, 3, 4 , 6.
- EXERCISE: Now try removing each top element of 11, 5, 10, 3, 4, 6 , using end-to-top and sift-down (or swap-higher-child) to get the numbers

in descending order.

pen-and-paper-solution

- 11 leaves * , 5, 10, 3, 4, 6 -> 6 , 5, 10, 3, 4 -> **sift-down** -> choose greater child 5 (2*n+0) or 10 (2*n+1) -> is 6 > 10 ? no -> swap 10 and 6 ->
  - 10, 5, *6, 3, 4 -> 4 is greatest child as no +1 child. is 6 > 4 ? yes, stop.
- 10 leaves * , 5 , 6 , 3, 4 -> *4, 5, 6, 3 -> is left(0) or right(+1) child greater -> +1 is greater; is 4 > +1 child ? no , swap
  - 6,5, *4, 3 -> *4 has no children so stop.
- 6 leaves *, 5, 4, 3 -> *3, 5, 4 -> +0 child is greater -> is 3 > 5 ? no , so swap -> 5, *3, 4 , *3 has no child so stop. is
- 5 leaves so 3, 4 -> *4, 3 -> +0 child greatest as no right child -> is 4 > 3 ? no , so exit
- 4 leaves 3 .
- 3 leaves *.
- numbers extracted in descending order 11, 10, 6, 5, 4, 3.

- a priority queue allows elements to be inserted with a priority , and extracted according to priority. (This can happen usefully, if the element has a paired structure, one part is the key, and the other part the data. Otherwise, it is just a mechanism for sorting).
- EXERCISE: Using the above technique of insert-back/challenge-parent, and delete-front/last-to-front/defer-higher-child, implement either heap sort or a priority queue.

Dijsktra's algorithm

Dijsktra's algorithm is a searching algorithm using a priority queue. It begins with inserting the start node with a priority value of 0. All other nodes are inserted with priority values of large N. Each node has an adjacency list of other nodes, a current distance to start node, and previous pointer to previous node used to calculate current node. Alternative to an adjacency list, is an adjacency matrix, which needs n x n boolean adjacencies.

The agorithm basically iterates over the priority queue, removing the front node, examining the adjacent nodes, and updating with a distance equal to the sum of the front nodes distance for each adjacent node , and the distance given by the adjacency information for an adjacent node.

After each node's update, the extra operation **"update priority"** is used on that node :

while the node's distance is less than it's parents node (for this priority queue, parents have lesser distances than the children), the node is swapped with the parent.

After this, *while* the node is greater distance than one or more of its children, it is swapped with the least distant child, so the least distant child becomes parent of its greater distant sibling, and parent to the greater distant current node.

With updating the priority, the adjacent node to the current node has a back pointer changed to the current node.

The algorithm ends when the target node becomes the current node removed, and the path to the start node can be recorded in an array by following back pointers, and then doing something like a quick sort partition to reverse the order of the array , to give the shortest path to target node from the start node.

Quick sort

3. Write a C program to recursively sort using the Quick sort partition exchange algorithm.

- you can use the "driver", or the random number test data from Q1. on mergesort. This is "re-use", which is encouraged in general.

- an advantage of reuse is less writing time, debugging time, testing time.

- the concept of partition exchange is that a partition element is (randomly) selected, and every thing that needs sorted is put into 3 equivalance

classes : those elements less than the partition value, the partition element, and everything above (and equal to) the partition element.

- this can be done without allocating more space than one temporary element space for swapping two elements. e.g a temporary integer for integer data.
- However, where the partition element should be using the original array space is not known.
- This is usually implemented with putting the partition on the end of the array to be sorted, and then putting two pointers , one at the start of the array,

and one at the element next to the partition element , and repeatedly scanning the left pointer right, and the right pointer left.

- the left scan stops when an element equal to or greater than the partition is found, and the right scan stops for a smaller element than the partition value,

and these are swapped, which uses the temporary extra space.

- the left scan will always stop if it reaches the partition element , which is the last element; this means the entire array is less than partition value.
- the right scan could reach the first element, if the entire array is greater than the partition , and this needs to be tested for, else the scan doesn't stop.
- the outer loop exits when then left and right pointers cross. Testing for pointer crossing and outer loop exit

should occur before swapping, otherwise the right pointer may be swapping a less-than-partition element previously scanned by the left pointer.

- finally, the partition element needs to be put between the left and right partitions, once the pointers cross.
- At pointer crossing, the left pointer may be stopped at the partition element's last position in the array, and the right pointer not progressed past the

element just before the last element. This happens when all the elements are less than the partition.

- if the right pointer is chosen to swap with the partition, then an incorrect state results where the last element of the left array becomes less than the partition element value.

- if the left pointer is chosen to swap with the partition, then the left array will be less than the partition, and partition will have swapped with an element with value greater than the partition or the partition itself.

- The corner case of quicksorting a 2 element **out-of-order** array has to be examined.

- The left pointer stops on the first **out of order** element. The right pointer begins on the first **out-of-order** element, but the outer loop exits because this is the leftmost element. The partition element is then swapped with the left pointer's first element, and the two elements are now **in order**.

- In the case of a 2 element **in order** array, the leftmost pointer skips the first element which is less than the partition, and stops on the partition. The right pointer begins on the first element and exits because it is the first position. The pointers have crossed so the outer loop exits. The partition swaps with itself, so the in-ordering is preserved.

- After doing a swap, the left pointer should be incremented and right pointer decremented, so the same positions aren't scanned again, because an endless loop can result (possibly when the left pointer exits when the element is equal to or greater than the partition, and the right element is equal to the partition value). One implementation, Sedgewick, starts the pointers with the left pointer minus one and right pointer

the plus one the intended initial scan positions, and use the pre-increment and pre-decrement operators e.g. (++i, --i) .

Solution One possible solution , can be to adapt this word sorting use of quicksort to sort integers. Otherwise , an exercise would be to re-write non-generic qsort functions of qsortsimp, partition, and swap for integers.

```
/*
 * qsortsimp.h
 *
 *  Created on: 17/03/2013
 *      Author: anonymous
 */

#ifndef QSORTSIMP_H_
#define QSORTSIMP_H_
#include <stdlib.h>
void qsortsimp( void* a, size_t elem_sz, int len, int(*cmp) (void*,void*) );
void shutdown_qsortsimp();

#endif /* QSORTSIMP_H_ */

//-----------------------------------------------------------------------------

/*   qsortsimp.c
 *   author : anonymous
 */
#include "qsortsimp.h"
#include<stdlib.h>
#include<string.h>

        static void * swap_buf =0;
        static int bufsz = 0;


void swap( void* a, int i, int j, size_t elem_sz) {
        if (i==j)return;
        if (bufsz == 0 || bufsz < elem_sz) {
                swap_buf = realloc(swap_buf, elem_sz);
                bufsz=elem_sz;
        }

        memcpy( swap_buf, a+i*elem_sz, elem_sz);
        memcpy( a+i*elem_sz, a+j*elem_sz, elem_sz);
        memcpy( a+j*elem_sz, swap_buf, elem_sz);
}

void shutdown_qsortsimp() {
        if (swap_buf) {
                free(swap_buf);
        }
}

int partition( void* a, size_t elem_sz, int len, int (*cmp)(void*,void*) ) {

        int i = -1;
        int j = len-1;
        void* v = a + j * elem_sz;

        for(;;) {
                while( (*cmp)(a + ++i * elem_sz , v  ) < 0);
                while ( (*cmp)(v, a + --j * elem_sz) < 0 ) if (j==0) break ;
                if( i>=j)break;
                swap(a, i, j, elem_sz);
        }
        swap( a, i, len-1, elem_sz);
        return i;

}

void qsortsimp( void* a, size_t elem_sz, int len, int(*cmp) (void*,void*) ) {
        if ( len > 2) {
                int p = partition(a, elem_sz, len, cmp);
                qsortsimp( a, elem_sz, p, cmp);
                qsortsimp( a+(p+1)*elem_sz, elem_sz, len - p -1, cmp );
        }

}



//------------------------------------------------------------------------------

/*
Name        : words_quicksort.c
 Author      : anonymous
 Version     :
 Copyright   :
 Description : quick sort the words in moby dick in C, Ansi-style
 ============================================================================
 */

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include "qsortsimp.h"


void printArray(const char* a[], int n) {
        int i;
        for(i=0; i < n; ++i) {
                        if(i!=0 && i% 5 == 0) {
                                printf("\n");
                        }
                        if (i%1000000 ==0) {
                                fprintf(stderr,"printed %d words\n", i);
                        }
                        printf("%s  ", a[i]);

        }
        printf("\n");

}

const int MAXCHARS=250;
char ** wordlist=0;
int nwords=0;
int remaining_block;
const size_t NWORDS_PER_BLOCK = 1000;

//const char* spaces=" \t\n\r";
//inline isspace(const char ch) {
//        int i=0;
//        while(spaces[i]!='\0') {
//                if(spaces[i++] == ch)
//                        return 1;
//        }
//        return 0;
//}

void freeMem() {
        int i = nwords;
        while(i > 0 ) {
                        free(wordlist[--i]);

        }
        free(wordlist);

}

static char * fname="~/Downloads/books/pg2701-moby-dick.txt";

void getWords() {

        char buffer[MAXCHARS];
        FILE* f = fopen(fname,"r");
        int state=0;
        int ch;
        int i;
        while ((ch=fgetc(f))!=EOF) {
                if (isalnum(ch) && state==0) {
                        state=1;
                        i=0;
                        buffer[i++]=ch;
                } else if (isalnum(ch)  && i < MAXCHARS-1) {
                        buffer[i++]=ch;
                } else if (state == 1) {
                        state =0;
                        buffer[i++]= '\0';
                        char* dynbuf = malloc(i);
                        int j;
                        for(j=0; j < i; ++j) {
                                dynbuf[j] = buffer[j];
                        }
                        i=0;
                        if (wordlist == 0 ) {

                                wordlist = calloc(NWORDS_PER_BLOCK, sizeof(char*));
                                remaining_block = NWORDS_PER_BLOCK;
                        } else if ( remaining_block == 0) {
                                wordlist = realloc(wordlist, (NWORDS_PER_BLOCK + nwords)*
 sizeof(char*));
                                remaining_block = NWORDS_PER_BLOCK;
                                fprintf(stderr,"allocated block %d , nwords = %d\n",
 remaining_block, nwords);

                        }
                        wordlist[nwords++]= dynbuf;
                        --remaining_block;
                }

        }
        fclose(f);


}
void testPrintArray() {

        int i;

        for(i=0; i < nwords;++i) {
                printf("%s | ", wordlist[i]);

        }
        putchar('\n');
        printf("stored %d words. \n",nwords);
}

int cmp_str_1( void* a, void *b) {
                int r = strcasecmp( *((char**)a),*((char**)b));
                return r;
}

int main(int argc, char* argv[]) {
        if (argc > 1) {
                fname = argv[1];
        }
        getWords();
        testPrintArray();

        qsortsimp(wordlist, sizeof(char*), nwords, &cmp_str_1);

        testPrintArray();

        shutdown_qsortsimp();
        freeMem();
        puts("!!!Hello World!!!"); /* prints !!!Hello World!!! */
        return EXIT_SUCCESS;
}
```

et:Programmeerimiskeel C/Harjutused[2] pl:C/Ćwiczenia dla początkujących[3]

2 http://et.wikibooks.org/wiki/Programmeerimiskeel%20C%2FHarjutused
3 http://pl.wikibooks.org/wiki/C%2F%C4%86wiczenia%20dla%20pocz%C4%85tkuj%C4%85cych

23 In-depth C ideas

24 Arrays

Arrays in C act to store related data under a single variable name with an index, also known as a *subscript*. It is easiest to think of an array as simply a list or ordered grouping for variables of the same type. As such, arrays often help a programmer organize collections of data efficiently and intuitively.

Later we will consider the concept of a *pointer*, fundamental to C, which extends the nature of the array (array can be termed as a constant pointer). For now, we will consider just their declaration and their use.

24.1 Arrays

If we want an 2D array of six integers (or numbers), we write in C:

```
int numbers[3][2];
```

For a SIX character array called letters,

```
char letters[6];
```

and so on.

If we wish to initialize as we declare, we write:

```
int point[6]={0,0,1,0,0,0};
```

Though when the array is initialized as in this case, the array dimension may be omitted, and the array will be automatically sized to hold the initial data:

```
int point[]={0,0,1,0,0,0};
```

This is very useful in that the size of the array can be controlled by simply adding or removing initializer elements from the definition without the need to adjust the dimension.

If the dimension is specified, but not all elements in the array are initialized, the remaining elements will contain a value of 0. This is very useful, especially when we have very large arrays.

```
int numbers[2000]={245};
```

The above example sets the first value of the array to 245, and the rest to 0.

If we want to access a variable stored in an array, for example with the above declaration, the following code will store a 1 in the variable `x`

```
int x;
x = point[2];
```

Arrays in C are indexed starting at 0, as opposed to starting at 1. The first element of the array above is `point[0]`. The index to the last value in the array is the array size minus one. In the example above the subscripts run from 0 through 5. C does not guarantee bounds checking on array accesses. The compiler may not complain about the following (though the best compilers do):

```
char y;
int z = 9;
char point[6] = { 1, 2, 3, 4, 5, 6 };
//examples of accessing outside the array. A compile error is not always raised
y = point[15];
y = point[-4];
y = point[z];
```

During program execution, an out of bounds array access does not always cause a run time error. Your program may happily continue after retrieving a value from point[-1]. To alleviate indexing problems, the sizeof() expression is commonly used when coding loops that process arrays.

```
int ix;
short anArray[]= { 3, 6, 9, 12, 15 };

for (ix=0; ix< (sizeof(anArray)/sizeof(short)); ++ix) {
  DoSomethingWith("%d", anArray[ix] );
}
```

Notice in the above example, the size of the array was not explicitly specified. The compiler knows to size it at 5 because of the five values in the initializer list. Adding an additional value to the list will cause it to be sized to six, and because of the sizeof expression in the `for` loop, the code automatically adjusts to this change. Good programming practice is to declare a variable *size* , and store the number of elements in the array in it.

size = sizeof(anArray)/sizeof(short)

C also supports multi dimensional arrays (or, rather, arrays of arrays). The simplest type is a two dimensional array. This creates a rectangular array - each row has the same number of columns. To get a char array with 3 rows and 5 columns we write in C

```
  char two_d[3][5];
```

To access/modify a value in this array we need two subscripts:

```
char ch;
ch = two_d[2][4];
```

or

```
two_d[0][0] = 'x';
```

Similarly, a multi-dimensional array can be initialized like this:

```
int two_d[2][3] = {{ 5, 2, 1 },
                   { 6, 7, 8 }};
```

The amount of columns must be explicitly stated; however, the compiler will find the appropriate amount of rows based on the initializer list.

There are also weird notations possible:

```
int a[100];
int i = 0;
if (a[i]==i[a])
{
  printf("Hello world!\n");
}
```

a[i] and i[a] refer to the same location. (This is explained later in the next Chapter.)

24.2 Strings

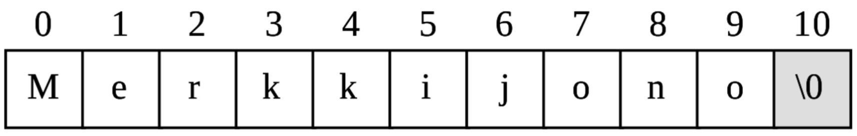


Figure 1 String "Merkkijono" stored in memory

C has no string handling facilities built in; consequently, strings are defined as arrays of characters. C allows a character array to be represented by a character string rather than a list of characters, with the null terminating character automatically added to the end. For example, to store the string "Merkkijono", we would write

```
char string[] = "Merkkijono";
```

or

```
char string[] = {'M', 'e', 'r', 'k', 'k', 'i', 'j', 'o', 'n', 'o', '\0'};
```

In the first example, the string will have a null character automatically appended to the end by the compiler; by convention, library functions expect strings to be terminated by a null character. The latter declaration indicates individual elements, and as such the null terminator needs to be added manually.

Strings do not always have to be linked to an explicit variable. As you have seen already, a string of characters can be created directly as an unnamed string that is used directly (as with the printf functions.)

To create an extra long string, you will have to split the string into multiple sections, by closing the first section with a quote, and recommencing the string on the next line (also starting and ending in a quote):

```
char string[] = "This is a very, very long "
                "string that requires two lines.";
```

While strings may also span multiple lines by putting the backslash character at the end of the line, this method is deprecated.

There is a useful library of string handling routines which you can use by including another header file.

```
#include <string.h>  //new header file
```

This standard string library will allow various tasks to be performed on strings, and is discussed in the Strings[1] chapter.

et:Programmeerimiskeel C/Massiivid[2] it:C/Vettori e puntatori/Vettori[3] pl:C/Tablice[4] fi:C/Taulukot[5]

1 Chapter 27 on page 205
2 http://et.wikibooks.org/wiki/Programmeerimiskeel%20C%2FMassiivid
3 http://it.wikibooks.org/wiki/C%2FVettori%20e%20puntatori%2FVettori
4 http://pl.wikibooks.org/wiki/C%2FTablice
5 http://fi.wikibooks.org/wiki/C%2FTaulukot

25 Pointers and arrays

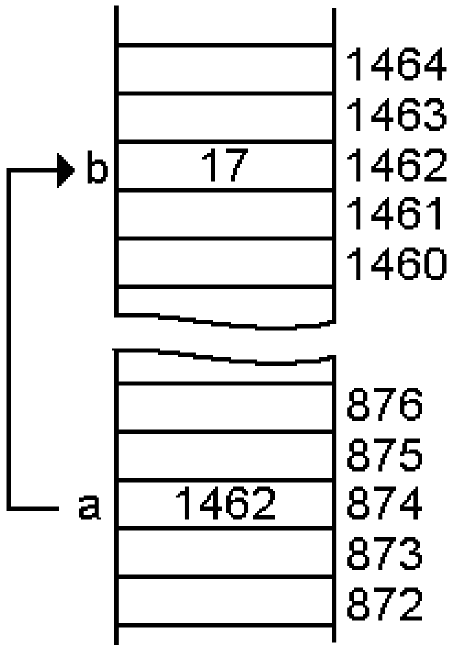


Figure 2 Pointer *a* pointing variable *b*. Note that *b* stores number, whereas *a* stores address of *b* in memory (1462)

A **pointer**[1] is a value that designates the address (i.e., the location in memory), of some value. There are four fundamental things you need to know about pointers:

- How to declare them
- How to assign to them
- How to reference the value to which the pointer points (known as *dereferencing*) and
- How they relate to arrays

We'll also discuss the relationship of pointers with text strings and the more advanced concept of function pointers.

Pointers are variables that hold a memory location. One can access the value of the variable pointed to using the dereferencing operator '`*`'.

Pointers can reference any data type, even functions.

1 `http://en.wikipedia.org/wiki/Pointer%20%28computing%29`

The vast majority of arrays in C are simple lists, also called "1 dimensional arrays". We will briefly cover multi-dimensional arrays in a later chapter[2].

25.1 Declaring pointers

Consider the following snippet of code which declares two pointers:

<source lang="c" line start=1>

```
struct MyStruct {
    int   m_aNumber;
    float num2;
};

int          * pJ2;
struct MyStruct * pAnItem;
```

</source>

Lines 1-4 define a structure[3]. Line 6 declares a variable which points to an `int`, and line 7 declares a variable which points to something with structure MyStruct. So to declare a variable as something which points to some type, rather than contains some type, the asterisk (`*`) is placed before the variable name.

In the following, line 1 declares `var1` as a pointer to a long and `var2` as a long and not a pointer to a long. In line 2, `p3` is declared as a pointer to a pointer to an int.

```
<source lang="c" line start=1>
```

long * var1, var2;

```
int   ** p3;
```

</source>

Pointer types are often used as parameters to function calls. The following shows how to declare a function which uses a pointer as an argument. Since C passes function arguments by value, in order to allow a function to modify a value from the calling routine, a pointer to the value must be passed. Pointers to structures are also used as function arguments even when nothing in the struct will be modified in the function. This is done to avoid copying the complete contents of the structure onto the stack. More about pointers as function arguments later.

2 Chapter 30.1 on page 231
3 Chapter 28.1.2 on page 222

```
int MyFunction( struct MyStruct *pStruct );
```

25.2 Assigning values to pointers

So far we've discussed how to declare pointers. The process of assigning values to pointers is next. To assign a pointer the address of a variable, the & or 'address of' operator is used.

```
int   myInt;
int  *pPointer;
struct MyStruct   dvorak;
struct MyStruct  *pKeyboard;

pPointer = &myInt;
pKeyboard = &dvorak;
```

Here, pPointer will now reference myInt and pKeyboard will reference dvorak.

Pointers can also be assigned to reference dynamically allocated memory. The malloc() and calloc() functions are often what are used to do this.

```
#include <stdlib.h>
/* ... */
struct MyStruct *pKeyboard;
/* ... */
pKeyboard = malloc(sizeof *pKeyboard);
```

The malloc function returns a pointer to dynamically allocated memory (or NULL if unsuccessful). The size of this memory will be appropriately sized to contain the MyStruct structure.

The following is an example showing one pointer being assigned to another and of a pointer being assigned a return value from a function.

```
static struct MyStruct val1, val2, val3, val4;

struct MyStruct *ASillyFunction( int b )
{
   struct MyStruct *myReturn;

   if (b == 1) myReturn = &val1;
   else if (b==2) myReturn = &val2;
   else if (b==3) myReturn = &val3;
   else myReturn = &val4;

   return myReturn;
}

struct MyStruct *strPointer;
int     *c, *d;
int     j;

c = &j;                           /* pointer assigned using & operator */
d = c;                            /* assign one pointer to another     */
strPointer = ASillyFunction( 3 ); /* pointer returned from a function. */
```

When returning a pointer from a function, do not return a pointer that points to a value that is local to the function or that is a pointer to a function argument. Pointers to local variables become invalid when the function exits. In the above function, the value returned points to a static variable. Returning a pointer to dynamically allocated memory is also valid.

25.3 Pointer dereferencing

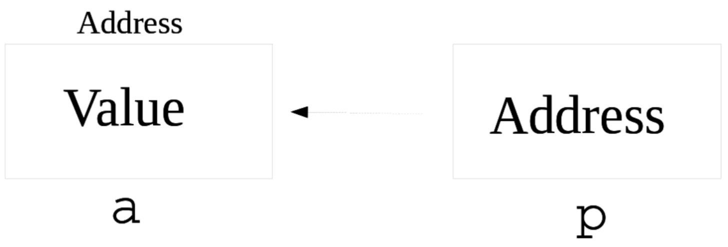


Figure 3 The pointer `p` points to the variable `a`.

To access a value to which a pointer points, the `*` operator is used. Another operator, the `->` operator is used in conjunction with pointers to structures. Here's a short example.

```
int   c, d;
int   *pj;
struct MyStruct astruct;
struct MyStruct *bb;

c   = 10;
pj  = &c;             /* pj points to c */
d   = *pj;            /* d is assigned the value to which pj points, 10 */
pj  = &d;             /* now points to d */
*pj = 12;             /* d is now 12 */

bb = &astruct;
(*bb).m_aNumber = 3;  /* assigns 3 to the m_aNumber member of astruct */
bb->num2 = 44.3;      /* assigns 44.3 to the num2 member of astruct   */
*pj = bb->m_aNumber;  /* eqivalent to d = astruct.m_aNumber;          */
```

The expression `bb->m_aNumber` is entirely equivalent to `(*bb).m_aNumber`. They both access the `m_aNumber` element of the structure pointed to by `bb`. There is one more way of dereferencing a pointer, which will be discussed in the following section.

When dereferencing a pointer that points to an invalid memory location, an error often occurs which results in the program terminating. The error is often reported as a segmentation error. A common cause of this is failure to initialize a pointer before trying to dereference it.

C is known for giving you just enough rope to hang yourself, and pointer dereferencing is a prime example. You are quite free to write code that accesses memory outside that which you have explicitly requested from the system. And many times, that memory may appear as available to your program due to the vagaries of system memory allocation. However, even if 99 executions allow your program to run without fault, that 100th execution may be the time when your "memory pilfering" is caught by the system and the program fails. Be careful to ensure that your pointer offsets are within the bounds of allocated memory!

The declaration `void *somePointer;` is used to declare a pointer of some nonspecified type. You can assign a value to a void pointer, but you must cast the variable to point to some specified type before you can dereference it. Pointer arithmetic is also not valid with `void *` pointers.

25.4 Pointers and Arrays

Up to now, we've carefully been avoiding discussing arrays in the context of pointers. The interaction of pointers and arrays can be confusing but here are two fundamental statements about it:

- A variable declared as an array of some type acts as a pointer to that type. When used by itself, it points to the first element of the array.
- A pointer can be indexed like an array name.

The first case often is seen to occur when an array is passed as an argument to a function. The function declares the parameter as a pointer, but the actual argument may be the name of an array. The second case often occurs when accessing dynamically allocated memory. Let's look at examples of each. In the following code, the call to calloc() effectively allocates an array of struct MyStruct items.

```
float KrazyFunction( struct MyStruct *parm1, int p1size, int bb )
{
  int ix; //declaring an integer variable//
  for (ix=0; ix<p1size; ix++) {
     if (parm1[ix].m_aNumber == bb )
         return parm1[ix].num2;
  }
  return 0.0f;
}

/* ... */
struct MyStruct myArray[4];
#define MY_ARRAY_SIZE (sizeof(myArray)/sizeof(*myArray))
float v3;
struct MyStruct *secondArray;
int   someSize;
int   ix;
/* initialization of myArray ... */
v3 = KrazyFunction( myArray, MY_ARRAY_SIZE, 4 );
/* ... */
secondArray = calloc( someSize, sizeof(myArray) );
for (ix=0; ix<someSize; ix++) {
    secondArray[ix].m_aNumber = ix *2;
    secondArray[ix].num2 = .304 * ix * ix;
}
```

Pointers and array names can pretty much be used interchangeably. There are exceptions. You cannot assign a new pointer value to an array name. The array name will always point to the first element of the array. In the function `KrazyFunction` above, you could however assign a new value to parm1, as it is just a pointer to the first element of myArray. It is also valid for a function to return a pointer to one of the array elements from an array passed as an argument to a function. A function should never return a pointer to a local variable, even though the compiler will probably not complain.

When declaring parameters to functions, declaring an array variable without a size is equivalent to declaring a pointer. Often this is done to emphasize the fact that the pointer variable will be used in a manner equivalent to an array.

```
/* two equivalent function definitions */
int LittleFunction( int *paramN );
int LittleFunction( int paramN[] );
```

Now we're ready to discuss pointer arithmetic. You can add and subtract integer values to/from pointers. If myArray is declared to be some type of array, the expression `*(myArray+j)`, where j is an integer, is equivalent to `myArray[j]`. So for instance in the above example where we had the expression secondArray[i].num2, we could have written that as `*(secondArray+i).num2` or more simply `(secondArray+i)->num2`.

Note that for addition and subtraction of integers and pointers, the value of the pointer is not adjusted by the integer amount, but is adjusted by the amount multiplied by the size (in bytes) of the type to which the pointer refers. One pointer may also be subtracted from another, provided they point to elements of the same array (or the position just beyond the end of the array). If you have a pointer that points to an element of an array, the index of the element is the result when the array name is subtracted from the pointer. Here's an example.

```
struct MyStruct someArray[20];
struct MyStruct *p2;
int idx;

.
/* array initialization .. */
.
for (p2 = someArray; p2 < someArray+20;  ++p2) {
   if (p2->num2 > testValue) break;
}
idx = p2 - someArray;
```

You may be wondering how pointers and multidimensional arrays interact. Let's look at this a bit in detail. Suppose A is declared as a two dimensional array of floats (`float A[D1][D2];`) and that pf is declared a pointer to a float. If pf is initialized to point to A[0][0], then *(pf+1) is equivalent to A[0][1] and *(pf+D2) is equivalent to A[1][0]. The elements of the array are stored in row-major order.

```
float A[6][8];
float *pf;
pf = &A[0][0];
*(pf+1) = 1.3;   /* assigns 1.3 to A[0][1] */
*(pf+8) = 2.3;   /* assigns 2.3 to A[1][0] */
```

Let's look at a slightly different problem. We want to have a two dimensional array, but we don't need to have all the rows the same length. What we do is declare an array of pointers. The second line below declares A as an array of pointers. Each pointer points to a float. Here's some applicable code:

```
float  linearA[30];
float *A[6];
```

```
A[0] = linearA;             /*  5 - 0 = 5 elements in row  */
A[1] = linearA + 5;         /* 11 - 5 = 6 elements in row  */
A[2] = linearA + 11;        /* 15 - 11 = 4 elements in row */
A[3] = linearA + 15;        /* 21 - 15 = 6 elements        */
A[4] = linearA + 21;        /* 25 - 21 = 4 elements        */
A[5] = linearA + 25;        /* 30 - 25 = 5 elements        */

A[3][2] = 3.66;         /* assigns 3.66 to linearA[17];     */
A[3][-3] = 1.44;        /* refers to linearA[12];
                           negative indices are sometimes useful. But avoid
using them as much as possible. */
```

We also note here something curious about array indexing. Suppose myArray is an array and idx is an integer value. The expression myArray[idx] is equivalent to idx[myArray]. The first is equivalent to *(myArray+idx), and the second is equivalent to *(idx+myArray). These turn out to be the same, since the addition is commutative.

Pointers can be used with preincrement or post decrement, which is sometimes done within a loop, as in the following example. The increment and decrement applies to the pointer, not to the object to which the pointer refers. In other words, *pArray++ is equivalent to *(pArray++).

```
long  myArray[20];
long  *pArray;
int  i;

/* Assign values to the entries of myArray */
pArray = myArray;
for (i=0; i<10; ++i) {
  *pArray++ = 5 + 3*i + 12*i*i;
  *pArray++ = 6 + 2*i + 7*i*i;
}
```

25.5 Pointers in Function Arguments

Often we need to invoke a function with an argument that is itself a pointer. In many instances, the variable is itself a parameter for the current function and may be a pointer to some type of structure. The ampersand character is not needed in this circumstance to obtain a pointer value, as the variable is itself a pointer. In the example below, the variable `pStruct`, a pointer, is a parameter to function `FunctTwo,` and is passed as an argument to `FunctOne`. The second parameter to `FunctOne` is an int. Since in function `FunctTwo, mValue` is a pointer to an int, the pointer must first be dereferenced using the * operator, hence the second argument in the call is `*mValue`. The third parameter to function `FunctOne` is a pointer to a long. Since `pAA` is itself a pointer to a long, no ampersand is needed when it is used as the third argument to the function.

```
int FunctOne( struct SomeStruct *pValue, int iValue, long *lValue )
{
   /*  do some stuff ... */
   return 0;
}
int FunctTwo( struct someStruct *pStruct, int *mValue )
{
   int j;
   long  AnArray[25];
   long *pAA;
   pAA = &AnArray[13];
   j = FunctOne( pStruct, *mValue, pAA );
   return j;
}
```

25.6 Pointers and Text Strings

Historically, text strings in C have been implemented as arrays of characters, with the last byte in the string being a zero, or the null character '\0'. Most C implementations come with a standard library of functions for manipulating strings. Many of the more commonly used functions expect the strings to be null terminated strings of characters. To use these functions requires the inclusion of the standard C header file "string.h".

A statically declared, initialized string would look similar to the following:

```
static const char *myFormat = "Total Amount Due: %d";
```

The variable `myFormat` can be viewed as an array of 21 characters. There is an implied null character ('\0') tacked on to the end of the string after the 'd' as the 21st item in the array. You can also initialize the individual characters of the array as follows:

```
static const char myFlower[] = { 'P', 'e', 't', 'u', 'n', 'i', 'a', '\0' };
```

An initialized array of strings would typically be done as follows:

```
static const char *myColors[] = {
   "Red", "Orange", "Yellow", "Green", "Blue", "Violet" };
```

The initilization of an especially long string can be split across lines of source code as follows.

```
static char *longString = "Hello. My name is Rudolph and I work as a reindeer "
  "around Christmas time up at the North Pole.  My boss is a really swell guy."
  " He likes to give everybody gifts.";
```

The library functions that are used with strings are discussed in a later chapter.

25.7 Pointers to Functions

C also allows you to create pointers to functions. Pointers to functions can get rather messy. Declaring a typedef to a function pointer generally clarifies the code. Here's an example that uses a function pointer, and a void * pointer to implement what's known as a callback. The `DoSomethingNice` function invokes a caller supplied function `TalkJive` with caller data. Note that `DoSomethingNice` really doesn't know anything about what `dataPointer`refers to.

```
typedef  int (*MyFunctionType)( int, void *);      /* a typedef for a function
pointer */

#define THE_BIGGEST 100

int DoSomethingNice( int aVariable, MyFunctionType aFunction, void *dataPointer
)
{
    int rv = 0;
    if (aVariable < THE_BIGGEST) {
       /* invoke function through function pointer (old style) */
       rv = (*aFunction)(aVariable, dataPointer );
     } else {
         /* invoke function through function pointer (new style) */
       rv = aFunction(aVariable, dataPointer );
    };
    return rv;
}

typedef struct {
    int    colorSpec;
    char   *phrase;
} DataINeed;

int TalkJive( int myNumber, void *someStuff )
{
    /* recast void * to pointer type specifically needed for this function */
    DataINeed *myData = someStuff;
    /* talk jive. */
    return 5;
}

static DataINeed  sillyStuff = { BLUE, "Whatcha talkin 'bout Willis?" };

/* ... */
 DoSomethingNice( 41, &TalkJive,  &sillyStuff );
```

Some versions of C may not require an ampersand preceding the `TalkJive` argument in the `DoSomethingNice` call. Some implementations may require specifically casting the argument to the `MyFunctionType` type, even though the function signature exacly matches that of the typedef.

Function pointers can be useful for implementing a form of polymorphism in C. First one declares a structure having as elements function pointers for the various operations to that can be specified polymorphically. A second base object structure containing a pointer to the previous structure is also declared. A class is defined by extending the second structure with the data specific for the class, and static variable of the type of the first structure, containing the addresses of the functions that are associated with the class. This type of polymorphism is used in the standard library when file I/O functions are called.

A similar mechanism can also be used for implementing a state machine in C. A structure is defined which contains function pointers for handling events that may occur within state, and for functions to be invoked upon entry to and exit from the state. An instance of this structure corresponds to a state. Each state is initialized with pointers to functions appropriate for the state. The current state of the state machine is in effect a pointer to one of these states. Changing the value of the current state pointer effectively changes the current state. When some event occurs, the appropriate function is called through a function pointer in the current state.

25.8 Practical use of function pointers in C

Function pointers are mainly used to reduce the complexity of switch statement. Example with switch statement:

```
#include <stdio.h>
int add(int a, int b);
int sub(int a, int b);
int mul(int a, int b);
int div(int a, int b);
int main()
{
    int i, result;
    int a=10;
    int b=5;
    printf("Enter the value between 0 and 3 : ");
    scanf("%d",&i);
    switch(i)
    {
        case 0:  result = add(a,b); break;
        case 1:  result = sub(a,b); break;
        case 2:  result = mul(a,b); break;
        case 3:  result = div(a,b); break;
    }
}
int add(int i, int j)
{
    return (i+j);
}
int sub(int i, int j)
{
    return (i-j);
}
 int mul(int i, int j)
{
    return (i*j);
}
int div(int i, int j)
{
    return (i/j);
}
```

Without using a switch statement:

```
#include <stdio.h>
int add(int a, int b);
int sub(int a, int b);
int mul(int a, int b);
int div(int a, int b);
int (*oper[4])(int a, int b) = {add, sub, mul, div};
int main()
{
    int i,result;
    int a=10;
    int b=5;
    printf("Enter the value between 0 and 3 : ");
    scanf("%d",&i);
    result = oper[i](a,b);
}
int add(int i, int j)
{
    return (i+j);
}
int sub(int i, int j)
```

```
{
    return (i-j);
}
int mul(int i, int j)
{
    return (i*j);
}
int div(int i, int j)
{
    return (i/j);
}
```

Function pointers may be used to create a struct member function:

```
typedef struct
{
    int (*open)(void);
    void (*close)(void);
    int (*register)(void);
} device;

int my_device_open(void)
{
    /* ... */
}

void my_device_close(void)
{
    /* ... */
}

void register_device(void)
{
    /* ... */
}

device create(void)
{
    device my_device;
    my_device.open = my_device_open;
    my_device.close = my_device_close;
    my_device.register = register_device;
    my_device.register();
    return my_device;
}
```

Use to implement this pointer (following code must be placed in library).

```
static struct device_data
{
    /* ... here goes data of structure ... */
};

static struct device_data obj;

typedef struct
{
    int (*open)(void);
    void (*close)(void);
    int (*register)(void);
} device;

static struct device_data create_device_data(void)
{
    struct device_data my_device_data;
    /* ... here goes constructor ... */
    return my_device_data;
}

/* here I omit the my_device_open, my_device_close and register_device functions
 */

device create_device(void)
{
    device my_device;
    my_device.open = my_device_open;
    my_device.close = my_device_close;
    my_device.register = register_device;
    my_device.register();
    return my_device;
}
```

25.9 Examples of pointer constructs

Below are some example constructs which may aid in creating your pointer.

```
int i;          // integer variable 'i'
int *p;         // pointer 'p' to an integer
int a[];        // array 'a' of integers
int f();        // function 'f' with return value of type integer
int **pp;       // pointer 'pp' to a pointer to an integer
int (*pa)[];    // pointer 'pa' to an array of integer
int (*pf)();    // pointer 'pf' to a function with return value integer
int *ap[];      // array 'ap' of pointers to an integer
int *fp();      // function 'fp' which returns a pointer to an integer
int ***ppp;     // pointer 'ppp' to a pointer to a pointer to an integer
int (**ppa)[]; // pointer 'ppa' to a pointer to an array of integers
int (**ppf)(); // pointer 'ppf' to a pointer to a function with return value of
type integer
int *(*pap)[]; // pointer 'pap' to an array of pointers to an integer
int *(*pfp)(); // pointer 'pfp' to function with return value of type pointer
to an integer
int **app[];    // array of pointers 'app' that point to pointers to integer
values
int (*apa[])[];// array of pointers 'apa' to arrays of integers
int (*apf[])();// array of pointers 'apf' to functions with return values of
type integer
int ***fpp();   // function 'fpp' which returns a pointer to a pointer to a
pointer to an int
int (*fpa())[];// function 'fpa' with return value of a pointer to array of
integers
int (*fpf())();// function 'fpf' with return value of a pointer to function
which returns an integer
```

25.10 sizeof

The sizeof operator is often used to refer to the size of a static array declared earlier in the same function.

To find the end of an array (example from wikipedia:Buffer overflow[4]):

4 http://en.wikipedia.org/wiki/Buffer%20overflow

```
/* better.c - demonstrates one method of fixing the problem */

#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
  char buffer[10];
  if (argc < 2)
  {
    fprintf(stderr, "USAGE: %s string\n", argv[0]);
    return 1;
  }
  strncpy(buffer, argv[1], sizeof(buffer));
  buffer[sizeof(buffer) - 1] = '\0';
  return 0;
}
```

To iterate over every element of an array, use

```
 #define NUM_ELEM(x) (sizeof (x) / sizeof (*(x)))

 for( i = 0; i < NUM_ELEM(array); i++ )
 {
     /* do something with array[i] */
     ;
 }
```

Note that the `sizeof` operator only works on things defined earlier in the same function. The compiler replaces it with some fixed constant number. In this case, the `buffer` was declared as an array of 10 char's earlier in the same function, and the compiler replaces `sizeof(buffer)` with the number 10 at compile time (equivalent to us hard-coding 10 into the code in place of `sizeof(buffer)`). The information about the length of `buffer` is not actually stored anywhere in memory (unless we keep track of it separately) and cannot be programmatically obtained at run time from the array/pointer itself.

Often a function needs to know the size of an array it was given -- an array defined in some other function. For example,

```
/* broken.c - demonstrates a flaw */

#include <stdio.h>
#include <string.h>
#define NUM_ELEM(x) (sizeof (x) / sizeof (*(x)))

int sum( int input_array[] ){
  int sum_so_far = 0;
  int i;
  for( i = 0; i < NUM_ELEM(input_array); i++ ) // WON'T WORK -- input_array
 wasn't defined in this function.
  {
    sum_so_far += input_array[i];
  };
  return( sum_so_far );
}

int main(int argc, char *argv[])
{
  int left_array[] = { 1, 2, 3 };
  int right_array[] = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };
  int the_sum = sum( left_array );
  printf( "the sum of left_array is: %d", the_sum );
  the_sum = sum( right_array );
  printf( "the sum of right_array is: %d", the_sum );

  return 0;
}
```

Unfortunately, (in C and C++) the length of the array cannot be obtained from an array passed in at run time, because (as mentioned above) the size of an array is not stored anywhere. The compiler always replaces sizeof with a constant. This sum() routine needs to handle more than just one constant length of an array.

There are some common ways to work around this fact:

- Write the function to require, for each array parameter, a "length" parameter (which has type "size_t"). (Typically we use sizeof at the point where this function is called).
- Use of a convention, such as a null-terminated string[5] to mark the end of the array.
- Instead of passing raw arrays, pass a structure that includes the length of the array (such as ".length") as well as the array (or a pointer to the first element); similar to the `string` or `vector` classes in C++.

```
/* fixed.c - demonstrates one work-around */

#include <stdio.h>
#include <string.h>
#define NUM_ELEM(x) (sizeof (x) / sizeof (*(x)))

int sum( int input_array[], size_t length ){
  int sum_so_far = 0;
  int i;
  for( i = 0; i < length; i++ )
  {
    sum_so_far += input_array[i];
  };
  return( sum_so_far );
}

int main(int argc, char *argv[])
{
  int left_array[] = { 1, 2, 3, 4 };
  int right_array[] = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };
  int the_sum = sum( left_array, NUM_ELEM(left_array) ); // works here, because
 left_array is defined in this function
  printf( "the sum of left_array is: %d", the_sum );
  the_sum = sum( right_array, NUM_ELEM(right_array) ); // works here, because
 right_array is defined in this function
  printf( "the sum of right_array is: %d", the_sum );

  return 0;
}
```

It's worth mentioning that sizeof operator has two variations: `sizeof (`*`type`*`)` (for instance: `sizeof (int)` or `sizeof (struct some_structure)`) and `sizeof` *`expression`* (for instance: `sizeof some_variable.some_field` or `sizeof 1`).

5 http://en.wikipedia.org/wiki/null-terminated%20string

25.11 External Links

- C Reference Card (ANSI)[6]
- "Common Pointer Pitfalls"[7] by Dave Marshall
- "Further insights into size_t"[8] by Dan Saks 2007
- "Pointer Fun with Binky"[9]

de:C-Programmierung: Zeiger[10] it:C/Vettori e puntatori/Interscambiabilità tra puntatori e vettori[11] pl:C/Wskaźniki[12]

6 `http://www.digilife.be/quickreferences/QRC/C%20Reference%20Card%20(ANSI)%202.2.pdf`
7 `http://www.cs.cf.ac.uk/Dave/C/node10.html#SECTION001080000000000000000`
8 `http://www.embedded.com/columns/programmingpointers/201803576`
9 `http://en.wikibooks.org/wiki/%3AFile%3APointer%20Fun%20with%20Binky%20%28C%29.ogg`
10 `http://de.wikibooks.org/wiki/C-Programmierung%3A%20Zeiger`
11 `http://it.wikibooks.org/wiki/C%2FVettori%20e%20puntatori%2FInterscambiabilit%C3%A0%20tra%20puntatori%20e%20vettori`
12 `http://pl.wikibooks.org/wiki/C%2FWska%C5%BAniki`

26 Memory management

In C, you have already considered creating variables for use in the program. You have created some arrays for use, but you may have already noticed some limitations:

- the size of the array must be known beforehand
- the size of the array cannot be changed in the duration of your program

Dynamic memory allocation in C is a way of circumventing these problems.

26.1 EXAMPLE

```
#include <stdlib.h>
void *calloc(size_t nmemb, size_t size);
void free(void *ptr);
void *malloc(size_t size);
void *realloc(void *ptr, size_t size);
```

The C function `malloc` is the means of implementing dynamic memory allocation. It is defined in stdlib.h or malloc.h, depending on what operating system you may be using. Malloc.h contains only the definitions for the memory allocation functions and not the rest of the other functions defined in stdlib.h. Usually you will not need to be so specific in your program, and if both are supported, you should use <stdlib.h>, since that is ANSI C, and what we will use here.

The corresponding call to release allocated memory back to the operating system is `free`.

When dynamically allocated memory is no longer needed, `free` should be called to release it back to the memory pool. Overwriting a pointer that points to dynamically allocated memory can result in that data becoming inaccessible. If this happens frequently, eventually the operating system will no longer be able to allocate more memory for the process. Once the process exits, the operating system is able to free all dynamically allocated memory associated with the process.

Let's look at how dynamic memory allocation can be used for arrays.

Normally when we wish to create an array we use a declaration such as

```
int array[10];
```

Recall `array` can be considered a pointer which we use as an array. We specify the length of this array is 10 `ints`. After `array[0]`, nine other integers have space to be stored consecutively.

Sometimes it is not known at the time the program is written how much memory will be needed for some data. In this case we would want to dynamically allocate required memory after the program has started executing. To do this we only need to declare a pointer, and invoke `malloc` when we wish to make space for the elements in our array, *or*, we can tell `malloc` to make space when we first initialize the array. Either way is acceptable and useful.

We also need to know how much an int takes up in memory in order to make room for it; fortunately this is not difficult, we can use C's builtin `sizeof` operator. For example, if `sizeof(int)` yields 4, then one `int` takes up 4 bytes. Naturally, `2*sizeof(int)` is how much memory we need for 2 `int`s, and so on.

So how do we `malloc` an array of ten `int`s like before? If we wish to declare and make room in one hit, we can simply say

```
int *array = malloc(10*sizeof(int));
```

We only need to declare the pointer; `malloc` gives us some space to store the 10 `int`s, and returns the pointer to the first element, which is assigned to that pointer.

Important note! `malloc` does *not* initialize the array; this means that the array may contain random or unexpected values! Like creating arrays without dynamic allocation, the programmer must initialize the array with sensible values before using it. Make sure you do so, too. (*See later the function `memset` for a simple method.)*

It is not necessary to immediately call `malloc` after declaring a pointer for the allocated memory. Often a number of statements exist between the declaration and the call to `malloc`, as follows:

```
int *array = NULL;
printf("Hello World!!!");
/* more statements */
array = malloc(10*sizeof(int)); /* delayed allocation */
/* use the array */
```

26.1.1 Error checking

When we want to use `malloc`, we have to be mindful that the pool of memory available to the programmer is *finite*. As such, we can conceivably run out of memory! In this case, `malloc` will return `NULL`. In order to stop the program crashing from having no more memory to use, one should always check that malloc has not returned `NULL` before attempting to use the memory; we can do this by

```
int *pt = malloc(3 * sizeof(int));
if(pt == NULL)
{
   fprintf(stderr, "Out of memory, exiting\n");
   exit(1);
}
```

Of course, suddenly quitting as in the above example is not always appropriate, and depends on the problem you are trying to solve and the architecture you are programming for. For example, if the program is a small, non critical application that's running on a desktop quitting may be appropriate. However if the program is some type of editor running on a desktop, you may want to give the operator the option of saving his tediously entered information instead of just exiting the program. A memory allocation failure in an embedded processor, such as might be in a washing machine, could cause an automatic reset of the machine. For this reason, many embedded systems designers avoid dynamic memory allocation altogether.

26.2 The `calloc` function

The `calloc` function allocates space for an array of items and initilizes the memory to zeros. The call `mArray = calloc( count, sizeof(struct V))` allocates `count` objects, each of whose size is sufficient to contain an instance of the structure `struct V`. The space is initialized to all bits zero. The function returns either a pointer to the allocated memory or, if the allocation fails, `NULL`.

26.3 The `realloc` function

```
void * realloc ( void * ptr, size_t size );
```

The `realloc` function changes the size of the object pointed to by `ptr` to the size specified by `size`. The contents of the object shall be unchanged up to the lesser of the new and old sizes. If the new size is larger, the value of the newly allocated portion of the object is indeterminate. If `ptr` is a null pointer, the `realloc` function behaves like the `malloc` function for the specified size. Otherwise, if `ptr` does not match a pointer earlier returned by the `calloc`, `malloc`, or `realloc` function, or if the space has been deallocated by a call to the `free` or `realloc` function, the behavior is undefined. If the space cannot be allocated, the object pointed to by `ptr` is unchanged. If `size` is zero and `ptr` is not a null pointer, the object pointed to is freed. The `realloc` function returns either a null pointer or a pointer to the possibly moved allocated object.

26.4 The `free` function

Memory that has been allocated using `malloc`, `realloc`, or `calloc` must be released back to the system memory pool once it is no longer needed. This is done to avoid perpetually allocating more and more memory, which could result in an eventual memory allocation failure. Memory that is not released with `free` is however released when the current program terminates on most operating systems. Calls to `free` are as in the following example.

```
int *myStuff = malloc( 20 * sizeof(int));
if (myStuff != NULL)
{
   /* more statements here */
   /* time to release myStuff */
   free( myStuff );
}
```

26.4.1 free with recursive data structures

It should be noted that `free` is neither intelligent nor recursive. The following code that depends on the recursive application of free to the internal variables of a struct[1] does not work.

```
typedef struct BSTNode
{
   int value;
   struct BSTNode* left;
   struct BSTNode* right;
} BSTNode;

// Later: ...

BSTNode* temp = (BSTNode*) calloc(1, sizeof(BSTNode));
temp->left = (BSTNode*) calloc(1, sizeof(BSTNode));

// Later: ...

free(temp); // WRONG! don't do this!
```

The statement "`free(temp);`" will **not** free `temp->left`, causing a memory leak.

Because C does not have a garbage collector, C programmers are responsible for making sure there is a `free()` exactly once for each time there is a `malloc()`. If a tree has been allocated one node at a time, then it needs to be freed one node at a time.

26.4.2 Don't free undefined pointers

Furthermore, using `free` when the pointer in question was never allocated in the first place often crashes or leads to mysterious bugs further along.

To avoid this problem, always initialize pointers when they are declared. Either use `malloc` at the point they are declared (as in most examples in this chapter), or set them to `NULL` when they are declared (as in the "delayed allocation" example in this chapter). [2]

26.5 References

1 Chapter 28 on page 221
2 "Bug 478901 ... libpng-1.2.34 and earlier might free undefined pointers" ^{`https://bugzilla.mozilla.org/show_bug.cgi?id=478901`}

27 Strings

A **string** in C is merely an array of characters. The length of a string is determined by a terminating null character: `'\0'`. So, a string with the contents, say, `"abc"` has four characters: `'a'`, `'b'`, `'c'`, and the terminating null character.

The terminating null character has the value zero.

27.1 Syntax

In C, string constants (literals) are surrounded by double quotes ("), e.g. "Hello world!" and are compiled to an array of the specified `char` values with an additional null terminating character (0-valued) code to mark the end of the string. The type of a string constant is `char *`.

String literals may not directly in the source code contain embedded newlines or other control characters, or some other characters of special meaning in string.

To include such characters in a string, the backslash escapes may be used, like this:

Escape	**Meaning**
`\\`	Literal backslash
`\"`	Double quote
`\'`	Single quote
`\n`	Newline (line feed)
`\r`	Carriage return
`\b`	Backspace
`\t`	Horizontal tab
`\f`	Form feed
`\a`	Alert (bell)
`\v`	Vertical tab
`\?`	Question mark (used to escape trigraphs[1])
`\`*nnn*	Character with octal value *nnn*
`\x`*hh*	Character with hexadecimal value *hh*

27.1.1 Wide character strings

C supports wide character strings, defined as arrays of the type `wchar_t`, 16-bit (at least) values. They are written with an L before the string like this

1 http://en.wikibooks.org/wiki/..%2FC%20trigraph%2F

```
wchar_t *p = L"Hello world!";
```

This feature allows strings where more than 256 different possible characters are needed (although also variable length `char` strings can be used). They end with a zero-valued `wchar_t`. These strings are not supported by the `<string.h>` functions. Instead they have their own functions, declared in `<wchar.h>`.

27.1.2 Character encodings

What character encoding the `char` and `wchar_t` represent is not specified by the C standard, except that the value 0x00 and 0x0000 specify the end of the string and not a character. It the input and output code which are directly affected by the character encoding. Other code should not be too affected. The editor should also be able to handle the encoding if strings shall be able to written in the source code.

There are three major types of encodings:

- One byte per character. Normally based on ASCII. There is a limit of 255 different characters plus the zero termination character.
- Variable length `char` strings, which allows many more than 255 different characters. Such strings are written as normal `char`-based arrays. These encodings are normally ASCII-based and examples are UTF-8[2] or Shift JIS[3].
- Wide character strings. They are arrays of `wchar_t` values. UTF-16[4] is the most common such encoding, and it is also variable-length, meaning that a character can be two `wchar_t`.

27.2 The `<string.h>` Standard Header

Because programmers find raw strings cumbersome to deal with, they wrote the code in the `<string.h>` library. It represents not a concerted design effort but rather the accretion of contributions made by various authors over a span of years.

First, three types of functions exist in the string library:

- the `mem` functions manipulate sequences of arbitrary characters without regard to the null character;
- the `str` functions manipulate null-terminated sequences of characters;
- the `strn` functions manipulate sequences of non-null characters.

27.2.1 The more commonly-used string functions

The nine most commonly used functions in the string library are:

- `strcat` - concatenate two strings

2 `http://en.wikibooks.org/wiki/UTF-8`
3 `http://en.wikibooks.org/wiki/Shift%20JIS`
4 `http://en.wikibooks.org/wiki/UTF-16`

- strchr - string scanning operation
- strcmp - compare two strings
- strcpy - copy a string
- strlen - get string length
- strncat - concatenate one string with part of another
- strncmp - compare parts of two strings
- strncpy - copy part of a string
- strrchr - string scanning operation

The strcat function

`char *strcat(char * restrict s1, const char * restrict s2);`

Some people recommend using `strncat()` *or* `strlcat()` *instead of strcat, in order to avoid buffer overflow.*

The `strcat()` function shall append a copy of the string pointed to by `s2` (including the terminating null byte) to the end of the string pointed to by `s1`. The initial byte of `s2` overwrites the null byte at the end of `s1`. If copying takes place between objects that overlap, the behavior is undefined. The function returns `s1`.

This function is used to attach one string to the end of another string. It is imperative that the first string (`s1`) have the space needed to store both strings.

Example:

```
   #include <stdio.h>
   #include <string.h>
   ...
   static const char *colors[] =
{"Red","Orange","Yellow","Green","Blue","Purple" };
   static const char *widths[] = {"Thin","Medium","Thick","Bold" };
   ...
   char penText[20];
   ...
   int penColor = 3, penThickness = 2;
   strcpy(penText, colors[penColor]);
   strcat(penText, widths[penThickness]);
   printf("My pen is %s\n", penText); // prints 'My pen is GreenThick'
```

Before calling `strcat()`, the destination must currently contain a null terminated string or the first character must have been initialized with the null character (e.g. `penText[0] = '\0';`).

The following is a public-domain implementation of `strcat`:

```
#include <string.h>
/* strcat */
char *(strcat)(char *restrict s1, const char *restrict s2)
{
    char *s = s1;
    /* Move s so that it points to the end of s1.  */
    while (*s != '\0')
        s++;
    /* Copy the contents of s2 into the space at the end of s1.  */
    strcpy(s, s2);
    return s1;
}
```

The `strchr` function

`char *strchr(const char *s, int c);`

The `strchr()` function shall locate the first occurrence of `c` (converted to a `char`) in the string pointed to by `s`. The terminating null byte is considered to be part of the string. The function returns the location of the found character, or a null pointer if the character was not found.

This function is used to find certain characters in strings.

At one point in history, this function was named `index`. The `strchr` name, however cryptic, fits the general pattern for naming.

The following is a public-domain implementation of `strchr`:

```
#include <string.h>
/* strchr */
char *(strchr)(const char *s, int c)
{
    /* Scan s for the character.  When this loop is finished,
       s will either point to the end of the string or the
       character we were looking for.  */
    while (*s != '\0' && *s != (char)c)
        s++;
    return ( (*s == c) ? (char *) s : NULL );
}
```

The `strcmp` function

`int strcmp(const char *s1, const char *s2);`

A rudimentary form of string comparison is done with the strcmp() function. It takes two strings as arguments and returns a value less than zero if the first is lexographically less than the second, a value greater than zero if the first is lexographically greater than the second, or zero if the two strings are equal. The comparison is done by comparing the coded (ascii) value of the chararacters, character by character.

This simple type of string comparison is nowadays generally considered unacceptable when sorting lists of strings. More advanced algorithms exist that are capable of producing lists in dictionary sorted order. They can also fix problems such as strcmp() considering the string "Alpha2" greater than "Alpha12". (In the previous example, "Alpha2" compares greater than "Alpha12" because '2' comes after '1' in the character set.) What we're saying is, don't use this `strcmp()` alone for general string sorting in any commercial or professional code.

The `strcmp()` function shall compare the string pointed to by `s1` to the string pointed to by `s2`. The sign of a non-zero return value shall be determined by the sign of the difference between the values of the first pair of bytes (both interpreted as type `unsigned char`) that differ in the strings being compared. Upon completion, `strcmp()` shall return an integer greater than, equal to, or less than 0, if the string pointed to by `s1` is greater than, equal to, or less than the string pointed to by `s2`, respectively.

Since comparing pointers by themselves is not practically useful unless one is comparing pointers within the same array, this function lexically compares the strings that two pointers point to.

This function is useful in comparisons, e.g.

```
if (strcmp(s, "whatever") == 0) /* do something */
    ;
```

The collating sequence used by `strcmp()` is equivalent to the machine's native character set. The only guarantee about the order is that the digits from `'0'` to `'9'` are in consecutive order.

The following is a public-domain implementation of `strcmp`:

```
#include <string.h>
/* strcmp */
int (strcmp)(const char *s1, const char *s2)
{
    unsigned char uc1, uc2;
    /* Move s1 and s2 to the first differing characters
       in each string, or the ends of the strings if they
       are identical.  */
    while (*s1 != '\0' && *s1 == *s2) {
        s1++;
        s2++;
    }
    /* Compare the characters as unsigned char and
       return the difference.  */
    uc1 = (*(unsigned char *) s1);
    uc2 = (*(unsigned char *) s2);
    return ((uc1 < uc2) ? -1 : (uc1 > uc2));
}
```

The `strcpy` function

`char *strcpy(char *restrict s1, const char *restrict s2);`

Some people recommend always using `strncpy()` *instead of strcpy, to avoid buffer overflow.*

The `strcpy()` function shall copy the C string pointed to by `s2` (including the terminating null byte) into the array pointed to by `s1`. If copying takes place between objects that overlap, the behavior is undefined. The function returns `s1`. There is no value used to indicate an error: if the arguments to `strcpy()` are correct, and the destination buffer is large enough, the function will never fail.

Example:

```
#include <stdio.h>
#include <string.h>
/* ... */
static const char *penType="round";
/* ... */
```

```
char penText[20];
/* ... */
strcpy(penText, penType);
```

Important: You must ensure that the destination buffer (`s1`) is able to contain all the characters in the source array, including the terminating null byte. Otherwise, `strcpy()` will overwrite memory past the end of the buffer, causing a buffer overflow, which can cause the program to crash, or can be exploited by hackers to compromise the security of the computer.

The following is a public-domain implementation of `strcpy`:

```
#include <string.h>
/* strcpy */
char *(strcpy)(char *restrict s1, const char *restrict s2)
{
    char *dst = s1;
    const char *src = s2;
    /* Do the copying in a loop.  */
    while ((*dst++ = *src++) != '\0')
        ;               /* The body of this loop is left empty. */
    /* Return the destination string.  */
    return s1;
}
```

The `strlen` function

`size_t strlen(const char *s);`

The `strlen()` function shall compute the number of bytes in the string to which `s` points, not including the terminating null byte. It returns the number of bytes in the string. No value is used to indicate an error.

The following is a public-domain implementation of `strlen`:

```
#include <string.h>
/* strlen */
size_t (strlen)(const char *s)
{
    const char *p = s;
    /* Loop over the data in s.  */
    while (*p != '\0')
        p++;
    return (size_t)(p - s);
}
```

The `strncat` function

`char *strncat(char *restrict s1, const char *restrict s2, size_t n);`

The `strncat()` function shall append not more than `n` bytes (a null byte and bytes that follow it are not appended) from the array pointed to by `s2` to the end of the string pointed to by `s1`. The initial byte of `s2` overwrites the null byte at the end of `s1`. A terminating null byte is always appended to the result. If copying takes place between objects that overlap, the behavior is undefined. The function returns `s1`.

The following is a public-domain implementation of strncat:

```
#include <string.h>
/* strncat */
char *(strncat)(char *restrict s1, const char *restrict s2, size_t n)
{
    char *s = s1;
    /* Loop over the data in s1.  */
    while (*s != '\0')
        s++;
    /* s now points to s1's trailing null character, now copy
       up to n bytes from s1 into s stopping if a null character
       is encountered in s2.
       It is not safe to use strncpy here since it copies EXACTLY n
       characters, NULL padding if necessary.  */
    while (n != 0 && (*s = *s2++) != '\0') {
        n--;
        s++;
    }
    if (*s != '\0')
        *s = '\0';
    return s1;
}
```

The strncmp function

```
int strncmp(const char *s1, const char *s2, size_t n);
```

The strncmp() function shall compare not more than n bytes (bytes that follow a null byte are not compared) from the array pointed to by s1 to the array pointed to by s2. The sign of a non-zero return value is determined by the sign of the difference between the values of the first pair of bytes (both interpreted as type unsigned char) that differ in the strings being compared. See strcmp for an explanation of the return value.

This function is useful in comparisons, as the strcmp function is.

The following is a public-domain implementation of strncmp:

```
#include <string.h>
/* strncmp */
int (strncmp)(const char *s1, const char *s2, size_t n)
{
    unsigned char uc1, uc2;
    /* Nothing to compare?  Return zero.  */
    if (n == 0)
        return 0;
    /* Loop, comparing bytes.  */
    while (n-- > 0 && *s1 == *s2) {
        /* If we've run out of bytes or hit a null, return zero
           since we already know *s1 == *s2.  */
        if (n == 0 || *s1 == '\0')
            return 0;
        s1++;
        s2++;
    }
    uc1 = (*(unsigned char *) s1);
    uc2 = (*(unsigned char *) s2);
    return ((uc1 < uc2) ? -1 : (uc1 > uc2));
}
```

The `strncpy` function

`char *strncpy(char *restrict s1, const char *restrict s2, size_t n);`

The `strncpy()` function shall copy not more than `n` bytes (bytes that follow a null byte are not copied) from the array pointed to by `s2` to the array pointed to by `s1`. If copying takes place between objects that overlap, the behavior is undefined. If the array pointed to by `s2` is a string that is shorter than `n` bytes, null bytes shall be appended to the copy in the array pointed to by `s1`, until `n` bytes in all are written. The function shall return s1; no return value is reserved to indicate an error.

It is possible that the function will **not** return a null-terminated string, which happens if the `s2` string is longer than `n` bytes.

The following is a public-domain version of `strncpy`:

```
#include <string.h>
/* strncpy */
char *(strncpy)(char *restrict s1, const char *restrict s2, size_t n)
{
    char *dst = s1;
    const char *src = s2;
    /* Copy bytes, one at a time.  */
    while (n > 0) {
        n--;
        if ((*dst++ = *src++) == '\0') {
            /* If we get here, we found a null character at the end
               of s2, so use memset to put null bytes at the end of
               s1.  */
            memset(dst, '\0', n);
            break;
        }
    }
    return s1;
}
```

The `strrchr` function

`char *strrchr(const char *s, int c);`

`strrchr` is similar to `strchr`, except the string is searched right to left.

The `strrchr()` function shall locate the last occurrence of `c` (converted to a `char`) in the string pointed to by `s`. The terminating null byte is considered to be part of the string. Its return value is similar to `strchr`'s return value.

At one point in history, this function was named `rindex`. The `strrchr` name, however cryptic, fits the general pattern for naming.

The following is a public-domain implementation of `strrchr`:

```
#include <string.h>
/* strrchr */
char *(strrchr)(const char *s, int c)
{
    const char *last = NULL;
    /* If the character we're looking for is the terminating null,
       we just need to look for that character as there's only one
       of them in the string.  */
    if (c == '\0')
        return strchr(s, c);
    /* Loop through, finding the last match before hitting NULL.  */
    while ((s = strchr(s, c)) != NULL) {
        last = s;
        s++;
    }
    return (char *) last;
}
```

27.2.2 The less commonly-used string functions

The less-used functions are:

- `memchr` - Find a byte in memory
- `memcmp` - Compare bytes in memory
- `memcpy` - Copy bytes in memory
- `memmove` - Copy bytes in memory with overlapping areas
- `memset` - Set bytes in memory
- `strcoll` - Compare bytes according to a locale-specific collating sequence
- `strcspn` - Get the length of a complementary substring
- `strerror` - Get error message
- `strpbrk` - Scan a string for a byte
- `strspn` - Get the length of a substring
- `strstr` - Find a substring
- `strtok` - Split a string into tokens
- `strxfrm` - Transform string

Copying functions

The `memcpy` function

```
void *memcpy(void * restrict s1, const void * restrict s2, size_t n);
```

The `memcpy()` function shall copy `n` bytes from the object pointed to by `s2` into the object pointed to by `s1`. If copying takes place between objects that overlap, the behavior is undefined. The function returns `s1`.

Because the function does not have to worry about overlap, it can do the simplest copy it can.

The following is a public-domain implementation of `memcpy`:

```
#include <string.h>
/* memcpy */
void *(memcpy)(void * restrict s1, const void * restrict s2, size_t n)
{
    char *dst = s1;
    const char *src = s2;
    /* Loop and copy.  */
    while (n-- != 0)
        *dst++ = *src++;
    return s1;
}
```

The memmove function

```
void *memmove(void *s1, const void *s2, size_t n);
```

The `memmove()` function shall copy `n` bytes from the object pointed to by `s2` into the object pointed to by `s1`. Copying takes place as if the `n` bytes from the object pointed to by `s2` are first copied into a temporary array of `n` bytes that does not overlap the objects pointed to by `s1` and `s2`, and then the `n` bytes from the temporary array are copied into the object pointed to by `s1`. The function returns the value of `s1`.

The easy way to implement this without using a temporary array is to check for a condition that would prevent an ascending copy, and if found, do a descending copy.

The following is a public-domain, though not completely portable, implementation of `memmove`:

```
#include <string.h>
/* memmove */
void *(memmove)(void *s1, const void *s2, size_t n)
{
   /* note: these don't have to point to unsigned chars */
   char *p1 = s1;
   const char *p2 = s2;
   /* test for overlap that prevents an ascending copy */
   if (p2 < p1 && p1 < p2 + n) {
       /* do a descending copy */
       p2 += n;
       p1 += n;
       while (n-- != 0)
           *--p1 = *--p2;
   } else
       while (n-- != 0)
           *p1++ = *p2++;
   return s1;
}
```

Comparison functions

The memcmp function

```
int memcmp(const void *s1, const void *s2, size_t n);
```

The `memcmp()` function shall compare the first `n` bytes (each interpreted as `unsigned char`) of the object pointed to by `s1` to the first `n` bytes of the object pointed to by `s2`. The sign of a non-zero return value shall be determined by the sign of the difference between the values of the first pair of bytes (both interpreted as type `unsigned char`) that differ in the objects being compared.

The following is a public-domain implementation of `memcmp`:

```
#include <string.h>
/* memcmp */
int (memcmp)(const void *s1, const void *s2, size_t n)
{
    const unsigned char *us1 = (const unsigned char *) s1;
    const unsigned char *us2 = (const unsigned char *) s2;
    while (n-- != 0) {
        if (*us1 != *us2)
            return (*us1 < *us2) ? -1 : +1;
        us1++;
        us2++;
    }
    return 0;
}
```

The strcoll and strxfrm functions

`int strcoll(const char *s1, const char *s2);`

`size_t strxfrm(char *s1, const char *s2, size_t n);`

The ANSI C Standard specifies two locale-specific comparison functions.

The `strcoll` function compares the string pointed to by `s1` to the string pointed to by `s2`, both interpreted as appropriate to the `LC_COLLATE` category of the current locale. The return value is similar to `strcmp`.

The `strxfrm` function transforms the string pointed to by `s2` and places the resulting string into the array pointed to by `s1`. The transformation is such that if the `strcmp` function is applied to the two transformed strings, it returns a value greater than, equal to, or less than zero, corresponding to the result of the `strcoll` function applied to the same two original strings. No more than `n` characters are placed into the resulting array pointed to by `s1`, including the terminating null character. If `n` is zero, `s1` is permitted to be a null pointer. If copying takes place between objects that overlap, the behavior is undefined. The function returns the length of the transformed string.

These functions are rarely used and nontrivial to code, so there is no code for this section.

Search functions

The memchr function

`void *memchr(const void *s, int c, size_t n);`

The `memchr()` function shall locate the first occurrence of `c` (converted to an `unsigned char`) in the initial `n` bytes (each interpreted as `unsigned char`) of the object pointed to by `s`. If `c` is not found, `memchr` returns a null pointer.

The following is a public-domain implementation of `memchr`:

```
#include <string.h>
/* memchr */
void *(memchr)(const void *s, int c, size_t n)
{
    const unsigned char *src = s;
    unsigned char uc = c;
    while (n-- != 0) {
        if (*src == uc)
            return (void *) src;
        src++;
    }
    return NULL;
}
```

The `strcspn`, `strpbrk`, and `strspn` functions

`size_t strcspn(const char *s1, const char *s2);`

`char *strpbrk(const char *s1, const char *s2);`

`size_t strspn(const char *s1, const char *s2);`

The `strcspn` function computes the length of the maximum initial segment of the string pointed to by `s1` which consists entirely of characters **not** from the string pointed to by `s2`.

The `strpbrk` function locates the first occurrence in the string pointed to by `s1` of any character from the string pointed to by `s2`, returning a pointer to that character or a null pointer if not found.

The `strspn` function computes the length of the maximum initial segment of the string pointed to by `s1` which consists entirely of characters from the string pointed to by `s2`.

All of these functions are similar except in the test and the return value.

The following are public-domain implementations of `strcspn`, `strpbrk`, and `strspn`:

```
#include <string.h>
/* strcspn */
size_t (strcspn)(const char *s1, const char *s2)
{
    const char *sc1;
    for (sc1 = s1; *sc1 != '\0'; sc1++)
        if (strchr(s2, *sc1) != NULL)
            return (sc1 - s1);
    return sc1 - s1;            /* terminating nulls match */
}
```

```
#include <string.h>
/* strpbrk */
char *(strpbrk)(const char *s1, const char *s2)
{
    const char *sc1;
    for (sc1 = s1; *sc1 != '\0'; sc1++)
        if (strchr(s2, *sc1) != NULL)
            return (char *)sc1;
    return NULL;                /* terminating nulls match */
}
```

```
#include <string.h>
/* strspn */
size_t (strspn)(const char *s1, const char *s2)
{
    const char *sc1;
    for (sc1 = s1; *sc1 != '\0'; sc1++)
        if (strchr(s2, *sc1) == NULL)
            return (sc1 - s1);
    return sc1 - s1;            /* terminating nulls don't match */
}
```

The `strstr` function

`char *strstr(const char *haystack, const char *needle);`

The `strstr()` function shall locate the first occurrence in the string pointed to by `haystack` of the sequence of bytes (excluding the terminating null byte) in the string pointed to by `needle`. The function returns the pointer to the matching string in `haystack` or a null pointer if a match is not found. If `needle` is an empty string, the function returns `haystack`.

The following is a public-domain implementation of `strstr`:

```
#include <string.h>
/* strstr */
char *(strstr)(const char *haystack, const char *needle)
{
    size_t needlelen;
    /* Check for the null needle case.  */
    if (*needle == '\0')
        return (char *) haystack;
    needlelen = strlen(needle);
    for (; (haystack = strchr(haystack, *needle)) != NULL; haystack++)
        if (strncmp(haystack, needle, needlelen) == 0)
            return (char *) haystack;
    return NULL;
}
```

The `strtok` function

`char *strtok(char *restrict s1, const char *restrict delimiters);`

A sequence of calls to `strtok()` breaks the string pointed to by `s1` into a sequence of tokens, each of which is delimited by a byte from the string pointed to by `delimiters`. The first call in the sequence has `s1` as its first argument, and is followed by calls with a null pointer as their first argument. The separator string pointed to by `delimiters` may be different from call to call.

The first call in the sequence searches the string pointed to by `s1` for the first byte that is not contained in the current separator string pointed to by `delimiters`. If no such byte is found, then there are no tokens in the string pointed to by `s1` and `strtok()` shall return a null pointer. If such a byte is found, it is the start of the first token.

The `strtok()` function then searches from there for a byte (or multiple, consecutive bytes) that is contained in the current separator string. If no such byte is found, the current token extends to the end of the string pointed to by `s1`, and subsequent searches for a token shall return a null pointer. If such a byte is found, it is overwritten by a null byte, which terminates the current token. The `strtok()` function saves a pointer to the following byte, from which the next search for a token shall start.

Each subsequent call, with a null pointer as the value of the first argument, starts searching from the saved pointer and behaves as described above.

The `strtok()` function need not be reentrant. A function that is not required to be reentrant is not required to be thread-safe.

Because the `strtok()` function must save state between calls, and you could not have two tokenizers going at the same time, the Single Unix Standard defined a similar function, `strtok_r()`, that does not need to save state. Its prototype is this:

`char *strtok_r(char *s, const char *delimiters, char **lasts);`

The `strtok_r()` function considers the null-terminated string `s` as a sequence of zero or more text tokens separated by spans of one or more characters from the separator string `delimiters`. The argument lasts points to a user-provided pointer which points to stored information necessary for `strtok_r()` to continue scanning the same string.

In the first call to `strtok_r()`, `s` points to a null-terminated string, `delimiters` to a null-terminated string of separator characters, and the value pointed to by `lasts` is ignored. The `strtok_r()` function shall return a pointer to the first character of the first token, write a null character into `s` immediately following the returned token, and update the pointer to which `lasts` points.

In subsequent calls, `s` is a null pointer and `lasts` shall be unchanged from the previous call so that subsequent calls shall move through the string `s`, returning successive tokens until no tokens remain. The separator string `delimiters` may be different from call to call. When no token remains in `s`, a NULL pointer shall be returned.

The following public-domain code for `strtok` and `strtok_r` codes the former as a special case of the latter:

```
#include <string.h>
/* strtok_r */
char *(strtok_r)(char *s, const char *delimiters, char **lasts)
{
    char *sbegin, *send;
    sbegin = s ? s : *lasts;
    sbegin += strspn(sbegin, delimiters);
    if (*sbegin == '\0') {
        *lasts = "";
        return NULL;
    }
    send = sbegin + strcspn(sbegin, delimiters);
    if (*send != '\0')
        *send++ = '\0';
    *lasts = send;
    return sbegin;
}
/* strtok */
char *(strtok)(char *restrict s1, const char *restrict delimiters)
{
    static char *ssave = "";
    return strtok_r(s1, delimiters, &ssave);
}
```

Miscellaneous functions

These functions do not fit into one of the above categories.

The `memset` function

```
void *memset(void *s, int c, size_t n);
```

The `memset()` function converts `c` into `unsigned char`, then stores the character into the first `n` bytes of memory pointed to by `s`.

The following is a public-domain implementation of `memset`:

```
#include <string.h>
/* memset */
void *(memset)(void *s, int c, size_t n)
{
    unsigned char *us = s;
    unsigned char uc = c;
    while (n-- != 0)
        *us++ = uc;
    return s;
}
```

The `strerror` function
`char *strerror(int errorcode);`

This function returns a locale-specific error message corresponding to the parameter. Depending on the circumstances, this function could be trivial to implement, but this author will not do that as it varies.

The Single Unix System Version 3 has a variant, `strerror_r`, with this prototype:

`int strerror_r(int errcode, char *buf, size_t buflen);`

This function stores the message in `buf`, which has a length of size `buflen`.

27.3 Examples

To determine the number of characters in a string, the `strlen()` function is used:

```
#include <stdio.h>
#include <string.h>
...
int length, length2;
char *turkey;
static char *flower= "begonia";
static char *gemstone="ruby ";

length = strlen(flower);
printf("Length = %d\n", length); // prints 'Length = 7'
length2 = strlen(gemstone);

turkey = malloc( length + length2 + 1);
if (turkey) {
  strcpy( turkey, gemstone);
  strcat( turkey, flower);
  printf( "%s\n", turkey); // prints 'ruby begonia'
  free( turkey );
}
```

Note that the amount of memory allocated for 'turkey' is one plus the sum of the lengths of the strings to be concatenated. This is for the terminating null character, which is not counted in the lengths of the strings.

27.3.1 Exercises

1. The string functions use a lot of looping constructs. Is there some way to portably unravel the loops?
2. What functions are possibly missing from the library as it stands now?

27.4 Further reading

- A Little C Primer/C String Function Library[5]
- C++ Programming/Code/IO/Streams/string[6]
- Because so many functions in the standard `string.h` library are vulnerable to buffer overflow errors, some people[7] recommend avoiding the `string.h` library and "C style strings" and instead using a dynamic string API, such as the ones listed in the String library comparison[8].
- There's a tiny public domain concat() function, which will allocate memory and safely concatenate any number of strings in portable C/C++ code[9]

pl:C/Napisy[10] pt:Programar em C/Strings[11]

5 `http://en.wikibooks.org/wiki/A%20Little%20C%20Primer%2FC%20String%20Function%20Library`
6 `http://en.wikibooks.org/wiki/C%2B%2B%20Programming%2FCode%2FIO%2FStreams%2Fstring`
7 `http://www.and.org/vstr/security`
8 `http://www.and.org/vstr/comparison`
9 `http://openwall.info/wiki/people/solar/software/public-domain-source-code/concat`
10 `http://pl.wikibooks.org/wiki/C%2FNapisy`
11 `http://pt.wikibooks.org/wiki/Programar%20em%20C%2FStrings`

28 Complex types

In the section C types[1] we looked at some basic types. However **C complex types** allow us greater flexibility in managing data in our C program.

28.1 Data structures

A data structure ("struct") contains multiple pieces of data. Each piece of data (called a "member") can be accessed by the name of the variable, followed by a '.', then the name of the member. (Another way to access a member is using the member operator '->'). The member variables of a struct can be of any data type and can even be an array or a pointer.

28.1.1 Pointers

Pointers are variables that don't hold the actual data. Instead they point to the memory location of some other variable. For example,

```
    int *pointer = &variable;
```

defines a pointer to an `int`, and also makes it point to the particular integer contained in `variable`.

The '*' is what makes this an integer pointer. To make the pointer point to a different integer, use the form

```
    pointer = &sandwiches;
```

Where & is the *address of* operator. Often programmers set the value of the pointer to NULL (a standard macro defined as 0 or (void*)0) like this:

```
    pointer = NULL;
```

This tells us that the pointer isn't currently pointing to any real location.

Additionally, to dereference (access the thing being pointed at) the pointer, use the form:

1 `http://en.wikibooks.org/wiki/C%20Programming%2FTypes`

```
    value = *pointer;
```

28.1.2 Structs

A data structure contains multiple pieces of data. One defines a data structure using the `struct` keyword. For example,

```
    struct mystruct
    {
        int int_member;
        double double_member;
        char string_member[25];
    } variable;
```

`variable` is an instance of `mystruct`. You can omit it from the end of the **`struct`** declaration and declare it later using:

```
struct mystruct variable;
```

It is often common practice to make a *type synonym* so we don't have to type "struct mystruct" all the time. C allows us the possibility to do so using a **`typedef`** statement, which aliases a type:

```
typedef struct
{
  ...
} Mystruct;
```

The **`struct`** itself has no name (by the absence of a name on the first line), but it is aliased as `Mystruct`. Then you can use

```
Mystruct structure;
```

Note that it is commonplace, and good style to capitalize the **first letter** of a type synonym. However in the actual definition we need to give the struct a *tag* so we can refer to it: we may have a *recursive data structure* of some kind. For trees or chained lists, we need a pointer to the same data type in the struct. During compilation, the type synonym is not known to the compiler and there will be an error. To avoid this, it is necessary to let the compiler know the name right from the start (Note that the **struct** keyword is used *only* inside the structure! After the declaration, the compiler *knows* that the type synonym refers to a **struct**):

```
typedef struct Mystruct
{
  ...
```

```
    struct Mystruct * pMystruct
} Mystruct;
```

28.1.3 Unions

The definition of a union is similar to that of a struct. The difference between the two is that in a struct, the members occupy different areas of memory, but in a union, the members occupy the same area of memory. Thus, in the following type, for example:

```
union {
    int i;
    double d;
} u;
```

The programmer can access either `u.i` or `u.d`, but not both at the same time. Since `u.i` and `u.d` occupy the same area of memory, modifying one modifies the value of the other, sometimes in unpredictable ways.

The size of a union is the size of its largest member.

28.2 Type modifiers

For "register", "volatile", "auto" and "extern", see ../Variables#Other_Modifiers[2].

de:C-Programmierung: Komplexe Datentypen[3] pl:C/Typy złożone[4]

2 Chapter 12.9 on page 54
3 `http://de.wikibooks.org/wiki/C-Programmierung%3A%20Komplexe%20Datentypen`
4 `http://pl.wikibooks.org/wiki/C%2FTypy%20z%C5%82o%C5%BCone`

29 Networking in UNIX

Network programming under UNIX is relatively simple in C.

This guide assumes you already have a good general idea about C, UNIX and networks.

29.1 A simple client

To start with, we'll look at one of the simplest things you can do: initialize a stream connection and receive a message from a remote server.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>

#define MAXRCVLEN 500
#define PORTNUM 2343

int main(int argc, char *argv[])
{
   char buffer[MAXRCVLEN + 1]; /* +1 so we can add null terminator */
   int len, mysocket;
   struct sockaddr_in dest;

   mysocket = socket(AF_INET, SOCK_STREAM, 0);

   memset(&dest, 0, sizeof(dest));                /* zero the struct */
   dest.sin_family = AF_INET;
   dest.sin_addr.s_addr = inet_addr("127.0.0.1"); /* set destination IP number
 */
   dest.sin_port = htons(PORTNUM);                /* set destination port number
 */

   connect(mysocket, (struct sockaddr *)&dest, sizeof(struct sockaddr));

   len = recv(mysocket, buffer, MAXRCVLEN, 0);

   /* We have to null terminate the received data ourselves */
   buffer[len] = '\0';

   printf("Received %s (%d bytes).\n", buffer, len);

   close(mysocket);
   return EXIT_SUCCESS;
}
```

This is the very bare bones of a client; in practice, we would check every function that we call for failure, however, error checking has been left out for clarity.

As you can see, the code mainly revolves around `dest` which is a struct of type `sockaddr_in`. This struct stores information about the machine we want to connect to.

```
mysocket = socket(AF_INET, SOCK_STREAM, 0);
```

The `socket()` function tells our OS that we want a file descriptor for a socket which we can use for a network stream connection; what the parameters mean is mostly irrelevant for now.

```
memset(&dest, 0, sizeof(dest));                /* zero the struct */
dest.sin_family = AF_INET;
dest.sin_addr.s_addr = inet_addr("127.0.0.1"); /* set destination IP number */
dest.sin_port = htons(PORTNUM);                /* set destination port number */
```

Now we get on to the interesting part:

The first line uses `memset()` to zero the struct.

The second line sets the address family. This should be the same value that was passed as the first parameter to `socket()`; for most purposes `AF_INET` will serve.

The third line is where we set the IP of the machine we need to connect to. The variable `dest.sin_addr.s_addr` is just an integer stored in Big Endian format, but we don't have to know that as the `inet_addr()` function will do the conversion from string into Big Endian integer for us.

The fourth line sets the destination port number. The `htons()` function converts the port number into a Big Endian short integer. If your program is going to be run solely on machines which use Big Endian numbers as default then `dest.sin_port = 21` would work just as well. However, for portability reasons `htons()` should always be used.

Now that all of the preliminary work is done, we can actually make the connection and use it:

```
connect(mysocket, (struct sockaddr *)&dest, sizeof(struct sockaddr));
```

This tells our OS to use the socket `mysocket` to create a connection to the machine specified in `dest`.

```
len = recv(mysocket, buffer, MAXRCVLEN, 0);
```

Now this receives up to `MAXRCVLEN` bytes of data from the connection and stores them in the buffer string. The number of characters received is returned by `recv()`. It is important to note that the data received will not automatically be null terminated when stored in the buffer, so we need to do it ourselves with `buffer[inputlen] = '\0'`.

And that's about it!

The next step after learning how to receive data is learning how to send it. If you've understood the previous section then this is quite easy. All you have to do is use the `send()` function, which uses the same parameters as `recv()`. If in our previous example `buffer` had the text we wanted to send and its length was stored in `len` we would write `send(mysocket, buffer, len, 0)`. `send()` returns the number of bytes that were sent. It is important to remember that `send()`, for various reasons, may not be able to send all of the bytes, so it is important to check that its return value is equal to the number of bytes you tried to send. In most cases this can be resolved by resending the unsent data.

29.2 A simple server

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>

#define PORTNUM 2343

int main(int argc, char *argv[])
{
    char msg[] = "Hello World !\n";

    struct sockaddr_in dest; /* socket info about the machine connecting to us
 */
    struct sockaddr_in serv; /* socket info about our server */
    int mysocket;            /* socket used to listen for incoming connections
 */
    socklen_t socksize = sizeof(struct sockaddr_in);

    memset(&serv, 0, sizeof(serv));           /* zero the struct before filling
 the fields */
    serv.sin_family = AF_INET;                /* set the type of connection to
 TCP/IP */
    serv.sin_addr.s_addr = htonl(INADDR_ANY); /* set our address to any
 interface */
    serv.sin_port = htons(PORTNUM);           /* set the server port number */


    mysocket = socket(AF_INET, SOCK_STREAM, 0);

    /* bind serv information to mysocket */
    bind(mysocket, (struct sockaddr *)&serv, sizeof(struct sockaddr));

    /* start listening, allowing a queue of up to 1 pending connection */
    listen(mysocket, 1);
    int consocket = accept(mysocket, (struct sockaddr *)&dest, &socksize);

    while(consocket)
    {
        printf("Incoming connection from %s - sending welcome\n",
 inet_ntoa(dest.sin_addr));
        send(consocket, msg, strlen(msg), 0);
        consocket = accept(mysocket, (struct sockaddr *)&dest, &socksize);
    }

    close(consocket);
    close(mysocket);
    return EXIT_SUCCESS;
}
```

Superficially, this is very similar to the client. The first important difference is that rather than creating a `sockaddr_in` with information about the machine we're connecting to, we create it with information about the server, and then we `bind()` it to the socket. This allows the machine to know the data received on the port specified in the `sockaddr_in` should be handled by our specified socket.

The `listen()` function then tells our program to start listening using the given socket. The second parameter of `listen()` allows us to specify the maximum number of connections that can be queued. Each time a connection is made to the server it is added to the queue. We take connections from the queue using the `accept()` function. If there is no connection waiting on the queue the program waits until a connection is received. The `accept()` function returns another socket. This socket is essentially a "session" socket, and can be used solely for communicating with connection we took off the queue. The original socket (`mysocket`) continues to listen on the specified port for further connections.

Once we have "session" socket we can handle it in the same way as with the client, using `send()` and `recv()` to handle data transfers.

Note that this server can only accept one connection at a time; if you want to simultaneously handle multiple clients then you'll need to `fork()` off separate processes, or use threads, to handle the connections.

29.3 Useful network functions

```
int gethostname(char *hostname, size_t size);
```

The parameters are a pointer to an array of chars and the size of that array. If possible, it finds the hostname and stores it in the array. On failure it returns -1.

```
struct hostent *gethostbyname(const char *name);
```

This function obtains information about a domain name and stores it in a `hostent` struct. The most useful part of a `hostent` structure is the `(char**) h_addr_list` field, which is a null terminated array of the IP addresses associated with that domain. The field `h_addr` is a pointer to the first IP address in the `h_addr_list` array. Returns `NULL` on failure.

29.4 FAQs

29.4.1 What about stateless connections?

If you don't want to exploit the properties of TCP in your program and would rather just use a UDP connection, then you can just replace `SOCK_STREAM` with `SOCK_DGRAM` in your call to `socket()` and use the result in the same way. It is important to remember that UDP does not guarantee delivery of packets and order of delivery, so checking is important.

If you want to exploit the properties of UDP, then you can use `sendto()` and `recvfrom()`, which operate like `send()` and `recv()` except you need to provide extra parameters specifying who you are communicating with.

29.4.2 How do I check for errors?

The functions `socket()`, `recv()` and `connect()` all return -1 on failure and use errno for further details.

30 Common practices

With its extensive use, a number of common practices and conventions have evolved to help avoid errors in C programs. These are simultaneously a demonstration of the application of good software engineering principles to a language and an indication of the limitations of C. Although few are used universally, and some are controversial, each of these enjoys wide use.

30.1 Dynamic multidimensional arrays

Although one-dimensional arrays are easy to create dynamically using malloc, and fixed-size multidimensional arrays are easy to create using the built-in language feature, dynamic multidimensional arrays are trickier. There are a number of different ways to create them, each with different tradeoffs. The two most popular ways to create them are:

- They can be allocated as a single block of memory, just like static multidimensional arrays. This requires that the array be *rectangular* (i.e. subarrays of lower dimensions are static and have the same size). The disadvantage is that the syntax of declaration the pointer is a little tricky for programmers at first. For example, if one wanted to create an array of ints of 3 columns and `rows` rows, one would do

```
int (*multi_array)[3] = malloc(rows * sizeof(int[3]));
```

  (Note that here `multi_array` is a pointer to an array of 3 ints.)

  Because of array-pointer interchangeability, you can index this just like static multidimensional arrays, i.e. `multi_array[5][2]` is the element at the 6th row and 3rd column.

- They can be allocated by first allocating an array of pointers, and then allocating subarrays and storing their addresses in the array of pointers (this approach is also known as an Iliffe vector[1]). The syntax for accessing elements is the same as for multidimensional arrays described above (even though they are stored very differently). This approach has the advantage of the ability to make ragged arrays (i.e. with subarrays of different sizes). However, it also uses more space and requires more levels of indirection to index into, and can have worse cache performance. It also requires many dynamic allocations, each of which can be expensive.

For more information, see the comp.lang.c FAQ, question 6.16[2].

1 http://en.wikipedia.org/wiki/Iliffe%20vector
2 http://www.eskimo.com/~scs/C-faq/q6.16.html

In some cases, the use of multi-dimensional arrays can best be addressed as an array of structures. Before user-defined data structures were available, a common technique was to define a multi-dimensional array, where each column contained different information about the row. This approach is also frequently used by beginner programmers. For example, columns of a two-dimensional character array might contain last name, first name, address, etc.

In cases like this, it is better to define a structure that contains the information that was stored in the columns, and then create an array of pointers to that structure. This is especially true when the number of data points for a given record might vary, such as the tracks on an album. In these cases, it is better to create a structure for the album that contains information about the album, along with a dynamic array for the list of songs on the album. Then an array of pointers to the album structure can be used to store the collection.

- Another useful way to create a dynamic multi-dimensional array is to flatten the array and index manually. For example, a 2-dimensional array with sizes x and y has x*y elements, therefore can be created by

```
int dynamic_multi_array[x*y];
```

The index is slightly trickier than before, but can still be obtained by y*i+j. You then access the array with

```
static_multi_array[i][j];
dynamic_multi_array[y*i+j];
```

Some more examples with higher dimensions:

```
int dim1[w];
int dim2[w*x];
int dim3[w*x*y];
int dim4[w*x*y*z];

dim1[i]
dim2[w*j+i];
dim3[w*(x*i+j)+k] // index is k + w*j + w*x*i
dim4[w*(x*(y*i+j)+k)+l] // index is w*x*y*i + w*x*j + w*k + l
```

Note that w*(x*(y*i+j)+k)+l is equal to w*x*y*i + w*x*j + w*k + l, but uses fewer operations (see Horner's Method[3]). It uses the same number of operations as accessing a static array by dim4[i][j][k][l], so should not be any slower to use.

The advantage to using this method is that the array can be passed freely between functions without knowing the size of the array at compile time (since C sees it as a 1-dimensional array, although some way of passing the dimensions is still necessary), and the entire array is contiguous in memory, so accessing consecutive elements should be fast. The disadvantage is that it can be difficult at first to get used to how to index the elements.

3 http://en.wikipedia.org/wiki/Horner%27s_method

30.2 Constructors and destructors

In most object-oriented languages, objects cannot be created directly by a client that wishes to use them. Instead, the client must ask the class to build an instance of the object using a special routine called a constructor. Constructors are important because they allow an object to enforce invariants about its internal state throughout its lifetime. Destructors, called at the end of an object's lifetime, are important in systems where an object holds exclusive access to some resource, and it is desirable to ensure that it releases these resources for use by other objects.

Since C is not an object-oriented language, it has no built-in support for constructors or destructors. It is not uncommon for clients to explicitly allocate and initialize records and other objects. However, this leads to a potential for errors, since operations on the object may fail or behave unpredictably if the object is not properly initialized. A better approach is to have a function that creates an instance of the object, possibly taking initialization parameters, as in this example:

```
struct string {
    size_t size;
    char *data;
};
```

```
struct string *create_string(const char *initial) {
    assert (initial != NULL);
    struct string *new_string = malloc(sizeof(*new_string));
    if (new_string != NULL) {
        new_string->size = strlen(initial);
        new_string->data = strdup(initial);
    }
    return new_string;
}
```

Similarly, if it is left to the client to destroy objects correctly, they may fail to do so, causing resource leaks. It is better to have an explicit destructor which is always used, such as this one:

```
void free_string(struct string *s) {
    assert (s != NULL);
    free(s->data);  /* free memory held by the structure */
    free(s);        /* free the structure itself */
}
```

It is often useful to combine destructors with *#Nulling freed pointers*[4].

Sometimes it is useful to hide the definition of the object to ensure that the client does not allocate it manually. To do this, the structure is defined in the source file (or a private header file not available to users) instead of the header file, and a forward declaration is put in the header file:

4 Chapter 30.3 on page 234

```
struct string;
struct string *create_string(const char *initial);
void free_string(struct string *s);
```

30.3 Nulling freed pointers

As discussed earlier, after `free()` has been called on a pointer, it becomes a dangling pointer. Worse still, most modern platforms cannot detect when such a pointer is used before being reassigned.

One simple solution to this is to ensure that any pointer is set to a null pointer immediately after being freed: [5]

```
free(p);
p = NULL;
```

Unlike dangling pointers, a hardware exception will arise on many modern architectures when a null pointer is dereferenced. Also, programs can include error checks for the null value, but not for a dangling pointer value. To ensure it is done at all locations, a macro can be used:

```
#define FREE(p)   do { free(p); (p) = NULL; } while(0)
```

(To see why the macro is written this way, see *#Macro conventions*[6].) Also, when this technique is used, destructors should zero out the pointer that they are passed, and their argument must be passed by reference to allow this. For example, here's the destructor from *#Constructors and destructors*[7] updated:

```
void free_string(struct string **s) {
    assert(s != NULL  &&  *s != NULL);
    FREE((*s)->data);  /* free memory held by the structure */
    FREE(*s);          /* free the structure itself */
    *s=NULL;           /* zero the argument */
}
```

Unfortunately, this idiom will not do anything to any other pointers that may be pointing to the freed memory. For this reason, some C experts regard this idiom as dangerous due to creating a false sense of security.

5 comp.lang.c FAQ list: "Why isn't a pointer null after calling free?" ^{`http://c-faq.com/malloc/ptrafterfree.html`} mentions that "it is often useful to set [pointer variables] to NULL immediately after freeing them".
6 Chapter 30.4 on page 235
7 Chapter 30.2 on page 233

30.4 Macro conventions

Because preprocessor macros in C work using simple token replacement, they are prone to a number of confusing errors, some of which can be avoided by following a simple set of conventions:

1. Placing parentheses around macro arguments wherever possible. This ensures that, if they are expressions, the order of operations does not affect the behavior of the expression. For example:
   - Wrong: `#define square(x) x*x`
   - Better: `#define square(x) (x)*(x)`
2. Placing parentheses around the entire expression if it is a single expression. Again, this avoids changes in meaning due to the order of operations.
   - Wrong: `#define square(x) (x)*(x)`
   - Better: `#define square(x) ((x)*(x))`
   - Dangerous, remember it replaces the text in verbatim. Suppose your code is `square (x++)`, after the macro invocation will x be incremented by 2
3. If a macro produces multiple statements, or declares variables, it can be wrapped in a **do** { ... } **while**(0) loop, with no terminating semicolon. This allows the macro to be used like a single statement in any location, such as the body of an if statement, while still allowing a semicolon to be placed after the macro invocation without creating a null statement. Care must be taken that any new variables do not potentially mask portions of the macro's arguments.
   - Wrong: `#define FREE(p) free(p); p = NULL;`
   - Better: `#define FREE(p) do { free(p); p = NULL; } while(0)`
4. Avoiding using a macro argument twice or more inside a macro, if possible; this causes problems with macro arguments that contain side effects, such as assignments.
5. If a macro may be replaced by a function in the future, considering naming it like a function.

30.5 Further reading

There are a huge number of C style guidelines.

- "C and C++ Style Guides"[8] by Chris Lott lists many popular C style guides.
- The Motor Industry Software Reliability Association (MISRA) publishes "MISRA-C: Guidelines for the use of the C language in critical systems". (Wikipedia: MISRA C[9]; `http://www.misra-c.com/`).

pl:C/Powszechne praktyki[10]

8 `http://www.chris-lott.org/resources/cstyle/`
9 `http://en.wikipedia.org/wiki/%20MISRA%20C`
10 `http://pl.wikibooks.org/wiki/C%2FPowszechne%20praktyki`

31 C and beyond

32 Language extensions

Most C compilers have one or more "extensions" to the standard C language, to do things that are inconvenient to do in standard, portable C.

Some examples of language extensions:

- in-line assembly language
- interrupt service routines
- variable-length data structure (a structure whose last item is a "zero-length array").[1]

[2]

- re-sizeable multidimensional arrays
- various "#pragma" settings to compile quickly, to generate fast code, or to generate compact code.
- bit manipulation, especially bit-rotations and things involving the "carry" bit
- storage alignment
- Arrays whose length is computed at run time.

32.1 External links

- GNU C: Extensions to the C Language[3]
- C/C++ interpreter Ch extensions to the C language for scripting[4]
- SDCC: Storage Class Language Extensions[5]

1

2 comp.lang.c FAQ list: Question 2.6 ^{http://c-faq.com/struct/structhack.html} : "C99 introduces the concept of a flexible array member, which allows the size of an array to be omitted if it is the last member in a structure, thus providing a well-defined solution."

3 http://gcc.gnu.org/onlinedocs/gcc-4.0.2/gcc/C-Extensions.html#C-Extensions

4 http://www.softintegration.com/support/faq/general.html#4

5 http://sdcc.sourceforge.net/doc/sdccman.html/node56.html

33 Mixing languages

33.1 Assembler

See Embedded Systems/Mixed C and Assembly Programming[1]

33.2 Cg

Make the main program (for CPU) in C, which loads and run the Cg[2] program (for GPU).[3][4][5]

33.2.1 Header Files

Add to C program :[6]

```
#include <Cg/cg.h> /* To include the core Cg runtime API into your  program */
#include <Cg/cgGL.h>  /* to include the OpenGL-specific Cg runtime API */
```

33.3 Java

Using the Java native interface (JNI), Java applications can call C libraries.

See also

- Java_Programming/Keywords/native[7]

1 http://en.wikibooks.org/wiki/Embedded%20Systems%2FMixed%20C%20and%20Assembly%20Programming
2 http://en.wikibooks.org/wiki/Cg_%28programming_language%29
3 Lesson: 47 from NeHe Productions ^{http://nehe.gamedev.net/data/lessons/lesson.asp?lesson=47}
4 Cg Bumpmapping by Razvan Surdulescu at GameDev ^{http://www.gamedev.net/reference/articles/article1903.asp}
5 [http://www.fusionindustries.com/default.asp?page=cg-hlsl-faq | Cg & HLSL Shading Language FAQ

```
by Fusion Industries]
```

6 http://http.developer.nvidia.com/CgTutorial/cg_tutorial_appendix_b.html NVidia Cg tutorial. Appendix B. The Cg Runtime
7 http://en.wikibooks.org/wiki/Java_Programming%2FKeywords%2Fnative

33.4 Perl

To mix Perl and C, we can use XS. XS is an interface description file format used to create an extension interface between Perl and C code (or a C library) which one wishes to use with Perl.

The basic procedure is very simple. We can create the necessary subdirectory structure by running "h2xs" application (e.g. "h2xs -A -n Modulename"). This will create - among others - a Makefile.PL, a .pm Perl module and a .xs XSUB file in the subdirectory tree. We can edit the .xs file by adding our code to that, let's say:

```
void
hello()
  CODE:
    printf("Hello, world!\n");
```

and we can successfully use our new command at Perl side, after running a "perl Makefile.PL" and "make".

Further details can be found on the perlxstut[8] perldoc[9] page.

33.5 Python

33.6 For further reading

- Embedded Systems/Mixed C and Assembly Programming[10]

33.7 References

pl:C/Łączenie z innymi językami[11]

8 http://perldoc.perl.org/perlxstut.html
9 http://perldoc.perl.org
10 http://en.wikibooks.org/wiki/Embedded%20Systems%2FMixed%20C%20and%20Assembly%20Programming
11 http://pl.wikibooks.org/wiki/C%2F%C5%81%C4%85czenie%20z%20innymi%20j%C4%99zykami

34 Code library

The following is an implementation of the Standard C99 version of `<assert.h>`:

```
/* assert.h header */
#undef assert
#ifdef NDEBUG
#define assert(_Ignore) ((void)0)
#else
void _Assertfail(char *, char *, int, char *);
#define assert(_Test)
((_Test)?((void)0):_Assertfail(#_Test,__FILE__,__LINE__,__func__))
#endif
/* END OF FILE */
```

```
/* xassertfail.c -- _Assertfail function */
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
void
_Assertfail(char *test, char *filename, int line_number, char *function_name)
{
   fprintf(stderr, "Assertion failed: %s, function %s, file %s, line %d.",
           test, function_name, filename, line_number);
   abort();
}
/* END OF FILE */
```

35 Computer Programming

The following articles are C adaptations from articles of the Computer programming[1] book.

1 http://en.wikibooks.org/wiki/Computer%20programming

36 Statements

A **statement** is a command given to the computer that instructs the computer to take a specific action, such as display to the screen, or collect input. A computer program is made up of a series of statements.

```
puts ("Hi there!");
```

```
puts ("Hi there!");
puts ("Strange things are afoot...");
```

Category:C Programming[1]

1 http://en.wikibooks.org/wiki/Category%3AC%20Programming

37 C Reference Tables

This section has some tables and lists of C entities.

38 Reference Tables

38.1 List of Keywords

ANSI C (C89)/ISO C (C90) keywords:

- **`auto`**
- **`break`**
- **`case`**
- **`char`**
- **`const`**
- **`continue`**
- **`default`**
- **`do`**
- **`double`**
- **`else`**
- **`enum`**
- **`extern`**
- **`float`**
- **`for`**
- **`goto`**
- **`if`**
- **`int`**
- **`long`**
- **`register`**
- **`return`**
- **`short`**
- **`signed`**
- **`sizeof`**
- **`static`**
- **`struct`**
- **`switch`**
- **`typedef`**
- **`union`**
- **`unsigned`**
- **`void`**
- **`volatile`**
- **`while`**

Keywords added to ISO C (C99) (Supported only in new compilers):

- **`_Bool`**
- **`_Complex`**
- **`_Imaginary`**
- **`inline`**
- **`restrict`**[1]

Specific compilers may (in a non-standard-compliant mode) also treat some other words as keywords, including **`asm`**, **`cdecl`**, **`far`**, **`fortran`**, **`huge`**, **`interrupt`**, **`near`**, **`pascal`**, **`typeof`**.

Very old compilers may not recognize some or all of the C89 keywords **`const`**, **`enum`**, **`signed`**, **`void`**, **`volatile`** as well as the C99 keywords.

See also the list of reserved identifiers[2].

38.2 List of Standard Headers

ANSI C (C89)/ISO C (C90) headers:

1 http://en.wikipedia.org/wiki/Restrict

2 http://publib.boulder.ibm.com/infocenter/comphelp/v7v91/topic/com.ibm.vacpp7a.doc/language/ref/clrc02reserved_identifiers.htm

- `<assert.h>`[3]
- `<ctype.h>`[4]
- `<errno.h>`[5]
- `<float.h>`[6]
- `<limits.h>`[7]
- `<locale.h>`[8]
- `<math.h>`[9]
- `<setjmp.h>`[10]
- `<signal.h>`[11]
- `<stdarg.h>`[12]
- `<stddef.h>`[13]
- `<stdio.h>`[14]
- `<stdlib.h>`[15]
- `<string.h>`[16]
- `<time.h>`[17]

Very old compilers may not include some or all of the following headers:

Headers added to ISO C (C94/C95) in Amendment 1 (AMD1):

- `<iso646.h>`[18]
- `<wchar.h>`[19]
- `<wctype.h>`[20]

Headers added to ISO C (C99) (Supported only in new compilers):

- `<complex.h>`[21]
- `<fenv.h>`[22]
- `<inttypes.h>`[23]
- `<stdbool.h>`[24]
- `<stdint.h>`[25]
- `<tgmath.h>`[26]

3 http://en.wikipedia.org/wiki/Assert.h
4 http://en.wikipedia.org/wiki/Ctype.h
5 http://en.wikipedia.org/wiki/Errno.h
6 http://en.wikipedia.org/wiki/Float.h
7 http://en.wikipedia.org/wiki/Limits.h
8 http://en.wikipedia.org/wiki/Locale.h
9 http://en.wikipedia.org/wiki/Math.h
10 http://en.wikipedia.org/wiki/Setjmp.h
11 http://en.wikipedia.org/wiki/Signal.h
12 http://en.wikipedia.org/wiki/Stdarg.h
13 http://en.wikipedia.org/wiki/Stddef.h
14 http://en.wikipedia.org/wiki/Stdio.h
15 http://en.wikipedia.org/wiki/Stdlib.h
16 http://en.wikipedia.org/wiki/String.h
17 http://en.wikipedia.org/wiki/Time.h
18 http://en.wikipedia.org/wiki/Iso646.h
19 http://en.wikipedia.org/wiki/Wchar.h
20 http://en.wikipedia.org/wiki/Wctype.h
21 http://en.wikipedia.org/wiki/Complex.h
22 http://en.wikipedia.org/wiki/Fenv.h
23 http://en.wikipedia.org/wiki/Inttypes.h
24 http://en.wikipedia.org/wiki/Stdbool.h
25 http://en.wikipedia.org/wiki/Stdint.h
26 http://en.wikipedia.org/wiki/Tgmath.h

38.3 Table of Operators

Operators in the same row of this table have the same **precedence** and the order of evaluation is decided by the **associativity** (*left-to-right* or *right-to-left*). Operators closer to the top of this table have *higher* precedence than those in a subsequent group.

Operators	**Description**	**Example Usage**	**Associativity**
Postfix operators			
`()`	function call operator	`swap (x, y)`	Left to right
`[]`	array index operator	`arr [i]`	
`.`	member access operator *for an object of struct/union type or a reference to it*	`obj.member`	
`->`	member access operator *for a pointer to an object of struct/union type*	`ptr->member`	
Unary Operators			
`!`	logical not operator	`!eof_reached`	
`~`	bitwise not operator	`~mask`	
`+ -`[1][27]	unary plus/minus operators	`-num`	
`++ --`	post-increment/decrement operators	`num++`	Right to left
`++ --`	pre-increment/decrement operators	`++num`	
`&`	address-of operator	`&data`	
`*`	indirection operator	`*ptr`	
`sizeof`	sizeof operator *for expressions*	**`sizeof`** `123`	
`sizeof``()`	sizeof operator *for types*	**`sizeof (int)`**	
`(`*`type`*`)`	cast operator	**`(float)`**`i`	
Multiplicative Operators			Left to right

27 Chapter 38.3.1 on page 255

`* / %`	multiplication, division and modulus operators	`celsius_diff * 9.0 / 5.0`	
Additive Operators			Left to right
`+ -`	addition and subtraction operators	`end - start + 1`	
Bitwise Shift Operators			
`<<`	left shift operator	`bits << shift_len`	Left to right
`>>`	right shift operator	`bits >> shift_len`	
Relational Inequality Operators			Left to right
`< > <= >=`	less-than, greater-than, less-than or equal-to, greater-than or equal-to operators	`i < num_elements`	
Relational Equality Operators			Left to right
`== !=`	equal-to, not-equal-to	`choice != 'n'`	
Bitwise And Operator			Left to right
`&`		`bits & clear_mask_complement`	
Bitwise Xor Operator			Left to right
`ˆ`		`bits ˆ invert_mask`	
Bitwise Or Operator			Left to right
|		bits | set_mask	
Logical And Operator			Left to right
`&&`		`arr != 0 && arr->len != 0`	
Logical Or Operator			Left to right

`||`	see Logical Expressions[28]	`arr == 0 || arr->len == 0`	
Conditional Operator			
`?:`		`size != 0 ? size : 0`	Right to left
Assignment Operators			
`=`	assignment operator	`i = 0`	Right to left
`+= -= *= /= %= &= |= ^= <<= >>=`	shorthand assignment operators (*foo* op= *bar*represents *foo* = *foo* op *bar*)	`num /= 10`	
Comma Operator			
`,`		`i = 0, j = i + 1, k = 0`	Left to right

38.3.1 Table of Operators Footnotes

[1]Very old compilers may not recognize the unary + operator.

et:Programmeerimiskeel C/Operaatorid[29]

38.4 Table of Data Types

28 Chapter 16.1.2 on page 85
29 http://et.wikibooks.org/wiki/Programmeerimiskeel%20C%2FOperaatorid

Type	Size in Bits	Comments	Alternative Names
Primitive Types in ANSI C (C89)/ISO C (C90)			
`char`	≥ 8	• `sizeof` gives the size in units of `char`s. These "C bytes" need not be 8-bit bytes (though commonly they are); the number of bits is given by the `CHAR_BIT` macro in the `limits.h` header. • Signedness is implementation-defined. • Any encoding of 8 bits or less (e.g. ASCII) can be used to store characters. • Integer operations can be performed portably only for the range 0 ~ 127. • All bits contribute to the value of the **char**, i.e. there are no "holes" or "padding" bits.	—
`signed char`	same as `char`	• Characters stored like for type `char`. • Can store integers in the range -127 ~ 127 portably[1][30].	—
`unsigned char`	same as `char`	• Characters stored like for type `char`. • Can store integers in the range 0 ~ 255 portably.	—
`short`	≥ 16, ≥ size of `char`	• Can store integers in the range -32767 ~ 32767 portably[2][31]. • Used to reduce memory usage (although the resulting executable may be larger and probably slower as compared to using `int`.	`short int`, `signed short`, `signed short int`

30 Chapter 38.4.1 on page 260
31 Chapter 38.4.1 on page 260

Type	**Size in Bits**	**Comments**	**Alternative Names**
Primitive Types in ANSI C (C89)/ISO C (C90)			
`unsigned short`	same as `short`	• Can store integers in the range 0 ~ 65535 portably. • Used to reduce memory usage (although the resulting executable may be larger and probably slower as compared to using `int`.	`unsigned short int`
`int`	≥ 16, ≥ size of `short`	• Represents the "normal" size of data the processor deals with (the word-size); this is the integral data-type used normally. • Can store integers in the range -32767 ~ 32767 portably[2][32].	`signed`, `signed int`
`unsigned int`	same as `int`	• Can store integers in the range 0 ~ 65535 portably.	`unsigned`
`long`	≥ 32, ≥ size of `int`	• Can store integers in the range -2147483647 ~ 2147483647 portably[3][33].	`long int`, `signed long`, `signed long int`
`unsigned long`	same as `long`	• Can store integers in the range 0 ~ 4294967295 portably.	`unsigned long int`
`float`	≥ size of `char`	• Used to reduce memory usage when the values used do not vary widely. • The floating-point format used is implementation defined and need not be the IEEE single-precision format. • `unsigned` cannot be specified.	—

32 Chapter 38.4.1 on page 260
33 Chapter 38.4.1 on page 260

Type	Size in Bits	Comments	Alternative Names
Primitive Types in ANSI C (C89)/ISO C (C90)			
`double`	≥ size of `float`	• Represents the "normal" size of data the processor deals with; this is the floating-point data-type used normally. • The floating-point format used is implementation defined and need not be the IEEE double-precision format. • `unsigned` cannot be specified.	—
`long double`	≥ size of `double`	• `unsigned` cannot be specified.	—
Primitive Types added to ISO C (C99)			
`long long`	≥ 64, ≥ size of `long`	• Can store integers in the range -9223372036854775807 ~ 9223372036854775807 portably[4]34 .	`long long int`, `signed long long`, `signed long long int`
`unsigned long long`	same as `long long`	• Can store integers in the range 0 ~ 18446744073709551615 portably.	`unsigned long long int`
`intmax_t`	the maximum width supported by the platform	• Can store integers in the range -(1 << n-1)+1 ~ (1 << n-1)-1, with 'n' the width of intmax_t. • Used by the "j" length modifier in the C Programming/File IO#Formatted output functions: the printf family of functions35.	—
`uintmax_t`	same as `intmax_t`	• Can store integers in the range 0 ~ (1 << n)-1, with 'n' the width of uintmax_t.	—
User Defined Types			

34 Chapter 38.4.1 on page 260
35 Chapter 21.4 on page 137

Type	Size in Bits	Comments	Alternative Names
Primitive Types in ANSI C (C89)/ISO C (C90)			
`struct`	≥ sum of size of each member	• Said to be an *aggregate type.*	—
`union`	≥ size of the largest member	• Said to be an *aggregate type.*	—
`enum`	≥ size of `char`	• Enumerations are a separate type from `int`s, though they are mutually convertible.	—
`typedef`	same as the type being given a name	• `typedef` has syntax similar to a storage class like `static`, `register` or `extern`.	—
Derived Types[5][36]			
`type``*` (pointer)	≥ size of `char`	• `0` always represents the null pointer (an address where no data can be placed), irrespective of what bit sequence represents the value of a null pointer. • Pointers to different types may have different representations, which means they could also be of different sizes. So they are not convertible to one another. • Even in an implementation which guarantess all data pointers to be of the same size, function pointers and data pointers are in general incompatible with each other. • For functions taking variable number of arguments, the arguments passed must be of appropriate type, so even `0` must be cast to the appropriate type in such function-calls.	—

36 Chapter 38.4.1 on page 260

Type	Size in Bits	Comments	Alternative Names
Primitive Types in ANSI C (C89)/ISO C (C90)			
`type [integer`[6][37]`]` (array)	$\geq$ *integer* × size of *`type`*	• The brackets (`[]`) *follow* the identifier name in a declaration. • In a declaration which also initializes the array (including a function parameter declaration), the size of the array (the *integer*) can be omitted. • *`type`* `[]` is not the same as *`type`*`*`. Only under some circumstances one can be converted to the other.	—
`type (comma-delimited list of types/declarations)` (function)	—	• Functions declared without any storage class are **`extern`**. • The parentheses (`()`) *follow* the identifier name in a declaration, e.g. a 2-arg function pointer: **`int (* fptr) (int arg1, int arg2)`**.	—

38.4.1 Table of Data Types Footnotes

[1] -128 can be stored in two's-complement machines (i.e. most machines in existence). Very old compilers may not recognize the **`signed`** keyword.
[2] -32768 can be stored in two's-complement machines (i.e. most machines in existence). Very old compilers may not recognize the **`signed`** keyword.
[3] -2147483648 can be stored in two's-complement machines (i.e. most machines in existence). Very old compilers may not recognize the **`signed`** keyword.
[4] -9223372036854775808 can be stored in two's-complement machines (i.e. most machines in existence).
[5] The precedences in a declaration are:
`[]`, `()` (left associative) — Highest
`*` (right associative) — Lowest
[6] The standards do NOT place any restriction on the size/type of the integer, it's implementation dependent. The only mention in the standards is a reference that an implementation may have limits to the maximum size of memory block which can be allocated, and as such the limit on integer will be size_of_max_block/sizeof(type).

37 Chapter 38.4.1 on page 260

pl:C/Składnia[38]

38 http://pl.wikibooks.org/wiki/C%2FSk%C5%82adnia

39 Compilers

39.1 Free (or with a free version)

- Ch_interpreter[1] (`http://www.softintegration.com`) - The software works in Windows, Linux, Mac OS X, Freebsd, Solaris, AIX and HP-UX. The Ch Standard Edition is free for noncommercial use.
- lcc-win32[2] (`http://www.cs.virginia.edu/~lcc-win32`) - Software copyrighted by Jacob Navia. It is free for non-commercial use. Windows (98/ME/XP/2000/NT).
- GNU Compiler Collection[3] (`http://gcc.gnu.org`) - GNU Compiler Collection. GNU General Public License / GNU Lesser General Public License.
  - MinGW[4] (`http://www.mingw.org/`) provides GCC for Windows
- Open Watcom[5] (`http://www.openwatcom.org`) Open Source development community to maintain and enhance the Watcom C/C++ and Fortran cross compilers and tools. Version 1.4 released in December 2005.
  - **Host Platforms:** Win32 systems (IDE and command line), 32-bit OS/2 (IDE and command line), DOS (command line), and Windows 3.x (IDE)
  - **Target Platforms:** DOS (16-bit), Windows 3.x (16-bit), OS/2 1.x (16-bit), Extended DOS, Win32s, Windows 95/98/Me, Windows NT/2000/XP, 32-bit OS/2, and Novell NLMs
  - **Experimental / Development:** Linux, BSD, *nix, PowerPC, Alpha AXP, MIPS, and Sparc v8
- Tiny C Compiler[6]
- Portable C Compiler[7] (`http://pcc.ludd.ltu.se`) - Portable C Compiler. BSD Style License(s).
- Small Device C Compiler[8] (SDCC)
  - target platforms: Intel 8051-compatibles; Freescale (Motorola) HC08; Microchip PIC16 and PIC18.
- FpgaC[9]. Target platform: FPGA hardware via XNF or VHDL files.
- Interactive C[10] (`http://www.botball.org/educational-resources/ic.php`).
  - target platform: Handy Board (Freescale 68HC11); Lego RCX

1 `http://en.wikipedia.org/wiki/Ch_interpreter`
2 `http://en.wikipedia.org/wiki/lcc-win32`
3 `http://en.wikipedia.org/wiki/GNU%20Compiler%20Collection`
4 `http://en.wikipedia.org/wiki/MinGW`
5 `http://en.wikipedia.org/wiki/Open%20Watcom`
6 `http://en.wikipedia.org/wiki/Tiny%20C%20Compiler`
7 `http://en.wikipedia.org/wiki/Portable%20C%20Compiler`
8 `http://en.wikipedia.org/wiki/Small%20Device%20C%20Compiler`
9 `http://en.wikipedia.org/wiki/FpgaC`
10 `http://en.wikipedia.org/wiki/Interactive%20C`

- C compilers for many digital signal processors (DSPs), many of them free, are listed in the comp.dsp FAQ[11].

39.2 Commercial

- Intel C Compiler[12] (`http://software.intel.com/en-us/intel-compilers`) - Windows, Linux, Mac, QNX, and embedded C/C++ compilers. Optimized for Intel 32-bit and 64-bit CPUs.
- Microsoft Visual C++[13] (`http://msdn.microsoft.com/visualc`) - Free (partially limited) version available (Express edition)
- Impulse C[14] - Target platform: FPGA hardware via Hardware Description Language (HDL) files.

11 `http://www.bdti.com/faq/3.htm`
12 `http://en.wikipedia.org/wiki/Intel%20C%20Compiler`
13 `http://en.wikipedia.org/wiki/Microsoft%20Visual%20C%2B%2B`
14 `http://en.wikipedia.org/wiki/Impulse%20C`

40 Contributors

Edits	User
1	16@r[1]
30	A thing[2]
8	A3 nm[3]
1	Ab8uu[4]
1	Abdull[5]
12	Adam majewski[6]
13	Adrignola[7]
1	Aentity[8]
5	AlbertCahalan[9]
6	Albmont[10]
1	AlistairMcMillan[11]
3	AllenZh[12]
1	Alsocal[13]
1	Alvin-cs[14]
4	Andrew Eugene[15]
1	Angus Lepper[16]
1	Arbitrarily0[17]
43	Astone42[18]
1	Asymmetric[19]
7	Avicennasis[20]
2	Az1568[21]

1 http://en.wikibooks.org/wiki/User:16@r
2 http://en.wikibooks.org/wiki/User:A_thing
3 http://en.wikibooks.org/wiki/User:A3_nm
4 http://en.wikibooks.org/wiki/User:Ab8uu
5 http://en.wikibooks.org/wiki/User:Abdull
6 http://en.wikibooks.org/wiki/User:Adam_majewski
7 http://en.wikibooks.org/wiki/User:Adrignola
8 http://en.wikibooks.org/wiki/User:Aentity
9 http://en.wikibooks.org/wiki/User:AlbertCahalan
10 http://en.wikibooks.org/wiki/User:Albmont
11 http://en.wikibooks.org/wiki/Special:Contributions/AlistairMcMillan
12 http://en.wikibooks.org/wiki/User:AllenZh
13 http://en.wikibooks.org/wiki/User:Alsocal
14 http://en.wikibooks.org/wiki/Special:Contributions/Alvin-cs
15 http://en.wikibooks.org/wiki/User:Andrew_Eugene
16 http://en.wikibooks.org/wiki/Special:Contributions/Angus_Lepper
17 http://en.wikibooks.org/wiki/User:Arbitrarily0
18 http://en.wikibooks.org/wiki/User:Astone42
19 http://en.wikibooks.org/wiki/Special:Contributions/Asymmetric
20 http://en.wikibooks.org/wiki/User:Avicennasis
21 http://en.wikibooks.org/wiki/User:Az1568

1	BL[22]
1	BOTarate[23]
1	Berkunt[24]
3	Bevo[25]
9	BimBot[26]
2	Binksternet[27]
1	Blanchardb[28]
1	Bobo192[29]
1	Bpringlemeir[30]
1	Buggi22[31]
12	CarsracBot[32]
14	CharmlessCoin[33]
2	Chobot[34]
2	Ckorff[35]
1	CrQAZ[36]
1	Cryptic[37]
1	Cybiko123[38]
2	Cyp[39]
1	D6[40]
1	DHN-bot[41]
1	Da monster under your bed[42]
4	Dan Polansky[43]
2	Darklama[44]
60	DavidCary[45]
1	Deathanatos[46]

22 http://en.wikibooks.org/wiki/Special:Contributions/BL
23 http://en.wikibooks.org/wiki/Special:Contributions/BOTarate
24 http://en.wikibooks.org/wiki/Special:Contributions/Berkunt
25 http://en.wikibooks.org/wiki/User:Bevo
26 http://en.wikibooks.org/wiki/User:BimBot
27 http://en.wikibooks.org/wiki/Special:Contributions/Binksternet
28 http://en.wikibooks.org/wiki/Special:Contributions/Blanchardb
29 http://en.wikibooks.org/wiki/Special:Contributions/Bobo192
30 http://en.wikibooks.org/wiki/Special:Contributions/Bpringlemeir
31 http://en.wikibooks.org/wiki/User:Buggi22
32 http://en.wikibooks.org/wiki/User:CarsracBot
33 http://en.wikibooks.org/wiki/User:CharmlessCoin
34 http://en.wikibooks.org/wiki/Special:Contributions/Chobot
35 http://en.wikibooks.org/wiki/Special:Contributions/Ckorff
36 http://en.wikibooks.org/wiki/Special:Contributions/CrQAZ
37 http://en.wikibooks.org/wiki/User:Cryptic
38 http://en.wikibooks.org/wiki/User:Cybiko123
39 http://en.wikibooks.org/wiki/User:Cyp
40 http://en.wikibooks.org/wiki/Special:Contributions/D6
41 http://en.wikibooks.org/wiki/Special:Contributions/DHN-bot
42 http://en.wikibooks.org/wiki/Special:Contributions/Da_monster_under_your_bed
43 http://en.wikibooks.org/wiki/User:Dan_Polansky
44 http://en.wikibooks.org/wiki/User:Darklama
45 http://en.wikibooks.org/wiki/User:DavidCary
46 http://en.wikibooks.org/wiki/Special:Contributions/Deathanatos

13	Derbeth[47]
1	Deryck Chan[48]
4	Dirk Hünniger[49]
1	Doshell[50]
2	Duplode[51]
15	EdC[52]
1	Edudobay[53]
8	Emperorbma[54]
5	Eric119[55]
1	Erkan Yilmaz[56]
2	Ervinn[57]
3	Felipebm[58]
1	Fourier.courier[59]
1	Frigotoni[60]
1	Giftlite[61]
1	Golftheman[62]
1	Grandscribe[63]
1	Graue[64]
4	Gsonnenf[65]
1	Guanabot[66]
1	Gulmammad[67]
3	Gwern[68]
2	Hagindaz[69]
1	HasharBot[70]
1	Hdante[71]

47 `http://en.wikibooks.org/wiki/User:Derbeth`
48 `http://en.wikibooks.org/wiki/User:Deryck_Chan`
49 `http://en.wikibooks.org/wiki/User:Dirk_H%25C3%25BCnniger`
50 `http://en.wikibooks.org/wiki/Special:Contributions/Doshell`
51 `http://en.wikibooks.org/wiki/User:Duplode`
52 `http://en.wikibooks.org/wiki/Special:Contributions/EdC`
53 `http://en.wikibooks.org/wiki/User:Edudobay`
54 `http://en.wikibooks.org/wiki/User:Emperorbma`
55 `http://en.wikibooks.org/wiki/User:Eric119`
56 `http://en.wikibooks.org/wiki/User:Erkan_Yilmaz`
57 `http://en.wikibooks.org/wiki/User:Ervinn`
58 `http://en.wikibooks.org/wiki/User:Felipebm`
59 `http://en.wikibooks.org/wiki/Special:Contributions/Fourier.courier`
60 `http://en.wikibooks.org/wiki/User:Frigotoni`
61 `http://en.wikibooks.org/wiki/Special:Contributions/Giftlite`
62 `http://en.wikibooks.org/wiki/Special:Contributions/Golftheman`
63 `http://en.wikibooks.org/wiki/Special:Contributions/Grandscribe`
64 `http://en.wikibooks.org/wiki/Special:Contributions/Graue`
65 `http://en.wikibooks.org/wiki/Special:Contributions/Gsonnenf`
66 `http://en.wikibooks.org/wiki/User:Guanabot`
67 `http://en.wikibooks.org/wiki/User:Gulmammad`
68 `http://en.wikibooks.org/wiki/User:Gwern`
69 `http://en.wikibooks.org/wiki/User:Hagindaz`
70 `http://en.wikibooks.org/wiki/Special:Contributions/HasharBot`
71 `http://en.wikibooks.org/wiki/Special:Contributions/Hdante`

1	HethrirBot[72]
1	Hongooi[73]
6	Hoxel[74]
1	I do not exist[75]
1	Icewedge[76]
3	Imran[77]
3	Intermediate-Hacker[78]
1	Iopq[79]
2	J.delanoy[80]
1	JAnDbot[81]
1	JL-Bot[82]
2	JackPotte[83]
1	Jafeluv[84]
11	James Dennett[85]
1	JetRanger405[86]
22	Jfmantis[87]
4	Jguk[88]
1	Jni[89]
25	Jomegat[90]
1	Jorgenev[91]
1	Jwwicks[92]
1	Kayau[93]
1	Kazabubu[94]
23	Kevinpaladin[95]
1	Keytotime[96]

72 http://en.wikibooks.org/wiki/User:HethrirBot
73 http://en.wikibooks.org/wiki/Special:Contributions/Hongooi
74 http://en.wikibooks.org/wiki/User:Hoxel
75 http://en.wikibooks.org/wiki/Special:Contributions/I_do_not_exist
76 http://en.wikibooks.org/wiki/User:Icewedge
77 http://en.wikibooks.org/wiki/User:Imran
78 http://en.wikibooks.org/wiki/User:Intermediate-Hacker
79 http://en.wikibooks.org/wiki/User:Iopq
80 http://en.wikibooks.org/wiki/User:J.delanoy
81 http://en.wikibooks.org/wiki/Special:Contributions/JAnDbot
82 http://en.wikibooks.org/wiki/Special:Contributions/JL-Bot
83 http://en.wikibooks.org/wiki/User:JackPotte
84 http://en.wikibooks.org/wiki/User:Jafeluv
85 http://en.wikibooks.org/wiki/User:James_Dennett
86 http://en.wikibooks.org/wiki/Special:Contributions/JetRanger405
87 http://en.wikibooks.org/wiki/User:Jfmantis
88 http://en.wikibooks.org/wiki/User:Jguk
89 http://en.wikibooks.org/wiki/User:Jni
90 http://en.wikibooks.org/wiki/User:Jomegat
91 http://en.wikibooks.org/wiki/User:Jorgenev
92 http://en.wikibooks.org/wiki/User:Jwwicks
93 http://en.wikibooks.org/wiki/User:Kayau
94 http://en.wikibooks.org/wiki/Special:Contributions/Kazabubu
95 http://en.wikibooks.org/wiki/User:Kevinpaladin
96 http://en.wikibooks.org/wiki/User:Keytotime

1	Kiensvay[97]
2	Kinglag[98]
1	Kj[99]
25	Krischik[100]
1	Ksn[101]
1	Kvgk[102]
1	Leftspk[103]
2	Liam987[104]
1	Lightbot[105]
3	Lincher[106]
3	Logictheo[107]
1	Lynx7725[108]
8	M2s87[109]
1	MF-Warburg[110]
1	MaTT[111]
4	MadCowpoke[112]
51	Maffu[113]
1	Mahanga[114]
1	ManiacK[115]
2	ManuelGR[116]
1	Markhobley[117]
1	Martyn Lovell[118]
1	Mattb112885[119]
3	MeMoria[120]
1	Mecanismo[121]

97 http://en.wikibooks.org/wiki/User:Kiensvay
98 http://en.wikibooks.org/wiki/User:Kinglag
99 http://en.wikibooks.org/wiki/User:Kj
100 http://en.wikibooks.org/wiki/User:Krischik
101 http://en.wikibooks.org/wiki/Special:Contributions/Ksn
102 http://en.wikibooks.org/wiki/Special:Contributions/Kvgk
103 http://en.wikibooks.org/wiki/Special:Contributions/Leftspk
104 http://en.wikibooks.org/wiki/User:Liam987
105 http://en.wikibooks.org/wiki/Special:Contributions/Lightbot
106 http://en.wikibooks.org/wiki/User:Lincher
107 http://en.wikibooks.org/wiki/User:Logictheo
108 http://en.wikibooks.org/wiki/User:Lynx7725
109 http://en.wikibooks.org/wiki/User:M2s87
110 http://en.wikibooks.org/wiki/User:MF-Warburg
111 http://en.wikibooks.org/wiki/Special:Contributions/MaTT
112 http://en.wikibooks.org/wiki/User:MadCowpoke
113 http://en.wikibooks.org/wiki/User:Maffu
114 http://en.wikibooks.org/wiki/User:Mahanga
115 http://en.wikibooks.org/wiki/Special:Contributions/ManiacK
116 http://en.wikibooks.org/wiki/User:ManuelGR
117 http://en.wikibooks.org/wiki/User:Markhobley
118 http://en.wikibooks.org/wiki/Special:Contributions/Martyn_Lovell
119 http://en.wikibooks.org/wiki/User:Mattb112885
120 http://en.wikibooks.org/wiki/User:MeMoria
121 http://en.wikibooks.org/wiki/User:Mecanismo

8	Merrheim[122]
1	Michael Safyan[123]
3	Mike.lifeguard[124]
6	Mikeblas[125]
1	MithrandirAgain[126]
3	Monobi[127]
2	Mortense[128]
1	Mr.Z-man[129]
4	Mrquick[130]
1	Mshonle[131]
1	Mwtoews[132]
1	Netizen[133]
1	Newmanbe[134]
1	Nick[135]
2	NithinBekal[136]
10	Noogz[137]
1	OMouse[138]
1	Onion Bulb[139]
216	Orderud[140]
2	Otus[141]
1	Outlyer[142]
1	PGibbons[143]
18	Paddu[144]
33	Panic2k4[145]
20	Pcu123456789[146]

122 http://en.wikibooks.org/wiki/User:Merrheim
123 http://en.wikibooks.org/wiki/Special:Contributions/Michael_Safyan
124 http://en.wikibooks.org/wiki/User:Mike.lifeguard
125 http://en.wikibooks.org/wiki/Special:Contributions/Mikeblas
126 http://en.wikibooks.org/wiki/User:MithrandirAgain
127 http://en.wikibooks.org/wiki/User:Monobi
128 http://en.wikibooks.org/wiki/User:Mortense
129 http://en.wikibooks.org/wiki/User:Mr.Z-man
130 http://en.wikibooks.org/wiki/User:Mrquick
131 http://en.wikibooks.org/wiki/User:Mshonle
132 http://en.wikibooks.org/wiki/User:Mwtoews
133 http://en.wikibooks.org/wiki/Special:Contributions/Netizen
134 http://en.wikibooks.org/wiki/User:Newmanbe
135 http://en.wikibooks.org/wiki/User:Nick
136 http://en.wikibooks.org/wiki/User:NithinBekal
137 http://en.wikibooks.org/wiki/User:Noogz
138 http://en.wikibooks.org/wiki/User:OMouse
139 http://en.wikibooks.org/wiki/Special:Contributions/Onion_Bulb
140 http://en.wikibooks.org/wiki/User:Orderud
141 http://en.wikibooks.org/wiki/User:Otus
142 http://en.wikibooks.org/wiki/Special:Contributions/Outlyer
143 http://en.wikibooks.org/wiki/Special:Contributions/PGibbons
144 http://en.wikibooks.org/wiki/User:Paddu
145 http://en.wikibooks.org/wiki/User:Panic2k4
146 http://en.wikibooks.org/wiki/User:Pcu123456789

9	Phosgram[147]
3	Pietrodn[148]
1	Public Menace[149]
6	PurplePieman[150]
3	QUBot[151]
14	QuiteUnusual[152]
4	Qwerky[153]
12	Recent Runes[154]
1	Redlentil[155]
10	Remi0o[156]
1	RibotBOT[157]
1	SPat[158]
1	STBot[159]
1	SieBot[160]
1	Sietse Snel[161]
89	Sigma 7[162]
1	Snowolf[163]
1	Somercet[164]
3	SoniyaR[165]
2	Sprink[166]
1	Stassats[167]
1	Steven jones[168]
1	Superm401[169]
2	Suruena[170]
5	TakuyaMurata[171]

147 http://en.wikibooks.org/wiki/User:Phosgram
148 http://en.wikibooks.org/wiki/User:Pietrodn
149 http://en.wikibooks.org/wiki/Special:Contributions/Public_Menace
150 http://en.wikibooks.org/wiki/User:PurplePieman
151 http://en.wikibooks.org/wiki/User:QUBot
152 http://en.wikibooks.org/wiki/User:QuiteUnusual
153 http://en.wikibooks.org/wiki/User:Qwerky
154 http://en.wikibooks.org/wiki/User:Recent_Runes
155 http://en.wikibooks.org/wiki/User:Redlentil
156 http://en.wikibooks.org/wiki/User:Remi0o
157 http://en.wikibooks.org/wiki/Special:Contributions/RibotBOT
158 http://en.wikibooks.org/wiki/User:SPat
159 http://en.wikibooks.org/wiki/Special:Contributions/STBot
160 http://en.wikibooks.org/wiki/Special:Contributions/SieBot
161 http://en.wikibooks.org/wiki/Special:Contributions/Sietse_Snel
162 http://en.wikibooks.org/wiki/User:Sigma_7
163 http://en.wikibooks.org/wiki/User:Snowolf
164 http://en.wikibooks.org/wiki/Special:Contributions/Somercet
165 http://en.wikibooks.org/wiki/User:SoniyaR
166 http://en.wikibooks.org/wiki/User:Sprink
167 http://en.wikibooks.org/wiki/User:Stassats
168 http://en.wikibooks.org/wiki/Special:Contributions/Steven_jones
169 http://en.wikibooks.org/wiki/User:Superm401
170 http://en.wikibooks.org/wiki/User:Suruena
171 http://en.wikibooks.org/wiki/User:TakuyaMurata

1	TelecomNut[172]
1	Thijs!bot[173]
33	Thunderbunny[174]
1	TimR[175]
1	Ttv[176]
4	Webaware[177]
36	Whiteknight[178]
2	Whym[179]
1	Wik[180]
1	WikHead[181]
1	Wikidemon[182]
2	Wj32[183]
3	Xania[184]
1	Xerol[185]
1	Xqbot[186]
16	Yacht[187]
1	Ygfperson[188]
1	Zoohouse[189]
3	タチコマ robot[190]

172 http://en.wikibooks.org/wiki/Special:Contributions/TelecomNut
173 http://en.wikibooks.org/wiki/Special:Contributions/Thijs!bot
174 http://en.wikibooks.org/wiki/User:Thunderbunny
175 http://en.wikibooks.org/wiki/Special:Contributions/TimR
176 http://en.wikibooks.org/wiki/User:Ttv
177 http://en.wikibooks.org/wiki/User:Webaware
178 http://en.wikibooks.org/wiki/User:Whiteknight
179 http://en.wikibooks.org/wiki/User:Whym
180 http://en.wikibooks.org/wiki/User:Wik
181 http://en.wikibooks.org/wiki/User:WikHead
182 http://en.wikibooks.org/wiki/Special:Contributions/Wikidemon
183 http://en.wikibooks.org/wiki/User:Wj32
184 http://en.wikibooks.org/wiki/User:Xania
185 http://en.wikibooks.org/wiki/User:Xerol
186 http://en.wikibooks.org/wiki/Special:Contributions/Xqbot
187 http://en.wikibooks.org/wiki/User:Yacht
188 http://en.wikibooks.org/wiki/Special:Contributions/Ygfperson
189 http://en.wikibooks.org/wiki/User:Zoohouse
190 http://en.wikibooks.org/wiki/User:%25E3%2582%25BF%25E3%2583%2581%25E3%2582%25B3%25E3%2583%259E_robot

List of Figures

- EPL: Eclipse Public License. `http://www.eclipse.org/org/documents/epl-v10.php`

Copies of the GPL, the LGPL as well as a GFDL are included in chapter Licenses[191]. Please note that images in the public domain do not require attribution. You may click on the image numbers in the following table to open the webpage of the images in your webbrower.

191 Chapter 41 on page 277

1	Emijrpbot, Jarkko Piiroinen	
2	Berland, Derbeth, MGA73bot2, SchlurcherBot, Slobot, Ufo karadagli	
3	Pietrodn	CC-BY-SA-2.5

41 Licenses

41.1 GNU GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Copyright © 2007 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed. Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow. TERMS AND CONDITIONS 0. Definitions.

“This License” refers to version 3 of the GNU General Public License.

“Copyright” also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

“The Program” refers to any copyrightable work licensed under this License. Each licensee is addressed as “you”. “Licensees” and “recipients” may be individuals or organizations.

To “modify” a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a “modified version” of the earlier work or a work “based on” the earlier work.

A “covered work” means either the unmodified Program or a work based on the Program.

To “propagate” a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To “convey” a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays “Appropriate Legal Notices” to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion. 1. Source Code.

The “source code” for a work means the preferred form of the work for making modifications to it. “Object code” means any non-source form of a work.

A “Standard Interface” means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The “System Libraries” of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A “Major Component”, in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The “Corresponding Source” for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work. 2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary. 3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures. 4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee. 5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

* a) The work must carry prominent notices stating that you modified it, and giving a relevant date. * b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to “keep intact all notices”. * c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it. * d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an “aggregate” if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate. 6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

* a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange. * b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge. * c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b. * d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements. * e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A “User Product” is either (1) a “consumer product”, which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, “normally used” refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

“Installation Information” for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying. 7. Additional Terms.

“Additional permissions” are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

* a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or * b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or * c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or * d) Limiting the use for publicity purposes of names of licensors or authors of the material; or * e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or * f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered “further restrictions” within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way. 8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10. 9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so. 10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An “entity transaction” is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party’s predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it. 11. Patents.

A “contributor” is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor’s “contributor version”.

A contributor’s “essential patent claims” are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, “control” includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor’s essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a “patent license” is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To “grant” such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. “Knowingly relying” means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient’s use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is “discriminatory” if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law. 12. No Surrender of Others’ Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program. 13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such. 14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License “or any later version” applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy’s public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version. 15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. 16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. 17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the “copyright” line and a pointer to where the full notice is found.

<one line to give the program’s name and a brief idea of what it does.> Copyright (C) <year> <name of author>

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

<program> Copyright (C) <year> <name of author> This program comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’. This is free software, and you are welcome to redistribute it under certain conditions; type ‘show c’ for details.

The hypothetical commands ‘show w’ and ‘show c’ should show the appropriate parts of the General Public License. Of course, your program’s commands might be different; for a GUI interface, you would use an “about box”.

You should also get your employer (if you work as a programmer) or school, if any, to sign a “copyright disclaimer” for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see <http://www.gnu.org/licenses/>.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read <http://www.gnu.org/philosophy/why-not-lgpl.html>.

41.2 GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed. 0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference. 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License. 2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies. 3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document’s license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document. 4. MODIFICATIONS

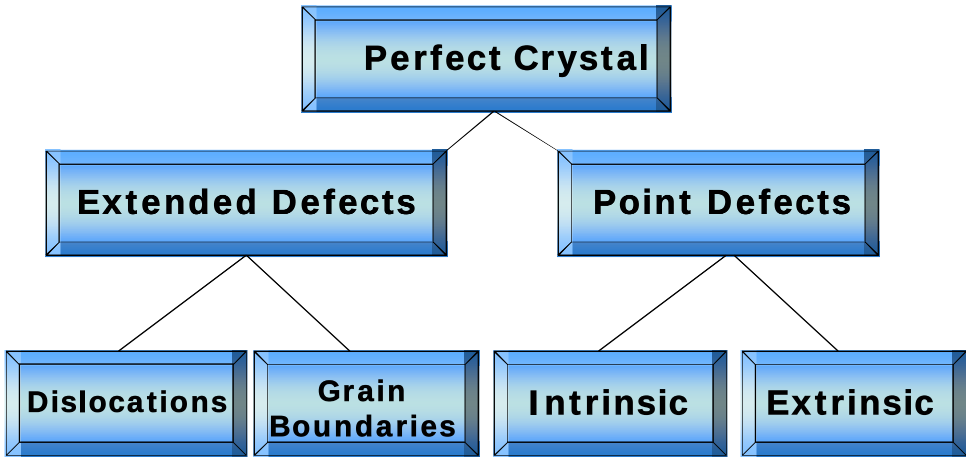
You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

* A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that

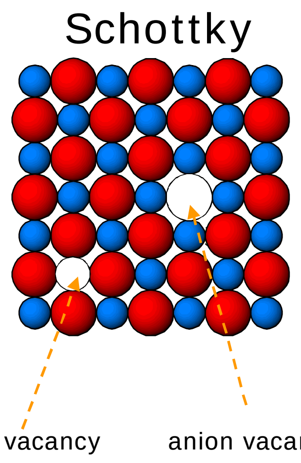
41.3 GNU Lesser General Public License

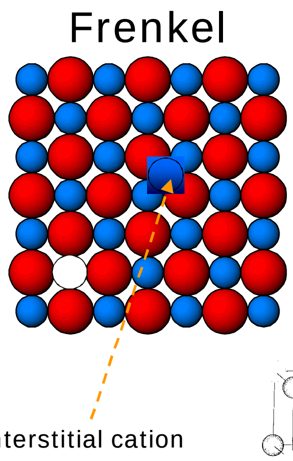
Chapter 5:

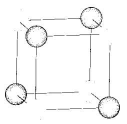
Defects and Non-stoichiometry



Point Defect - Intrinsic







$Na^+ + Cl^- \rightarrow V_{na} + V_{Cl}$ $Ag^+ \rightarrow V_{Ag} + Ag^+_{interstitial}$

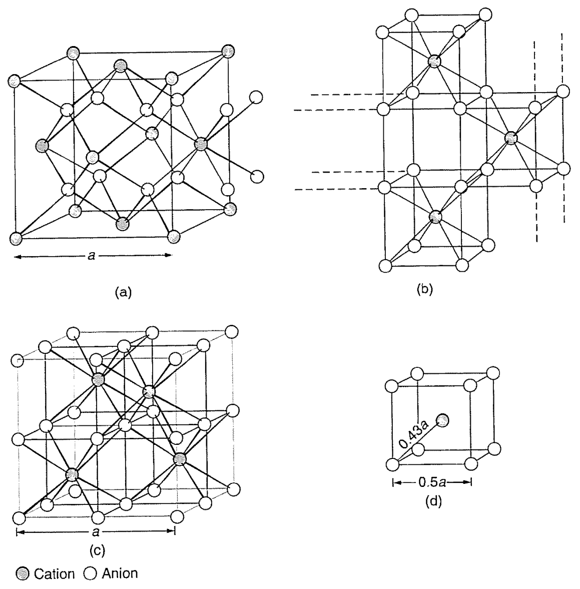
Anion Frenkel defect in fluorite

Example fluorites include CaF_2, SrF_2, PbF_2, ThO_2, UO_2, ZrO_2

Cation Frenkel defects are common because of the typically smaller size of a cation compared to an anion.

•However, anions in the fluorite structure have a lower electrical charge than the cations and don't find it as difficult to move nearer each other.

•The fluorite structure *ccp* cations with all tetrahedral holes occupied by the anions – thus all octahedral holes are unoccupied.



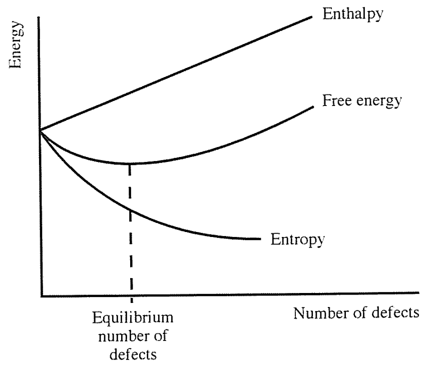
Concentration of defects

Energy is required to form a defect (endothermic process)

•Although there is a cost in ***energy***, there is a gain in ***entropy*** in the formation of a defect.

•At equilibrium, the overall change in free energy of the crystal due to the defect formation is zero according to:

$$\Delta G = \Delta H - T\Delta S$$



At any temperature, there will always be an equilibrium population of defects. The number of defects (for an MX crystal) is given by:

$$n_s \approx N \exp\left(\frac{-\Delta H_s}{2kT}\right)$$

where n_s is the number of Schottky defects per unit volume, at T K, in a crystal with N cations and N anion sites per unit cell volume, and ΔH_s is the enthalpy required to form one defect.

Concentration of defects, cont.

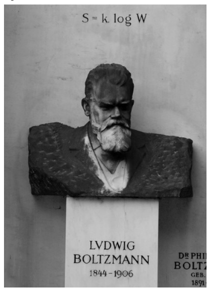
Estimate the configurational entropy, the change of entropy due to the vibrations of atoms around the defects and the arrangement of defects, using methods of statistical mechanics.

•If the number of Schottky defects is n_s per unit volume at T K, then there are n_s cation vacancies and n_s anion vacancies in a crystal containing N possible cation sites and N possible anion sites.

•The Boltzmann formula tells us that the entropy of such a system is:

$$S = k\ln W$$

where W is the number of ways of distributing n_s defects over N possible sites at random, and k is the Boltzmann constant (1.38x10^{-23} J/K)



•Probability theory shows that W is given by:

$$W = \frac{N!}{(N-n)!n!}$$

N! is 'factorial N'. N×(N-1)×(N-2)...×1

Concentration of defects, cont.

Number of ways on can distribute *cation* vacancies $W_c = \frac{N!}{(N-n_s)!n_s!}$

Number of ways on can distribute *anion* vacancies $W_a = \frac{N!}{(N-n_s)!n_s!}$

The total number of ways of distributing these defects, W, is:

$$W = W_c W_a$$

The *change* in entropy due to introducing defects into a perfect crystal:

$$\Delta S = k \ln W = k \ln\left(\frac{N!}{(N-n_s)!n_s!}\right)^2 = 2k \ln\left(\frac{N!}{(N-n_s)!n_s!}\right)$$

Concentration of defects, cont.

Simplify using Stirling's approximation:

$$\ln N! \approx N \ln N - N$$

thus:

$$\Delta S = 2k\{N \ln N - (N-n_s)\ln(N-n_s) - n_s \ln n_s\}$$

If the enthalpy change for the formation of a single defect is ΔH_s and then assume that the enthalpy change for the formation of n_s defects is $n_s \Delta H_s$ then the Gibbs free energy change is given by:

$$\Delta G = n_s \Delta H_s - 2kT\{N \ln N - (N-n_s)\ln(N-n_s) - n_s \ln n_s\}$$

At equilibrium, Gibbs free energy of the system must be a minimum with respect to changes in the number of defects, n_s. $\left(\frac{d\Delta G}{dn_s}\right) = 0$

$$\Delta H_s - 2kT \frac{d}{dn_s}\{N \ln N - (N-n_s)\ln(N-n_s) - n_s \ln n_s\} = 0$$

Concentration of defects, cont.

$$\Delta H_s - 2kT\frac{d}{dn_s}\{N\ln N - (N-n_s)\ln(N-n_s) - n_s\ln n_s\} = 0$$

*N*ln*N* is a constant, so differential becomes zero
Differential of:
ln *x* is 1/*x*
*x*ln*x* is (1+ ln*x*)

$$\Delta H_s - 2kT\{\ln(N-n_s)+1-\ln n_s - 1\} = 0$$

$$\Delta H_s = 2kT\ln\left[\frac{(N-n_s)}{n_s}\right] \longrightarrow n_s = (N-n_s)\exp\left(\frac{-\Delta H_s}{2kT}\right)$$

Since $N >> N_s$, approximate N-n_s as N

$$n_s \approx N\exp\left(\frac{-\Delta H_s}{2kT}\right) \xrightarrow[\text{quantities}]{\text{in molar}} n_s \approx N\exp\left(\frac{-\Delta H_s}{2RT}\right)$$

where R is 8.314 J/mol K

ΔH_s is the enthalpy required to form one mole of Schottky defects

Concentration of defects, cont.

The number of Frenkel defects present in a MX crystal is:

$$n_s \approx (NN_i)^{1/2}\exp\left(\frac{-\Delta H_F}{2kT}\right)$$

where n_f is the number of Frenkel defects per unit volume, *N* is the number of lattice sites, and N_i the number of interstitial sites available., and ΔH_F is the enthalpy of formation of one Frenkel defect:

If ΔH_F is the enthalpy of formation of one *mole* of Frenkel defects:

$$n_s \approx (NN_i)^{1/2}\exp\left(\frac{-\Delta H_F}{2RT}\right)$$

Knowing the enthalpy of formation for Schottky and Frenkel defects, one can *estimate* how many defects are present in a crystal.

Schottky Defects

Compound	ΔH (10^{-19} J)	ΔH (eV)
MgO	10.57	6.60
CaO	9.77	6.10
LiF	3.75	2.34
LiCl	3.40	2.12
LiBr	2.88	1.80
LiI	2.08	1.30
NaCl	3.69	2.30
KCl	3.62	2.26

Frenkel Defects

Compound	ΔH (10^{-19} J)	ΔH (eV)
UO_2	5.45	3.40
ZrO_2	6.57	4.10
CaF_2	4.49	2.80
SrF_2	1.12	0.70
AgCl	2.56	1.60
AgBr	1.92	1.20
β-AgI	1.12	0.70

Assuming $\Delta H_s = 5\times10^{-19}$ J, the proportion of vacant sites n_s/N at 300 K is 6.12×10^{-27}, whereas at 1000 K this increases to 1.37×10^{-8}

At room temperature there are very few Schottky defects, even at 1000K there are only about 1 or 2 defects *per hundred million* sites.

Depending on the value of ΔH, a Schottky or Frenkel defect may be present. The lower ΔH dominates, but in some crystals it is possible that *both* types of defects may be present.

Increasing temperature increases defects, in agreement with the endothermic process and Le Chatelier's principle.

Extrinsic defects

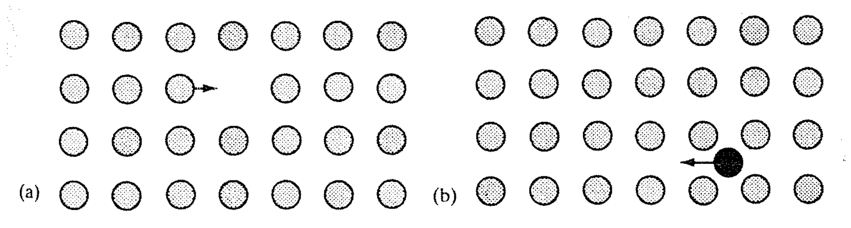
Doping with selected 'impurities' can introduce vacancies into a crystal.

•Consider incorporating $CaCl_2$ into NaCl, in which each Ca^{2+} replaces two Na^{+} and creates one cation vacancy.

Defects and Ionic Conductivity in Solids

Defects make it possible for atoms or ions to move, through diffusion through the lattice or ionic conductivity (ions under the influence of an external electric field) through the structure.

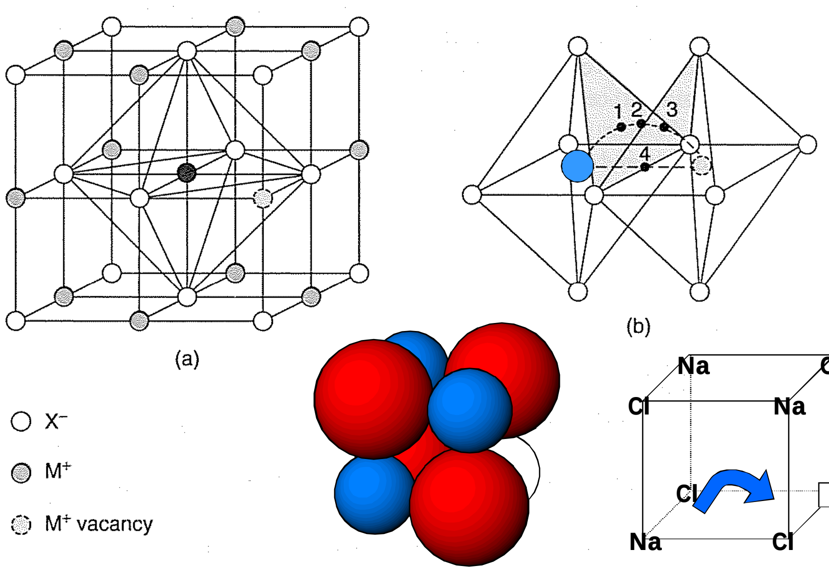
Two possible mechanisms for the movement of ions through a lattice:



Vacancy mechanism   Interstitial mechanism

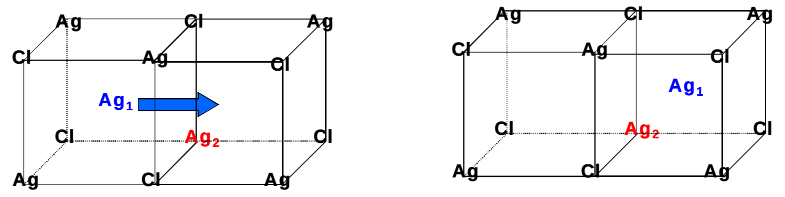
Ion Migration (Schottky defects)

Na^+ ions move, but meet resistance in the crystal structure

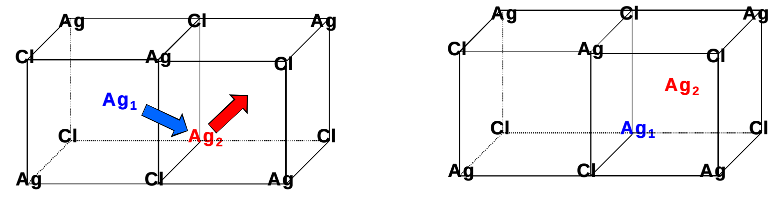


Ion Migration (Frenkel Defects)

The Frenkel defects in AgCl can migrate via two mechanisms.

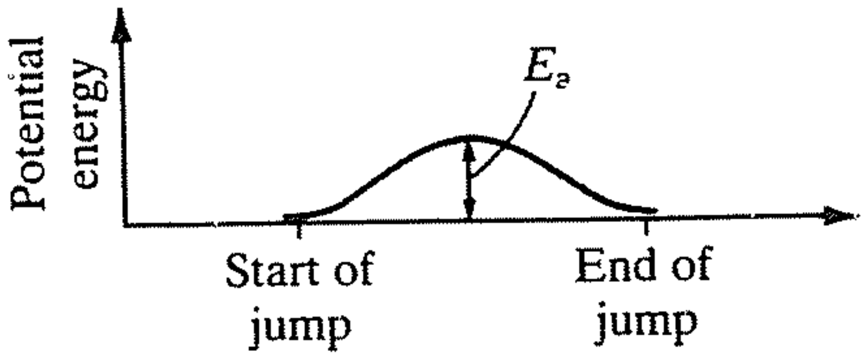


Direct Interstitial Jump



Interstitialcy Mechanism

The energy required to make the jump, E_a, is the activation energy.



Ionic Conductivity

Ionic Conductivity, σ, is defined the same as electrical conductivity:

$$\sigma = nZe\mu$$

where n is the number of charge carriers per unit volume, Ze is the charge ($e = 1.602189 \times 10^{-19}$ C), and μ is the mobility, which is a measure of the drift velocity in a constant electric field.

	Material	**Conductivity / (S m^{-1})**
Ionic Conductors	Ionic crystals	$<10^{-16} – 10^{-2}$
	Solid electrolytes	$10^{-1} – 10^{3}$
	Strong (liquid) electrolytes	$10^{-1} – 10^{3}$
Electronic conductors	Metals	$10^{3} – 10^{7}$
	Semiconductors	$10^{-3} – 10^{4}$
	Insulators	$<10^{-10}$

Ionic Conductivity

The temperature dependence of the mobility of the ions can be expressed by an Arrhenius equation.

$\mu \propto \exp\left(\frac{-E_a}{kT}\right)$ or $\mu = \mu_0 \exp\left(\frac{-E_a}{kT}\right)$ where μ_0 is a proportionality constant known as the pre-exponential factor

μ_0 depends on the attempt frequency (frequency of vibration of the lattice 10^{12}-10^{13} Hz), distance moved by ion, and the size of the external field.

If the external field is small (up to 300 V cm^{-1}), a temperature dependence of 1/T is present in the pre exponential factor.

An expression for the variation of ionic conductivity: $\sigma = \frac{\sigma_0}{T}\exp\left(\frac{-E_a}{T}\right)$

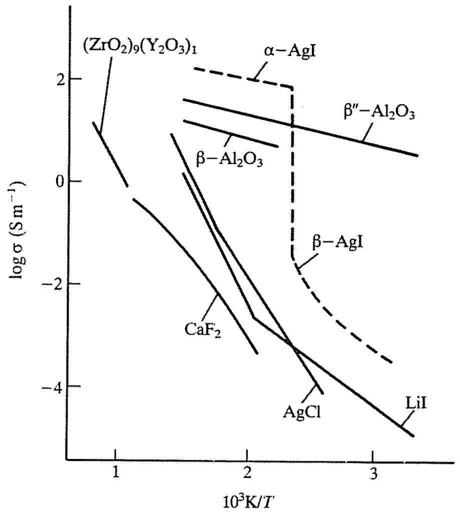
The term σ_0 contains *n* and *Ze* as well as the attempt frequency and jump distance. Taking logs...

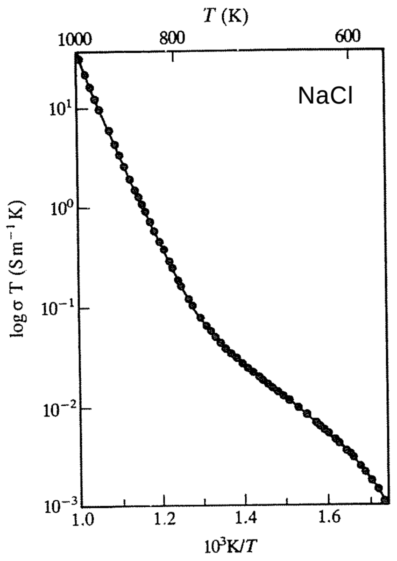
$$\ln \sigma T = \ln \sigma_0 - \left(\frac{E_a}{T}\right)$$

Plotting lnσT vs 1/T should produce a straight line with a slope of $-E_a$.

lnσ vs 1/T is also used

Conductivities of Solid Electrolytes vs Temperature





Differences in slopes are evident, even in very pure crystals.

- At **low** temperatures extrinsic vacancies are most important.
  - The concentration of intrinsic vacancies are so small at low temperature that they may be ignored
  - The number of vacancies will be essentially constant
  - μ in the extrinsic region thus will only depend on the cation mobility due to extrinsic defects, with the temperature dependence:

$$\mu = \mu_0 \exp\left(\frac{-E_a}{kT}\right)$$

At **high** temperatures the concentration of *intrinsic* defects has increased so that it is similar or greater then the concentration of *extrinsic* defects

$$n_s \approx N \exp\left(\frac{-\Delta H_s}{2kT}\right)$$

The conductivity in this intrinsic region on the left side of the plot:

$$\sigma = \frac{\sigma'}{T}\exp\left(\frac{-E_a}{2RT}\right)\exp\left(\frac{\Delta H_s}{2kT}\right)$$

A plot of $\ln\sigma T$ vs $1/T$ gives a larger value for the activation energy (E_s), because it depends on both the activation energy for the cation jump (E_a) and the enthalpy of formation of a Schottky defect. $E_s = E_a + 1/2\Delta H_s$

For a system with Frenkel defects, $E_F = E_a + 1/2\Delta H_F$

Activation energies typically lie in the range of 0.05 to 1.1 eV.

Solid Electrolytes

Ionic conductivity of solids is important towards the development of solid state batteries.

- Primary batteries are not reversible and are discarded after use.
- Secondary or storage batteries are reversible and significant research is performed to improve properties of materials.

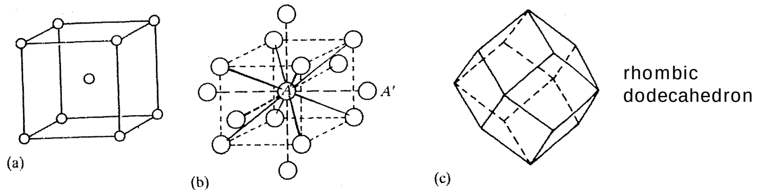
Fast ion conductors: α-AgI

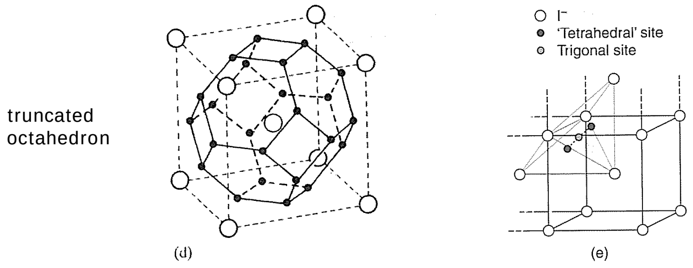
α-AgI exists as two phases below 146°C

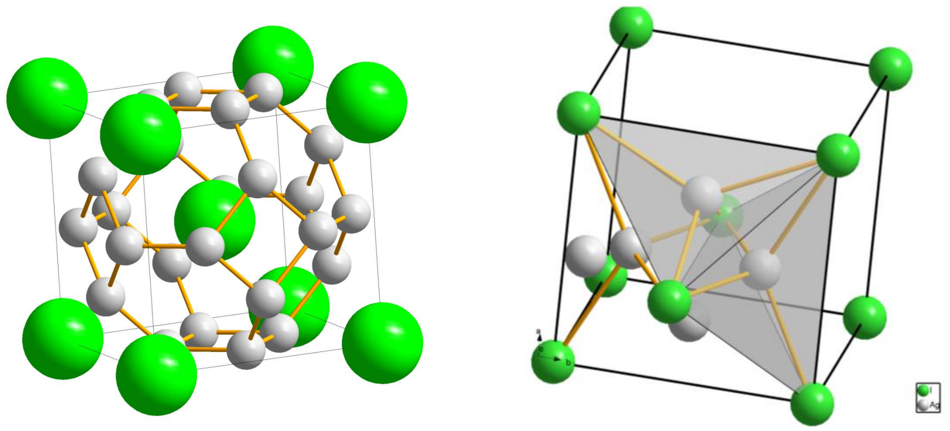
γ-AgI (zinc blende) and β-AgI (wurtzite), both → ccp I^-

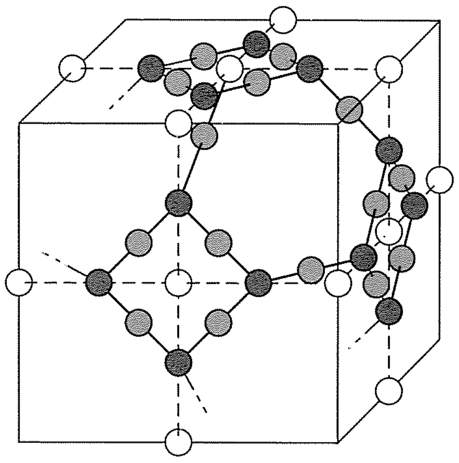
above 146°C α-AgI has I^- in body centered cubic arrangement.

The conductivity of α-AgI is very high, ~10^4 higher than γ-AgI & β-AgI









● 'Tetrahedral' site

● Trigonal site

○ Distorted octahedral site

•There are many possible positions for the Ag^+ to occupy, 6 are distorted octahedral, 12 are ~tetrahedral, and 24 are trigonal, giving 42 possible sites.

•Structural determinations indicate the Ag^+ ions are statistically distributed among the twelve tetrahedral sites.

•There are five spare sites available per Ag^+ atom.

•Silver moves from site to site by jumping though a vacant trigonal site, changing the coordination from 4-3-4, the activation energy is very low, 0.05 eV.

•Described as a molten sublattice of Ag^+.

Fast ion conductors: $RbAg_4I_5$

The 146°C is higher than desired for many applications, so a search for other solids with high ionic conductivity resulted in $RbAg_4I_5$.

•Has an ionic conductivity of 25 S/m and activation energy of 0.07 eV.

•The Rb^+ and I^- form a rigid array while the Ag^+ ions are randomly distributed over a network of tetrahedral sites.

•A conducting solid electrolyte must have high ionic conductivity, but negligible electronic conductivity (otherwise a short circuit).

•A cell may be constructed with Ag and RbI_3 electrodes.

•Cells constructed operate between -55 to +200°C, have long shelf life, and can withstand mechanical shock.

Anion Structure	bcc	ccp	hcp	other
	α-AgI	α-CuI	β-CuBr	$RbAg_4I_5$
	α-CuBr	α-Ag_2Te		
	α-Ag_2S	α-Cu_2Se		
	α-Ag_2Se	α-Ag_2HgI_4		

Fast ion conductors: oxygen ion conductors

Fluorite structure has some empty space that may enable an F^- ion to move into an interstitial site. Some fluorites (PbF_2) exhibits low ionic conductivity at room temperature, but increases to ~500 S/m at 500°C.

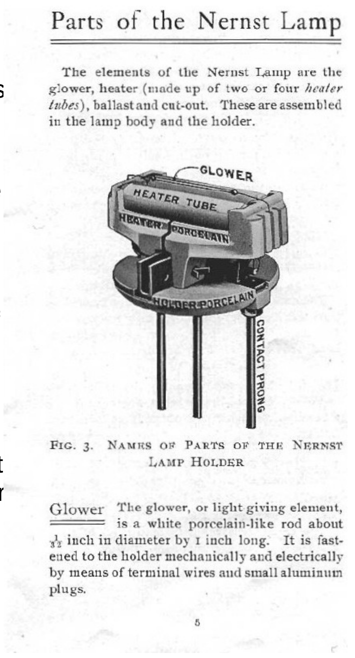
•Many oxides also adopt the fluorite structure (UO_2, ThO_2, CeO_2)

•Nernst found that mixed oxides of Y_2O_3 and ZrO_2 glowed white hot if an electrical current passed, which was attributed to conduction of oxide ions.

•These doped zirconia oxides were used for filaments in 'glower' electric lights.

•The cubic form of ZrO_2 (fluorite) is formed at high temperature or when doped with another element.

•Addition of either Y_2O_3 (yttria stabilized zirconia, YSZ) or CaO to ZrO_2



Parts of the Nernst Lamp

The elements of the Nernst Lamp are the glower, heater (made up of two or four *heater tubes*), ballast and cut-out. These are assembled in the lamp body and the holder.

GLOWER
HEATER TUBE
HEATER PORCELAIN
HOLDER PORCELAIN
CONTACT PRONG

FIG. 3. NAMES OF PARTS OF THE NERNST LAMP HOLDER

Glower The glower, or light giving element, is a white porcelain-like rod about $\frac{1}{32}$ inch in diameter by 1 inch long. It is fastened to the holder mechanically and electrically by means of terminal wires and small aluminum plugs.

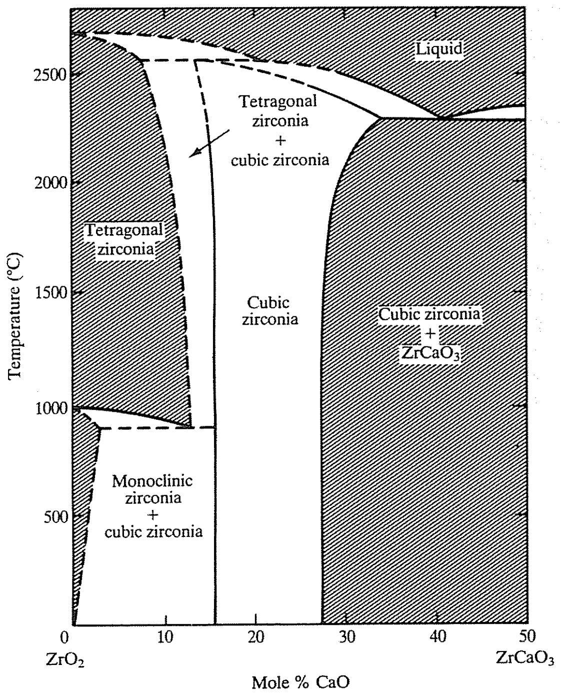
5

•If Ca^{2+} ions are located on Zr^{4+} sites, then compensating vacancies are created in the O^{2-} sublattice.

•Ca-doped ZrO_2 are very good fast-ion conductors of O^{2-} ions.

•Conductivity maximizes at relatively low concentrations of dopant, when the crystal lattice is distorted as little as possible.

•Two of the best oxygen ion conductors are ZrO_2 doped with Sc_2O_3, and CeO_2 doped with Gd_2O_3, and others based on CeO_2, ThO_2, HfO_2, and ZrO_2 are doped with other transition metals.



Oxygen ion conductors: Other

Perovskite:

Materials based on the perovskite lanthanum gallate, $LaGaO_3$, doped with Sr^{2+} and Mg^{2+}, produce $La_{1-x}Sr_xGa_{1-y}Mg_yO_{3-\delta}$ (LSGM).

•Has similar conductivities to zirconias, but at a lower operating temperature.

•For a cathode material in a solid oxide fuel cell, a material is needed that can conduct both ions and electrons. The Sr^{2+} doped $LaMnO_3$ (LSM) and $LaCrO_3$ (Sr) have both these properties.

LAMOX:

•Materials based on $La_2Mo_2O_9$ has high conductivity above 600°C, but tend to be susceptible to reduction by hydrogen.

BIMEVOX:

•Materials based on Bi_2O_3 have high conductivity above 600°C

Apatite:

•Structure based on $La_{10-x}M_6O_{26+y}$ (M = Si, Ge) conduct well at high temperatures.

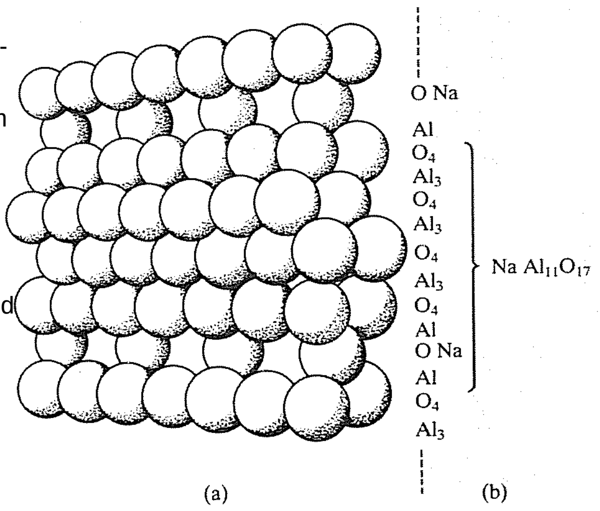
Fast-ion conductors: sodium ion conductors

•β-alumina is a series of compounds that exhibit fast-ion conducting properties.

•Parent compound is sodium β-alumina, $NaAl_{11}O_{17}$

•General formula is M_2O-nX_2O_3, where n can range from 5 to 11 and M is a monovalent cation (alkali metal)$^+$, Cu^+, Ag^+, NH_4^+, and X is a trivalent cation Al^{3+}, Ga^{3+}, or Fe^{3+}.

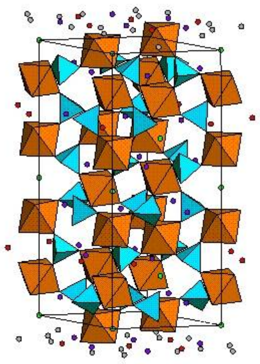
•High conductivity of the compound is related to the crystal structure.



•Close-packed layers of oxide ions, but in every fifth layer three-fourths of the oxygens are missing. The four close packed layers contain Al^{3+} ions in both octahedral and tetrahedral holes. The Na^+ are found in the fifth oxide layer [B(ABCA) C (ACBA) B]. Na^+ ions move in the conduction plane.

Fast-ion conductors: NASICON

$NaZr_2(PO_4)_3$ (NZP) consists of corner-linked ZrO_6 octahedra joined by PO_4 tetrahedra, each of which corner-shares to four octahedra.



•This arrangement creates a 3D system of channels with two types of vacant sites:

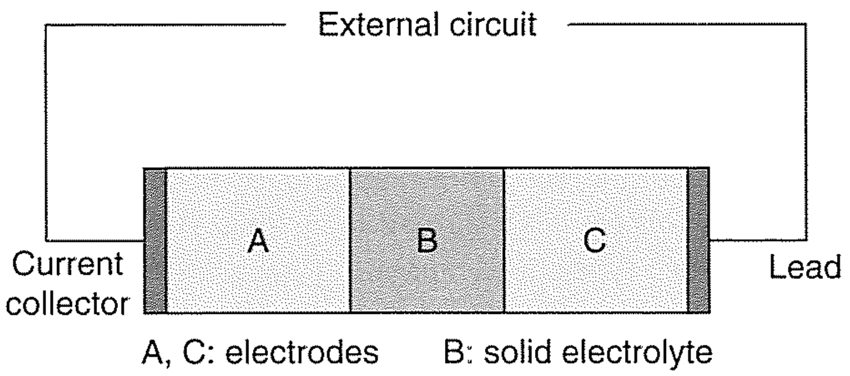
•Type I – a single distorted octahedral site occupied by Na^+ ions in NZP.

•Type II – a larger vacant site

•The structure type is very versatile and hundreds of compounds adopt it by varying the charge balancing 'A' cation with alkali or alkaline earth metals, the structural 'M' cation with transition metal, Ti, Zr, Nb, Cr, or Fe, and the P may be substituted with Si.

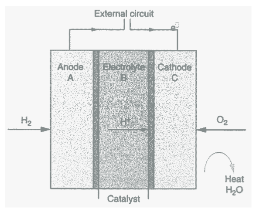
•The NASICON (**Na S**uper**I**onic **Con**ductor) has a conductivity of 20 S/m at 300°C and has the formula $Na_3Zr_2(PO_4)(SiO_4)_2$ and has three out of the four vacant sites occupied by Na^+.

Solid State Ionic Devices

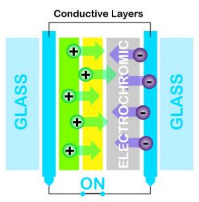


Batteries

Fuel Cells



Electrochromic Devices



Batteries

A battery is an electrochemical cell that produces an electric current at a constant voltage as a result of a chemical reaction.

•Ions travel through an **electrolyte** and are oxidized or reduced at the electrode.

•Oxidation occurs at the **anode**.

•Reduction occurs at the **cathode**.

•The solid electrolyte must have *high conductivity of <u>ions</u>*, but *not electrons* (electronic insulator).

•The electromotive force (emf), or voltage, produced by the cell under standard open circuit conditions is related to Gibb's free energy.

$$\Delta G^\circ = -nE^\circ F$$

where n is the number of electrons transferred in the reaction, E° is the standard emf of the cell (voltage delivered under standard, zero-current conditions) and F is the Faraday constant (96485 C/mol or 96485 J/V).

•Energy stored in a battery is related to the energy generated by the cell reaction and the amount of material used.

•Typically expressed in watt-hours (current*voltage*discharge time)

Lithium batteries

Energy density (watt-hours/battery volume in L) or specific energy (watt-hours divided by battery weight in kg) is a more useful indicator in applications where size or weight is critical.

•LiI has relatively low ionic conductivity, but was used in heart pacemaker batteries in the early 1970's, where a low current, small, long lasting, and *generate no gases during discharge*.

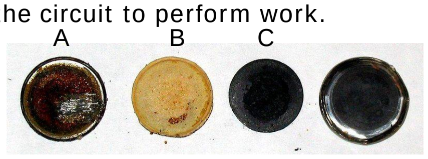
$$\begin{array}{ccc} A & B & C \\ \text{Li} \;// & \text{LiI} \;// & I_2 \text{ and polymer} \end{array}$$

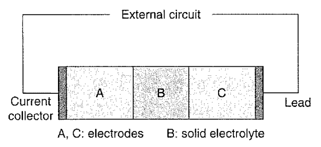
where the cathode is a conducting polymer with embedded iodine, poly-2-vinyl-pyridine

•Electrode reactions are:

Anode A: $2Li(s) \rightarrow 2Li^+(s) + 2e^-$ Cathode C: $I_2(s) + 2e^- \rightarrow 2I^-(s)$

•LiI contains *intrinsic Schottky defects* and the small Li^+ cations are able to pass through the solid electrolyte, while the electrons travel through the circuit to perform work.





Lithium-ion batteries

- Sony developed rechargeable lithium-ion batteries the are able to undergo many charge-discharge cycles.
- The lightweight batteries find use in many applications from mobile phones and laptop computers, etc.
- Driving reaction is that of Li with CoO_2 to form an intercalation compound, Li_xCoO_2 and anode is Li in graphitic carbon.

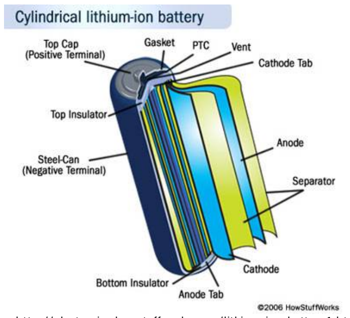
A B C

Li/C // Li^+ electrolyte // CoO_2

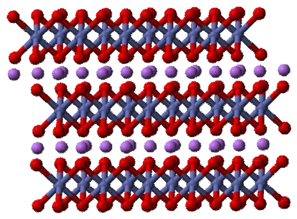
- Electrode reactions are:

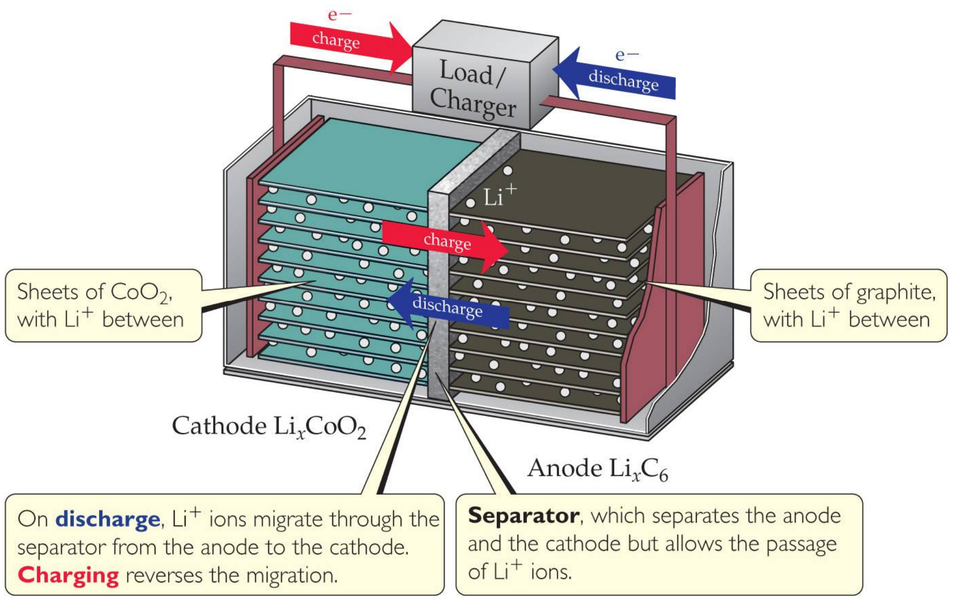
Anode A: $Li_xC_6(s) \rightarrow xLi^+ + 6C + xe^-$

Cathode C: $xLi^+ + CoO_2(s) + xe^- \rightarrow Li_xCoO_2(s)$



http://electronics.howstuffworks.com/lithium-ion-battery1.htm




e−
charge
Load / Charger
e−
discharge
Li+
charge
discharge
Sheets of CoO_2, with Li^+ between
Sheets of graphite, with Li^+ between
Cathode Li_xCoO_2
Anode Li_xC_6
On discharge, Li^+ ions migrate through the separator from the anode to the cathode. Charging reverses the migration.
Separator, which separates the anode and the cathode but allows the passage of Li^+ ions.

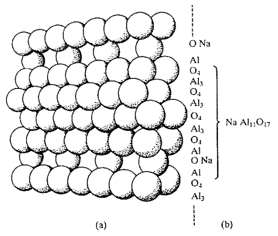
Sodium batteries

Na^+ conduction has been used in a high-temperature secondary battery, the sodium sulfur battery.

•Uses NASICON and β-alumina as the electrolyte.

•110 Wh/kg, with lightweight Na and energetic rxn.

•Electrolyte separates molten sulfur/molten sodium



A B C
Na(l) // β-alumina // S(l) and C(graphite)

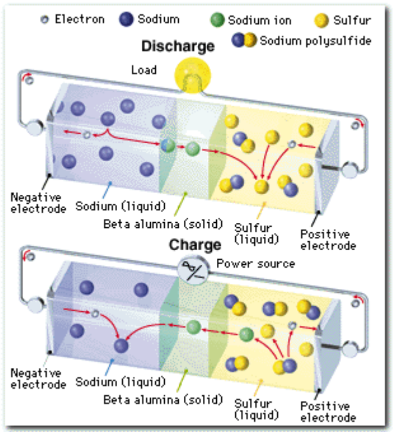
Anode A: $2Na(l) \rightarrow 2\,Na^+ + 2e^-$

Cathode C: $2Na^+ + 5S(l) + 2e^- \rightarrow Na_2S_5(l)$

Overall Reaction: $2Na(l) + 5S(l) \rightarrow Na_2S_5(l)$

•Reaction is completed when low polysulfides are formed, terminating with Na_2S_3

•Wind to battery project



Zebra batteries

Uses β-alumina as a Na^+ ion conductor.

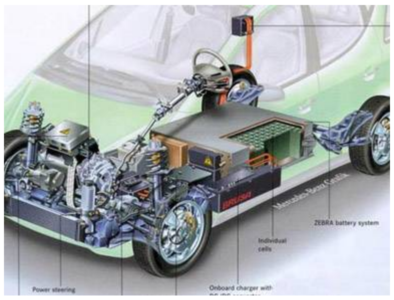
•Nickel chloride or a mixture of ferrous and nickel chlorides are used as the cathode.

•Current flow is improved by adding a second liquid electrolyte, molten $NaAlCl_4$ between the electrode and the β-alumina

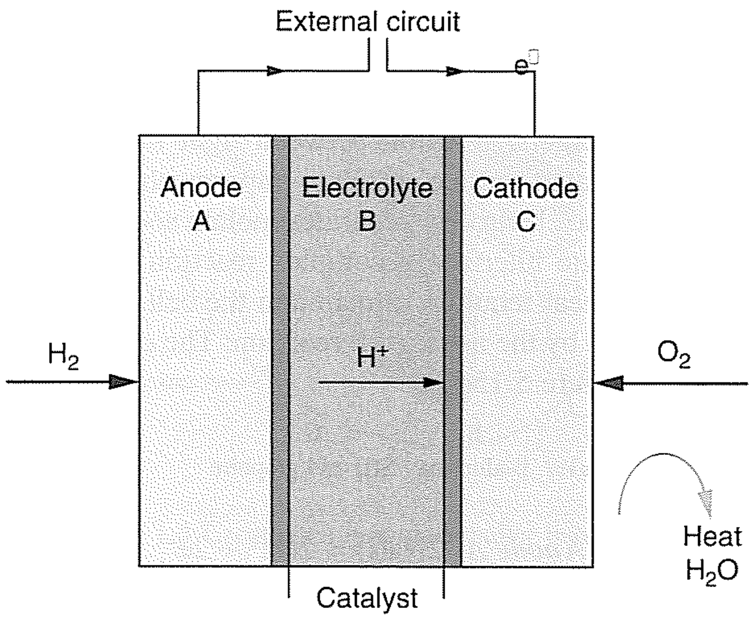
•Overall cell reaction is $2Na + NiCl_2 \rightarrow Ni + 2NaCl$

•Has high specific energy > 100 Wh/kg and gives electric vehicles a range of up to 250 km.

•Fully rechargeable, safe, and need no maintenance to over 100,000 km



Fuel Cells



Fuel Cells

Fuel cells differ from conventional batteries in that the fuel is fed in externally to the electrodes.

•Advantage: cell can operate continuously as long as fuel is available, unlike a battery that must be discarded (primary) or recharged (secondary).

•The fuels used are usually hydrogen and oxygen (air), which react electrochemically to produce water, electricity and heat.

•H_2 is fed to the anode where it is oxidized to H^+ ions and electrons.

•Electrons travel through the external circuit and the H^+ ions travel through the electrolyte to the cathode, where they react with O^{2-}.

•The reaction process is ‘green’ with byproducts of water and heat.

•The low temperature of the reaction means NO_x are avoided.

•Efficiency is up to about 50% or more, compared to 15-20% for ICE and 30% for diesel engines.

•Reduction of oxygen at the cathode is rather slow at low temperatures, therefore a Pt catalyst is incorporated into the carbon electrodes.

Fuel Cells

A B C

$H_2(g)//Pt/C$ electrode// hydrogen electrolyte //Pt/C electrode//$O_2(g)$

•Electrode reactions are:

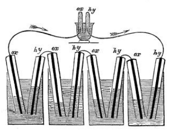
$$\text{Anode A: } H_2(g) \rightarrow 2H^+ + 2e^-$$

$$\text{Cathode C: } 1/2O_2(g) + 2H^+ + 2e^- \rightarrow H_2O$$

•The theoretical emf is $E^0 = 1.229$ V at 298K, but decreases to ~1 V at 500 K, so a compromise is needed between voltage and operating temperature.

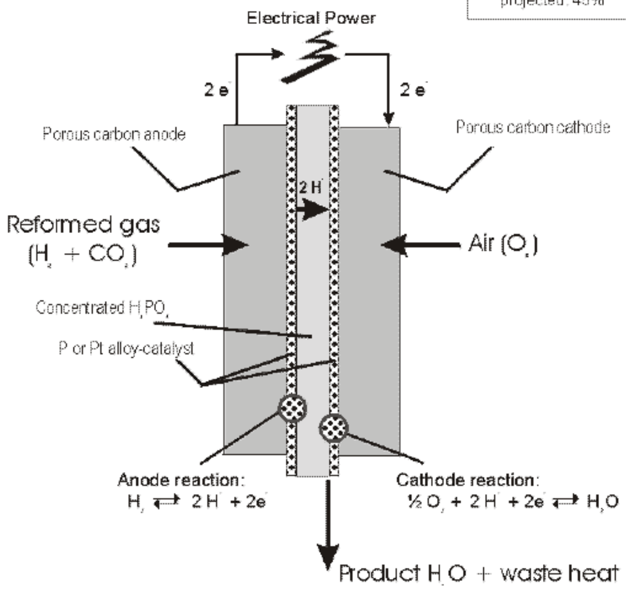
•Hydrogen storage is difficult, as well as the transportation (heavy cylinders for transportation), and changeover of the current infrastructure.

•Production of very pure hydrogen is energy intensive, thus cheap sources of electricity must be found (solar, hydroelectric, nuclear) or through the use of reforming reactions from methane or methanol with steam to produce hydrogen and CO_2.

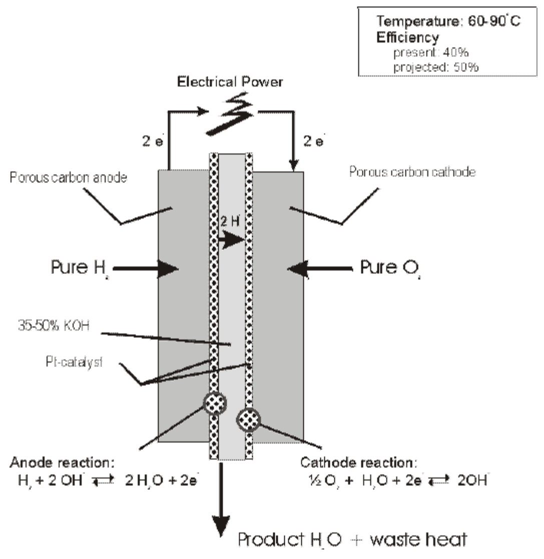


Phosphoric Acid Fuel Cell (PAFC)

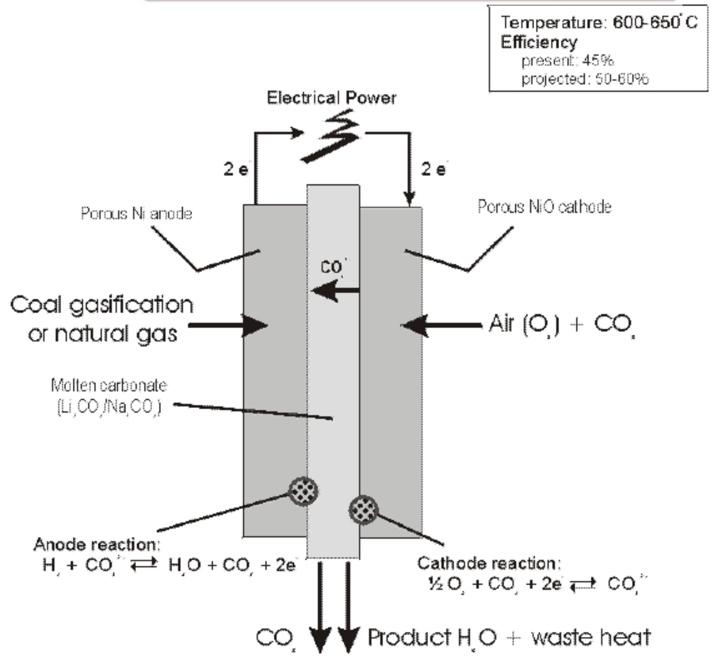
Temperature: 160-220°C
Efficiency
present: 40%
projected: 45%


Electrical Power
2 e⁻
2 e⁻
Porous carbon anode
Porous carbon cathode
2 H⁺
Reformed gas (H_2 + CO_2)
Air (O_2)
Concentrated H_3PO_4
P or Pt alloy-catalyst
Anode reaction: $H_2 \rightleftarrows 2H^+ + 2e^-$
Cathode reaction: $\frac{1}{2}O_2 + 2H^+ + 2e^- \rightleftarrows H_2O$
Product H_2O + waste heat

Alkaline Fuel Cell (AFC)

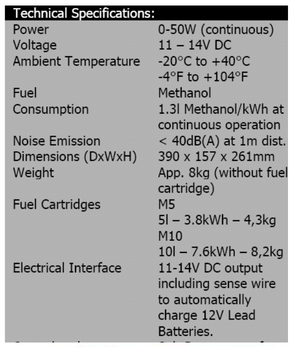


Molten Carbonate Fuel Cell (MCFC)


Temperature: 600-650°C
Efficiency
present: 45%
projected: 50-60%
Electrical Power
2 e⁻
2 e⁻
Porous Ni anode
Porous NiO cathode
$CO_3^=$
Coal gasification or natural gas
Air (O_2) + CO_2
Molten carbonate (Li_2CO_3/Na_2CO_3)
Anode reaction:
$H_2 + CO_3^{2-} \rightleftarrows H_2O + CO_2 + 2e^-$
Cathode reaction:
$½ O_2 + CO_2 + 2e^- \rightleftarrows CO_3^{2-}$
CO_2
Product H_2O + waste heat

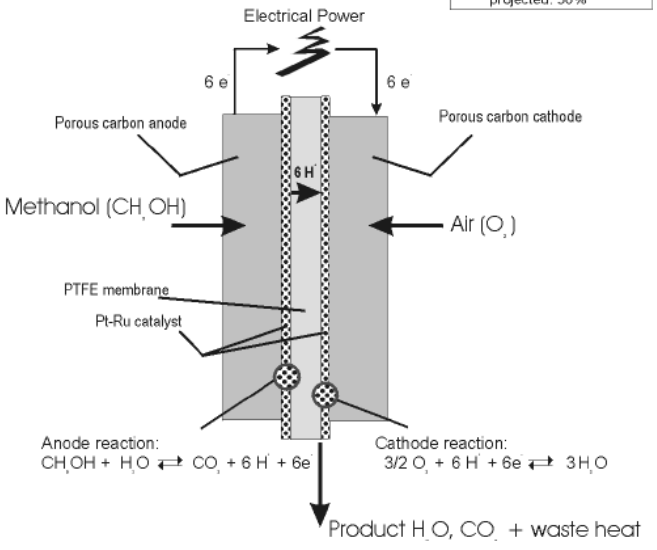
Direct Methanol Fuel Cell (DMFC)

Temperature: 50-85°C
Efficiency
present: 45%
projected: 50%



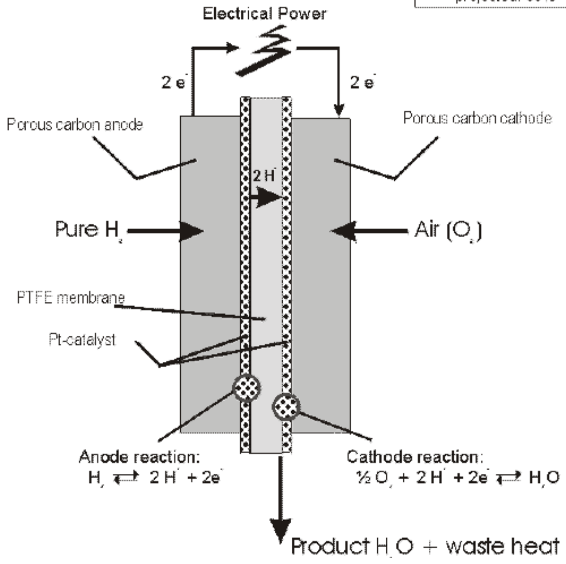
Technical Specifications:	
Power	0-50W (continuous)
Voltage	11 – 14V DC
Ambient Temperature	-20°C to +40°C -4°F to +104°F
Fuel	Methanol
Consumption	1.3l Methanol/kWh at continuous operation
Noise Emission	< 40dB(A) at 1m dist.
Dimensions (DxWxH)	390 x 157 x 261mm
Weight	App. 8kg (without fuel cartridge)
Fuel Cartridges	M5 5l – 3.8kWh – 4,3kg M10 10l – 7.6kWh – 8,2kg
Electrical Interface	11-14V DC output including sense wire to automatically charge 12V Lead Batteries.

http://www.udomi.de

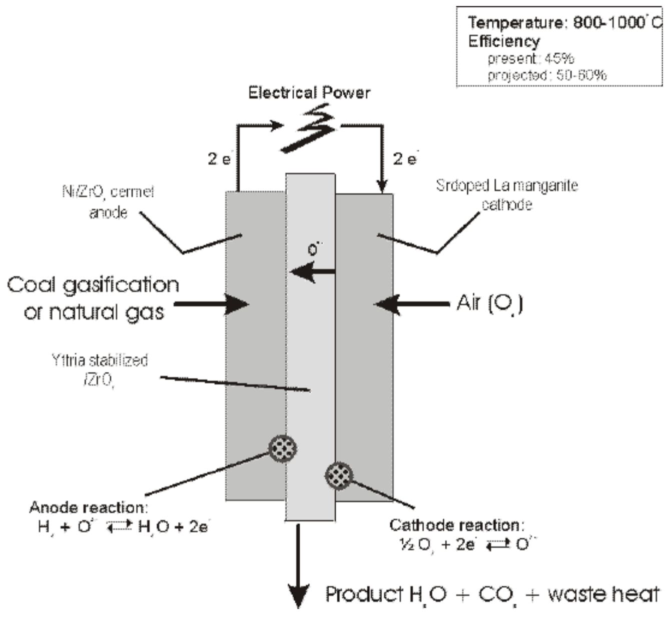


Polymer Electrolyte Fuel Cell

Temperature: 50-85°C
Efficiency
present: 45%
projected: 50%



Solid Oxide Fuel Cell


Temperature: 800-1000°C
Efficiency
present: 45%
projected: 50-60%
Electrical Power
2 e
2 e
Ni/ZrO2 cermet anode
Srdoped La manganite cathode
O2-
Coal gasification or natural gas
Air (O2)
Yttria stabilized /ZrO2
Anode reaction: H2 + O2- ⇄ H2O + 2e-
Cathode reaction: ½ O2 + 2e- ⇄ O2-
Product H2O + CO2 + waste heat

Solid Oxide Fuel Cell

SOFC's employ a ceramic oxide (ceria- or yttria-doped zirconia, Y_2O_3/ZrO_2), which becomes O^{2-} conducting at very high temperature (800-1000°C) and requires heating, but at this temperature reforming and H^+ production can take place internally without Pt.

A B C

$H_2(g)$//electrode// solid oxide electrolyte // electrode//$O_2(g)$

•Electrode reactions are:

Anode A: $H_2(g) + O^{2-} \rightarrow H_2O + 2e^-$

Cathode C: $1/2O_2(g) + 2e^- \rightarrow O^{2-}$

•The solid oxide electrolyte can withstand the extreme conditions of H_2 at the anode at 800°C. (many oxides would be reduced)

•Cathode materials must be able to conduct both oxide ions and electrons, and must similar thermal expansion coefficients as the electrolyte.

Proton Exchange Membrane Fuel Cells (PEM)

Proton Exchange membrane Fuel cells operate at ~80°C

•Electrolyte is a conducting polymer membrane (Nafion), which is a sulfonated fluoropolymer.

•The strongly acidic –SO_2OH group allows movement of H^+, but not e^-.

•Output voltage is ~1V at 80°C with a current flow of 0.5A/cm^2

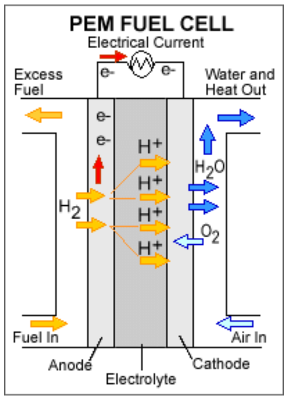
•Ohmic losses reduce this to 0.5V

•A membrane of 1 m^2 provides about 1 kW

•Cells are placed together to form a stack

•Large fuel cells are produced and can power banks, hospitals (250 kW)

•Medium cells (7 kW) can power a house and heat hot water. Fuel cells powered the Space Shuttle.

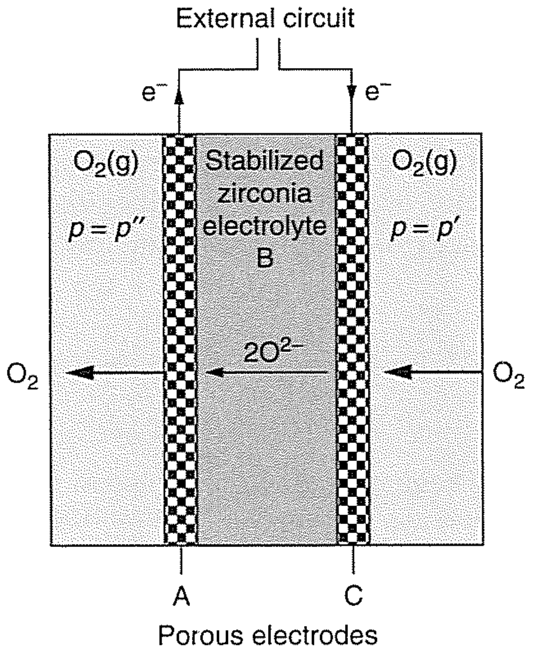


Sensors: Oxygen meters and oxygen sensors

CSZ is used in O_2 detection, in oxygen meters and oxygen sensors.

•Gas pressures tend to equalize, and if p' > p" oxygen ions pass through the stabilized zirconia.

•A potential difference, because the ions are charged, is formed, indicating the oxygen is present (in the sensor) and a measurement of the potential gives the oxygen pressure difference (in the oxygen meter).



Sensors: Oxygen meters and oxygen sensors

Oxygen gas is reduced to O^{2-} at the right-hand electrode (C). The oxide ions are able to pass through the doped zirconia and are oxidized to oxygen gas at the left-hand electrode (A).

•Electrode reactions are:

Anode A: $2O^{2-} \rightarrow O_2(p") + 4e^-$

Cathode C: $O_2(p') + 4e^- \rightarrow 2O^{2-}$

Overall: $O_2(p') \rightarrow O_2(p")$

Under standard conditions, the change in Gibb's Free energy is related to the standard emf of the cell:

$$\Delta G^o = -nE^oF$$

Nernst equation - allows calculation of the cell emf under nonstandard conditions, E. Assume the cell reaction is given by a general equation: $aA = bB + \ldots + ne \rightarrow xX\ yY + \ldots$

$$E = E^o - \frac{2.303RT}{nF}\log\left\{\frac{a_X^x a_Y^y}{a_A^a a_B^b}\right\}$$

where the quantities a_x are the activities of the reactants and products.

Applying the Nernst equation to the cell reaction in an oxygen meter:

$$E = E^o - \frac{2.303RT}{4F}\log\left\{\frac{p"}{p'}\right\}$$

E^o is zero, since under standard conditions the oxygen pressure is equal.

•Typically the pressure of the oxygen on one side of the cell (p") is set to be a known reference pressure, usually either pure oxygen at 1 atm or atmospheric oxygen pressure (~0.21 atm).

$$E = E^o - \frac{2.303RT}{4F}\log\left\{\frac{p'}{p_{ref}}\right\}$$

•All of the quantities in the equation are known or can be measured, enabling a direct measure of the unknown oxygen pressure p'.

•In order for a oxygen sensor of meter to operate, there must not be any electronic conduction through the electrolyte.

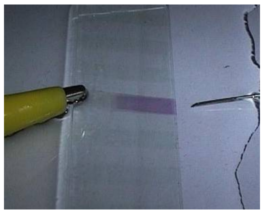
•Oxygen meters find use in detection of waste gases in chimneys, exhaust pipes, etc.

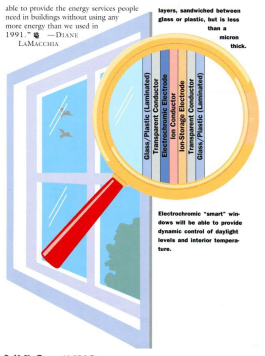
•Sensors for other gases operate using different electrolytes in the detection of H_2, F_2, Cl_2, CO_2, SO_x, NO_x.

Electrochromic Devices

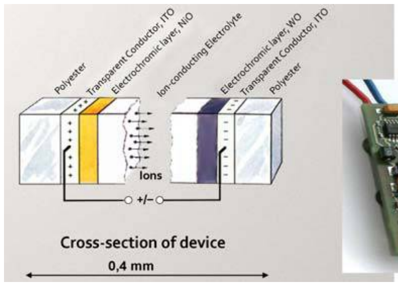
•Electric current is applied to the cell, causing a movement of ions through the electrolyte and creating a colored compound in one of the electrodes.

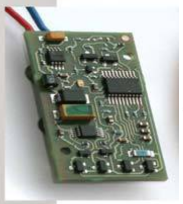
•Li⁺ ions flow from the anode, through the colorless electrolyte to form Li_xWO_3 at the cathode, changing it from colorless to deep blue.

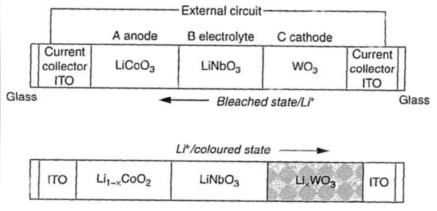



able to provide the energy services people need in buildings without using any more energy than we used in 1991." —DIANE LAMACCHIA
layers, sandwiched between glass or plastic, but is less than a micron thick.
Glass/Plastic (Laminated)
Transparent Conductor
Electrochromic Electrode
Ion Conductor
Ion-Storage Electrode
Transparent Conductor
Glass/Plastic (Laminated)
Electrochromic "smart" windows will be able to provide dynamic control of daylight levels and interior temperature.

A
$LiCoO_2$ // $LiNbO_3$ // WO_3

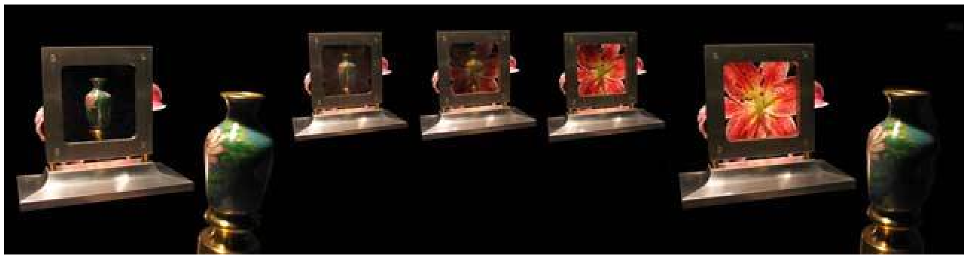

Polyester
Transparent Conductor, ITO
Electrochromic layer, NiO
Ion-conducting Electrolyte
Electrochromic layer, WO
Transparent Conductor, ITO
Polyester
Ions
+/-
Cross-section of device
0,4 mm





Granqvist et al Appl. Phys. A 89, 29–35 (2007)

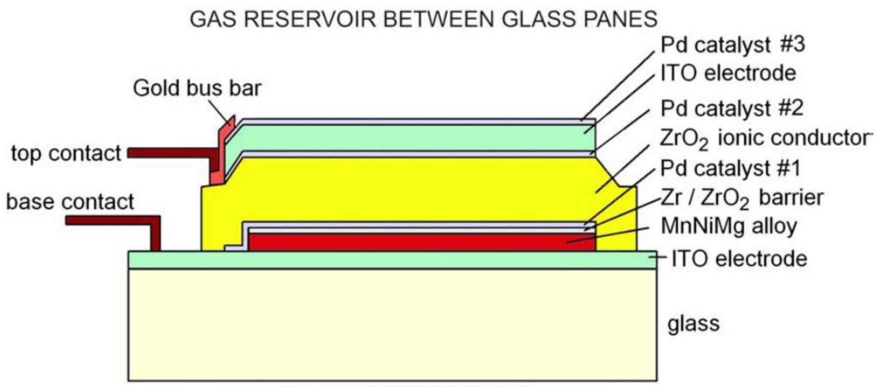
Transition-Metal Switchable Mirrors

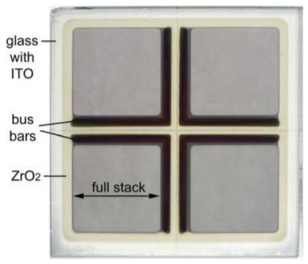


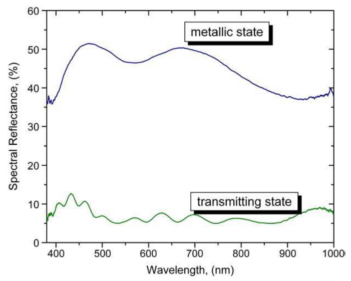
•The film used for the switchable mirror is made of an alloy of magnesium and one or more transition-metals.

Thin Ni-Mg films, on exposure to hydrogen gas or on reduction in alkaline electrolyte, the films become transparent. The transition is believed to result from formation of nickel magnesium hydride, Mg_2NiH_4.

http://eetd.lbl.gov/l2m2/tms-mirrors.html







Anders et al, Thin Solid Films 517 (2008) 1021–1026

Photography

A photographic emulsion contains small crystallites of AgBr (or AgBr-AgI) dispersed in gelatin that is supported on a paper or thin plastic to form a photographic film.

•Crystallites are small triangular or hexagonal platelets, known as grains.

•Grains are grown *in situ,* few defects and range in size 0.05 to 2x10^{-6} m.

•Light causes formation of Ag atoms from the salt, forming the dark part of the image. The grains that are affected by the light contain the latent image. Formation of the latent image depends on the presence of point defects.

•AgBr and AgI have the NaCl-structure type and AgBr has **Frenkel** defects in the form of interstitial Ag$^+$ ions.

•For a grain to possess a latent image, a cluster forms as small as **4** Ag atoms on the surface of the grain.

•When light strikes the AgBr crystals, an electron is promoted from the valence band to the conduction band. (band gap AgBr = 2.7 eV)

$$Ag_i^+ + e^- \rightarrow Ag$$

$Ag + e^- \rightarrow Ag^-$

$Ag^- + Ag_i^+ \rightarrow Ag_2$

$Ag_2 + Ag_i^+ \rightarrow Ag_3^+$

$Ag_3^+ + e^- \rightarrow Ag_3$

$Ag_3 + e- \rightarrow Ag_3^-$

$Ag_3^- + Ag_i^+ \rightarrow Ag_4 + \ldots$

Only the odd numbered clusters appear to interact with the electrons.

•Sensitizers, sulfur or organic dye, are added and absorb light of longer wavelength and extend the spectra range.

•Sensitizers form traps for the photoelectrons on the surfaces of the grains and transfer from an excited energy level of the sensitizer to the conduction band of AgBr.

The film containing the latent image is treated to produce a negative.

•Developed by using a reducing agent – such as an alkaline solution of hydroquinone, to reduce the AgBr crystals to Ag.

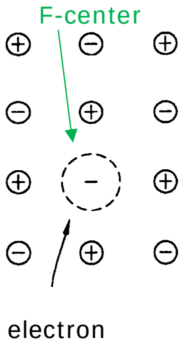
•The clusters of Ag atoms act as a catalyst to the reduction process, all of the grains with a latent image are reduced to Ag.

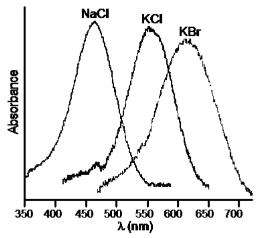
•Process is rate controlled, so all grains that have not reacted to light are unaffected by the developer (except for long developing times).

•Final stage is to dissolve out the remaining light sensitive AgBr using hypo-sodium thiosulfate ($Na_2S_2O_3$) which forms a water soluble complex with Ag^+ ions.

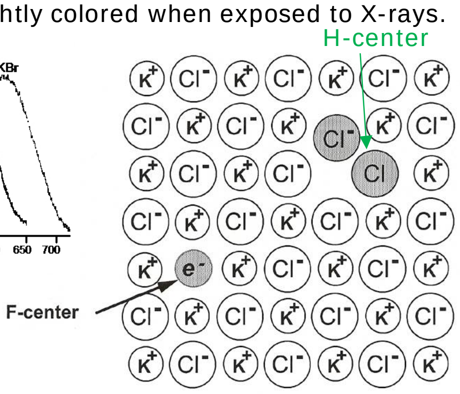
Color Centers

Crystals of alkali halides become brightly colored when exposed to X-rays.





A color center is known as a Farbenzentre (F-center).



Electron Spin Resonance (ESR) spectroscopy confirms the presence of unpaired electrons trapped at vacant lattice (anion site).

A **H-center** may also be formed by heating NaCl in Cl_2 gas, with the Cl_2^- ion occupying an anion site.

Non-stoichiometric Compounds

Impurity induced defects are extrinsic and maintain charge neutrality.

Color centers in NaCl may be formed heat heating in the presence of Na vapor, becoming $Na_{1+x}Cl$, where the sodium atom occupies cation sites, creating anion vacancies. The sodium atoms oxidize to form a sodium cation with an electron at the anion vacancy.

•The resulting compound is known as a non-stoichiometric compound because the number of atomic components is no longer 1:1.

•Ionic compounds may also be non-stoichiometric when it contains an element with a variable valency, then a change in the number of ions of that element can be compensated with changes in ion charge.

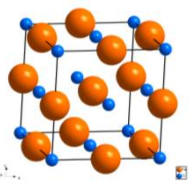
Compound		Composition range
TiO_x	[≈TiO]	$0.65 < x < 1.25$
	[≈TiO_2]	$1.998 < x < 2.000$
VO_x	[≈VO]	$0.79 < x < 1.29$
Mn_xO	[≈MnO]	$0.848 < x < 1.000$
Fe_xO	[≈FeO]	$0.833 < x < 0.957$
Co_xO	[≈CoO]	$0.998 < x < 1.000$
Ni_xO	[≈NiO]	$0.999 < x < 1.000$
CeO_x	[≈Ce_2O_3]	$1.50 < x < 1.52$
ZrO_x	[≈ZrO_2]	$1.700 < x < 2.004$
UO_x	[≈UO_2]	$1.65 < x < 2.25$
Li_xWO_3		$0 < x < 0.50$

•Isolated point defects are not randomly located in non-stoichiometric compounds, but are dispersed in a regular pattern.

•Conventional XRD gives average structure, local structure may be investigated using high resolution electron microscopy (HREM) and direct lattice imaging.

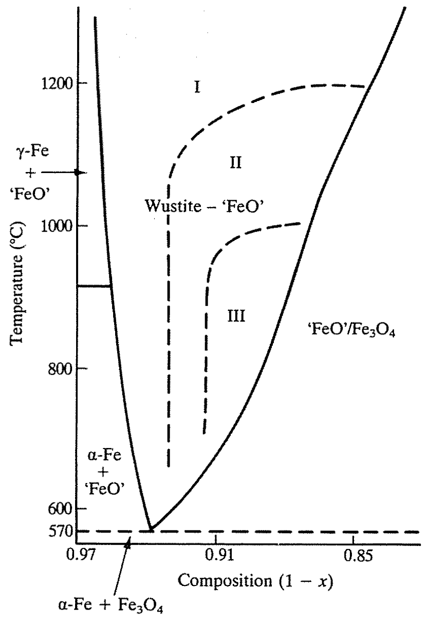
Non-stoichiometry in Wustite (FeO)

Ferrous oxide, or wustite (FeO) has the NaCl structure type.



- Chemical analysis indicates it is non-stoichiometric and always deficient in iron. Stoichiometric FeO isn't stable, and below 570°C disproportionates into α-Fe and Fe_3O_4.
- Iron deficiency may be accommodated in the structure on one of two ways:

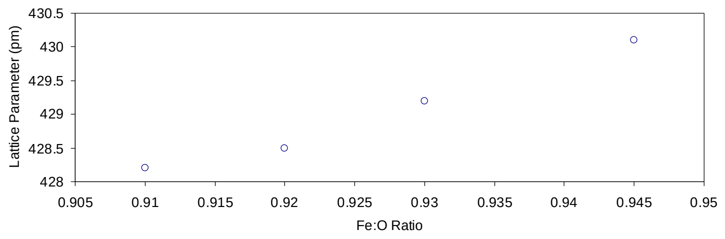
1. Iron vacancies, giving $Fe_{1-x}O$
2. Excess of oxygen in interstitial positions, giving FeO_{1+x}



Vegard's Law

It is often found that non-stoichiometric compounds have a unit cell size that varies smoothly with composition but has symmetry that is unchanged, which is known as Vegard's Law.

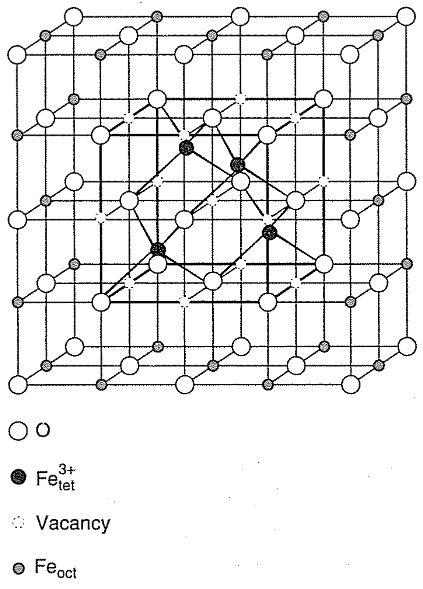
O:Fe ratio	Fe:O ratio	Lattice parameter /pm	Observed density (g/cm^3)	Interstitial O (g/cm^3)	Fe Vacancies (g/cm^3)
1.058	0.945	430.1	5.728	6.075	5.742
1.075	0.930	429.2	5.658	6.136	5.706
1.087	0.920	428.5	5.624	6.181	5.687
1.099	0.910	428.2	5.613	6.210	5.652



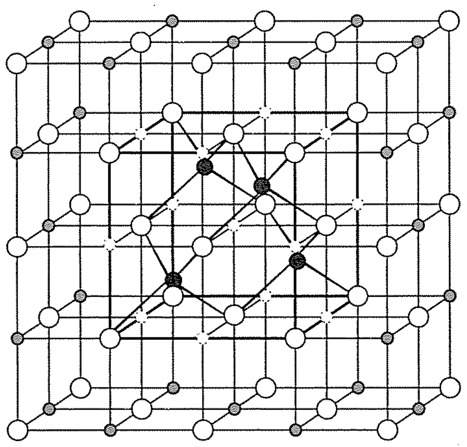
Electronic Defects and Structure in FeO

Compensation for iron deficiency is energetically more favorable to oxidize Fe(II), requiring the oxidation of two Fe^{2+} to Fe^{3+} for every Fe^{2+} cation vacancy.

- In the case of excess metal, neighboring cations would be reduced.
- In the case of an Fe^{2+} vacancy, the two Fe^{3+} vacancies are determined to be neighboring as confirmed by Mossbauer spectroscopy.
- Some Fe^{3+} ions are found to be in a tetrahedral site.
- If the tetrahedral site is occupied, then the thirteen neighboring octahedral Fe^{2+} sites are empty.
- This type of defect is found for low values of *x*.
- At high values of *x*, the structure contains various types of defect clusters, one possibility is a Koch-Cohen cluster.



Koch-Cohen Cluster



○ O

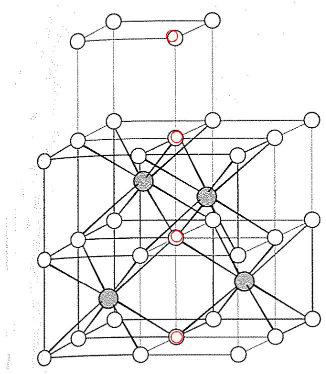
● Fe^{3+}_{tet}

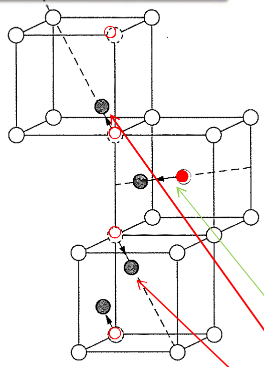
◌ Vacancy

● Fe_{oct}

- A *defect cluster* is a region of the crystal where defects form an *ordered structure*.
- Surrounding the central defect unit cell, the other octahedral iron sites (Fe_{oct}) are occupied, but may contain either Fe^{2+} or Fe^{3+}.
- Clusters sometimes referred to the ratio of cation vacancies to interstitial Fe^{3+} in tetrahedral holes (13:4).

Uranium Dioxide


(a)


(b)

● Uranium

○ Oxygen

● Ideal interstitial site for oxygen

● Interstitial oxygen

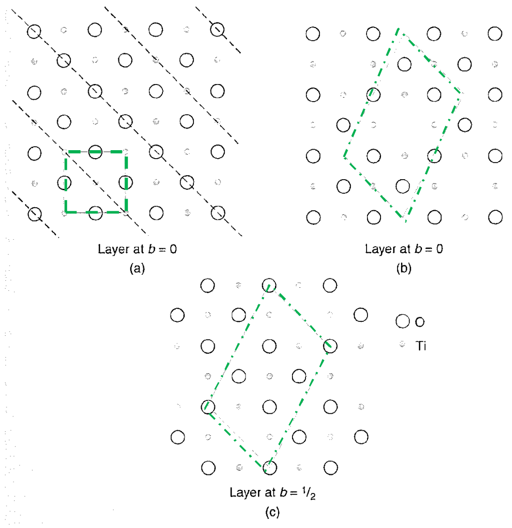
◌ Vacancy

- Above 1127°C, a single oxygen-rich non-stoichiometric phase of UO_2 is found with formula UO_2 to $UO_{2.25}$ (U_4O_9)
- Interstitial anions are present in the fluorite structure.
- Interstitial O' causes O'' displacement.
- A defect cluster, considered as two vacancies, one interstitial of one kind O', and two of another O'', is called a 2:1:2 Willis cluster.
- The movement of the interstitial oxide O' is along the direction towards the diagonal of the cube face (110) direction, whereas the O'' is along cube diagonal (111)
- Can consider UO_2 as containing microdomains of U_4O_9 structure within UO_2.

Titanium Monoxide (TiO)

•Composition ranges from $TiO_{0.65}$ to $TiO_{1.25}$, with a stoichiometric 1:1 composition resembling the NaCl-type structure with vacancies in both the metal and oxygen sublattices: 1/6 of Ti and 1/6 of O are missing.

•Vacancies are randomly distributed above 900°C, but below are ordered.



•The structure appears stoichiometric, but contains defects on both the cation and anion sublattices.

•Note that every other atom along every third diagonal plane is missing.

•The new unit cell is monoclinic (β ≠ 90°)

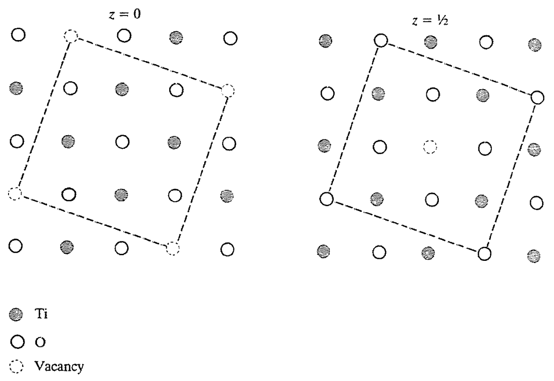
•Vacancies permit sufficient contraction of the lattice to enable 3d orbitals on Ti to overlap, broadening the conduction band and allowing electronic conduction.

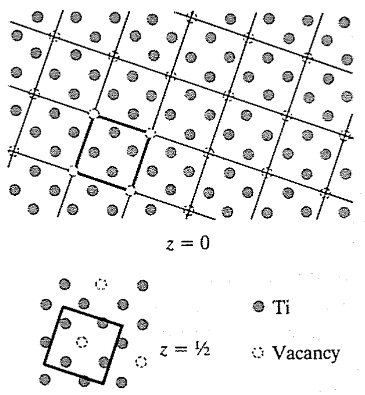
Titanium Monoxide ($TiO_{1.25}$)

•$TiO_{1.25}$ has all of the oxygens present and one in every five Ti missing.

•Ordering produces a superlattice with a different unit cell.

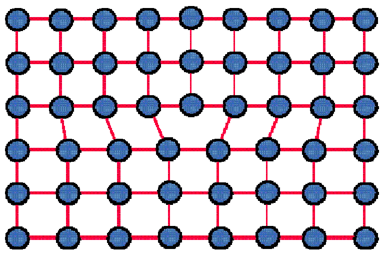
•Formula would be more correctly written as $Ti_{0.8}O$ ($Ti_{1-x}O$) because this indicates the structure contains interstitial vacancies.

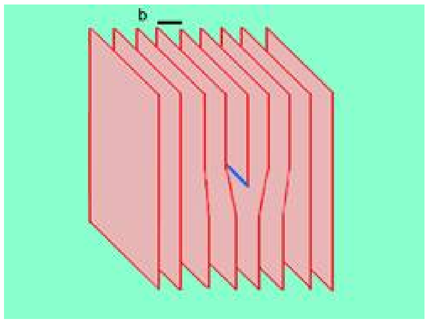




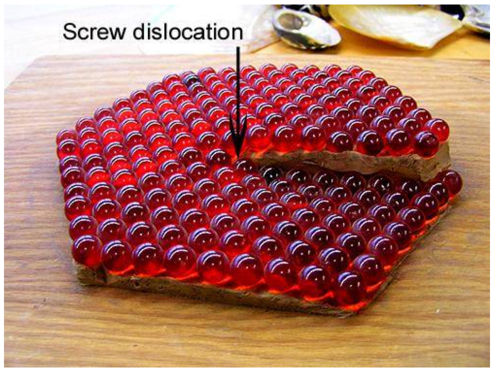
Extended Defects

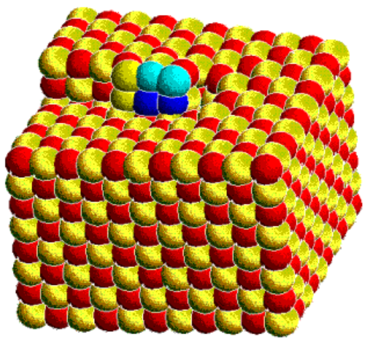
The simplest *linear* defect is a **dislocation** where there is a fault in the arrangement of atoms in a line through the crystal lattice.



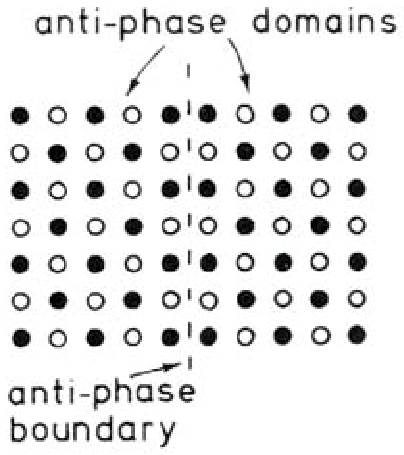


Another *linear* defect is a **screw dislocation.** This occurs when a stress is applied to the crystal and the dislocation of the line of atoms is perpendicular to the stress.

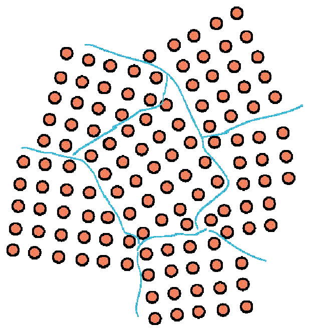




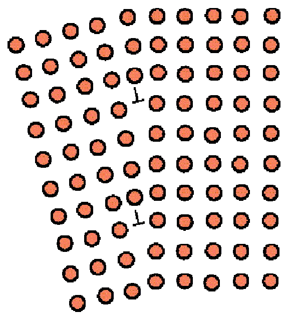
Antiphase domain: the grain has the reverse structure from the surrounding structure.



There are also *planar defects* such as **grain boundaries**.



Chemical twinning (*planar defects*) contains unit cells mirrored about the twin plane through the crystal.



Crystallographic Shear Planes

Non-stoichiometric compounds are found for the higher oxides of tungsten (WO_{3-x}), molybdenum (MoO_{3-x}), and titanium (TiO_{2-x}).

•In these systems a series of closely related compounds with similar formula exist (Mo_nO_{3n-1}, W_nO_{3n-1} and W_nO_{3n-2}, and Ti_nO_{2n-1}, where *n* can take values of 4 and above). The resulting series of oxides is known as a homologous series.

•These compounds have regions of corner-sharing octahedra separated from each other by regions of a different structure known as a crystallographic shear plane.

•The different members of a homologous series are determined by the fixed spacing between the crystallographic shear planes.

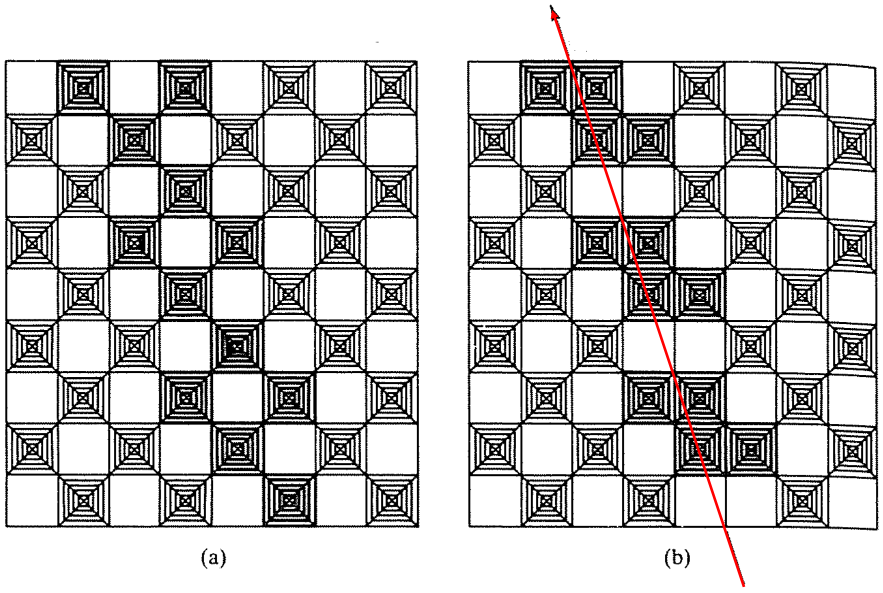
•Above 900°C, the WO_3 structure is that of ReO_3, which has $[WO_6]$ octahedra sharing corners with any octahedron lined to four others in the same layer.

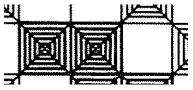
•Non-stoichiometry is WO_{3-x} is achieved by some of the octahedra in this structure changing from corner-sharing to edge-sharing.

•Shearing occurs at regular intervals and creates groups of four octahedra which share edges.

•Direction of maximum density of edge sharing groups in the CS plane.

Formation of Shear Structure





The four octahedra consist of four W atoms and 18 O atoms.

•14 of the O are linked to other octahedra and 4 O are involved in edge sharing within the group.

•The overall stoichiometry is given by 4W + (14*1/2)O+4O = W_4O_{11}

•If groups of W_4O_{11} are interspersed throughout the WO_3 structure, the groups can be written as WO_{3-x}.

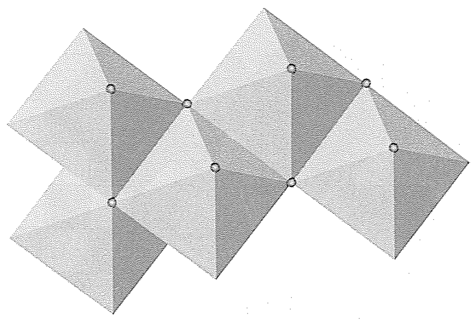
•$W_4O_{11} + WO_3 = W_5O_{14}$

•$W_4O_{11} + 2WO_3 = W_6O_{17}$

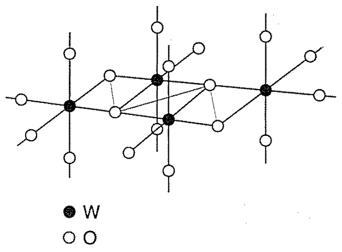
•$W_4O_{11} + 3WO_3 = W_7O_{20}$

•$W_4O_{11} + 4WO_3 = W_8O_{23}$

•Simplifies to W_nO_{3n-1}

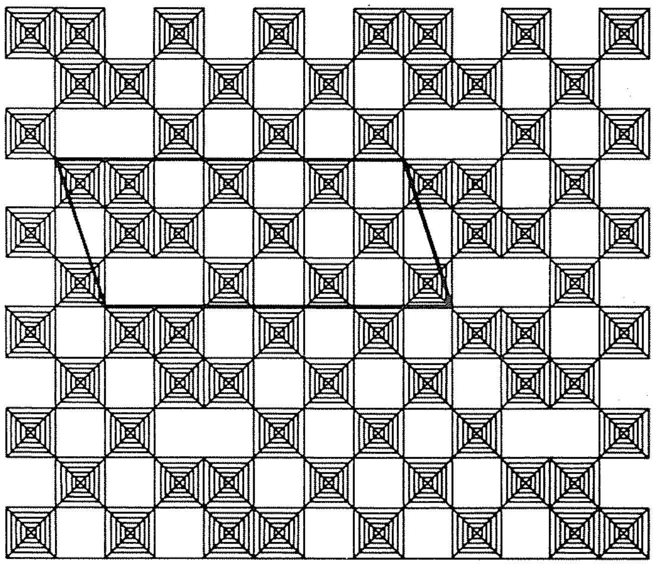


(a)



(b)

Shear Structure: $W_4O_{11} + 7WO_3 \rightarrow W_{11}O_{32}$



(2.) Planar Intergrowths: Tungsten Bronze

The term bronze is applied to metallic oxides that have a deep color (yellow to red or deep purple), metallic luster, and are metallic or semiconducting.

•Color depends on x in Na_xWO_3.

•Structure is a 3d network of channels throughout the structure, with alkali metals in the channels.

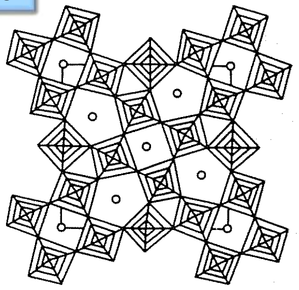
•Three main types of structures:

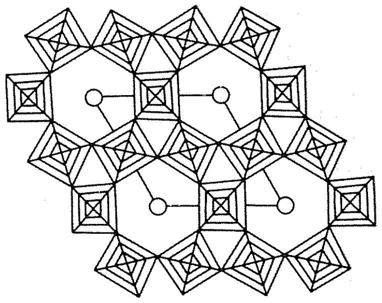
1. Cubic, 2. Tetragonal, 3. Hexagonal

•Charge compensation occurs with the M^+ presence, reducing the metal M^{5+}.

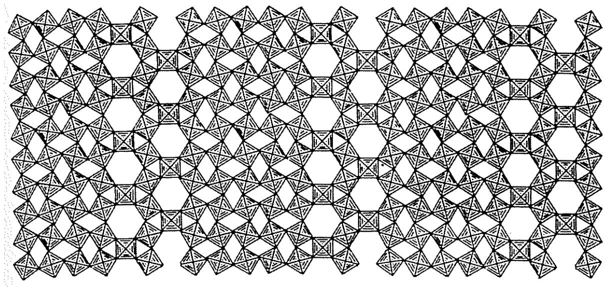
•In the case of K, stability lies in the range of $K_{0.19}WO_3$ to $K_{0.33}WO_3$, below 0.19 the structure has WO_3 intergrown with the hexagonal structure in a regular fashion.

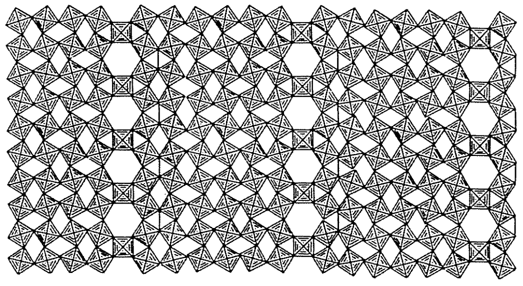
•Layers of hexagonal structure 1 or 2 tunnels wide.

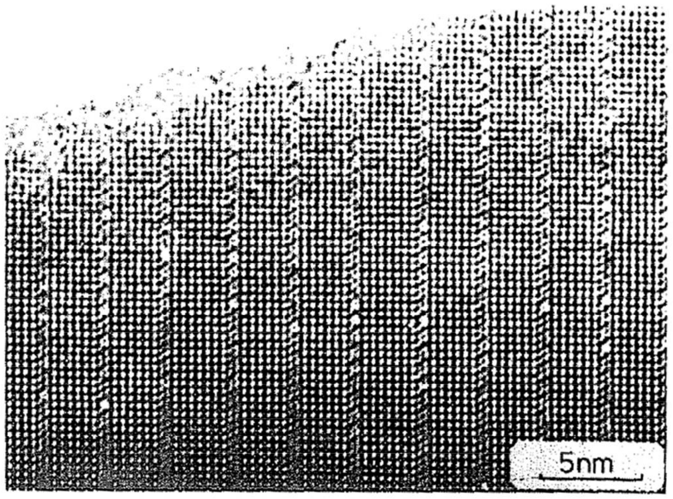

(a) Tetragonal


(b) Hexagonal

(2.) Planar Intergrowths: Tungsten Bronze


(a)


(b)

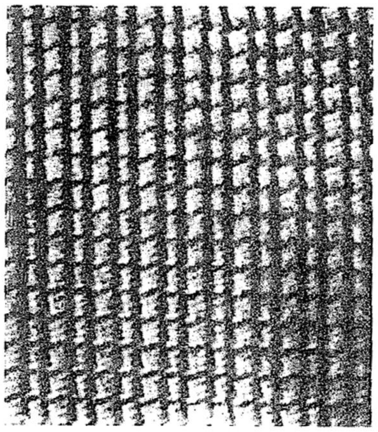


Electron micrograph of intergrowth Ba_xWO_3

Three-Dimensional Defects: Block Structures

In O-deficient Nb_2O_5, and mixed oxides of Nb and Ti, and Nb and W, the crystallographic shear planes occur in two sets at right angles to each other.

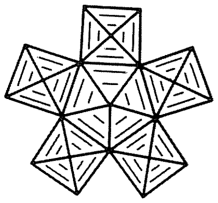
- •Intervening regions of perfect structure change from infinite sheets to infinite columns or blocks, which are known as double shear or block structures.
- •Characterized by the cross sectional size of the blocks.
- •May also have blocks of two or three different sizes arranged in an ordered fashion, such as the 4x4 and 3x4 blocks in $W_4Nb_{26}O_{77}$.

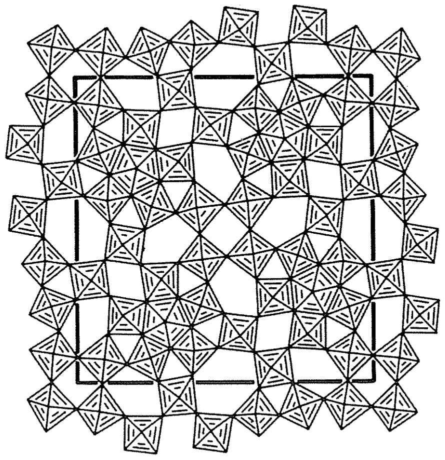


High-resolution electron micrograph of $W_4Nb_{26}O_{77}$

Three-Dimensional Defects: Pentagonal Columns

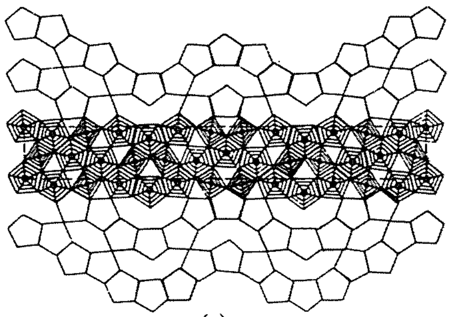
- •Structure consists of a pentagonal ring of five $[MO_6]$ octahedra, which when stacked form a pentagonal column with alternating M and O atoms.
- •The pentagonal columns fit inside an ReO_3 type structure in an ordered way and, depending on the spacing, form a homologous series.
- •One example is the Mo_5O_{14} structure.

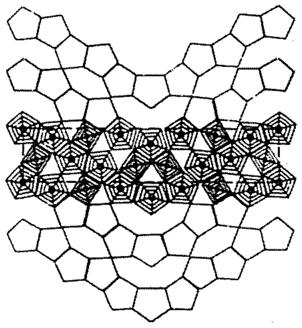

(a)


(b) Mo_5O_{14}

Three-Dimensional Defects: Infinitely Adaptive Structures

- •A large number of compounds form in the Ta_2O_5-WO_3 system, built from fitting together pentagonal columns.
- •Structure have a *wavelike* skeleton of pentagonal columns.
- •As the composition varies, the 'wavelength' of the backbone changes, giving rise to a huge number of possible ordered structures, know as infinitely adaptive structures.


(a) $Ta_{22}W_4O_{67}$


(b) $Ta_{30}W_2O_{81}$

Electronic Properties of Non-stoichiometric Oxides

Four basic types of compounds are non-stoichiometric:

Metal excess (reduced metal)

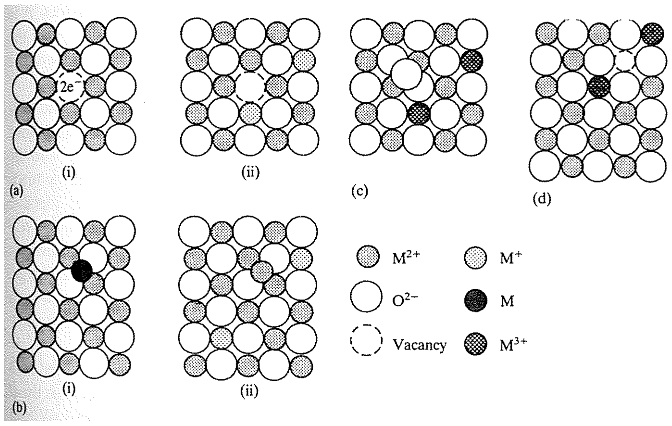
Type A: anion vacancies present → formula MO_{1-x} (e.g. TiO, VO, ZrS)

Type B: interstitial cations → formula $M_{1+x}O$ (e.g. CdO, ZnO)

Metal deficiency (oxidized metal)

Type C: interstitial anions → formula MO_{1+x}

Type D: cation vacancies → $M_{1-x}O$ (e.g. TiO, VO, MnO, FeO, CoO)



Type A oxides: Compensate for metal excess with *anion vacancies*. Two electrons have to be introduced for each anion vacancy, which may be trapped at the vacant anion site. More likely to find electrons associated with the reduction of nearby metal cations from M^{2+} to M^{+}.

Type B oxides: Have a *metal excess* incorporated into the lattice in interstitial positions. Most likely that an interstitial atom is ionized, reducing nearby metal cations from M^{2+} to M^{+}.

Type C oxides: Compensate for the lack of metal with *interstitial anions*. Charge balance is maintained by creation of two M^{3+} cations for each interstitial anion (O^{2-}).

Conductivity:

(i) depends on d-orbital overlap – the bigger the overlap the greater the band *width* and electrons in the band are *delocalized* over the whole structure.

(ii) interelectronic repulsion tends to keep electrons localized on individual atoms.

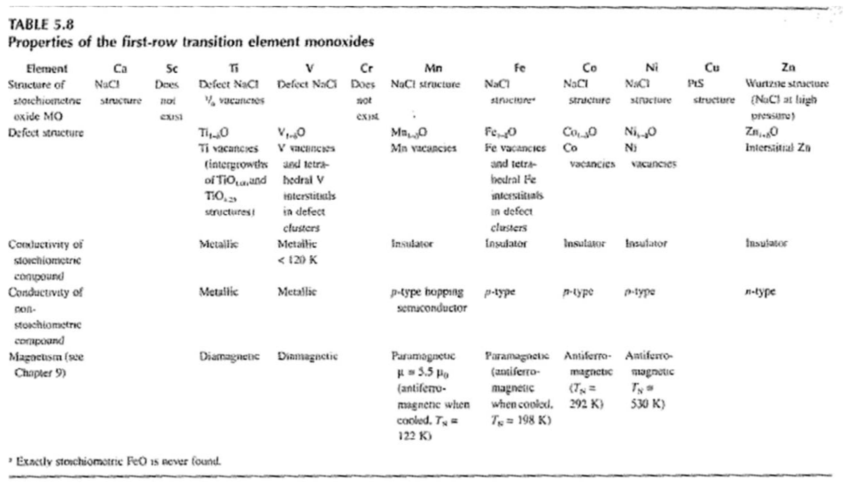


TABLE 5.8
Properties of the first-row transition element monoxides

Element	Ca	Sc	Ti	V	Cr	Mn	Fe	Co	Ni	Cu	Zn
Structure of stoichiometric oxide MO	NaCl structure	Does not exist	Defect NaCl $^1/_6$ vacancies	Defect NaCl	Does not exist	NaCl structure	NaCl structure[a]	NaCl structure	NaCl structure	PtS structure	Wurtzite structure (NaCl at high pressure)
Defect structure			$Ti_{1-\delta}O$ Ti vacancies (intergrowths of $TiO_{1.00}$ and $TiO_{1.25}$ structures)	$V_{1-\delta}O$ V vacancies and tetrahedral V interstitials in defect clusters		$Mn_{1-\delta}O$ Mn vacancies	$Fe_{1-\delta}O$ Fe vacancies and tetrahedral Fe interstitials in defect clusters	$Co_{1-\delta}O$ Co vacancies	$Ni_{1-\delta}O$ Ni vacancies		$Zn_{1+\delta}O$ Interstitial Zn
Conductivity of stoichiometric compound			Metallic	Metallic < 120 K		Insulator	Insulator	Insulator	Insulator		Insulator
Conductivity of non-stoichiometric compound			Metallic	Metallic		*p*-type hopping semiconductor	*p*-type	*p*-type	*p*-type		*n*-type
Magnetism (see Chapter 9)			Diamagnetic	Diamagnetic		Paramagnetic $\mu = 5.5\ \mu_0$ (antiferromagnetic when cooled, T_N = 122 K)	Paramagnetic (antiferromagnetic when cooled, T_N = 198 K)	Antiferromagnetic (T_N = 292 K)	Antiferromagnetic T_N = 530 K)		

[a] Exactly stoichiometric FeO is never found.

TiO and VO are *metallic conductors* – have good overlap between d-orbitals (forming a d-electron band), partly because Ti and V are early in the transition series and the *d* orbitals have not suffered contraction due to increase nuclear charge as is later in the series.

-TiO also has 1/6 of the Ti and O missing from NaCl structure type, leading to a contraction of the structure and better *d* overlap.

MnO, FeO, CoO, and NiO are insulators - the d-orbitals are too contracted to overlap much, with typical band width 1 eV, and the overlap is not sufficient to overcome the *localizing* influence of interelectronic repulsions. Gives rise to magnetic properties.

Non-stoichiometric oxides, types A and B metal excess monoxides, have extra electrons to compensate for excess metal in the structure. These electrons can be free to move through the lattice and are not necessarily bound to a particular atom.

-thermal energy is enough to make the electrons move and conductivity increases with temperature (like a semiconductor).

Compounds of type A and B would produce *n-type* semiconductors because the conduction is produced by electrons.

•Consider the conduction electrons (or holes) as localized, or trapped, at atoms or defects instead of delocalized in bands.

•Conduction occurs by jumping or hopping from one site to another under the influence of an electric field.

- •Energetically, electron ‘jumps’ between two valence states (e.g. Zn^+ and Zn^{2+}), it doesn’t take much energy.
- •These are called hopping semiconductors and can be described in the same way as ionic conduction.

•The mobility (μ) for a charge carrier (electron or positive hole), is an activated process.

$$\mu \text{ is proportional to } e^{(-E_a/kT)}$$

where E_a is the activation energy of the hop (0.1 to 0.5 eV).

The hopping conductivity is $\sigma = ne\mu$

where n is the number of mobile charge carriers per unit volume and e is the electronic charge. n doesn’t depend on temperature, only on composition.

Compounds of type C and D monoxides have M^{3+} ions, which can be regarded as a positive hole compared to the M^{2+} cation.

•if sufficient energy is available, conduction can be thought to occur via the *positive hole* hopping to another M^{2+} cation, giving rise to *p-type* electronic conductivity.

•This type of conductivity is found for MnO, CoO, NiO, and FeO

Non-stoichiometric oxides cover the entire range from metal to insulator.

•Others, such as TiO_2 and WO_3 require a different description.

•Each structure needs to be examined *individually* in terms of conductivity.

Doping compounds with an impurity *extends the range of properties*:

$$0.5x\ Li_2O + (1\text{-}x)NiO + 1/4x\ O_2 \rightarrow Li_xNi_{1\text{-}x}O$$

where Ni^{2+} is oxidized to Ni^{3+}, creating a high concentration of positive holes located at Ni cations. This process is known as **valence induction**.

In greatly increase the conductivity range of NiO – at high Li concentration the conductivity approaches that of a metal.

Ch 3.1: Second Order Linear Homogeneous Equations with Constant Coefficients

- A **second order ordinary differential equation** has the general form

$$y'' = f(t, y, y')$$

  where f is some given function.

- This equation is said to be **linear** if f is linear in y and y':

$$y'' = g(t) - p(t)y' - q(t)y$$

  Otherwise the equation is said to be **nonlinear**.

- A second order linear equation often appears as

$$P(t)y'' + Q(t)y' + R(t)y = G(t)$$

- If $G(t) = 0$ for all t, then the equation is called **homogeneous**. Otherwise the equation is **nonhomogeneous**.

Homogeneous Equations, Initial Values

- In Sections 3.6 and 3.7, we will see that once a solution to a homogeneous equation is found, then it is possible to solve the corresponding nonhomogeneous equation, or at least express the solution in terms of an integral.
- The focus of this chapter is thus on homogeneous equations; and in particular, those with constant coefficients:

$$ay'' + by' + cy = 0$$

  We will examine the variable coefficient case in Chapter 5.
- Initial conditions typically take the form

$$y(t_0) = y_0,\ \ y'(t_0) = y_0'$$

- Thus solution passes through (t_0, y_0), and slope of solution at (t_0, y_0) is equal to y_0'.

Example 1: Infinitely Many Solutions (1 of 3)

- Consider the second order linear differential equation

$$y'' - y = 0$$

- Two solutions of this equation are

$$y_1(t) = e^t, \quad y_2(t) = e^{-t}$$

- Other solutions include

$$y_3(t) = 3e^t, \quad y_4(t) = 5e^{-t}, \quad y_5(t) = 3e^t + 5e^{-t}$$

- Based on these observations, we see that there are infinitely many solutions of the form

$$y(t) = c_1 e^t + c_2 e^{-t}$$

- It will be shown in Section 3.2 that all solutions of the differential equation above can be expressed in this form.

Example 1: Initial Conditions (2 of 3)

- Now consider the following initial value problem for our equation:

$$y'' - y = 0, \quad y(0) = 3,\ y'(0) = 1$$

- We have found a general solution of the form

$$y(t) = c_1 e^t + c_2 e^{-t}$$

- Using the initial equations,

$$\left.\begin{aligned} y(0) &= c_1 + c_2 = 3 \\ y'(0) &= c_1 - c_2 = 1 \end{aligned}\right\} \Rightarrow c_1 = 2,\ c_2 = 1$$

- Thus

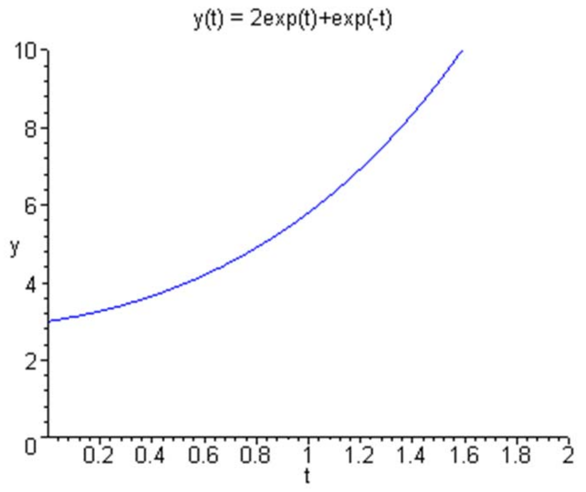
$$y(t) = 2e^t + e^{-t}$$

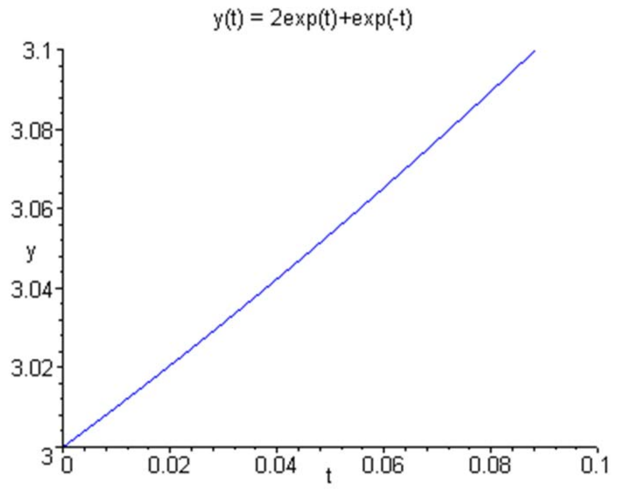
Example 1: Solution Graphs (3 of 3)

- Our initial value problem and solution are

$$y'' - y = 0, \quad y(0) = 3,\ y'(0) = 1 \;\Rightarrow\; y(t) = 2e^{t} + e^{-t}$$

- Graphs of this solution are given below. The graph on the right suggests that both initial conditions are satisfied.





Characteristic Equation

- To solve the 2nd order equation with constant coefficients,

$$ay'' + by' + cy = 0,$$

  we begin by assuming a solution of the form $y = e^{rt}$.
- Substituting this into the differential equation, we obtain

$$ar^2e^{rt} + bre^{rt} + ce^{rt} = 0$$

- Simplifying,

$$e^{rt}(ar^2 + br + c) = 0$$

  and hence

$$ar^2 + br + c = 0$$

- This last equation is called the **characteristic equation** of the differential equation.
- We then solve for *r* by factoring or using quadratic formula.

General Solution

- Using the quadratic formula on the characteristic equation

$$ar^2 + br + c = 0,$$

  we obtain two solutions, r_1 and r_2.
- There are three possible results:
  - The roots r_1, r_2 are real and $r_1 \neq r_2$.
  - The roots r_1, r_2 are real and $r_1 = r_2$.
  - The roots r_1, r_2 are complex.

$$r = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- In this section, we will assume r_1, r_2 are real and $r_1 \neq r_2$.
- In this case, the general solution has the form

$$y(t) = c_1 e^{r_1 t} + c_2 e^{r_2 t}$$

Initial Conditions

- For the initial value problem

$$ay'' + by' + cy = 0, \quad y(t_0) = y_0, \quad y'(t_0) = y_0',$$

  we use the general solution

$$y(t) = c_1 e^{r_1 t} + c_2 e^{r_2 t}$$

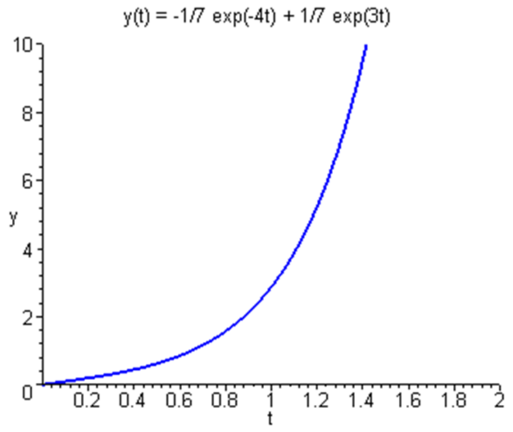
  together with the initial conditions to find c_1 and c_2. That is,

$$\left.\begin{aligned} c_1 e^{r_1 t_0} + c_2 e^{r_2 t_0} &= y_0 \\ c_1 r_1 e^{r_1 t_0} + c_2 r_2 e^{r_2 t_0} &= y_0' \end{aligned}\right\} \Rightarrow c_1 = \frac{y_0' - y_0 r_2}{r_1 - r_2} e^{-r_1 t_0}, \; c_2 = \frac{y_0 r_1 - y_0'}{r_1 - r_2} e^{-r_2 t_0}$$

- Since we are assuming $r_1 \neq r_2$, it follows that a solution of the form $y = e^{rt}$ to the above initial value problem will always exist, for any set of initial conditions.

Example 2

- Consider the initial value problem
$$y'' + y' - 12y = 0, \ \ y(0) = 0, \ \ y'(0) = 1$$
- Assuming exponential soln leads to characteristic equation:
$$y(t) = e^{rt} \quad \Rightarrow \quad r^2 + r - 12 = 0 \ \Leftrightarrow \ (r+4)(r-3) = 0$$
- Factoring yields two solutions, r_1 = -4 and r_2 = 3
- The general solution has the form
$$y(t) = c_1 e^{-4t} + c_2 e^{3t}$$
- Using the initial conditions:
$$\left.\begin{aligned} c_1 + c_2 &= 0 \\ -4c_1 + 3c_2 &= 1 \end{aligned}\right\} \Rightarrow c_1 = \frac{-1}{7}, \ c_2 = \frac{1}{7}$$
- Thus $$y(t) = \frac{-1}{7} e^{-4t} + \frac{1}{7} e^{3t}$$



Example 3

- Consider the initial value problem

$$2y'' + 3y' = 0, \quad y(0) = 1, \quad y'(0) = 3$$

- Then

$$y(t) = e^{rt} \quad \Rightarrow \quad 2r^2 + 3r = 0 \iff r(2r+3) = 0$$

- Factoring yields two solutions, $r_1 = 0$ and $r_2 = -3/2$
- The general solution has the form

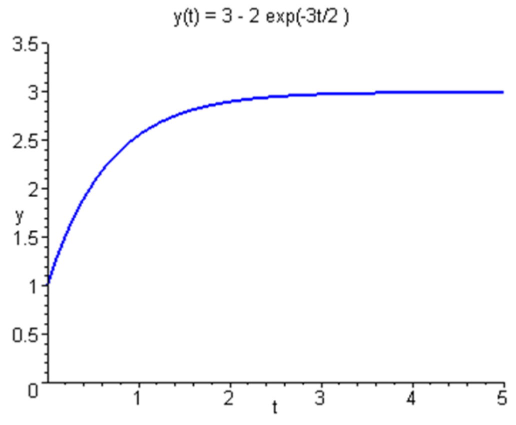
$$y(t) = c_1 e^{0t} + c_2 e^{-3t/2} = c_1 + c_2 e^{-3t/2}$$

- Using the initial conditions:

$$\left.\begin{aligned} c_1 + c_2 &= 1 \\ -\frac{3c_2}{2} &= 3 \end{aligned}\right\} \Rightarrow c_1 = 3,\ c_2 = -2$$

- Thus

$$y(t) = 3 - 2e^{-3t/2}$$



Example 4: Initial Value Problem (1 of 2)

- Consider the initial value problem

$$y'' + 5y' + 6y = 0, \quad y(0) = 2, \; y'(0) = 3$$

- Then

$$y(t) = e^{rt} \quad \Rightarrow \quad r^2 + 5r + 6 = 0 \;\Leftrightarrow\; (r+2)(r+3) = 0$$

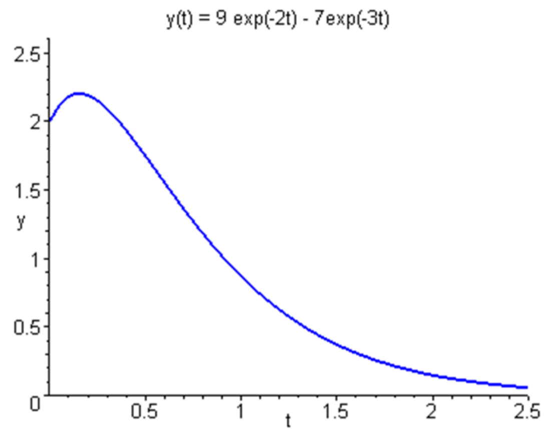
- Factoring yields two solutions, r_1 = -2 and r_2 = -3
- The general solution has the form

$$y(t) = c_1 e^{-2t} + c_2 e^{-3t}$$

- Using initial conditions:

$$\left.\begin{aligned} c_1 + c_2 &= 2 \\ -2c_1 - 3c_2 &= 3 \end{aligned}\right\} \Rightarrow c_1 = 9, \; c_2 = -7$$

- Thus $y(t) = 9e^{-2t} - 7e^{-3t}$



Example 4: Find Maximum Value (2 of 2)

- Find the maximum value attained by the solution.

$$y(t) = 9e^{-2t} - 7e^{-3t}$$

$$y'(t) = -18e^{-2t} + 21e^{-3t} \overset{set}{=} 0$$

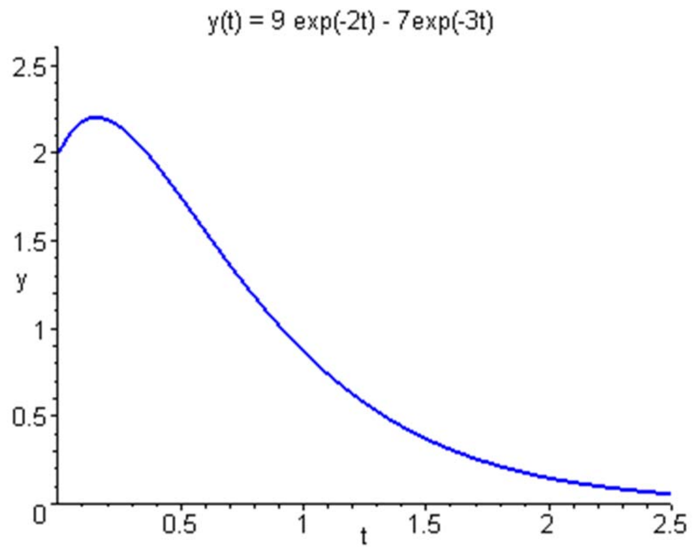
$$6e^{-2t} = 7e^{-3t}$$

$$e^{t} = 7/6$$

$$t = \ln(7/6)$$

$$t \approx 0.1542$$

$$y \approx 2.204$$



Ch 3.2: Fundamental Solutions of Linear Homogeneous Equations

- Let p, q be continuous functions on an interval $I = (\alpha, \beta)$, which could be infinite. For any function y that is twice differentiable on I, define the differential operator L by

$$L[y] = y'' + p\,y' + q\,y$$

- Note that $L[y]$ is a function on I, with output value

$$L[y](t) = y''(t) + p(t)\,y'(t) + q(t)\,y(t)$$

- For example,

$$p(t) = t^2,\ q(t) = e^{2t}, y(t) = \sin(t),\ \ I = (0, 2\pi)$$
$$L[y](t) = -\sin(t) + t^2\cos(t) + 2e^{2t}\sin(t)$$

Differential Operator Notation

- In this section we will discuss the second order linear homogeneous equation $L[y](t) = 0$, along with initial conditions as indicated below:

$$L[y] = y'' + p(t)\,y' + q(t)\,y = 0$$
$$y(t_0) = y_0,\ y'(t_0) = y_1$$

- We would like to know if there are solutions to this initial value problem, and if so, are they unique.
- Also, we would like to know what can be said about the form and structure of solutions that might be helpful in finding solutions to particular problems.
- These questions are addressed in the theorems of this section.

Theorem 3.2.1

- Consider the initial value problem

$$y'' + p(t)\,y' + q(t)\,y = g(t)$$
$$y(t_0) = y_0,\; y'(t_0) = y_0'$$

- where *p*, *q*, and *g* are continuous on an open interval *I* that contains t_0. Then there exists a unique solution $y = \phi(t)$ on *I*.

- Note: While this theorem says that a solution to the initial value problem above exists, it is often not possible to write down a useful expression for the solution. This is a major difference between first and second order linear equations.

$$y'' + p(t)\,y' + q(t)\,y = g(t)$$
$$y(t_0) = y_0,\; y'(t_0) = y_1$$

Example 1

- Consider the second order linear initial value problem
$$y'' - y = 0,\; y(0) = 3,\;\; y'(0) = 1$$
- In Section 3.1, we showed that this initial value problem had the following solution:
$$y(t) = 2e^{t} + e^{-t}$$
- Note that *p*(*t*) = 0, *q*(*t*) = -1, *g*(*t*) = 0 are each continuous on (-∞, ∞), and the solution *y* is defined and twice differentiable on (-∞, ∞).

Example 2

- Consider the second order linear initial value problem

$$y'' + p(t)y' + q(t)y = 0,\ y(0) = 0,\ \ y'(0) = 0$$

  where p, q are continuous on an open interval I containing t_0.
- In light of the initial conditions, note that $y = 0$ is a solution to this homogeneous initial value problem.
- Since the hypotheses of Theorem 3.2.1 are satisfied, it follows that $y = 0$ is the only solution of this problem.

Example 3

- Determine the longest interval on which the given initial value problem is certain to have a unique twice differentiable solution. Do not attempt to find the solution.

$$(t+1)y'' - (\cos t)y' + 3y = 1, \; y(0) = 1, \; y'(0) = 0$$

- First put differential equation into standard form:

$$y'' - \frac{\cos t}{t+1}y' + \frac{3}{t+1}y = \frac{1}{t+1}, \; y(0) = 1, \; y'(0) = 0$$

- The longest interval containing the point $t = 0$ on which the coefficient functions are continuous is $(-1, \infty)$.
- It follows from Theorem 3.2.1 that the longest interval on which this initial value problem is certain to have a twice differentiable solution is also $(-1, \infty)$.

Theorem 3.2.2 (Principle of Superposition)

- If $y_1$and y_2 are solutions to the equation
$$L[y] = y'' + p(t)\,y' + q(t)\,y = 0$$
then the linear combination $c_1y_1 + y_2c_2$ is also a solution, for all constants c_1 and c_2.

- To prove this theorem, substitute $c_1y_1 + y_2c_2$ in for y in the equation above, and use the fact that y_1 and y_2 are solutions.
- Thus for any two solutions y_1 and y_2, we can construct an infinite family of solutions, each of the form $y = c_1y_1 + c_2\,y_2$.
- Can all solutions can be written this way, or do some solutions have a different form altogether? To answer this question, we use the Wronskian determinant.

The Wronskian Determinant (1 of 3)

- Suppose y_1 and y_2 are solutions to the equation

$$L[y] = y'' + p(t)\,y' + q(t)\,y = 0$$

- From Theorem 3.2.2, we know that $y = c_1y_1 + c_2y_2$ is a solution to this equation.
- Next, find coefficients such that $y = c_1y_1 + c_2y_2$ satisfies the initial conditions

$$y(t_0) = y_0,\ y'(t_0) = y_0'$$

- To do so, we need to solve the following equations:

$$\begin{aligned} c_1y_1(t_0) + c_2y_2(t_0) &= y_0 \\ c_1y_1'(t_0) + c_2y_2'(t_0) &= y_0' \end{aligned}$$

$$c_1 y_1(t_0) + c_2 y_2(t_0) = y_0$$
$$c_1 y_1'(t_0) + c_2 y_2'(t_0) = y_0'$$

The Wronskian Determinant (2 of 3)

- Solving the equations, we obtain

$$c_1 = \frac{y_0 y_2'(t_0) - y_0' y_2(t_0)}{y_1(t_0) y_2'(t_0) - y_1'(t_0) y_2(t_0)}$$
$$c_2 = \frac{-y_0 y_1'(t_0) + y_0' y_1(t_0)}{y_1(t_0) y_2'(t_0) - y_1'(t_0) y_2(t_0)}$$

- In terms of determinants:

$$c_1 = \frac{\begin{vmatrix} y_0 & y_2(t_0) \\ y_0' & y_2'(t_0) \end{vmatrix}}{\begin{vmatrix} y_1(t_0) & y_2(t_0) \\ y_1'(t_0) & y_2'(t_0) \end{vmatrix}}, \quad c_2 = \frac{\begin{vmatrix} y_1(t_0) & y_0 \\ y_1'(t_0) & y_0' \end{vmatrix}}{\begin{vmatrix} y_1(t_0) & y_2(t_0) \\ y_1'(t_0) & y_2'(t_0) \end{vmatrix}}$$

The Wronskian Determinant (3 of 3)

- In order for these formulas to be valid, the determinant W in the denominator cannot be zero:

$$c_1 = \frac{\begin{vmatrix} y_0 & y_2(t_0) \\ y_0' & y_2'(t_0) \end{vmatrix}}{W}, \quad c_2 = \frac{\begin{vmatrix} y_1(t_0) & y_0 \\ y_1'(t_0) & y_0' \end{vmatrix}}{W}$$

$$W = \begin{vmatrix} y_1(t_0) & y_2(t_0) \\ y_1'(t_0) & y_2'(t_0) \end{vmatrix} = y_1(t_0)y_2'(t_0) - y_1'(t_0)y_2(t_0)$$

- W is called the **Wronskian determinant**, or more simply, the Wronskian of the solutions y_1 and y_2. We will sometimes use the notation $W(y_1, y_2)(t_0)$

Theorem 3.2.3

- Suppose y_1 and y_2 are solutions to the equation

$$L[y] = y'' + p(t)\,y' + q(t)\,y = 0 \qquad (1)$$

and that the Wronskian

$$W = y_1 y_2' - y_1' y_2$$

is not zero at the point t_0 where the initial conditions

$$y(t_0) = y_0,\; y'(t_0) = y_0' \qquad (2)$$

are assigned. Then there is a choice of constants c_1, c_2 for which $y = c_1 y_1 + c_2 y_2$ is a solution to the differential equation (1) and initial conditions (2).

Example 4

- Recall the following initial value problem and its solution:

$$y'' - y = 0,\ y(0) = 3,\ y'(0) = 1 \ \Rightarrow\ y(t) = 2e^t + e^{-t}$$

- Note that the two functions below are solutions to the differential equation:

$$y_1 = e^t, y_2 = e^{-t}$$

- The Wronskian of y_1 and y_2 is

$$W = \begin{vmatrix} y_1 & y_2 \\ y_1' & y_2' \end{vmatrix} = y_1 y_2' - y_1' y_2 = -e^t e^{-t} - e^t e^{-t} = -2e^0 = -2$$

- Since $W \neq 0$ for all t, linear combinations of y_1 and y_2 can be used to construct solutions of the IVP for any initial value t_0.

$$y = c_1 y_1 + c_2 y_2$$

Theorem 3.2.4 (Fundamental Solutions)

- Suppose y_1 and y_2 are solutions to the equation
$$L[y] = y'' + p(t)\,y' + q(t)\,y = 0.$$
If there is a point t_0 such that $W(y_1,y_2)(t_0) \neq 0$, then the family of solutions $y = c_1y_1 + c_2y_2$ with arbitrary coefficients c_1, c_2 includes every solution to the differential equation.

- The expression $y = c_1y_1 + c_2y_2$ is called the **general solution** of the differential equation above, and in this case y_1 and y_2 are said to form a **fundamental set of solutions** to the differential equation.

Example 5

- Recall the equation below, with the two solutions indicated:

$$y'' - y = 0, \quad y_1 = e^t, y_2 = e^{-t}$$

- The Wronskian of y_1 and y_2 is

$$W = \begin{vmatrix} y_1 & y_2 \\ y_1' & y_2' \end{vmatrix} = -e^t e^{-t} - e^t e^{-t} = -2e^0 = -2 \neq 0 \text{ for all } t.$$

- Thus y_1 and y_2 form a fundamental set of solutions to the differential equation above, and can be used to construct all of its solutions.
- The general solution is

$$y = c_1 e^t + c_2 e^{-t}$$

Example 6

- Consider the general second order linear equation below, with the two solutions indicated:

$$y'' + p(t)\,y' + q(t)\,y = 0$$

- Suppose the functions below are solutions to this equation:

$$y_1 = e^{r_1 t},\, y_2 = e^{r_2 t}, \quad r_1 \neq r_2$$

- The Wronskian of y_1 and y_2 is

$$W = \begin{vmatrix} y_1 & y_2 \\ y_1' & y_2' \end{vmatrix} = \begin{vmatrix} e^{r_1 t} & e^{r_2 t} \\ r_1 e^{r_1 t} & r_2 e^{r_2 t} \end{vmatrix} = (r_2 - r_1) e^{(r_1 + r_2)t} \neq 0 \text{ for all } t.$$

- Thus y_1 and y_2 form a fundamental set of solutions to the equation, and can be used to construct all of its solutions.
- The general solution is

$$y = c_1 e^{r_1 t} + c_2 e^{r_2 t}$$

Example 7: Solutions (1 of 2)

- Consider the following differential equation:

$$2t^2 y'' + 3t\, y' - y = 0, \;\; t > 0$$

- Show that the functions below are fundamental solutions:

$$y_1 = t^{1/2}, y_2 = t^{-1}$$

- To show this, first substitute y_1 into the equation:

$$2t^2\left(\frac{-t^{-3/2}}{4}\right) + 3t\left(\frac{t^{-1/2}}{2}\right) - t^{1/2} = \left(-\frac{1}{2} + \frac{3}{2} - 1\right)t^{1/2} = 0$$

- Thus y_1 is a indeed a solution of the differential equation.
- Similarly, y_2 is also a solution:

$$2t^2\left(2t^{-3}\right) + 3t\left(-t^{-2}\right) - t^{-1} = (4-3-1)t^{-1} = 0$$

Example 7: Fundamental Solutions (2 of 2)

- Recall that

$$y_1 = t^{1/2}, y_2 = t^{-1}$$

- To show that y_1 and y_2 form a fundamental set of solutions, we evaluate the Wronskian of y_1 and y_2:

$$W = \begin{vmatrix} y_1 & y_2 \\ y_1' & y_2' \end{vmatrix} = \begin{vmatrix} t^{1/2} & t^{-1} \\ \frac{1}{2}t^{-1/2} & -t^{-2} \end{vmatrix} = -t^{-3/2} - \frac{1}{2}t^{-3/2} = -\frac{3}{2}t^{-3/2} = -\frac{3}{2\sqrt{t^3}}$$

- Since $W \neq 0$ for $t > 0$, y_1, y_2 form a fundamental set of solutions for the differential equation

$$2t^2 y'' + 3t\, y' - y = 0, \;\; t > 0$$

Theorem 3.2.5: Existence of Fundamental Set of Solutions

- Consider the differential equation below, whose coefficients *p* and *q* are continuous on some open interval *I*:

$$L[y] = y'' + p(t)\,y' + q(t)\,y = 0$$

- Let t_0 be a point in *I*, and y_1 and y_2 solutions of the equation with y_1 satisfying initial conditions

$$y_1(t_0) = 1,\; y_1'(t_0) = 0$$

  and y_2 satisfying initial conditions

$$y_2(t_0) = 0,\; y_2'(t_0) = 1$$

- Then y_1, y_2 form a fundamental set of solutions to the given differential equation.

Example 7: Theorem 3.2.5 (1 of 3)

- Find the fundamental set specified by Theorem 3.2.5 for the differential equation and initial point

$$y'' - y = 0, \quad t_0 = 0$$

- We showed previously that

$$y_1 = e^t, y_2 = e^{-t}$$

were fundamental solutions, since $W(y_1, y_2)(t_0) = -2 \neq 0$.

- But these two solutions don’t satisfy the initial conditions stated in Theorem 3.2.5, and thus they do not form the fundamental set of solutions mentioned in that theorem.
- Let y_3 and y_4 be the fundamental solutions of Thm 3.2.5.

$$y_3(0) = 1, \; y_3'(0) = 0; \quad y_4(0) = 0, \; y_4'(0) = 1$$

Example 7: General Solution (2 of 3)

- Since y_1 and y_2 form a fundamental set of solutions,

$$y_3 = c_1 e^t + c_2 e^{-t}, \quad y_3(0) = 1, y_3'(0) = 0$$

$$y_4 = d_1 e^t + d_2 e^{-t}, \quad y_4(0) = 0, y_4'(0) = 1$$

- Solving each equation, we obtain

$$y_3(t) = \frac{1}{2}e^t + \frac{1}{2}e^{-t} = \cosh(t), \quad y_4(t) = \frac{1}{2}e^t - \frac{1}{2}e^{-t} = \sinh(t)$$

- The Wronskian of y_3 and y_4 is

$$W = \begin{vmatrix} y_1 & y_2 \\ y_1' & y_2' \end{vmatrix} = \begin{vmatrix} \cosh t & \sinh t \\ \sinh t & \cosh t \end{vmatrix} = \cosh^2 t - \sinh^2 t = 1 \neq 0$$

- Thus y_3, y_4 forms the fundamental set of solutions indicated in Theorem 3.2.5, with general solution in this case

$$y(t) = k_1 \cosh(t) + k_2 \sinh(t)$$

Example 7: Many Fundamental Solution Sets (3 of 3)

- Thus

$$S_1 = \left\{ e^t, e^{-t} \right\}, \quad S_2 = \left\{ \cosh t, \sinh t \right\}$$

  both form fundamental solution sets to the differential equation and initial point

$$y'' - y = 0, \quad t_0 = 0$$

- In general, a differential equation will have infinitely many different fundamental solution sets. Typically, we pick the one that is most convenient or useful.

Summary

- To find a general solution of the differential equation

$$y'' + p(t)\,y' + q(t)\,y = 0, \quad \alpha < t < \beta$$

  we first find two solutions y_1 and y_2.
- Then make sure there is a point t_0 in the interval such that $W(y_1, y_2)(t_0) \neq 0$.
- It follows that y_1 and y_2 form a fundamental set of solutions to the equation, with general solution $y = c_1 y_1 + c_2 y_2$.
- If initial conditions are prescribed at a point t_0 in the interval where $W \neq 0$, then c_1 and c_2 can be chosen to satisfy those conditions.

Ch 3.3: Linear Independence and the Wronskian

- Two functions f and g are **linearly dependent** if there exist constants c_1 and c_2, not both zero, such that
$$c_1 f(t) + c_2 g(t) = 0$$
for all t in I. Note that this reduces to determining whether f and g are multiples of each other.
- If the only solution to this equation is $c_1 = c_2 = 0$, then f and g are **linearly independent**.
- For example, let $f(x) = \sin 2x$ and $g(x) = \sin x \cos x$, and consider the linear combination
$$c_1 \sin 2x + c_2 \sin x \cos x = 0$$
This equation is satisfied if we choose $c_1 = 1$, $c_2 = -2$, and hence f and g are linearly dependent.

Solutions of 2 x 2 Systems of Equations

- When solving

$$c_1x_1 + c_2x_2 = a$$
$$c_1y_1 + c_2y_2 = b$$

  for c_1 and c_2, it can be shown that

$$c_1 = \frac{ay_2 - bx_2}{x_1y_2 - y_1x_2} = \frac{ay_2 - bx_2}{D},$$
$$c_2 = \frac{-ay_1 + bx_1}{x_1y_2 - y_1x_2} = \frac{-ay_1 + bx_1}{D}, \quad \text{where } D = \begin{vmatrix} x_1 & x_2 \\ y_1 & y_2 \end{vmatrix}$$

- Note that if $a = b = 0$, then the only solution to this system of equations is $c_1 = c_2 = 0$, provided $D \neq 0$.

Example 1: Linear Independence (1 of 2)

- Show that the following two functions are linearly independent on any interval:

$$f(t) = e^{t}, \quad g(t) = e^{-t}$$

- Let c_1 and c_2 be scalars, and suppose

$$c_1 f(t) + c_2 g(t) = 0$$

for all t in an arbitrary interval (α, β).

- We want to show $c_1 = c_2 = 0$. Since the equation holds for all t in (α, β), choose t_0 and t_1 in (α, β), where $t_0 \neq t_1$. Then

$$c_1 e^{t_0} + c_2 e^{-t_0} = 0$$

$$c_1 e^{t_1} + c_2 e^{-t_1} = 0$$

Example 1: Linear Independence (2 of 2)

- The solution to our system of equations

$$c_1 e^{t_0} + c_2 e^{-t_0} = 0$$

$$c_1 e^{t_1} + c_2 e^{-t_1} = 0$$

  will be $c_1 = c_2 = 0$, provided the determinant D is nonzero:

$$D = \begin{vmatrix} e^{t_0} & e^{-t_0} \\ e^{t_1} & e^{-t_1} \end{vmatrix} = e^{t_0}e^{-t_1} - e^{-t_0}e^{t_1} = e^{t_0 - t_1} - e^{t_1 - t_0}$$

- Then

$$D = 0 \Leftrightarrow e^{t_0 - t_1} = e^{t_1 - t_0} \Leftrightarrow e^{t_0 - t_1} = \frac{1}{e^{t_0 - t_1}} \Leftrightarrow \left(e^{t_0 - t_1}\right)^2 = 1$$

$$\Leftrightarrow e^{t_0 - t_1} = 1 \Leftrightarrow t_0 = t_1$$

- Since $t_0 \neq t_1$, it follows that $D \neq 0$, and therefore f and g are linearly independent.

Theorem 3.3.1

- If f and g are differentiable functions on an open interval I and if $W(f, g)(t_0) \neq 0$ for some point t_0 in I, then f and g are linearly independent on I. Moreover, if f and g are linearly dependent on I, then $W(f, g)(t) = 0$ for all t in I.

- Proof (outline): Let c_1 and c_2 be scalars, and suppose

$$c_1 f(t) + c_2 g(t) = 0$$

- for all t in I. In particular, when $t = t_0$ we have

$$c_1 f(t_0) + c_2 g(t_0) = 0$$
$$c_1 f'(t_0) + c_2 g'(t_0) = 0$$

- Since $W(f, g)(t_0) \neq 0$, it follows that $c_1 = c_2 = 0$, and hence f and g are linearly independent.

Theorem 3.3.2 (Abel's Theorem)

- Suppose y_1 and y_2 are solutions to the equation

$$L[y] = y'' + p(t)\,y' + q(t)\,y = 0$$

where p and q are continuous on some open interval I. Then $W(y_1,y_2)(t)$ is given by

$$W(y_1, y_2)(t) = ce^{-\int p(t)dt}$$

where c is a constant that depends on y_1 and y_2 but not on t.

- Note that $W(y_1,y_2)(t)$ is either zero for all t in I (if $c = 0$) or else is never zero in I (if $c \neq 0$).

Example 2: Wronskian and Abel's Theorem

- Recall the following equation and two of its solutions:

$$y'' - y = 0, \quad y_1 = e^t, y_2 = e^{-t}$$

- The Wronskian of y_1 and y_2 is

$$W = \begin{vmatrix} y_1 & y_2 \\ y_1' & y_2' \end{vmatrix} = -e^t e^{-t} - e^t e^{-t} = -2e^0 = -2 \neq 0 \text{ for all } t.$$

- Thus y_1 and y_2 are linearly independent on any interval *I*, by Theorem 3.3.1. Now compare *W* with Abel's Theorem:

$$W(y_1, y_2)(t) = ce^{-\int p(t)dt} = ce^{-\int 0 dt} = c$$

- Choosing c = -2, we get the same *W* as above.

Theorem 3.3.3

- Suppose y_1 and y_2 are solutions to equation below, whose coefficients p and q are continuous on some open interval I:

$$L[y] = y'' + p(t)\,y' + q(t)\,y = 0$$

Then y_1 and y_2 are linearly dependent on I iff $W(y_1, y_2)(t) = 0$ for all t in I. Also, y_1 and y_2 are linearly independent on I iff $W(y_1, y_2)(t) \neq 0$ for all t in I.

Summary

- Let y_1 and y_2 be solutions of

$$y'' + p(t)\,y' + q(t)\,y = 0$$

  where p and q are continuous on an open interval I.
- Then the following statements are equivalent:
  - The functions y_1 and y_2 form a fundamental set of solutions on I.
  - The functions y_1 and y_2 are linearly independent on I.
  - $W(y_1,y_2)(t_0) \neq 0$ for some t_0 in I.
  - $W(y_1,y_2)(t) \neq 0$ for all t in I.

Linear Algebra Note

- Let V be the set

$$V = \{y : y'' + p(t)\,y' + q(t)\,y = 0,\ t \in (\alpha, \beta)\}$$

Then V is a vector space of dimension two, whose bases are given by any fundamental set of solutions y_1 and y_2.

- For example, the solution space V to the differential equation

$$y'' - y = 0$$

has bases

$$S_1 = \{e^t, e^{-t}\},\quad S_2 = \{\cosh t, \sinh t\}$$

with

$$V = \operatorname{Span} S_1 = \operatorname{Span} S_2$$

Ch 3.4: Complex Roots of Characteristic Equation

- Recall our discussion of the equation

$$ay'' + by' + cy = 0$$

  where a, b and c are constants.
- Assuming an exponential soln leads to characteristic equation:

$$y(t) = e^{rt} \quad \Rightarrow \quad ar^2 + br + c = 0$$

- Quadratic formula (or factoring) yields two solutions, r_1 & r_2:

$$r = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- If $b^2 - 4ac < 0$, then complex roots: $r_1 = \lambda + i\mu$, $r_2 = \lambda - i\mu$
- Thus

$$y_1(t) = e^{(\lambda + i\mu)t}, \quad y_2(t) = e^{(\lambda - i\mu)t}$$

Euler's Formula; Complex Valued Solutions

- Substituting *it* into Taylor series for e^t, we obtain **Euler's formula**:

$$e^{it} = \sum_{n=0}^{\infty} \frac{(it)^n}{n!} = \sum_{n=0}^{\infty} \frac{(-1)^n t^{2n}}{(2n)!} + i \sum_{n=1}^{\infty} \frac{(-1)^{n-1} t^{2n-1}}{(2n-1)!} = \cos t + i \sin t$$

- Generalizing Euler's formula, we obtain

$$e^{i\mu t} = \cos \mu t + i \sin \mu t$$

- Then

$$e^{(\lambda + i\mu)t} = e^{\lambda t} e^{i\mu t} = e^{\lambda t}\left[\cos \mu t + i \sin \mu t\right] = e^{\lambda t} \cos \mu t + i e^{\lambda t} \sin \mu t$$

- Therefore

$$y_1(t) = e^{(\lambda + i\mu)t} = e^{\lambda t} \cos \mu t + i e^{\lambda t} \sin \mu t$$
$$y_2(t) = e^{(\lambda - i\mu)t} = e^{\lambda t} \cos \mu t - i e^{\lambda t} \sin \mu t$$

Real Valued Solutions

- Our two solutions thus far are complex-valued functions:

$$y_1(t) = e^{\lambda t}\cos\mu t + ie^{\lambda t}\sin\mu t$$
$$y_2(t) = e^{\lambda t}\cos\mu t - ie^{\lambda t}\sin\mu t$$

- We would prefer to have real-valued solutions, since our differential equation has real coefficients.
- To achieve this, recall that linear combinations of solutions are themselves solutions:

$$y_1(t) + y_2(t) = 2e^{\lambda t}\cos\mu t$$
$$y_1(t) - y_2(t) = 2ie^{\lambda t}\sin\mu t$$

- Ignoring constants, we obtain the two solutions

$$y_3(t) = e^{\lambda t}\cos\mu t,\ \ y_4(t) = e^{\lambda t}\sin\mu t$$

Real Valued Solutions: The Wronskian

- Thus we have the following real-valued functions:

$$y_3(t) = e^{\lambda t}\cos\mu t, \;\; y_4(t) = e^{\lambda t}\sin\mu t$$

- Checking the Wronskian, we obtain

$$W = \begin{vmatrix} e^{\lambda t}\cos\mu t & e^{\lambda t}\sin\mu t \\ e^{\lambda t}\left(\lambda\cos\mu t - \mu\sin\mu t\right) & e^{\lambda t}\left(\lambda\sin\mu t + \mu\cos\mu t\right) \end{vmatrix}$$

$$= \mu e^{2\lambda t} \neq 0$$

- Thus y_3 and y_4 form a fundamental solution set for our ODE, and the general solution can be expressed as

$$y(t) = c_1 e^{\lambda t}\cos\mu t + c_2 e^{\lambda t}\sin\mu t$$

Example 1

- Consider the equation

$$y'' + y' + y = 0$$

- Then

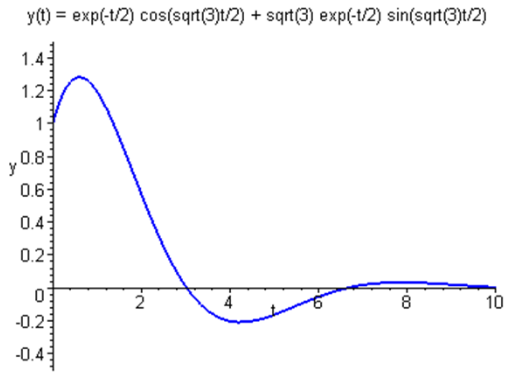
$$y(t) = e^{rt} \;\Rightarrow\; r^2 + r + 1 = 0 \;\Leftrightarrow\; r = \frac{-1 \pm \sqrt{1-4}}{2} = \frac{-1 \pm \sqrt{3}\,i}{2} = -\frac{1}{2} \pm \frac{\sqrt{3}}{2} i$$

- Therefore

$$\lambda = -1/2, \;\mu = \sqrt{3}/2$$

and thus the general solution is

$$y(t) = c_1 e^{-t/2}\cos\left(\sqrt{3}t/2\right) + c_2 e^{-t/2}\sin\left(\sqrt{3}t/2\right)$$



Example 2

- Consider the equation

$$y'' + 4y = 0$$

- Then

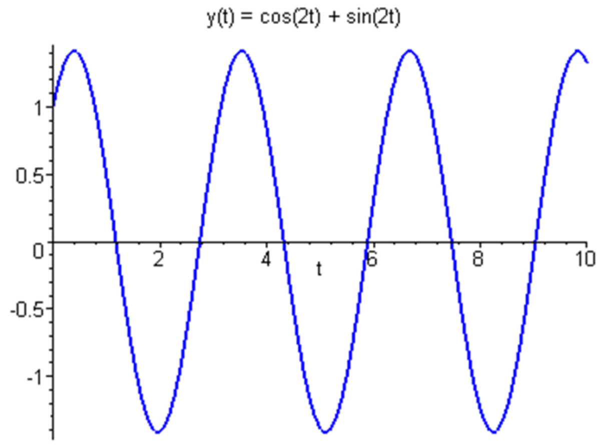
$$y(t) = e^{rt} \quad \Rightarrow \quad r^2 + 4 = 0 \;\Leftrightarrow\; r = \pm 2i$$

- Therefore

$$\lambda = 0,\ \mu = 2$$

and thus the general solution is

$$y(t) = c_1 \cos(2t) + c_2 \sin(2t)$$



Example 3

- Consider the equation

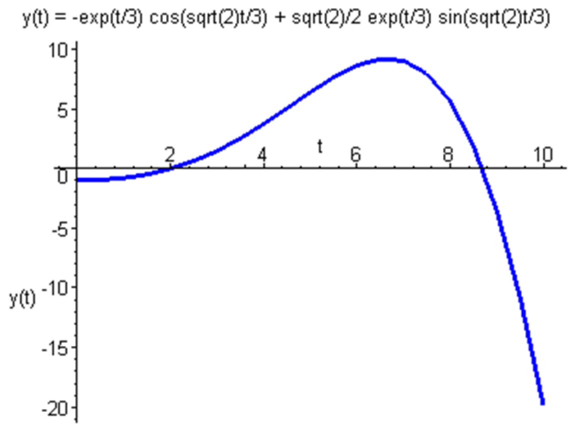
$$3y'' - 2y' + y = 0$$

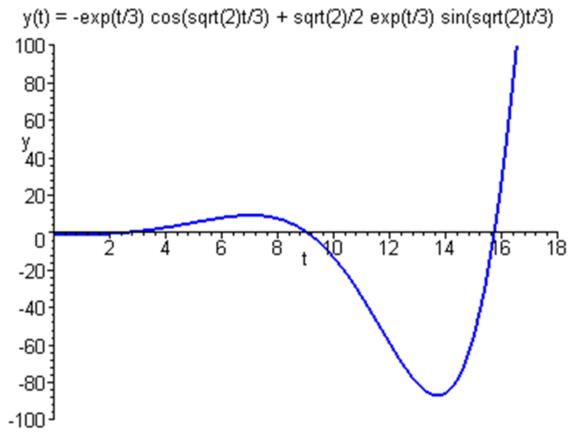
- Then

$$y(t) = e^{rt} \Rightarrow 3r^2 - 2r + 1 = 0 \Leftrightarrow r = \frac{2 \pm \sqrt{4-12}}{6} = \frac{1}{3} \pm \frac{\sqrt{2}}{3}i$$

- Therefore the general solution is

$$y(t) = c_1 e^{t/3} \cos\left(\sqrt{2}t/3\right) + c_2 e^{t/3} \sin\left(\sqrt{2}t/3\right)$$





Example 4: Part (a) (1 of 2)

- For the initial value problem below, find (a) the solution $u(t)$ and (b) the smallest time T for which $|u(t)| \le 0.1$

$$y'' + y' + y = 0, \quad y(0) = 1, \; y'(0) = 1$$

- We know from Example 1 that the general solution is

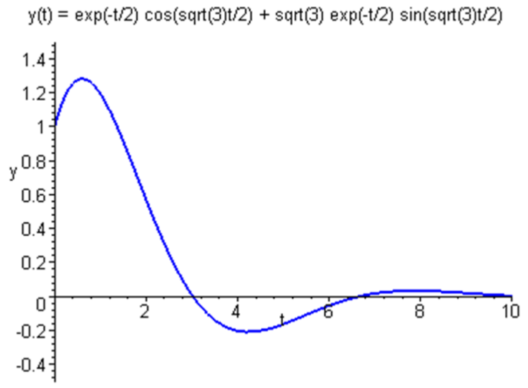
$$u(t) = c_1 e^{-t/2} \cos\left(\sqrt{3}t/2\right) + c_2 e^{-t/2} \sin\left(\sqrt{3}t/2\right)$$

- Using the initial conditions, we obtain

$$\left.\begin{aligned} c_1 \qquad\qquad &= 1 \\ -\frac{1}{2}c_1 + \frac{\sqrt{3}}{2}c_2 &= 1 \end{aligned}\right\} \Rightarrow c_1 = 1, \; c_2 = \frac{3}{\sqrt{3}} = \sqrt{3}$$

- Thus

$$u(t) = e^{-t/2} \cos\left(\sqrt{3}t/2\right) + \sqrt{3}\, e^{-t/2} \sin\left(\sqrt{3}t/2\right)$$

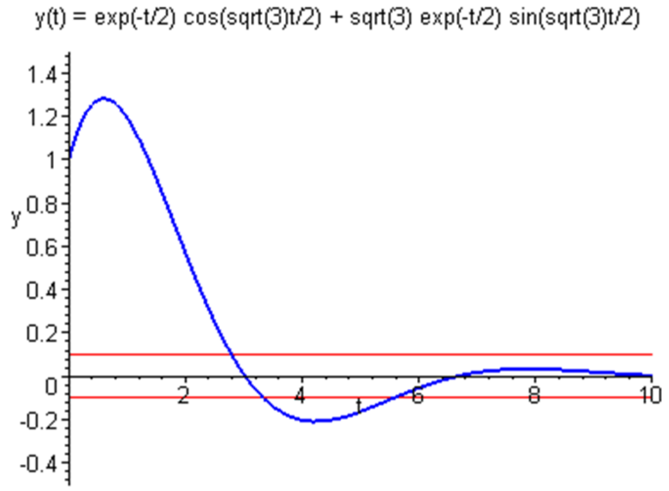


Example 4: Part (b) (2 of 2)

- Find the smallest time T for which $|u(t)| \le 0.1$
- Our solution is

$$u(t) = e^{-t/2}\cos\left(\sqrt{3}t/2\right) + \sqrt{3}\,e^{-t/2}\sin\left(\sqrt{3}t/2\right)$$

- With the help of graphing calculator or computer algebra system, we find that $T \cong 2.79$. See graph below.



Ch 3.5: Repeated Roots; Reduction of Order

- Recall our 2nd order linear homogeneous ODE

$$ay'' + by' + cy = 0$$

- where a, b and c are constants.
- Assuming an exponential soln leads to characteristic equation:

$$y(t) = e^{rt} \quad \Rightarrow \quad ar^2 + br + c = 0$$

- Quadratic formula (or factoring) yields two solutions, r_1 & r_2:

$$r = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- When $b^2 - 4ac = 0$, $r_1 = r_2 = -b/2a$, since method only gives one solution:

$$y_1(t) = ce^{-bt/2a}$$

Second Solution: Multiplying Factor *v*(*t*)

- We know that

$$y_1(t) \text{ a solution} \Rightarrow y_2(t) = cy_1(t) \text{ a solution}$$

- Since y_1 and y_2 are linearly dependent, we generalize this approach and multiply by a function *v*, and determine conditions for which y_2 is a solution:

$$y_1(t) = e^{-bt/2a} \text{ a solution} \Rightarrow \text{ try } \; y_2(t) = v(t)e^{-bt/2a}$$

- Then

$$y_2(t) = v(t)e^{-bt/2a}$$

$$y_2'(t) = v'(t)e^{-bt/2a} - \frac{b}{2a}v(t)e^{-bt/2a}$$

$$y_2''(t) = v''(t)e^{-bt/2a} - \frac{b}{2a}v'(t)e^{-bt/2a} - \frac{b}{2a}v'(t)e^{-bt/2a} + \frac{b^2}{4a^2}v(t)e^{-bt/2a}$$

$$ay'' + by' + cy = 0$$

Finding Multiplying Factor *v*(*t*)

- Substituting derivatives into ODE, we seek a formula for *v*:

$$e^{-bt/2a}\left\{a\left[v''(t) - \frac{b}{a}v'(t) + \frac{b^2}{4a^2}v(t)\right] + b\left[v'(t) - \frac{b}{2a}v(t)\right] + cv(t)\right\} = 0$$

$$av''(t) - bv'(t) + \frac{b^2}{4a}v(t) + bv'(t) - \frac{b^2}{2a}v(t) + cv(t) = 0$$

$$av''(t) + \left(\frac{b^2}{4a} - \frac{b^2}{2a} + c\right)v(t) = 0$$

$$av''(t) + \left(\frac{b^2}{4a} - \frac{2b^2}{4a} + \frac{4ac}{4a}\right)v(t) = 0 \quad \Leftrightarrow \quad av''(t) + \left(\frac{-b^2}{4a} + \frac{4ac}{4a}\right)v(t) = 0$$

$$av''(t) - \left(\frac{b^2 - 4ac}{4a}\right)v(t) = 0$$

$$v''(t) = 0 \quad \Rightarrow \quad v(t) = k_3 t + k_4$$

General Solution

- To find our general solution, we have:

$$\begin{aligned} y(t) &= k_1 e^{-bt/2a} + k_2 v(t) e^{-bt/2a} \\ &= k_1 e^{-bt/2a} + \left(k_3 t + k_4\right) e^{-bt/2a} \\ &= c_1 e^{-bt/2a} + c_2 t e^{-bt/2a} \end{aligned}$$

- Thus the general solution for repeated roots is

$$y(t) = c_1 e^{-bt/2a} + c_2 t e^{-bt/2a}$$

Wronskian

- The general solution is
$$y(t) = c_1 e^{-bt/2a} + c_2 t e^{-bt/2a}$$
- Thus every solution is a linear combination of
$$y_1(t) = e^{-bt/2a},\ y_2(t) = te^{-bt/2a}$$
- The Wronskian of the two solutions is
$$\begin{aligned} W(y_1, y_2)(t) &= \begin{vmatrix} e^{-bt/2a} & te^{-bt/2a} \\ -\dfrac{b}{2a} e^{-bt/2a} & \left(1 - \dfrac{bt}{2a}\right) e^{-bt/2a} \end{vmatrix} \\ &= e^{-bt/a}\left(1 - \frac{bt}{2a}\right) + e^{-bt/a}\left(\frac{bt}{2a}\right) \\ &= e^{-bt/a} \neq 0 \quad \text{for all } t \end{aligned}$$
- Thus y_1 and y_2 form a fundamental solution set for equation.

Example 1

- Consider the initial value problem

$$y'' + 2y' + y = 0, \quad y(0) = 1, \ \ y'(0) = 1$$

- Assuming exponential soln leads to characteristic equation:

$$y(t) = e^{rt} \quad \Rightarrow \quad r^2 + 2r + 1 = 0 \ \Leftrightarrow \ (r+1)^2 = 0 \ \Leftrightarrow r = -1$$

- Thus the general solution is

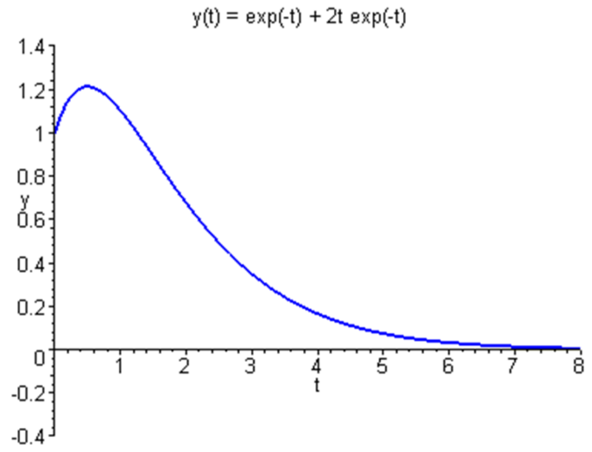
$$y(t) = c_1 e^{-t} + c_2 t e^{-t}$$

- Using the initial conditions:

$$\left.\begin{array}{rcrcl} c_1 & & & = & 1 \\ -c_1 & + & c_2 & = & 1 \end{array}\right\} \Rightarrow c_1 = 1, \ c_2 = 2$$

- Thus

$$y(t) = e^{-t} + 2te^{-t}$$



Example 2

- Consider the initial value problem

$$y'' - y' + 0.25y = 0, \quad y(0) = 2, \quad y'(0) = 1/2$$

- Assuming exponential soln leads to characteristic equation:

$$y(t) = e^{rt} \quad \Rightarrow \quad r^2 - r + 0.25 = 0 \;\Leftrightarrow\; (r - 1/2)^2 = 0 \;\Leftrightarrow r = 1/2$$

- Thus the general solution is

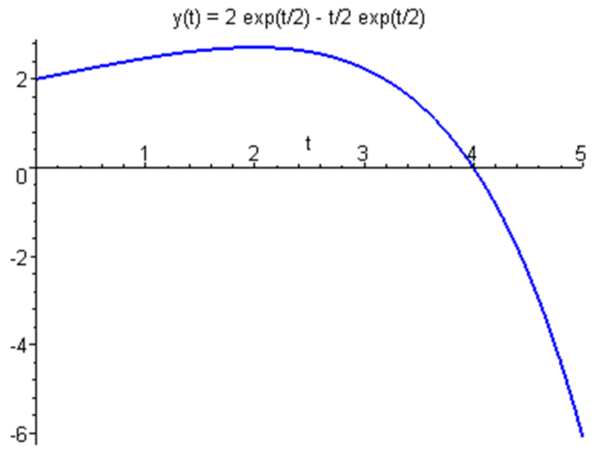
$$y(t) = c_1 e^{t/2} + c_2 t e^{t/2}$$

- Using the initial conditions:

$$\left.\begin{array}{rcl} c_1 & = & 2 \\ \frac{1}{2}c_1 + c_2 & = & \frac{1}{2} \end{array}\right\} \Rightarrow c_1 = 2,\; c_2 = -\frac{1}{2}$$

- Thus

$$y(t) = 2e^{t/2} - \frac{1}{2} t e^{t/2}$$



Example 3

- Consider the initial value problem

$$y'' - y' + 0.25y = 0, \quad y(0) = 2, \quad y'(0) = 3/2$$

- Assuming exponential soln leads to characteristic equation:

$$y(t) = e^{rt} \quad \Rightarrow \quad r^2 - r + 0.25 = 0 \;\Leftrightarrow\; (r - 1/2)^2 = 0 \;\Leftrightarrow r = 1/2$$

- Thus the general solution is

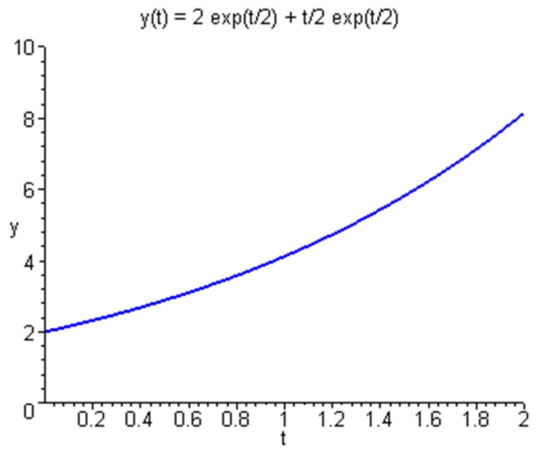
$$y(t) = c_1 e^{t/2} + c_2 t e^{t/2}$$

- Using the initial conditions:

$$\left.\begin{array}{rcrcl} c_1 & & & = & 2 \\ \frac{1}{2}c_1 & + & c_2 & = & \frac{3}{2} \end{array}\right\} \Rightarrow c_1 = 2,\; c_2 = \frac{1}{2}$$

- Thus

$$y(t) = 2e^{t/2} + \frac{1}{2} t e^{t/2}$$



Reduction of Order

- The method used so far in this section also works for equations with nonconstant coefficients:

$$y'' + p(t)y' + q(t)y = 0$$

- That is, given that y_1 is solution, try $y_2 = v(t)y_1$:

$$y_2(t) = v(t)y_1(t)$$

$$y_2'(t) = v'(t)y_1(t) + v(t)y_1'(t)$$

$$y_2''(t) = v''(t)y_1(t) + 2v'(t)y_1'(t) + v(t)y_1''(t)$$

- Substituting these into ODE and collecting terms,

$$y_1 v'' + (2y_1' + py_1)v' + (y_1'' + py_1' + qy_1)v = 0$$

- Since y_1 is a solution to the differential equation, this last equation reduces to a first order equation in v':

$$y_1 v'' + (2y_1' + py_1)v' = 0$$

Example 4: Reduction of Order (1 of 3)

- Given the variable coefficient equation and solution y_1,

$$t^2 y'' + 3ty' + y = 0, \quad t > 0; \quad y_1(t) = t^{-1},$$

use reduction of order method to find a second solution:

$$y_2(t) = v(t)\,t^{-1}$$

$$y_2'(t) = v'(t)\,t^{-1} - v(t)\,t^{-2}$$

$$y_2''(t) = v''(t)\,t^{-1} - 2v'(t)\,t^{-2} + 2v(t)\,t^{-3}$$

- Substituting these into ODE and collecting terms,

$$t^2\left(v''t^{-1} - 2v't^{-2} + 2vt^{-3}\right) + 3t\left(v't^{-1} - vt^{-2}\right) + vt^{-1} = 0$$

$$\Leftrightarrow v''t - 2v' + 2vt^{-1} + 3v' - 3vt^{-1} + vt^{-1} = 0$$

$$\Leftrightarrow \; tv'' + v' = 0$$

$$\Leftrightarrow \; tu' + u = 0, \; \text{ where } u(t) = v'(t)$$

Example 4: Finding *v*(*t*) (2 of 3)

- To solve

$$tu' + u = 0, \quad u(t) = v'(t)$$

for *u*, we can use the separation of variables method:

$$t\frac{du}{dt} + u = 0 \quad \Leftrightarrow \quad \int \frac{du}{u} = -\int \frac{1}{t}\,dt \quad \Leftrightarrow \quad \ln|u| = -\ln|t| + C$$

$$\Leftrightarrow \quad |u| = |t|^{-1}e^{C} \quad \Leftrightarrow \quad u = ct^{-1}, \quad \text{since } t > 0.$$

- Thus

$$v' = \frac{c}{t}$$

and hence

$$v(t) = c\ln t + k$$

Example 4: General Solution (3 of 3)

- We have
$$v(t) = c\ln t + k$$
- Thus
$$y_2(t) = \left(c\ln t + k\right)t^{-1} = ct^{-1}\ln t + k\,t^{-1}$$
- Recall
$$y_1(t) = t^{-1}$$
and hence we can neglect the second term of y_2 to obtain
$$y_2(t) = t^{-1}\ln t.$$
- Hence the general solution to the differential equation is
$$y(t) = c_1 t^{-1} + c_2 t^{-1}\ln t$$

Ch 3.6: Nonhomogeneous Equations; Method of Undetermined Coefficients

- Recall the nonhomogeneous equation

$$y'' + p(t)y' + q(t)y = g(t)$$

  where *p*, *q*, *g* are continuous functions on an open interval *I*.
- The associated homogeneous equation is

$$y'' + p(t)y' + q(t)y = 0$$

- In this section we will learn the method of undetermined coefficients to solve the nonhomogeneous equation, which relies on knowing solutions to homogeneous equation.

Theorem 3.6.1

- If Y_1, Y_2 are solutions of nonhomogeneous equation

$$y'' + p(t)y' + q(t)y = g(t)$$

then Y_1 - Y_2 is a solution of the homogeneous equation

$$y'' + p(t)y' + q(t)y = 0$$

- If y_1, y_2 form a fundamental solution set of homogeneous equation, then there exists constants c_1, c_2 such that

$$Y_1(t) - Y_2(t) = c_1 y_1(t) + c_2 y_2(t)$$

Theorem 3.6.2 (General Solution)

- The general solution of nonhomogeneous equation

$$y'' + p(t)y' + q(t)y = g(t)$$

can be written in the form

$$y(t) = c_1 y_1(t) + c_2 y_2(t) + Y(t)$$

where y_1, y_2 form a fundamental solution set of homogeneous equation, c_1, c_2 are arbitrary constants and Y is a specific solution to the nonhomogeneous equation.

Method of Undetermined Coefficients

- Recall the nonhomogeneous equation

$$y'' + p(t)y' + q(t)y = g(t)$$

with general solution

$$y(t) = c_1 y_1(t) + c_2 y_2(t) + Y(t)$$

- In this section we use the method of **undetermined coefficients** to find a particular solution Y to the nonhomogeneous equation, assuming we can find solutions y_1, y_2 for the homogeneous case.
- The method of undetermined coefficients is usually limited to when p and q are constant, and $g(t)$ is a polynomial, exponential, sine or cosine function.

Example 1: Exponential $g(t)$

- Consider the nonhomogeneous equation

$$y'' - 3y' - 4y = 3e^{2t}$$

- We seek Y satisfying this equation. Since exponentials replicate through differentiation, a good start for Y is:

$$Y(t) = Ae^{2t} \Rightarrow Y'(t) = 2Ae^{2t},\ Y''(t) = 4Ae^{2t}$$

- Substituting these derivatives into differential equation,

$$4Ae^{2t} - 6Ae^{2t} - 4Ae^{2t} = 3e^{2t}$$

$$\Leftrightarrow \quad -6Ae^{2t} = 3e^{2t} \quad \Leftrightarrow \quad A = -1/2$$

- Thus a particular solution to the nonhomogeneous ODE is

$$Y(t) = -\frac{1}{2}e^{2t}$$

Example 2: Sine $g(t)$, First Attempt (1 of 2)

- Consider the nonhomogeneous equation

$$y'' - 3y' - 4y = 2\sin t$$

- We seek Y satisfying this equation. Since sines replicate through differentiation, a good start for Y is:

$$Y(t) = A\sin t \Rightarrow Y'(t) = A\cos t,\ Y''(t) = -A\sin t$$

- Substituting these derivatives into differential equation,

$$-A\sin t - 3A\cos t - 4A\sin t = 2\sin t$$

$$\Leftrightarrow\ (2+5A)\sin t + 3A\cos t = 0$$

$$\Leftrightarrow\ c_1\sin t + c_2\cos t = 0$$

- Since sin(x) and cos(x) are linearly independent (they are not multiples of each other), we must have $c_1 = c_2 = 0$, and hence $2 + 5A = 3A = 0$, which is impossible.

$$y'' - 3y' - 4y = 2\sin t$$

Example 2: Sine *g*(*t*), Particular Solution (2 of 2)

- Our next attempt at finding a *Y* is

$$Y(t) = A\sin t + B\cos t$$
$$\Rightarrow Y'(t) = A\cos t - B\sin t,\ Y''(t) = -A\sin t - B\cos t$$

- Substituting these derivatives into ODE, we obtain

$$\left(-A\sin t - B\cos t\right) - 3\left(A\cos t - B\sin t\right) - 4\left(A\sin t + B\cos t\right) = 2\sin t$$
$$\Leftrightarrow\ \left(-5A + 3B\right)\sin t + \left(-3A - 5B\right)\cos t = 2\sin t$$
$$\Leftrightarrow -5A + 3B = 2,\ -3A - 5B = 0$$
$$\Leftrightarrow\ A = -5/17,\ B = 3/17$$

- Thus a particular solution to the nonhomogeneous ODE is

$$Y(t) = \frac{-5}{17}\sin t + \frac{3}{17}\cos t$$

Example 3: Polynomial $g(t)$

- Consider the nonhomogeneous equation

$$y'' - 3y' - 4y = 4t^2 - 1$$

- We seek Y satisfying this equation. We begin with

$$Y(t) = At^2 + Bt + C \Rightarrow Y'(t) = 2At + B, \; Y''(t) = 2A$$

- Substituting these derivatives into differential equation,

$$2A - 3(2At + B) - 4\left(At^2 + Bt + C\right) = 4t^2 - 1$$

$$\Leftrightarrow \; -4At^2 - (6A + 4B)t + (2A - 3B - 4C) = 4t^2 - 1$$

$$\Leftrightarrow \; -4A = 4, \; 6A + 4B = 0, \; 2A - 3B - 4C = -1$$

$$\Leftrightarrow \; A = -1, \; B = 3/2, \; C = -11/8$$

- Thus a particular solution to the nonhomogeneous ODE is

$$Y(t) = -t^2 + \frac{3}{2}t - \frac{11}{8}$$

Example 4: Product $g(t)$

- Consider the nonhomogeneous equation

$$y'' - 3y' - 4y = -8e^t \cos 2t$$

- We seek Y satisfying this equation, as follows:

$$Y(t) = Ae^t \cos 2t + Be^t \sin 2t$$

$$Y'(t) = Ae^t \cos 2t - 2Ae^t \sin 2t + Be^t \sin 2t + 2Be^t \cos 2t$$

$$= (A + 2B)e^t \cos 2t + (-2A + B)e^t \sin 2t$$

$$Y''(t) = (A + 2B)e^t \cos 2t - 2(A + 2B)e^t \sin 2t + (-2A + B)e^t \sin 2t$$

$$+ 2(-2A + B)e^t \cos 2t$$

$$= (-3A + 4B)e^t \cos 2t + (-4A - 3B)e^t \sin 2t$$

- Substituting derivatives into ODE and solving for A and B:

$$A = \frac{10}{13},\ B = \frac{2}{13} \quad \Rightarrow Y(t) = \frac{10}{13}e^t \cos 2t + \frac{2}{13}e^t \sin 2t$$

Discussion: Sum $g(t)$

- Consider again our general nonhomogeneous equation

$$y'' + p(t)y' + q(t)y = g(t)$$

- Suppose that $g(t)$ is sum of functions:

$$g(t) = g_1(t) + g_2(t)$$

- If Y_1, Y_2 are solutions of

$$y'' + p(t)y' + q(t)y = g_1(t)$$
$$y'' + p(t)y' + q(t)y = g_2(t)$$

respectively, then $Y_1 + Y_2$ is a solution of the nonhomogeneous equation above.

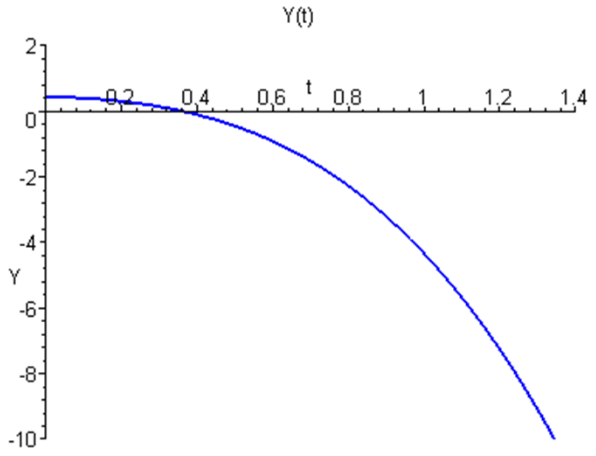
Example 5: Sum $g(t)$

- Consider the equation

$$y'' - 3y' - 4y = 3e^{2t} + 2\sin t - 8e^{t}\cos 2t$$

- Our equations to solve individually are

$$y'' - 3y' - 4y = 3e^{2t}$$
$$y'' - 3y' - 4y = 2\sin t$$
$$y'' - 3y' - 4y = -8e^{t}\cos 2t$$



- Our particular solution is then

$$Y(t) = -\frac{1}{2}e^{2t} + \frac{3}{17}\cos t - \frac{5}{17}\sin t + \frac{10}{13}e^{t}\cos 2t + \frac{2}{13}e^{t}\sin 2t$$

Example 6: First Attempt (1 of 3)

- Consider the equation

$$y'' + 4y = 3\cos 2t$$

- We seek Y satisfying this equation. We begin with

$$Y(t) = A\sin 2t + B\cos 2t$$

$$\Rightarrow Y'(t) = 2A\cos 2t - 2B\sin 2t,\ Y''(t) = -4A\sin 2t - 4B\cos 2t$$

- Substituting these derivatives into ODE:

$$\begin{aligned}(-4A\sin 2t - 4B\cos 2t) + 4(A\sin 2t + B\cos 2t) &= 3\cos 2t\\ (-4A + 4A)\sin 2t + (-4B + 4B)\cos 2t &= 3\cos 2t\\ 0 &= 3\cos 2t\end{aligned}$$

- Thus no particular solution exists of the form

$$Y(t) = A\sin 2t + B\cos 2t$$

Example 6: Homogeneous Solution (2 of 3)

- Thus no particular solution exists of the form

$$Y(t) = A\sin 2t + B\cos 2t$$

- To help understand why, recall that we found the corresponding homogeneous solution in Section 3.4 notes:

$$y'' + 4y = 0 \quad \Rightarrow \quad y(t) = c_1 \cos 2t + c_2 \sin 2t$$

- Thus our assumed particular solution solves homogeneous equation

$$y'' + 4y = 0$$

instead of the nonhomogeneous equation.

$$y'' + 4y = 3\cos 2t$$

$$y'' + 4y = 3\cos 2t$$

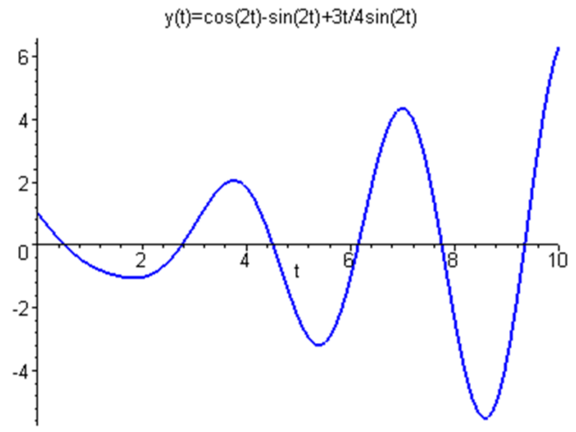
Example 6: Particular Solution (3 of 3)

- Our next attempt at finding a Y is:

$$Y(t) = At\sin 2t + Bt\cos 2t$$
$$Y'(t) = A\sin 2t + 2At\cos 2t + B\cos 2t - 2Bt\sin 2t$$
$$Y''(t) = 2A\cos 2t + 2A\cos 2t - 4At\sin 2t - 2B\sin 2t - 2B\sin 2t - 4Bt\cos 2t$$
$$= 4A\cos 2t - 4B\sin 2t - 4At\sin 2t - 4Bt\cos 2t$$

- Substituting derivatives into ODE,

$$4A\cos 2t - 4B\sin 2t = 3\cos 2t$$
$$\Rightarrow A = 3/4,\ B = 0$$
$$\Rightarrow Y(t) = \frac{3}{4}t\sin 2t$$



Ch 3.7: Variation of Parameters

- Recall the nonhomogeneous equation

$$y'' + p(t)y' + q(t)y = g(t)$$

  where p, q, g are continuous functions on an open interval I.
- The associated homogeneous equation is

$$y'' + p(t)y' + q(t)y = 0$$

- In this section we will learn the **variation of parameters** method to solve the nonhomogeneous equation. As with the method of undetermined coefficients, this procedure relies on knowing solutions to homogeneous equation.
- Variation of parameters is a general method, and requires no detailed assumptions about solution form. However, certain integrals need to be evaluated, and this can present difficulties.

Example: Variation of Parameters (1 of 6)

- We seek a particular solution to the equation below.

$$y'' + 4y = 3\csc t$$

- We cannot use method of undetermined coefficients since $g(t)$ is a quotient of $\sin t$ or $\cos t$, instead of a sum or product.
- Recall that the solution to the homogeneous equation is

$$y_C(t) = c_1 \cos 2t + c_2 \sin 2t$$

- To find a particular solution to the nonhomogeneous equation, we begin with the form

$$y(t) = u_1(t)\cos 2t + u_2(t)\sin 2t$$

- Then

$$y'(t) = u_1'(t)\cos 2t - 2u_1(t)\sin 2t + u_2'(t)\sin 2t + 2u_2(t)\cos 2t$$

- or

$$y'(t) = -2u_1(t)\sin 2t + 2u_2(t)\cos 2t + u_1'(t)\cos 2t + u_2'(t)\sin 2t$$

Example: Derivatives, 2nd Equation (2 of 6)

- From the previous slide,

$$y'(t) = -2u_1(t)\sin 2t + 2u_2(t)\cos 2t + u_1'(t)\cos 2t + u_2'(t)\sin 2t$$

- Note that we need two equations to solve for u_1 and u_2. The first equation is the differential equation. To get a second equation, we will require

$$u_1'(t)\cos 2t + u_2'(t)\sin 2t = 0$$

- Then

$$y'(t) = -2u_1(t)\sin 2t + 2u_2(t)\cos 2t$$

- Next,

$$y''(t) = -2u_1'(t)\sin 2t - 4u_1(t)\cos 2t + 2u_2'(t)\cos 2t - 4u_2(t)\sin 2t$$

Example: Two Equations (3 of 6)

- Recall that our differential equation is

$$y'' + 4y = 3\csc t$$

- Substituting y'' and y into this equation, we obtain

$$\begin{aligned} &-2u_1'(t)\sin 2t - 4u_1(t)\cos 2t + 2u_2'(t)\cos 2t - 4u_2(t)\sin 2t \\ &\quad + 4\big(u_1(t)\cos 2t + u_2(t)\sin 2t\big) = 3\csc t \end{aligned}$$

- This equation simplifies to

$$-2u_1'(t)\sin 2t + 2u_2'(t)\cos 2t = 3\csc t$$

- Thus, to solve for u_1 and u_2, we have the two equations:

$$\begin{aligned} -2u_1'(t)\sin 2t + 2u_2'(t)\cos 2t &= 3\csc t \\ u_1'(t)\cos 2t + u_2'(t)\sin 2t &= 0 \end{aligned}$$

Example: Solve for u_1' (4 of 6)

- To find u_1 and u_2 , we need to solve the equations

$$-2u_1'(t)\sin 2t + 2u_2'(t)\cos 2t = 3\csc t$$
$$u_1'(t)\cos 2t + u_2'(t)\sin 2t = 0$$

- From second equation,

$$u_2'(t) = -u_1'(t)\frac{\cos 2t}{\sin 2t}$$

- Substituting this into the first equation,

$$-2u_1'(t)\sin 2t + 2\left[-u_1'(t)\frac{\cos 2t}{\sin 2t}\right]\cos 2t = 3\csc t$$
$$-2u_1'(t)\sin^2(2t) - 2u_1'(t)\cos^2(2t) = 3\csc t\sin 2t$$
$$-2u_1'(t)\left[\sin^2(2t) + \cos^2(2t)\right] = 3\left[\frac{2\sin t\cos t}{\sin t}\right]$$
$$u_1'(t) = -3\cos t$$

Example : Solve for u_1 and u_2 (5 of 6)

- From the previous slide,

$$u_1'(t) = -3\cos t, \quad u_2'(t) = -u_1'(t)\frac{\cos 2t}{\sin 2t}$$

- Then

$$u_2'(t) = 3\cos t\left[\frac{\cos 2t}{\sin 2t}\right] = 3\cos t\left[\frac{1-2\sin^2 t}{2\sin t\cos t}\right] = 3\left[\frac{1-2\sin^2 t}{2\sin t}\right]$$

$$= 3\left[\frac{1}{2\sin t} - \frac{2\sin^2 t}{2\sin t}\right] = \frac{3}{2}\csc t - 3\sin t$$

- Thus

$$u_1(t) = \int u_1'(t)dt = \int -3\cos t dt = -3\sin t + c_1$$

$$u_2(t) = \int u_2'(t)dt = \int\left(\frac{3}{2}\csc t - 3\sin t\right)dt = \frac{3}{2}\ln\left|\csc t - \cot t\right| + 3\cos t + c_2$$

Example: General Solution (6 of 6)

- Recall our equation and homogeneous solution y_C:

$$y'' + 4y = 3\csc t, \quad y_C(t) = c_1 \cos 2t + c_2 \sin 2t$$

- Using the expressions for u_1 and u_2 on the previous slide, the general solution to the differential equation is

$$\begin{aligned}
y(t) &= u_1(t)\cos 2t + u_2(t)\sin 2t + y_C(t) \\
&= -3\sin t\cos 2t + \frac{3}{2}\ln\left|\csc t - \cot t\right|\sin 2t + 3\cos t\sin 2t + y_C(t) \\
&= 3\left[\cos t\sin 2t - \sin t\cos 2t\right] + \frac{3}{2}\ln\left|\csc t - \cot t\right|\sin 2t + y_C(t) \\
&= 3\left[2\sin t\cos^2 t - \sin t\left(2\cos^2 t - 1\right)\right] + \frac{3}{2}\ln\left|\csc t - \cot t\right|\sin 2t + y_C(t) \\
&= 3\sin t + \frac{3}{2}\ln\left|\csc t - \cot t\right|\sin 2t + c_1\cos 2t + c_2\sin 2t
\end{aligned}$$

$$y'' + p(t)y' + q(t)y = g(t)$$
$$y(t) = u_1(t)y_1(t) + u_2(t)y_2(t)$$

Summary

- Suppose y_1, y_2 are fundamental solutions to the homogeneous equation associated with the nonhomogeneous equation above, where we note that the coefficient on y'' is 1.
- To find u_1 and u_2, we need to solve the equations
$$u_1'(t)y_1(t) + u_2'(t)y_2(t) = 0$$
$$u_1'(t)y_1'(t) + u_2'(t)y_2'(t) = g(t)$$
- Doing so, and using the Wronskian, we obtain
$$u_1'(t) = -\frac{y_2(t)g(t)}{W(y_1, y_2)(t)}, \quad u_2'(t) = \frac{y_1(t)g(t)}{W(y_1, y_2)(t)}$$
- Thus
$$u_1(t) = -\int \frac{y_2(t)g(t)}{W(y_1, y_2)(t)}dt + c_1, \quad u_2(t) = \int \frac{y_1(t)g(t)}{W(y_1, y_2)(t)}dt + c_2$$

Theorem 3.7.1

- Consider the equations

$$y'' + p(t)y' + q(t)y = g(t) \qquad (1)$$

$$y'' + p(t)y' + q(t)y = 0 \qquad (2)$$

- If the functions *p*, *q* and *g* are continuous on an open interval *I*, and if y_1 and y_2 are fundamental solutions to Eq. (2), then a particular solution of Eq. (1) is

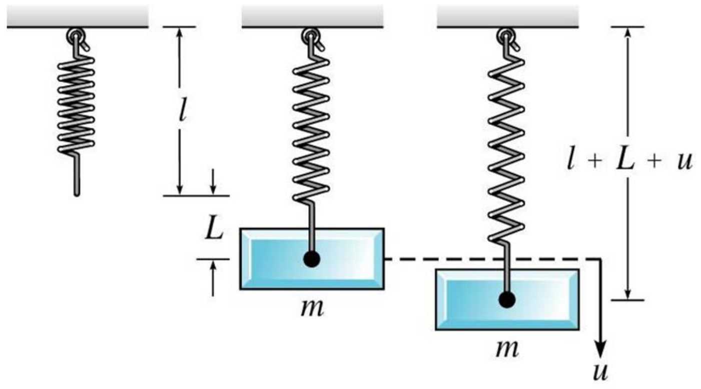
$$Y(t) = -y_1(t)\int \frac{y_2(t)g(t)}{W(y_1, y_2)(t)}\,dt + y_2(t)\int \frac{y_1(t)g(t)}{W(y_1, y_2)(t)}\,dt$$

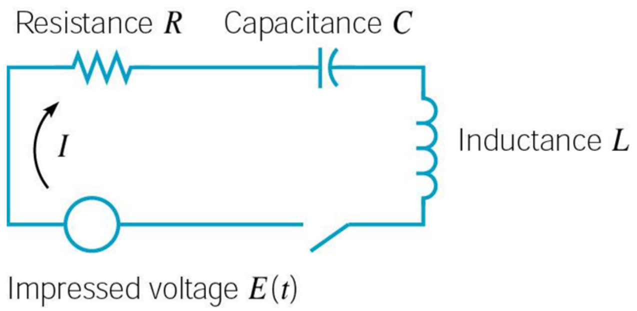
and the general solution is

$$y(t) = c_1 y_1(t) + c_2 y_2(t) + Y(t)$$

Ch 3.8: Mechanical & Electrical Vibrations

- Two important areas of application for second order linear equations with constant coefficients are in modeling mechanical and electrical oscillations.
- We will study the motion of a mass on a spring in detail.
- An understanding of the behavior of this simple system is the first step in investigation of more complex vibrating systems.



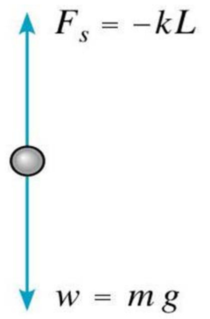


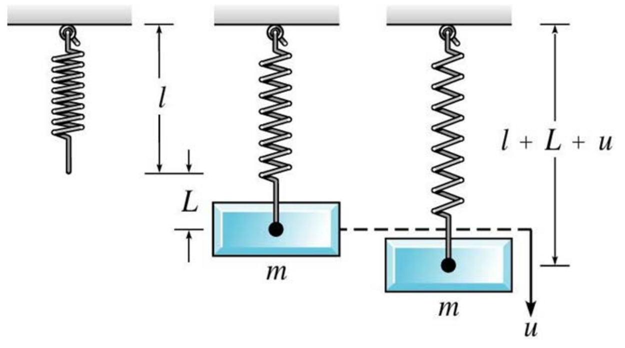
Spring – Mass System

- Suppose a mass *m* hangs from vertical spring of original length *l*. The mass causes an elongation *L* of the spring.
- The force F_G of gravity pulls mass down. This force has magnitude *mg*, where *g* is acceleration due to gravity.
- The force F_S of spring stiffness pulls mass up. For small elongations *L*, this force is proportional to *L*.

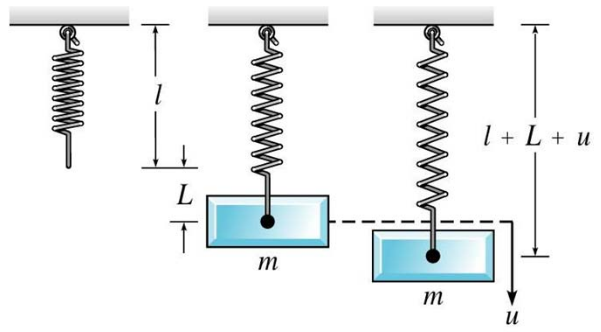
  That is, $F_s = kL$ (Hooke's Law).
- Since mass is in equilibrium, the forces balance each other:

$$mg = kL$$





Spring Model

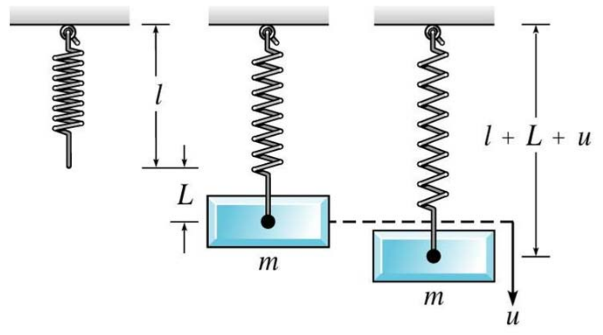


- We will study motion of mass when it is acted on by an external force (forcing function) or is initially displaced.
- Let $u(t)$ denote the displacement of the mass from its equilibrium position at time t, measured downward.
- Let f be the net force acting on mass. Newton's 2nd Law:

$$mu''(t) = f(t)$$

- In determining f, there are four separate forces to consider:
  - Weight: $w = mg$ (downward force)
  - Spring force: $F_s = -k(L + u)$ (up or down force, see next slide)
  - Damping force: $F_d(t) = -\gamma u'(t)$ (up or down, see following slide)
  - External force: $F(t)$ (up or down force, see text)

Spring Model: Spring Force Details



- The spring force F_s acts to restore spring to natural position, and is proportional to $L + u$. If $L + u > 0$, then spring is extended and the spring force acts upward. In this case

$$F_s = -k(L+u)$$

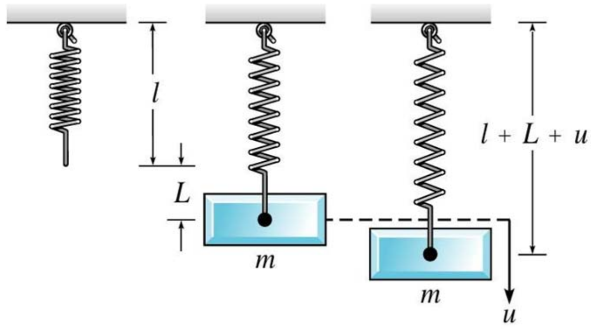
- If $L + u < 0$, then spring is compressed a distance of $|L + u|$, and the spring force acts downward. In this case

$$F_s = k|L+u| = k[-(L+u)] = -k(L+u)$$

- In either case,

$$F_s = -k(L+u)$$

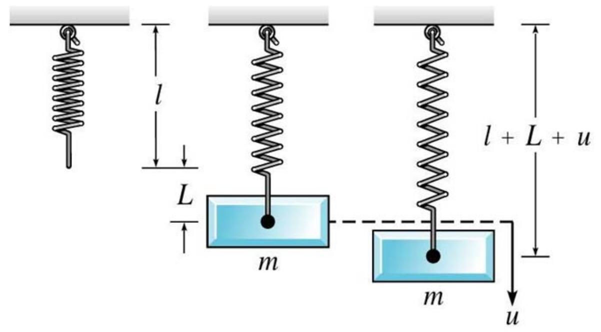
Spring Model: Damping Force Details



- The damping or resistive force F_d acts in opposite direction as motion of mass. Can be complicated to model.
- F_d may be due to air resistance, internal energy dissipation due to action of spring, friction between mass and guides, or a mechanical device (dashpot) imparting resistive force to mass.
- We keep it simple and assume F_d is proportional to velocity.
- In particular, we find that
  - If $u' > 0$, then u is increasing, so mass is moving downward. Thus F_d acts upward and hence $F_d = -\gamma u'$, where $\gamma > 0$.
  - If $u' < 0$, then u is decreasing, so mass is moving upward. Thus F_d acts downward and hence $F_d = -\gamma u'$, $\gamma > 0$.
- In either case,

$$F_d(t) = -\gamma u'(t), \quad \gamma > 0$$

Spring Model: Differential Equation



- Taking into account these forces, Newton's Law becomes:

$$\begin{aligned} mu''(t) &= mg + F_s(t) + F_d(t) + F(t) \\ &= mg - k\left[L + u(t)\right] - \gamma u'(t) + F(t) \end{aligned}$$

- Recalling that *mg* = *kL*, this equation reduces to

$$mu''(t) + \gamma u'(t) + ku(t) = F(t)$$

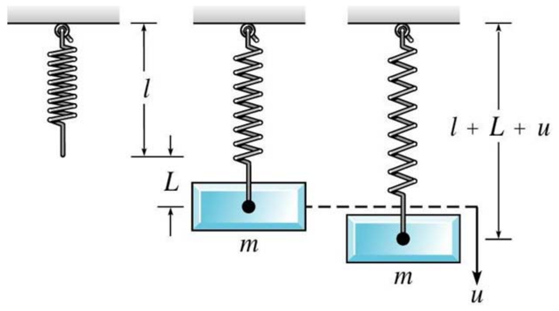
where the constants *m*, γ, and *k* are positive.

- We can prescribe initial conditions also:

$$u(0) = u_0, \; u'(0) = v_0$$

- It follows from Theorem 3.2.1 that there is a unique solution to this initial value problem. Physically, if mass is set in motion with a given initial displacement and velocity, then its position is uniquely determined at all future times.

Example 1: Find Coefficients (1 of 2)



- A 4 lb mass stretches a spring 2". The mass is displaced an additional 6" and then released; and is in a medium that exerts a viscous resistance of 6 lb when velocity of mass is 3 ft/sec. Formulate the IVP that governs motion of this mass:

$$mu''(t) + \gamma u'(t) + ku(t) = F(t),\ \ u(0) = u_0,\ \ u'(0) = v_0$$

- Find *m*:

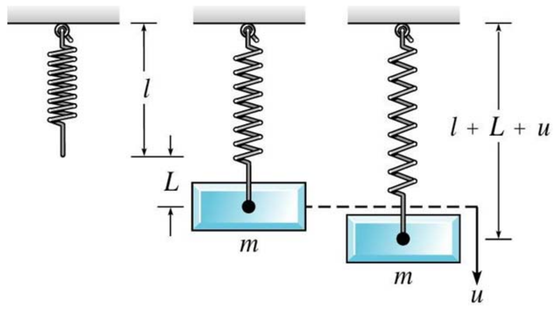
$$w = mg \Rightarrow m = \frac{w}{g} \Rightarrow m = \frac{4\,\text{lb}}{32\text{ft}/\sec^2} \Rightarrow m = \frac{1}{8}\frac{\text{lb}\sec^2}{\text{ft}}$$

- Find γ:

$$\gamma u' = 6\,\text{lb} \Rightarrow \gamma = \frac{6\,\text{lb}}{3\text{ft}/\sec} \Rightarrow \gamma = 2\frac{\text{lb}\sec}{\text{ft}}$$

- Find *k*:

$$F_s = -k\,L \Rightarrow k = \frac{4\,\text{lb}}{2\,\text{in}} \Rightarrow \ \ k = \frac{4\,\text{lb}}{1/6\,\text{ft}} \Rightarrow k = 24\frac{\text{lb}}{\text{ft}}$$



Example 1: Find IVP (2 of 2)

- Thus our differential equation becomes

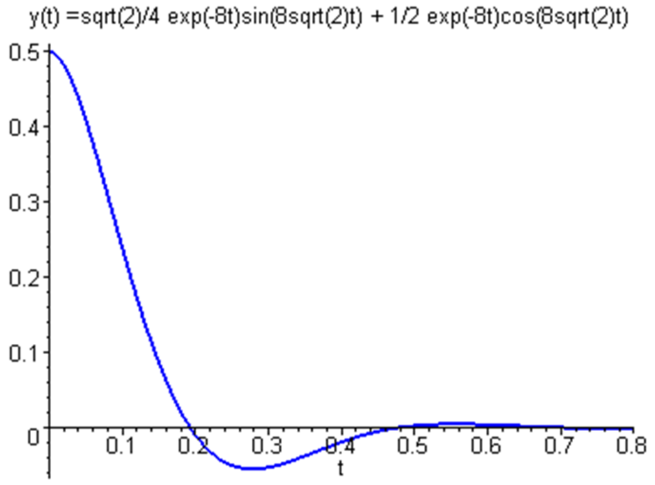
$$\frac{1}{8}u''(t) + 2u'(t) + 24u(t) = 0$$

and hence the initial value problem can be written as

$$u''(t) + 16u'(t) + 192u(t) = 0$$

$$u(0) = \frac{1}{2}, \quad u'(0) = 0$$

- This problem can be solved using methods of Chapter 3.4. Given on right is the graph of solution.



Spring Model: Undamped Free Vibrations (1 of 4)

- Recall our differential equation for spring motion:

$$mu''(t) + \gamma u'(t) + ku(t) = F(t)$$

- Suppose there is no external driving force and no damping. Then $F(t) = 0$ and $\gamma = 0$, and our equation becomes

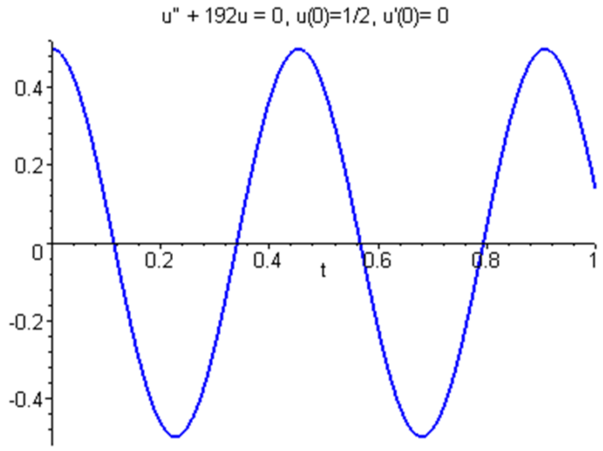
$$mu''(t) + ku(t) = 0$$

- The general solution to this equation is

$$u(t) = A\cos\omega_0 t + B\sin\omega_0 t,$$

where

$$\omega_0^2 = k/m$$



Spring Model: Undamped Free Vibrations (2 of 4)

- Using trigonometric identities, the solution

$$u(t) = A\cos\omega_0 t + B\sin\omega_0 t, \quad \omega_0^2 = k/m$$

can be rewritten as follows:

$$u(t) = A\cos\omega_0 t + B\sin\omega_0 t \quad \Leftrightarrow \quad u(t) = R\cos(\omega_0 t - \delta)$$

$$\Leftrightarrow u(t) = R\cos\delta\cos\omega_0 t + R\sin\delta\sin\omega_0 t,$$

where

$$A = R\cos\delta, \;\; B = R\sin\delta \;\; \Rightarrow \;\; R = \sqrt{A^2 + B^2}, \quad \tan\delta = \frac{B}{A}$$

- Note that in finding δ, we must be careful to choose correct quadrant. This is done using the signs of $\cos\delta$ and $\sin\delta$.

Spring Model: Undamped Free Vibrations (3 of 4)

- Thus our solution is

$$u(t) = A\cos\omega_0 t + B\sin\omega_0 t \quad = R\cos(\omega_0 t - \delta)$$

where

$$\omega_0 = \sqrt{k/m}$$

- The solution is a shifted cosine (or sine) curve, that describes simple harmonic motion, with period

$$T = \frac{2\pi}{\omega_0} = 2\pi\sqrt{\frac{m}{k}}$$

- The circular frequency ω_0 (radians/time) is **natural frequency** of the vibration, R is the **amplitude** of max displacement of mass from equilibrium, and δ is the **phase** (dimensionless).

Spring Model: Undamped Free Vibrations (4 of 4)

- Note that our solution

$$u(t) = A\cos\omega_0 t + B\sin\omega_0 t = R\cos(\omega_0 t - \delta), \quad \omega_0 = \sqrt{k/m}$$

  is a shifted cosine (or sine) curve with period

$$T = 2\pi\sqrt{\frac{m}{k}}$$

- Initial conditions determine *A* & *B*, hence also the amplitude *R*.
- The system always vibrates with same frequency ω_0, regardless of initial conditions.
- The period *T* increases as *m* increases, so larger masses vibrate more slowly. However, *T* decreases as *k* increases, so stiffer springs cause system to vibrate more rapidly.

Example 2: Find IVP (1 of 3)

- A 10 lb mass stretches a spring 2". The mass is displaced an additional 2" and then set in motion with initial upward velocity of 1 ft/sec. Determine position of mass at any later time. Also find period, amplitude, and phase of the motion.

$$mu''(t) + ku(t) = 0,\ \ u(0) = u_0,\ \ u'(0) = v_0$$

- Find *m*:

$$w = mg \Rightarrow m = \frac{w}{g} \Rightarrow m = \frac{10\,\text{lb}}{32\text{ft}/\sec^2} \Rightarrow m = \frac{5}{16}\,\frac{\text{lb}\sec^2}{\text{ft}}$$

- Find *k*:

$$F_s = -k\,L \Rightarrow k = \frac{10\,\text{lb}}{2\,\text{in}} \Rightarrow\ k = \frac{10\,\text{lb}}{1/6\,\text{ft}} \Rightarrow k = 60\frac{\text{lb}}{\text{ft}}$$

- Thus our IVP is

$$5/16\,u''(t) + 60u(t) = 0,\ \ \ u(0) = 1/6,\ \ u'(t) = -1$$

Example 2: Find Solution (2 of 3)

- Simplifying, we obtain

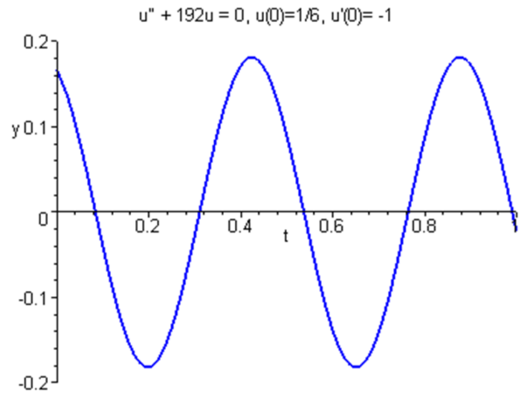
$$u''(t) + 192u(t) = 0, \quad u(0) = 1/6, \quad u'(0) = -1$$

- To solve, use methods of Ch 3.4 to obtain

$$u(t) = \frac{1}{6}\cos\sqrt{192}t - \frac{1}{\sqrt{192}}\sin\sqrt{192}t$$

or

$$u(t) = \frac{1}{6}\cos 8\sqrt{3}t - \frac{1}{8\sqrt{3}}\sin 8\sqrt{3}t$$



$$u(t) = \frac{1}{6}\cos 8\sqrt{3}t - \frac{1}{8\sqrt{3}}\sin 8\sqrt{3}t$$

Example 2:
Find Period, Amplitude, Phase (3 of 3)

- The natural frequency is

$$\omega_0 = \sqrt{k/m} = \sqrt{192} = 8\sqrt{3} \cong 13.856 \text{ rad/sec}$$

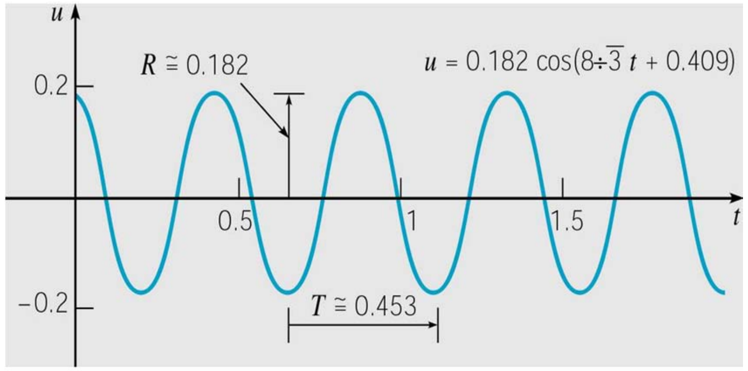
- The period is

$$T = 2\pi/\omega_0 \cong 0.45345 \text{ sec}$$

- The amplitude is

$$R = \sqrt{A^2 + B^2} \cong 0.18162 \text{ ft}$$

- Next, determine the phase δ:



$$A = R\cos\delta, \quad B = R\sin\delta, \quad \tan\delta = B/A$$

$$\tan\delta = \frac{B}{A} \Rightarrow \tan\delta = \frac{-\sqrt{3}}{4} \Rightarrow \delta = \tan^{-1}\left(\frac{-\sqrt{3}}{4}\right) \cong -0.40864 \text{ rad}$$

Thus $u(t) = 0.182\cos\left(8\sqrt{3}t + 0.409\right)$

Spring Model: Damped Free Vibrations (1 of 8)

- Suppose there is damping but no external driving force $F(t)$:

$$mu''(t) + \gamma u'(t) + ku(t) = 0$$

- What is effect of damping coefficient γ on system?
- The characteristic equation is

$$r_1, r_2 = \frac{-\gamma \pm \sqrt{\gamma^2 - 4mk}}{2m} = \frac{\gamma}{2m}\left[-1 \pm \sqrt{1 - \frac{4mk}{\gamma^2}}\right]$$

- Three cases for the solution:

$$\gamma^2 - 4mk > 0: \quad u(t) = Ae^{r_1 t} + Be^{r_2 t}, \ \text{ where } r_1 < 0,\ r_2 < 0;$$

$$\gamma^2 - 4mk = 0: \quad u(t) = (A + Bt)e^{-\gamma t/2m}, \ \text{ where } \gamma/2m > 0;$$

$$\gamma^2 - 4mk < 0: \quad u(t) = e^{-\gamma t/2m}(A\cos\mu t + B\sin\mu t), \ \ \mu = \frac{\sqrt{4mk - \gamma^2}}{2m} > 0.$$

Note: In all three cases, $\lim_{t\to\infty} u(t) = 0$, as expected from damping term.

Damped Free Vibrations: Small Damping (2 of 8)

- Of the cases for solution form, the last is most important, which occurs when the damping is small:

$$\gamma^2 - 4mk > 0: \;\; u(t) = Ae^{r_1 t} + Be^{r_2 t}, \quad r_1 < 0,\; r_2 < 0$$

$$\gamma^2 - 4mk = 0: \;\; u(t) = (A + Bt)e^{-\gamma t/2m}, \quad \gamma/2m > 0$$

$$\gamma^2 - 4mk < 0: \;\; u(t) = e^{-\gamma t/2m}(A\cos\mu t + B\sin\mu t), \;\; \mu > 0$$

- We examine this last case. Recall

$$A = R\cos\delta, \;\; B = R\sin\delta$$

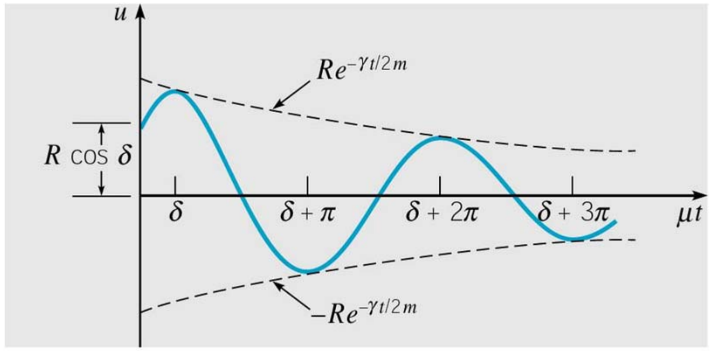
- Then

$$u(t) = Re^{-\gamma t/2m}\cos(\mu t - \delta)$$

and hence

$$|u(t)| \le Re^{-\gamma t/2m}$$

(damped oscillation)



Damped Free Vibrations: Quasi Frequency (3 of 8)

- Thus we have damped oscillations:

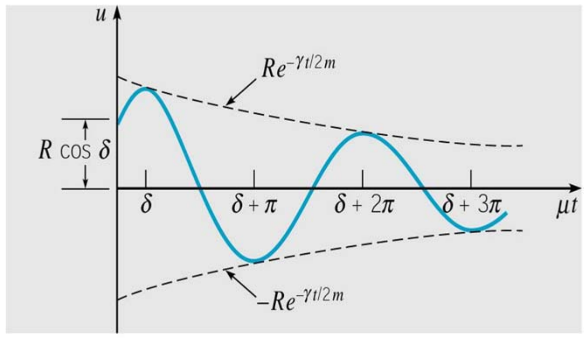
$$u(t) = Re^{-\gamma t/2m}\cos(\mu t - \delta) \quad \Rightarrow \quad |u(t)| \le Re^{-\gamma t/2m}$$

- Amplitude *R* depends on the initial conditions, since

$$u(t) = e^{-\gamma t/2m}\left(A\cos\mu t + B\sin\mu t\right), \; A = R\cos\delta, \; B = R\sin\delta$$

- Although the motion is not periodic, the parameter μ determines mass oscillation frequency.
- Thus μ is called the **quasi frequency**.
- Recall

$$\mu = \frac{\sqrt{4mk - \gamma^2}}{2m}$$



Damped Free Vibrations: Quasi Period (4 of 8)

- Compare μ with ω_0, the frequency of undamped motion:

$$\frac{\mu}{\omega_0} = \frac{\sqrt{4km-\gamma^2}}{2m\sqrt{k/m}} = \frac{\sqrt{4km-\gamma^2}}{\sqrt{4m^2}\sqrt{k/m}} = \frac{\sqrt{4km-\gamma^2}}{\sqrt{4km}} = \sqrt{1-\frac{\gamma^2}{4km}}$$

For small γ $\rightarrow$

$$\cong \sqrt{1-\frac{\gamma^2}{4km}+\frac{\gamma^4}{64k^2m^2}} = \sqrt{\left(1-\frac{\gamma^2}{8km}\right)^2} = 1-\frac{\gamma^2}{8km}$$

- Thus, small damping reduces oscillation frequency slightly.
- Similarly, **quasi period** is defined as $T_d = 2\pi/\mu$. Then

$$\frac{T_d}{T} = \frac{2\pi/\mu}{2\pi/\omega_0} = \frac{\omega_0}{\mu} = \left(1-\frac{\gamma^2}{4km}\right)^{-1/2} \cong \left(1-\frac{\gamma^2}{8km}\right)^{-1} \cong 1+\frac{\gamma^2}{8km}$$

- Thus, small damping increases quasi period.

Damped Free Vibrations: Neglecting Damping for Small $\gamma^2/4km$ (5 of 8)

- Consider again the comparisons between damped and undamped frequency and period:

$$\frac{\mu}{\omega_0} = \left(1 - \frac{\gamma^2}{4km}\right)^{1/2}, \quad \frac{T_d}{T} = \left(1 - \frac{\gamma^2}{4km}\right)^{-1/2}$$

- Thus it turns out that a small γ is not as telling as a small ratio $\gamma^2/4km$.
- For small $\gamma^2/4km$, we can neglect effect of damping when calculating quasi frequency and quasi period of motion. But if we want a detailed description of motion of mass, then we cannot neglect damping force, no matter how small.

Damped Free Vibrations: Frequency, Period (6 of 8)

- Ratios of damped and undamped frequency, period:

$$\frac{\mu}{\omega_0} = \left(1 - \frac{\gamma^2}{4km}\right)^{1/2}, \quad \frac{T_d}{T} = \left(1 - \frac{\gamma^2}{4km}\right)^{-1/2}$$

- Thus

$$\lim_{\gamma \to 2\sqrt{km}} \mu = 0 \text{ and } \lim_{\gamma \to 2\sqrt{km}} T_d = \infty$$

- The importance of the relationship between γ^2 and $4km$ is supported by our previous equations:

$$\gamma^2 - 4mk > 0: \; u(t) = Ae^{r_1 t} + Be^{r_2 t}, \quad r_1 < 0, \, r_2 < 0$$

$$\gamma^2 - 4mk = 0: \; u(t) = (A + Bt)e^{-\gamma t/2m}, \quad \gamma / 2m > 0$$

$$\gamma^2 - 4mk < 0: \; u(t) = e^{-\gamma t/2m}(A\cos \mu t + B\sin \mu t), \quad \mu > 0$$

Damped Free Vibrations: Critical Damping Value (7 of 8)

- Thus the nature of the solution changes as γ passes through the value $2\sqrt{km}$.
- This value of γ is known as the **critical damping** value, and for larger values of γ the motion is said to be **overdamped**.
- Thus for the solutions given by these cases,

$$\gamma^2 - 4mk > 0: \quad u(t) = Ae^{r_1 t} + Be^{r_2 t}, \quad r_1 < 0,\ r_2 < 0 \qquad (1)$$

$$\gamma^2 - 4mk = 0: \quad u(t) = (A + Bt)e^{-\gamma t/2m}, \quad \gamma/2m > 0 \qquad (2)$$

$$\gamma^2 - 4mk < 0: \quad u(t) = e^{-\gamma t/2m}(A\cos\mu t + B\sin\mu t), \quad \mu > 0 \qquad (3)$$

  we see that the mass creeps back to its equilibrium position for solutions (1) and (2), but does not oscillate about it, as for small γ in solution (3).
- Soln (1) is overdamped and soln (2) is critically damped.

Damped Free Vibrations: Characterization of Vibration (8 of 8)

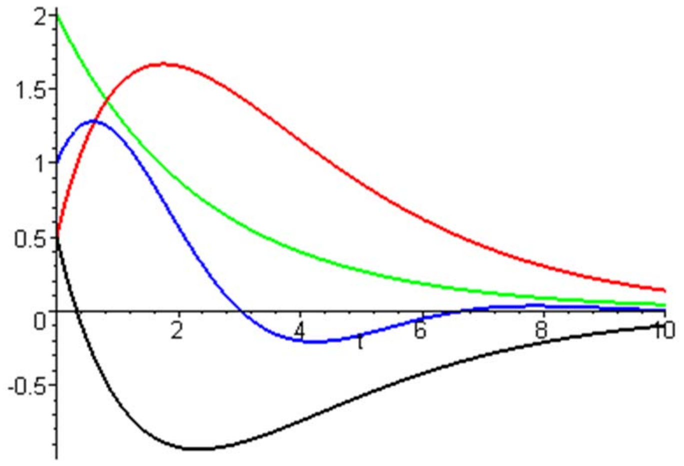
- Mass creeps back to equilibrium position for solns (1) & (2), but does not oscillate about it, as for small γ in solution (3).

$$\gamma^2 - 4mk > 0: \;\; u(t) = Ae^{r_1 t} + Be^{r_2 t}, \;\; r_1 < 0,\, r_2 < 0 \quad \text{(Green)} \qquad (1)$$

$$\gamma^2 - 4mk = 0: \;\; u(t) = (A + Bt)e^{-\gamma t/2m}, \;\; \gamma/2m > 0 \quad \text{(Red, Black)} \qquad (2)$$

$$\gamma^2 - 4mk < 0: \;\; u(t) = e^{-\gamma t/2m}(A\cos\mu t + B\sin\mu t) \quad \text{(Blue)} \qquad (3)$$

- Soln (1) is overdamped and soln (2) is critically damped.



Example 3: Initial Value Problem (1 of 4)

- Suppose that the motion of a spring-mass system is governed by the initial value problem

$$u'' + 0.125u' + u = 0, \quad u(0) = 2, \; u'(0) = 0$$

- Find the following:
  - (a) quasi frequency and quasi period;
  - (b) time at which mass passes through equilibrium position;
  - (c) time τ such that $|u(t)| < 0.1$ for all $t > \tau$.
- For Part (a), using methods of this chapter we obtain:

$$u(t) = e^{-t/16}\left(2\cos\frac{\sqrt{255}}{16}t + \frac{2}{\sqrt{255}}\sin\frac{\sqrt{255}}{16}t\right) = \frac{32}{\sqrt{255}}e^{-t/16}\cos\left(\frac{\sqrt{255}}{16}t - \delta\right)$$

where

$$\tan\delta = \frac{1}{\sqrt{255}} \Rightarrow \delta \cong 0.06254 \quad (\text{recall } A = R\cos\delta, \; B = R\sin\delta)$$

Example 3: Quasi Frequency & Period (2 of 4)

- The solution to the initial value problem is:

$$u(t) = e^{-t/16}\left(2\cos\frac{\sqrt{255}}{16}t + \frac{2}{\sqrt{255}}\sin\frac{\sqrt{255}}{16}t\right) = \frac{32}{\sqrt{255}}e^{-t/16}\cos\left(\frac{\sqrt{255}}{16}t - \delta\right)$$

- The graph of this solution, along with solution to the corresponding undamped problem, is given below.
- The quasi frequency is

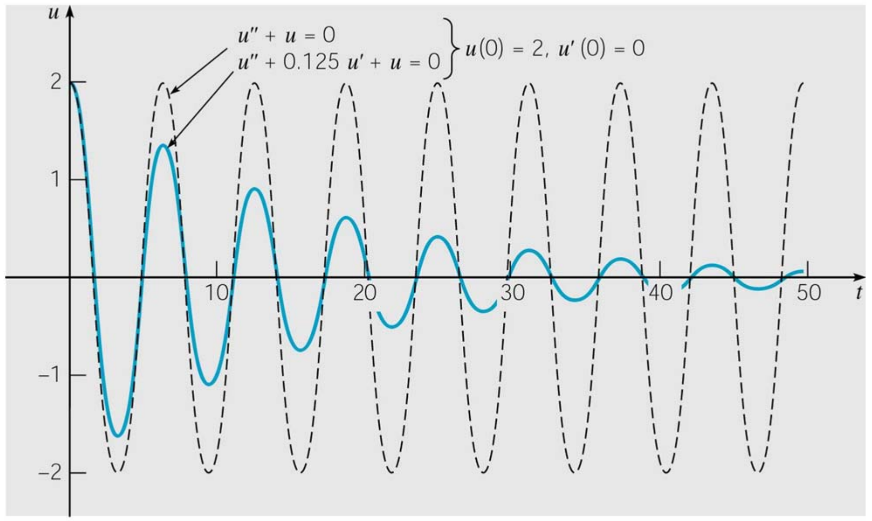
$$\mu = \sqrt{255}/16 \cong 0.998$$

and quasi period

$$T_d = 2\pi/\mu \cong 6.295$$

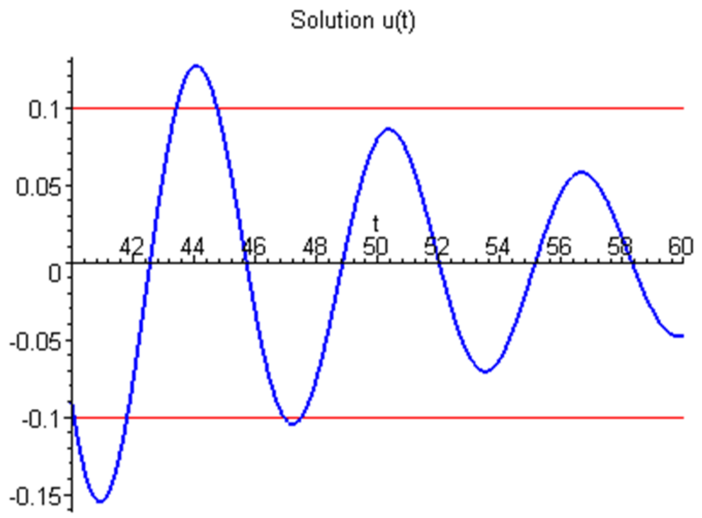
- For undamped case:

$$\omega_0 = 1,\ T = 2\pi \cong 6.283$$



Example 3: Quasi Frequency & Period (3 of 4)

- The damping coefficient is $\gamma = 0.125 = 1/8$, and this is 1/16 of the critical value $2\sqrt{km} = 2$
- Thus damping is small relative to mass and spring stiffness. Nevertheless the oscillation amplitude diminishes quickly.
- Using a solver, we find that $|u(t)| < 0.1$ for $t > \tau \approx 47.515$ sec



Example 3: Quasi Frequency & Period (4 of 4)

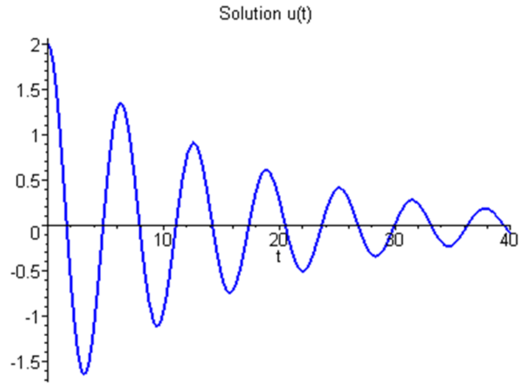
- To find the time at which the mass first passes through the equilibrium position, we must solve

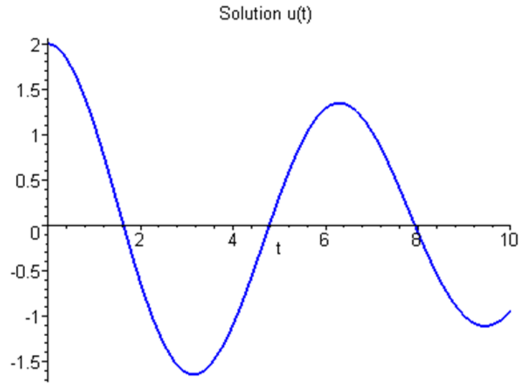
$$u(t) = \frac{32}{\sqrt{255}} e^{-t/16} \cos\left(\frac{\sqrt{255}}{16} t - \delta \right) = 0$$

- Or more simply, solve

$$\frac{\sqrt{255}}{16} t - \delta = \frac{\pi}{2}$$

$$\Rightarrow t = \frac{16}{\sqrt{255}} \left(\frac{\pi}{2} + \delta \right) \cong 1.637 \text{ sec}$$





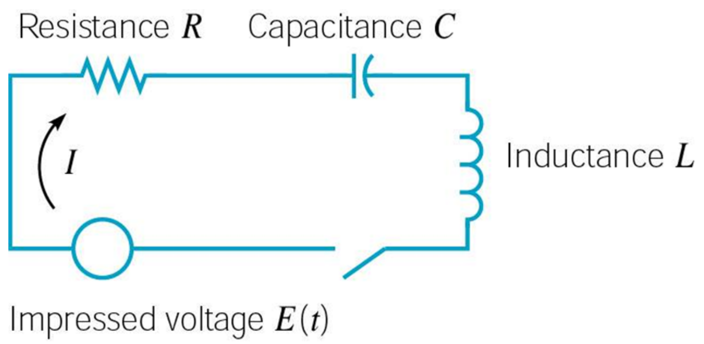
Electric Circuits

- The flow of current in certain basic electrical circuits is modeled by second order linear ODEs with constant coefficients:

$$L\,I''(t) + R\,I'(t) + \frac{1}{C}I(t) = E'(t)$$

$$I(0) = I_0, \;\; I'(0) = I'_0$$

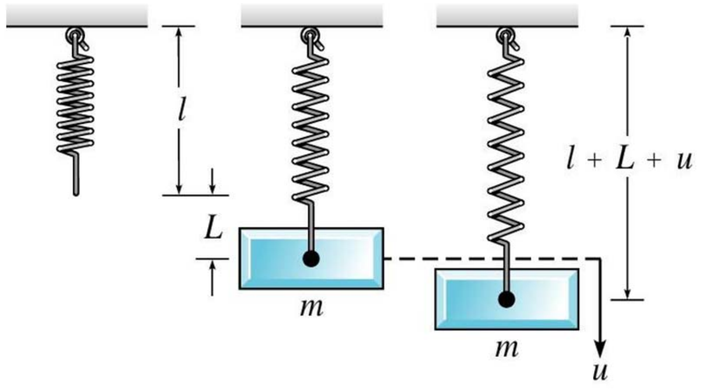
- It is interesting that the flow of current in this circuit is mathematically equivalent to motion of spring-mass system.
- For more details, see text.



Ch 3.9: Forced Vibrations

- We continue the discussion of the last section, and now consider the presence of a periodic external force:

$$mu''(t) + \gamma u'(t) + ku(t) = F_0 \cos \omega t$$



Forced Vibrations with Damping

- Consider the equation below for damped motion and external forcing funcion $F_0\cos\omega t$.

$$mu''(t) + \gamma u'(t) + ku(t) = F_0 \cos \omega t$$

- The general solution of this equation has the form

$$u(t) = c_1u_1(t) + c_2u_2(t) + A\cos(\omega t) + B\sin(\omega t) = u_C(t) + U(t)$$

where the general solution of the homogeneous equation is

$$u_C(t) = c_1u_1(t) + c_2u_2(t)$$

and the particular solution of the nonhomogeneous equation is

$$U(t) = A\cos(\omega t) + B\sin(\omega t)$$

Homogeneous Solution

- The homogeneous solutions u_1 and u_2 depend on the roots r_1 and r_2 of the characteristic equation:

$$mr^2 + \gamma r + kr = 0 \quad \Rightarrow \quad r = \frac{-\gamma \pm \sqrt{\gamma^2 - 4mk}}{2m}$$

- Since m, γ, and k are are all positive constants, it follows that r_1 and r_2 are either real and negative, or complex conjugates with negative real part. In the first case,

$$\lim_{t\to\infty} u_C(t) = \lim_{t\to\infty}\left(c_1 e^{r_1 t} + c_2 e^{r_2 t}\right) = 0,$$

while in the second case

$$\lim_{t\to\infty} u_C(t) = \lim_{t\to\infty}\left(c_1 e^{\lambda t}\cos\mu t + c_2 e^{\lambda t}\sin\mu t\right) = 0.$$

- Thus in either case,

$$\lim_{t\to\infty} u_C(t) = 0$$

Transient and Steady-State Solutions

- Thus for the following equation and its general solution,

$$mu''(t) + \gamma u'(t) + ku(t) = F_0 \cos \omega t$$

$$u(t) = \underbrace{c_1 u_1(t) + c_2 u_2(t)}_{u_C(t)} + \underbrace{A\cos(\omega t) + B\sin(\omega t)}_{U(t)},$$

  we have

$$\lim_{t\to\infty} u_C(t) = \lim_{t\to\infty}\left(c_1 u_1(t) + c_2 u_2(t)\right) = 0$$

- Thus $u_C(t)$ is called the **transient solution**. Note however that

$$U(t) = A\cos(\omega t) + B\sin(\omega t)$$

  is a steady oscillation with same frequency as forcing function.
- For this reason, $U(t)$ is called the **steady-state solution**, or **forced response**.

Transient Solution and Initial Conditions

- For the following equation and its general solution,

$$mu''(t) + \gamma u'(t) + ku(t) = F_0 \cos \omega t$$

$$u(t) = \underbrace{c_1 u_1(t) + c_2 u_2(t)}_{u_C(t)} + \underbrace{A\cos(\omega t) + B\sin(\omega t)}_{U(t)}$$

  the transient solution $u_C(t)$ enables us to satisfy whatever initial conditions might be imposed.

- With increasing time, the energy put into system by initial displacement and velocity is dissipated through damping force. The motion then becomes the response $U(t)$ of the system to the external force $F_0 \cos \omega t$.
- Without damping, the effect of the initial conditions would persist for all time.

Rewriting Forced Response

- Using trigonometric identities, it can be shown that

$$U(t) = A\cos(\omega t) + B\sin(\omega t)$$

can be rewritten as

$$U(t) = R\cos(\omega t - \delta)$$

- It can also be shown that

$$R = \frac{F_0}{\sqrt{m^2(\omega_0^2 - \omega^2)^2 + \gamma^2\omega^2}},$$

$$\cos\delta = \frac{m(\omega_0^2 - \omega^2)}{\sqrt{m^2(\omega_0^2 - \omega^2)^2 + \gamma^2\omega^2}}, \ \sin\delta = \frac{\gamma\,\omega}{\sqrt{m^2(\omega_0^2 - \omega^2)^2 + \gamma^2\omega^2}}$$

where $\omega_0^2 = k / m$

Amplitude Analysis of Forced Response

- The amplitude R of the steady state solution

$$R = \frac{F_0}{\sqrt{m^2(\omega_0^2 - \omega^2)^2 + \gamma^2\omega^2}},$$

  depends on the driving frequency ω. For low-frequency excitation we have

$$\lim_{\omega\to 0} R = \lim_{\omega\to 0} \frac{F_0}{\sqrt{m^2(\omega_0^2 - \omega^2)^2 + \gamma^2\omega^2}} = \frac{F_0}{m\omega_0^2} = \frac{F_0}{k}$$

  where we recall $(\omega_0)^2 = k/m$. Note that F_0/k is the static displacement of the spring produced by force F_0.

- For high frequency excitation,

$$\lim_{\omega\to \infty} R = \lim_{\omega\to \infty} \frac{F_0}{\sqrt{m^2(\omega_0^2 - \omega^2)^2 + \gamma^2\omega^2}} = 0$$

Maximum Amplitude of Forced Response

- Thus

$$\lim_{\omega \to 0} R = F_0 / k, \quad \lim_{\omega \to \infty} R = 0$$

- At an intermediate value of ω, the amplitude R may have a maximum value. To find this frequency ω, differentiate R and set the result equal to zero. Solving for $\omega_{\max}$, we obtain

$$\omega_{\max}^2 = \omega_0^2 - \frac{\gamma^2}{2m^2} = \omega_0^2 \left(1 - \frac{\gamma^2}{2mk} \right)$$

where $(\omega_0)^2 = k/m$. Note $\omega_{\max} < \omega_0$, and $\omega_{\max}$ is close to ω_0 for small γ. The maximum value of R is

$$R_{\max} = \frac{F_0}{\gamma \omega_0 \sqrt{1 - (\gamma^2 / 4mk)}}$$

Maximum Amplitude for Imaginary $\omega_{\max}$

- We have

$$\omega_{\max}^2 = \omega_0^2\left(1 - \frac{\gamma^2}{2mk}\right)$$

and

$$R_{\max} = \frac{F_0}{\gamma\omega_0\sqrt{1-(\gamma^2/4mk)}} \cong \frac{F_0}{\gamma\omega_0}\left(1 + \frac{\gamma^2}{8mk}\right)$$

where the last expression is an approximation for small γ. If $\gamma^2/(mk) > 2$, then $\omega_{\max}$ is imaginary. In this case, $R_{\max} = F_0/k$, which occurs at $\omega = 0$, and R is a monotone decreasing function of ω. Recall from Section 3.8 that critical damping occurs when $\gamma^2/(mk) = 4$.

Resonance

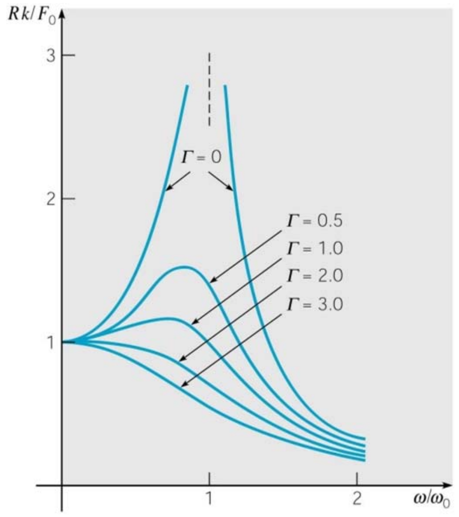
- From the expression

$$R_{\max} = \frac{F_0}{\gamma\omega_0\sqrt{1-(\gamma^2/4mk)}} \cong \frac{F_0}{\gamma\omega_0}\left(1+\frac{\gamma^2}{8mk}\right)$$

  we see that $R_{\max} \cong F_0/(\gamma\,\omega_0)$ for small γ.
- Thus for lightly damped systems, the amplitude R of the forced response is large for ω near ω_0, since $\omega_{\max} \cong \omega_0$ for small γ.
- This is true even for relatively small external forces, and the smaller the γ the greater the effect.
- This phenomena is known as **resonance**. Resonance can be either good or bad, depending on circumstances; for example, when building bridges or designing seismographs.

Graphical Analysis of Quantities

- To get a better understanding of the quantities we have been examining, we graph the ratios $R/(F_0/k)$ vs. ω/ω_0 for several values of $\Gamma = \gamma^2/(mk)$, as shown below.
- Note that the peaks tend to get higher as damping decreases.
- As damping decreases to zero, the values of $R/(F_0/k)$ become asymptotic to $\omega = \omega_0$. Also, if $\gamma^2/(mk) > 2$, then $R_{max} = F_0/k$, which occurs at $\omega = 0$.



Analysis of Phase Angle

- Recall that the phase angle δ given in the forced response

  $$U(t) = R\cos(\omega t - \delta)$$

  is characterized by the equations

  $$\cos\delta = \frac{m(\omega_0^2 - \omega^2)}{\sqrt{m^2(\omega_0^2 - \omega^2)^2 + \gamma^2\omega^2}}, \ \sin\delta = \frac{\gamma\,\omega}{\sqrt{m^2(\omega_0^2 - \omega^2)^2 + \gamma^2\omega^2}}$$

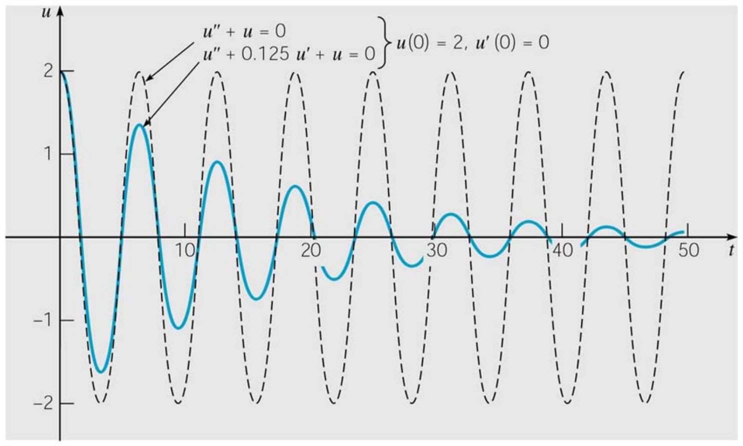
- If $\omega \cong 0$, then $\cos\delta \cong 1$, $\sin\delta \cong 0$, and hence $\delta \cong 0$. Thus the response is nearly in phase with the excitation.
- If $\omega = \omega_0$, then $\cos\delta = 0$, $\sin\delta = 1$, and hence $\delta \cong \pi/2$. Thus response lags behind excitation by nearly $\pi/2$ radians.
- If ω large, then $\cos\delta \cong -1$, $\sin\delta = 0$, and hence $\delta \cong \pi$. Thus response lags behind excitation by nearly π radians, and hence they are nearly out of phase with each other.

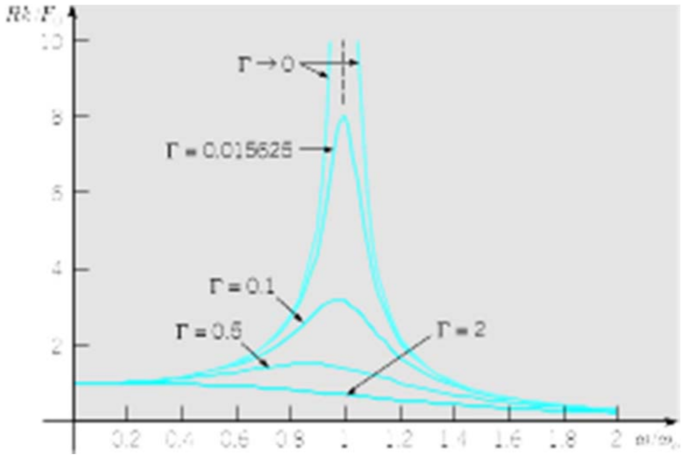
Example 1: Forced Vibrations with Damping (1 of 4)

- Consider the initial value problem

$$u''(t) + 0.125u'(t) + u(t) = 3\cos 2t, \quad u(0) = 2, \ u'(0) = 0$$

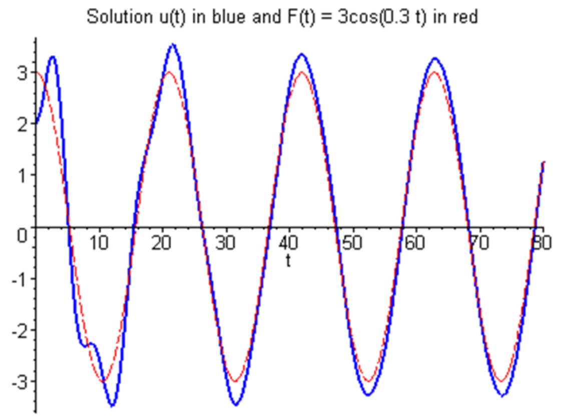
- Then $\omega_0 = 1$, $F_0 = 3$, and $\Gamma = \gamma^2/(mk) = 1/64 = 0.015625$.
- The unforced motion of this system was discussed in Ch 3.8, with the graph of the solution given below, along with the graph of the ratios $R/(F_0/k)$ vs. ω/ω_0 for different values of Γ.





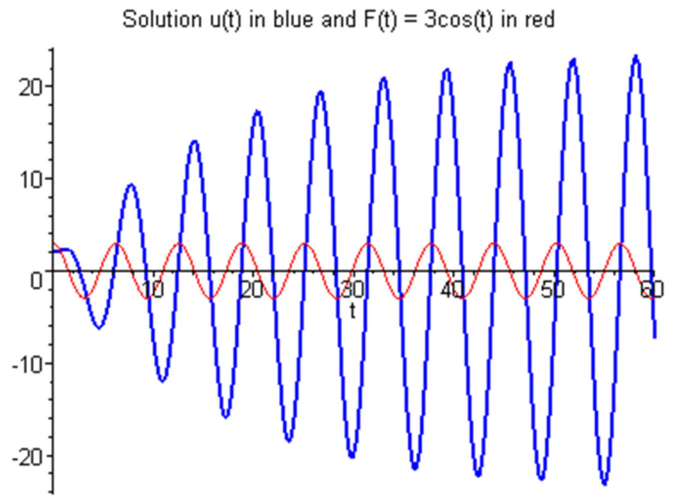
Example 1:
Forced Vibrations with Damping (2 of 4)

- Recall that $\omega_0 = 1$, $F_0 = 3$, and $\Gamma = \gamma^2/(mk) = 1/64 = 0.015625$.
- The solution for the low frequency case $\omega = 0.3$ is graphed below, along with the forcing function.
- After the transient response is substantially damped out, the steady-state response is essentially in phase with excitation, and response amplitude is larger than static displacement.
- Specifically, $R \cong 3.2939 > F_0/k = 3$, and $\delta \cong 0.041185$.



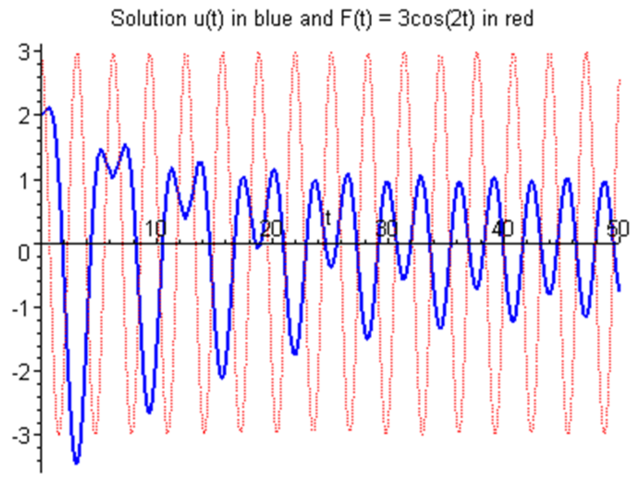
Example 1: Forced Vibrations with Damping (3 of 4)

- Recall that $\omega_0 = 1$, $F_0 = 3$, and $\Gamma = \gamma^2/(mk) = 1/64 = 0.015625$.
- The solution for the resonant case $\omega = 1$ is graphed below, along with the forcing function.
- The steady-state response amplitude is eight times the static displacement, and the response lags excitation by $\pi/2$ radians, as predicted. Specifically, $R = 24 > F_0/k = 3$, and $\delta = \pi/2$.



Example 1: Forced Vibrations with Damping (4 of 4)

- Recall that $\omega_0 = 1$, $F_0 = 3$, and $\Gamma = \gamma^2/(mk) = 1/64 = 0.015625$.
- The solution for the relatively high frequency case $\omega = 2$ is graphed below, along with the forcing function.
- The steady-state response is out of phase with excitation, and response amplitude is about one third the static displacement.
- Specifically, $R \cong 0.99655 \cong F_0/k = 3$, and $\delta \cong 3.0585 \cong \pi$.



Undamped Equation: General Solution for the Case $\omega_0 \neq \omega$

- Suppose there is no damping term. Then our equation is

$$mu''(t) + ku(t) = F_0 \cos \omega t$$

- Assuming $\omega_0 \neq \omega$, then the method of undetermined coefficients can be use to show that the general solution is

$$u(t) = c_1 \cos \omega_0 t + c_2 \sin \omega_0 t + \frac{F_0}{m(\omega_0^2 - \omega^2)} \cos \omega t$$

Undamped Equation: Mass Initially at Rest (1 of 3)

- If the mass is initially at rest, then the corresponding initial value problem is

$$mu''(t) + ku(t) = F_0 \cos \omega t, \;\; u(0) = 0, \; u'(0) = 0$$

- Recall that the general solution to the differential equation is

$$u(t) = c_1 \cos \omega_0 t + c_2 \sin \omega_0 t + \frac{F_0}{m(\omega_0^2 - \omega^2)} \cos \omega t$$

- Using the initial conditions to solve for c_1 and c_2, we obtain

$$c_1 = -\frac{F_0}{m(\omega_0^2 - \omega^2)}, \quad c_2 = 0$$

- Hence

$$u(t) = \frac{F_0}{m(\omega_0^2 - \omega^2)} \left(\cos \omega t - \cos \omega_0 t \right)$$

Undamped Equation: Solution to Initial Value Problem (2 of 3)

- Thus our solution is

$$u(t) = \frac{F_0}{m(\omega_0^2 - \omega^2)}\left(\cos \omega t - \cos \omega_0 t\right)$$

- To simplify the solution even further, let $A = (\omega_0 + \omega)/2$ and $B = (\omega_0 - \omega)/2$. Then $A + B = \omega_0 t$ and $A - B = \omega t$. Using the trigonometric identity

$$\cos(A \pm B) = \cos A \cos B \mp \sin A \sin B,$$

it follows that

$$\cos \omega t = \cos A \cos B + \sin A \sin B$$

$$\cos \omega_0 t = \cos A \cos B - \sin A \sin B$$

and hence

$$\cos \omega t - \cos \omega_0 t = 2 \sin A \sin B$$

Undamped Equation: Beats (3 of 3)

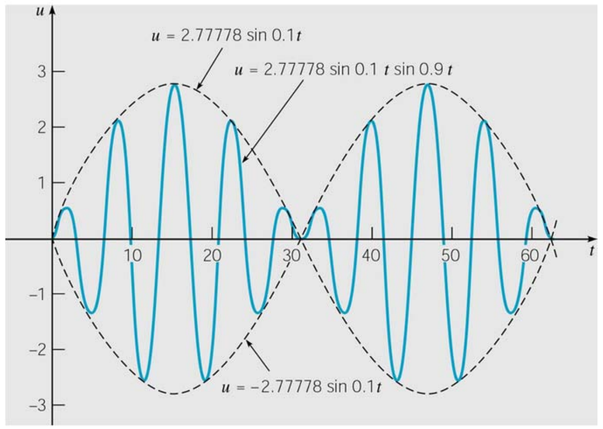
- Using the results of the previous slide, it follows that

$$u(t) = \left[\frac{2F_0}{m(\omega_0^2 - \omega^2)} \sin\frac{(\omega_0 - \omega)t}{2}\right] \sin\frac{(\omega_0 + \omega)t}{2}$$

- When $|\omega_0 - \omega| \cong 0$, $\omega_0 + \omega$ is much larger than $\omega_0 - \omega$, and $\sin[(\omega_0 + \omega)t/2]$ oscillates more rapidly than $\sin[(\omega_0 - \omega)t/2]$.
- Thus motion is a rapid oscillation with frequency $(\omega_0 + \omega)/2$, but with slowly varying sinusoidal amplitude given by

$$\frac{2F_0}{m\left|\omega_0^2 - \omega^2\right|}\left|\sin\frac{(\omega_0 - \omega)t}{2}\right|$$

- This phenomena is called a **beat**.
- Beats occur with two tuning forks of nearly equal frequency.



Example 2: Undamped Equation, Mass Initially at Rest (1 of 2)

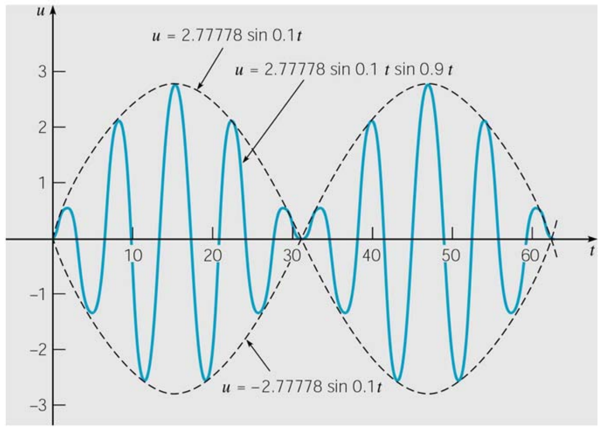
- Consider the initial value problem

$$u''(t) + u(t) = 0.5\cos 0.8t, \quad u(0) = 0,\ u'(0) = 0$$

- Then $\omega_0 = 1$, $\omega = 0.8$, and $F_0 = 0.5$, and hence the solution is

$$u(t) = 2.77778\left(\sin 0.1t\right)\left(\sin 0.9t\right)$$

- The displacement of the spring–mass system oscillates with a frequency of 0.9, slightly less than natural frequency $\omega_0 = 1$.
- The amplitude variation has a slow frequency of 0.1 and period of 20π.
- A half-period of 10π corresponds to a single cycle of increasing and then decreasing amplitude.

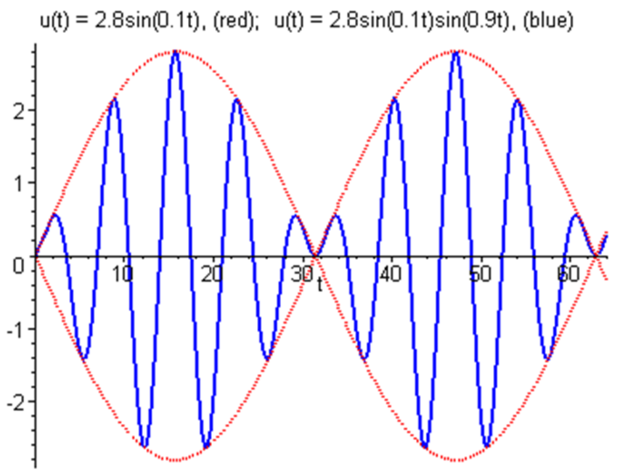


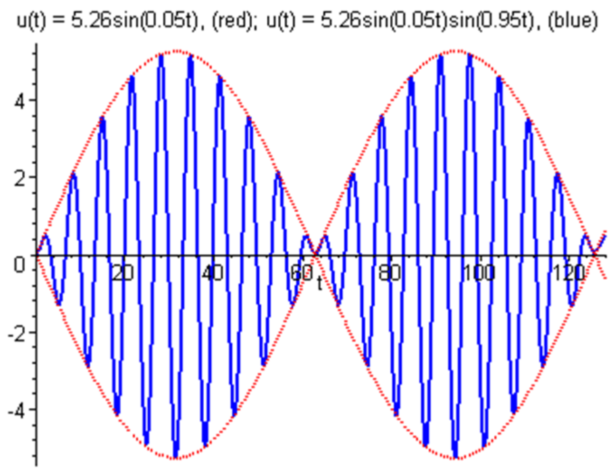
Example 2: Increased Frequency (2 of 2)

- Recall our initial value problem

$$u''(t) + u(t) = 0.5\cos 0.8t, \;\; u(0) = 0, \; u'(0) = 0$$

- If driving frequency ω is increased to $\omega = 0.9$, then the slow frequency is halved to 0.05 with half-period doubled to 20π.
- The multiplier 2.77778 is increased to 5.2632, and the fast frequency only marginally increased, to 0.095.





Undamped Equation: General Solution for the Case $\omega_0 = \omega$ (1 of 2)

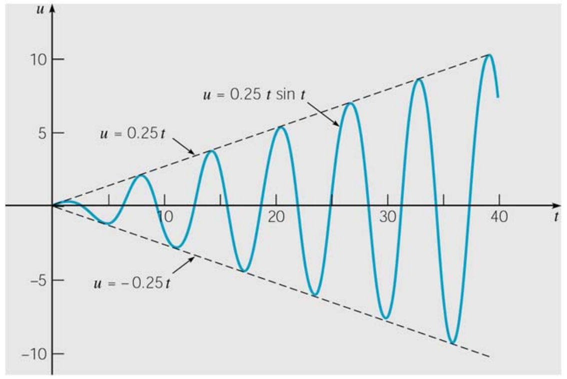
- Recall our equation for the undamped case:

$$mu''(t) + ku(t) = F_0 \cos \omega t$$

- If forcing frequency equals natural frequency of system, i.e., $\omega = \omega_0$, then nonhomogeneous term $F_0 \cos \omega t$ is a solution of homogeneous equation. It can then be shown that

$$u(t) = c_1 \cos \omega_0 t + c_2 \sin \omega_0 t + \frac{F_0}{2m\omega_0} t \sin \omega_0 t$$

- Thus solution u becomes unbounded as $t \to \infty$.
- Note: Model invalid when u gets large, since we assume small oscillations u.

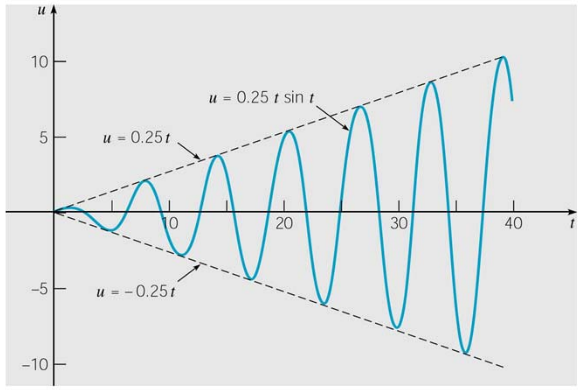


Undamped Equation: Resonance (2 of 2)

- If forcing frequency equals natural frequency of system, i.e., $\omega = \omega_0$, then our solution is

$$u(t) = c_1 \cos \omega_0 t + c_2 \sin \omega_0 t + \frac{F_0}{2m\omega_0} t \sin \omega_0 t$$

- Motion u remains bounded if damping present. However, response u to input $F_0 \cos \omega t$ may be large if damping is small and $|\omega_0 - \omega| \cong 0$, in which case we have resonance.



SECTION 15.1 Exact First-Order Equations

Exact Differential Equations • Integrating Factors

Exact Differential Equations

In Section 5.6, you studied applications of differential equations to growth and decay problems. In Section 5.7, you learned more about the basic ideas of differential equations and studied the solution technique known as separation of variables. In this chapter, you will learn more about solving differential equations and using them in real-life applications. This section introduces you to a method for solving the first-order differential equation

$$M(x, y)\,dx + N(x, y)\,dy = 0$$

for the special case in which this equation represents the exact differential of a function $z = f(x, y)$.

Definition of an Exact Differential Equation

The equation $M(x, y)\,dx + N(x, y)\,dy = 0$ is an **exact differential equation** if there exists a function f of two variables x and y having continuous partial derivatives such that

$$f_x(x, y) = M(x, y) \qquad \text{and} \qquad f_y(x, y) = N(x, y).$$

The general solution of the equation is $f(x, y) = C$.

From Section 12.3, you know that if f has continuous second partials, then

$$\frac{\partial M}{\partial y} = \frac{\partial^2 f}{\partial y\,\partial x} = \frac{\partial^2 f}{\partial x\,\partial y} = \frac{\partial N}{\partial x}.$$

This suggests the following test for exactness.

THEOREM 15.1 Test for Exactness

Let M and N have continuous partial derivatives on an open disc R. The differential equation $M(x, y)\,dx + N(x, y)\,dy = 0$ is exact if and only if

$$\frac{\partial M}{\partial y} = \frac{\partial N}{\partial x}.$$

Exactness is a fragile condition in the sense that seemingly minor alterations in an exact equation can destroy its exactness. This is demonstrated in the following example.

EXAMPLE 1 Testing for Exactness

NOTE Every differential equation of the form

$$M(x)\,dx + N(y)\,dy = 0$$

is exact. In other words, a separable variables equation is actually a special type of an exact equation.

a. The differential equation $(xy^2 + x)\,dx + yx^2\,dy = 0$ is exact because

$$\frac{\partial M}{\partial y} = \frac{\partial}{\partial y}[xy^2 + x] = 2xy \qquad \text{and} \qquad \frac{\partial N}{\partial x} = \frac{\partial}{\partial x}[yx^2] = 2xy.$$

But the equation $(y^2 + 1)\,dx + xy\,dy = 0$ is not exact, even though it is obtained by dividing both sides of the first equation by x.

b. The differential equation $\cos y\,dx + (y^2 - x\sin y)\,dy = 0$ is exact because

$$\frac{\partial M}{\partial y} = \frac{\partial}{\partial y}[\cos y] = -\sin y \qquad \text{and} \qquad \frac{\partial N}{\partial x} = \frac{\partial}{\partial x}[y^2 - x\sin y] = -\sin y.$$

But the equation $\cos y\,dx + (y^2 + x\sin y)\,dy = 0$ is not exact, even though it differs from the first equation only by a single sign.

Note that the test for exactness of $M(x, y)\,dx + N(x, y)\,dy = 0$ is the same as the test for determining whether $\mathbf{F}(x, y) = M(x, y)\mathbf{i} + N(x, y)\mathbf{j}$ is the gradient of a potential function (Theorem 14.1). This means that a general solution $f(x, y) = C$ to an exact differential equation can be found by the method used to find a potential function for a conservative vector field.

EXAMPLE 2 Solving an Exact Differential Equation

Solve the differential equation $(2xy - 3x^2)\,dx + (x^2 - 2y)\,dy = 0$.

Solution The given differential equation is exact because

$$\frac{\partial M}{\partial y} = \frac{\partial}{\partial y}[2xy - 3x^2] = 2x = \frac{\partial N}{\partial x} = \frac{\partial}{\partial x}[x^2 - 2y].$$

The general solution, $f(x, y) = C$, is given by

$$\begin{aligned} f(x, y) &= \int M(x, y)\,dx \\ &= \int (2xy - 3x^2)\,dx = x^2y - x^3 + g(y). \end{aligned}$$

In Section 14.1, you determined $g(y)$ by integrating $N(x, y)$ with respect to y and reconciling the two expressions for $f(x, y)$. An alternative method is to partially differentiate this version of $f(x, y)$ with respect to y and compare the result with $N(x, y)$. In other words,

$$f_y(x, y) = \frac{\partial}{\partial y}[x^2y - x^3 + g(y)] = x^2 + g'(y) = \overbrace{x^2 - 2y}^{N(x,\,y)}.$$

$$\boxed{g'(y) = -2y}$$

Thus, $g'(y) = -2y$, and it follows that $g(y) = -y^2 + C_1$. Therefore,

$$f(x, y) = x^2y - x^3 - y^2 + C_1$$

and the general solution is $x^2y - x^3 - y^2 = C$. Figure 15.1 shows the solution curves that correspond to $C = 1$, 10, 100, and 1000.

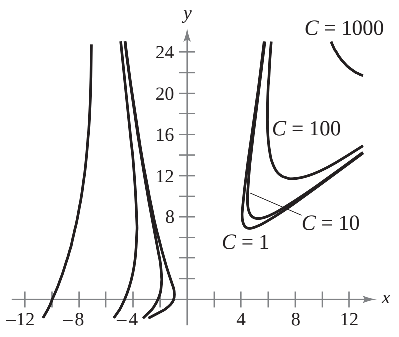


Figure 15.1

EXAMPLE 3 Solving an Exact Differential Equation

Find the particular solution of

$$(\cos x - x\sin x + y^2)\,dx + 2xy\,dy = 0$$

that satisfies the initial condition $y = 1$ when $x = \pi$.

Solution The differential equation is exact because

$$\overbrace{\frac{\partial}{\partial y}[\cos x - x\sin x + y^2]}^{\frac{\partial M}{\partial y}} = 2y = \overbrace{\frac{\partial}{\partial x}[2xy]}^{\frac{\partial N}{\partial x}}.$$

Because $N(x, y)$ is simpler than $M(x, y)$, it is better to begin by integrating $N(x, y)$.

$$f(x, y) = \int N(x, y)\,dy = \int 2xy\,dy = xy^2 + g(x)$$

$$f_x(x, y) = \frac{\partial}{\partial x}[xy^2 + g(x)] = y^2 + g'(x) = \overbrace{\cos x - x\sin x + y^2}^{M(x, y)}$$

$$\boxed{g'(x) = \cos x - x\sin x}$$

Thus, $g'(x) = \cos x - x\sin x$ and

$$g(x) = \int (\cos x - x\sin x)\,dx$$
$$= x\cos x + C_1$$

which implies that $f(x, y) = xy^2 + x\cos x + C_1$, and the general solution is

$$xy^2 + x\cos x = C. \qquad \text{General solution}$$

Applying the given initial condition produces

$$\pi(1)^2 + \pi\cos\pi = C$$

which implies that $C = 0$. Hence, the particular solution is

$$xy^2 + x\cos x = 0. \qquad \text{Particular solution}$$

The graph of the particular solution is shown in Figure 15.3. Notice that the graph consists of two parts: the ovals are given by $y^2 + \cos x = 0$, and the y-axis is given by $x = 0$.

TECHNOLOGY You can use a graphing utility to graph a particular solution that satisfies the initial condition of a differential equation. In Example 3, the differential equation and initial conditions are satisfied when $xy^2 + x\cos x = 0$, which implies that the particular solution can be written as $x = 0$ or $y = \pm\sqrt{-\cos x}$. On a graphing calculator screen, the solution would be represented by Figure 15.2 together with the y-axis.

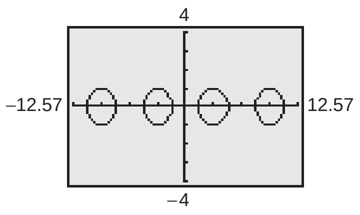


Figure 15.2

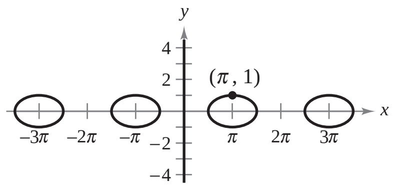


Figure 15.3

In Example 3, note that if $z = f(x, y) = xy^2 + x\cos x$, the total differential of z is given by

$$dz = f_x(x, y)\,dx + f_y(x, y)\,dy$$
$$= (\cos x - x\sin x + y^2)\,dx + 2xy\,dy$$
$$= M(x, y)\,dx + N(x, y)\,dy.$$

In other words, $M\,dx + N\,dy = 0$ is called an *exact* differential equation because $M\,dx + N\,dy$ is exactly the differential of $f(x, y)$.

Integrating Factors

If the differential equation $M(x, y)\,dx + N(x, y)\,dy = 0$ is not exact, it may be possible to make it exact by multiplying by an appropriate factor $u(x, y)$, which is called an **integrating factor** for the differential equation.

EXAMPLE 4 Multiplying by an Integrating Factor

a. If the differential equation

$$2y\,dx + x\,dy = 0 \qquad \text{Not an exact equation}$$

is multiplied by the integrating factor $u(x, y) = x$, the resulting equation

$$2xy\,dx + x^2\,dy = 0 \qquad \text{Exact equation}$$

is exact—the left side is the total differential of x^2y.

b. If the equation

$$y\,dx - x\,dy = 0 \qquad \text{Not an exact equation}$$

is multiplied by the integrating factor $u(x, y) = 1/y^2$, the resulting equation

$$\frac{1}{y}\,dx - \frac{x}{y^2}\,dy = 0 \qquad \text{Exact equation}$$

is exact—the left side is the total differential of x/y.

Finding an integrating factor can be difficult. However, there are two classes of differential equations whose integrating factors can be found routinely—namely, those that possess integrating factors that are functions of either x alone or y alone. The following theorem, which we present without proof, outlines a procedure for finding these two special categories of integrating factors.

THEOREM 15.2 Integrating Factors

Consider the differential equation $M(x, y)\,dx + N(x, y)\,dy = 0$.

1. If

$$\frac{1}{N(x, y)}\left[M_y(x, y) - N_x(x, y)\right] = h(x)$$

is a function of x alone, then $e^{\int h(x)\,dx}$ is an integrating factor.

2. If

$$\frac{1}{M(x, y)}\left[N_x(x, y) - M_y(x, y)\right] = k(y)$$

is a function of y alone, then $e^{\int k(y)\,dy}$ is an integrating factor.

STUDY TIP If either $h(x)$ or $k(y)$ is constant, Theorem 15.2 still applies. As an aid to remembering these formulas, note that the subtracted partial derivative identifies both the denominator and the variable for the integrating factor.

EXAMPLE 5 Finding an Integrating Factor

Solve the differential equation $(y^2 - x)\,dx + 2y\,dy = 0$.

Solution The given equation is not exact because $M_y(x, y) = 2y$ and $N_x(x, y) = 0$. However, because

$$\frac{M_y(x, y) - N_x(x, y)}{N(x, y)} = \frac{2y - 0}{2y} = 1 = h(x)$$

it follows that $e^{\int h(x)\,dx} = e^{\int dx} = e^x$ is an integrating factor. Multiplying the given differential equation by e^x produces the exact differential equation

$$(y^2e^x - xe^x)\,dx + 2ye^x\,dy = 0$$

whose solution is obtained as follows.

$$f(x, y) = \int N(x, y)\,dy = \int 2ye^x\,dy = y^2e^x + g(x)$$

$$f_x(x, y) = y^2e^x + g'(x) = \overbrace{y^2e^x - xe^x}^{M(x, y)}$$

$$\boxed{g'(x) = -xe^x}$$

Therefore, $g'(x) = -xe^x$ and $g(x) = -xe^x + e^x + C_1$, which implies that

$$f(x, y) = y^2e^x - xe^x + e^x + C_1.$$

The general solution is $y^2e^x - xe^x + e^x = C$, or $y^2 - x + 1 = Ce^{-x}$.

In the next example, we show how a differential equation can help in sketching a force field given by $\mathbf{F}(x, y) = M(x, y)\mathbf{i} + N(x, y)\mathbf{j}$.

EXAMPLE 6 An Application to Force Fields

Sketch the force field given by

$$\mathbf{F}(x, y) = \frac{2y}{\sqrt{x^2 + y^2}}\mathbf{i} - \frac{y^2 - x}{\sqrt{x^2 + y^2}}\mathbf{j}$$

by finding and sketching the family of curves tangent to $\mathbf{F}$.

Solution At the point (x, y) in the plane, the vector $\mathbf{F}(x, y)$ has a slope of

$$\frac{dy}{dx} = \frac{-(y^2 - x)/\sqrt{x^2 + y^2}}{2y/\sqrt{x^2 + y^2}} = \frac{-(y^2 - x)}{2y}$$

which, in differential form, is

$$2y\,dy = -(y^2 - x)\,dx$$

$$(y^2 - x)\,dx + 2y\,dy = 0.$$

From Example 5, we know that the general solution of this differential equation is $y^2 - x + 1 = Ce^{-x}$, or $y^2 = x - 1 + Ce^{-x}$. Figure 15.4 shows several representative curves from this family. Note that the force vector at (x, y) is tangent to the curve passing through (x, y).

Force field:

$$\mathbf{F}(x, y) = \frac{2y}{\sqrt{x^2 + y^2}}\mathbf{i} - \frac{y^2 - x}{\sqrt{x^2 + y^2}}\mathbf{j}$$

Family of tangent curves to $\mathbf{F}$:

$y^2 = x - 1 + Ce^{-x}$

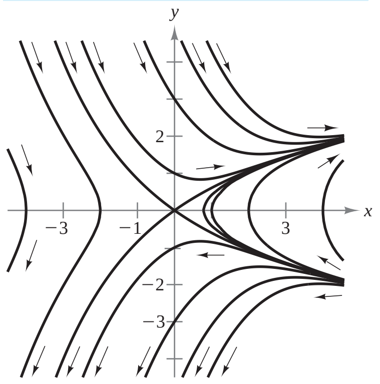


Figure 15.4

EXERCISES FOR SECTION 15.1

LAB SERIES Lab 20

In Exercises 1–10, determine whether the differential equation is exact. If it is, find the general solution.

1. $(2x - 3y)\,dx + (2y - 3x)\,dy = 0$
2. $ye^x\,dx + e^x\,dy = 0$
3. $(3y^2 + 10xy^2)\,dx + (6xy - 2 + 10x^2y)\,dy = 0$
4. $2\cos(2x - y)\,dx - \cos(2x - y)\,dy = 0$
5. $(4x^3 - 6xy^2)\,dx + (4y^3 - 6xy)\,dy = 0$
6. $2y^2e^{xy^2}\,dx + 2xye^{xy^2}\,dy = 0$
7. $\dfrac{1}{x^2 + y^2}(x\,dy - y\,dx) = 0$
8. $e^{-(x^2+y^2)}(x\,dx + y\,dy) = 0$
9. $\dfrac{1}{(x - y)^2}(y^2\,dx + x^2\,dy) = 0$
10. $e^y\cos xy\,[y\,dx + (x + \tan xy)\,dy] = 0$

In Exercises 11 and 12, (a) sketch an approximate solution of the differential equation satisfying the initial condition by hand on the direction field, (b) find the particular solution that satisfies the initial condition, and (c) use a graphing utility to graph the particular solution. Compare the graph with the hand-drawn graph of part (a).

	Differential Equation	*Initial Condition*
11.	$(2x\tan y + 5)\,dx + (x^2\sec^2 y)\,dy = 0$	$y\left(\frac{1}{2}\right) = \pi/4$
12.	$\dfrac{1}{\sqrt{x^2 + y^2}}(x\,dx + y\,dy) = 0$	$y(4) = 3$

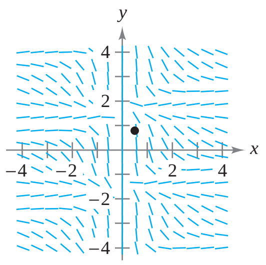


Figure for 11

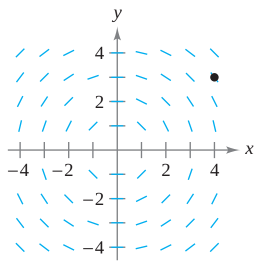


Figure for 12

In Exercises 13–16, find the particular solution that satisfies the initial condition.

	Differential Equation	*Initial Condition*
13.	$\dfrac{y}{x - 1}\,dx + [\ln(x - 1) + 2y]\,dy = 0$	$y(2) = 4$
14.	$\dfrac{1}{x^2 + y^2}(x\,dx + y\,dy) = 0$	$y(0) = 4$
15.	$e^{3x}(\sin 3y\,dx + \cos 3y\,dy) = 0$	$y(0) = \pi$
16.	$(x^2 + y^2)\,dx + 2xy\,dy = 0$	$y(3) = 1$

In Exercises 17–26, find the integrating factor that is a function of x or y alone and use it to find the general solution of the differential equation.

17. $y\,dx - (x + 6y^2)\,dy = 0$
18. $(2x^3 + y)\,dx - x\,dy = 0$
19. $(5x^2 - y)\,dx + x\,dy = 0$
20. $(5x^2 - y^2)\,dx + 2y\,dy = 0$
21. $(x + y)\,dx + \tan x\,dy = 0$
22. $(2x^2y - 1)\,dx + x^3\,dy = 0$
23. $y^2\,dx + (xy - 1)\,dy = 0$
24. $(x^2 + 2x + y)\,dx + 2\,dy = 0$
25. $2y\,dx + \left(x - \sin\sqrt{y}\right)dy = 0$
26. $(-2y^3 + 1)\,dx + (3xy^2 + x^3)\,dy = 0$

In Exercises 27–30, use the integrating factor to find the general solution of the differential equation.

27. $(4x^2y + 2y^2)\,dx + (3x^3 + 4xy)\,dy = 0$
    $u(x, y) = xy^2$
28. $(3y^2 + 5x^2y)\,dx + (3xy + 2x^3)\,dy = 0$
    $u(x, y) = x^2y$
29. $(-y^5 + x^2y)\,dx + (2xy^4 - 2x^3)\,dy = 0$
    $u(x, y) = x^{-2}y^{-3}$
30. $-y^3\,dx + (xy^2 - x^2)\,dy = 0$
    $u(x, y) = x^{-2}y^{-2}$

31. Show that each of the following is an integrating factor for the differential equation

$$y\,dx - x\,dy = 0.$$

(a) $\dfrac{1}{x^2}$ (b) $\dfrac{1}{y^2}$ (c) $\dfrac{1}{xy}$ (d) $\dfrac{1}{x^2 + y^2}$

32. Show that the differential equation

$$(axy^2 + by)\,dx + (bx^2y + ax)\,dy = 0$$

is exact only if $a = b$. If $a \neq b$, show that $x^m y^n$ is an integrating factor, where

$$m = -\frac{2b + a}{a + b}, \qquad n = -\frac{2a + b}{a + b}.$$

In Exercises 33–36, use a graphing utility to graph the family of tangent curves to the given force field.

33. $\mathbf{F}(x, y) = \dfrac{y}{\sqrt{x^2 + y^2}}\mathbf{i} - \dfrac{x}{\sqrt{x^2 + y^2}}\mathbf{j}$
34. $\mathbf{F}(x, y) = \dfrac{x}{\sqrt{x^2 + y^2}}\mathbf{i} - \dfrac{y}{\sqrt{x^2 + y^2}}\mathbf{j}$
35. $\mathbf{F}(x, y) = 4x^2y\,\mathbf{i} - \left(2xy^2 + \dfrac{x}{y^2}\right)\mathbf{j}$
36. $\mathbf{F}(x, y) = (1 + x^2)\,\mathbf{i} - 2xy\,\mathbf{j}$

In Exercises 37 and 38, find an equation for the curve with the specified slope passing through the given point.

	Slope	*Point*
37.	$\dfrac{dy}{dx} = \dfrac{y - x}{3y - x}$	$(2, 1)$
38.	$\dfrac{dy}{dx} = \dfrac{-2xy}{x^2 + y^2}$	$(0, 2)$

39. ***Cost*** If $y = C(x)$ represents the cost of producing x units in a manufacturing process, the **elasticity of cost** is defined as

$$E(x) = \frac{\text{marginal cost}}{\text{average cost}} = \frac{C'(x)}{C(x)/x} = \frac{x}{y}\frac{dy}{dx}.$$

Find the cost function if the elasticity function is

$$E(x) = \frac{20x - y}{2y - 10x}$$

where $C(100) = 500$ and $x \geq 100$.

40. ***Euler's Method*** Consider the differential equation $y' = F(x, y)$ with the initial condition $y(x_0) = y_0$. At any point (x_k, y_k) in the domain of F, $F(x_k, y_k)$ yields the slope of the solution at that point. Euler's Method gives a discrete set of estimates of the y values of a solution of the differential equation using the iterative formula

$$y_{k+1} = y_k + F(x_k, y_k)\,\Delta x$$

where $\Delta x = x_{k+1} - x_k$.

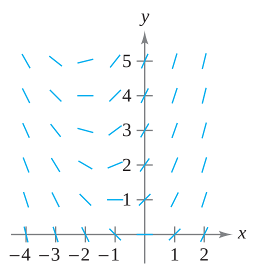
(a) Write a short paragraph describing the general idea of how Euler's Method works.

(b) How will decreasing the magnitude of Δx affect the accuracy of Euler's Method?

41. ***Euler's Method*** Use Euler's Method (see Exercise 40) to approximate $y(1)$ for the values of Δx given in the table if $y' = x + \sqrt{y}$ and $y(0) = 2$. (Note that the number of iterations increases as Δx decreases.) Sketch a graph of the approximate solution on the direction field in the figure.

Δx	0.50	0.25	0.10
Estimate of $y(1)$			

The value of $y(1)$, accurate to three decimal places, is 4.213.



42. ***Programming*** Write a program for a graphing utility or computer that will perform the calculations of Euler's Method for a specified differential equation, interval, Δx, and initial condition. The output should be a graph of the discrete points approximating the solution.

Euler's Method **In Exercises 43–46, (a) use the program of Exercise 42 to approximate the solution of the differential equation over the indicated interval with the specified value of Δx and the initial condition, (b) solve the differential equation analytically, and (c) use a graphing utility to graph the particular solution and compare the result with the graph of part (a).**

	Differential Equation	*Interval*	Δx	*Initial Condition*
43.	$y' = x\sqrt[3]{y}$	$[1, 2]$	0.01	$y(1) = 1$
44.	$y' = \dfrac{\pi}{4}(y^2 + 1)$	$[-1, 1]$	0.1	$y(-1) = -1$
45.	$y' = \dfrac{-xy}{x^2 + y^2}$	$[2, 4]$	0.05	$y(2) = 1$
46.	$y' = \dfrac{6x + y^2}{y(3y - 2x)}$	$[0, 5]$	0.2	$y(0) = 1$

47. ***Euler's Method*** Repeat Exercise 45 for $\Delta x = 1$ and discuss how the accuracy of the result changes.

48. ***Euler's Method*** Repeat Exercise 46 for $\Delta x = 0.5$ and discuss how the accuracy of the result changes.

True or False? **In Exercises 49–52, determine whether the statement is true or false. If it is false, explain why or give an example that shows it is false.**

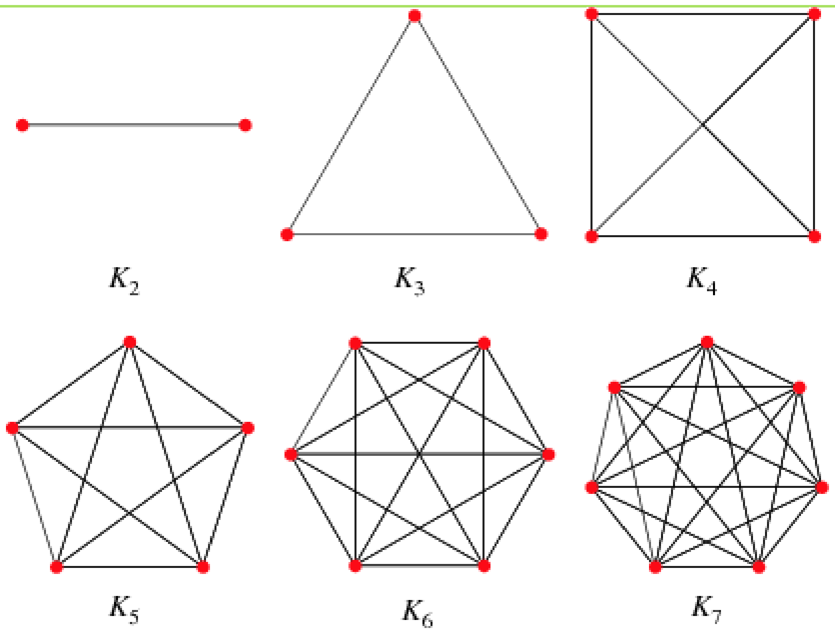
49. The differential equation $2xy\,dx + (y^2 - x^2)\,dy = 0$ is exact.

50. If $M\,dx + N\,dy = 0$ is exact, then $xM\,dx + xN\,dy = 0$ is also exact.

51. If $M\,dx + N\,dy = 0$ is exact, then $[f(x) + M]\,dx + [g(y) + N]\,dy = 0$ is also exact.

52. The differential equation $f(x)\,dx + g(y)\,dy = 0$ is exact.

Complete Graph



A complete graph is a graph in which each pair of graph vertices is connected by an edge. The complete graph with $n$graph vertices is denoted K_n and has $\binom{n}{2} = n(n-1)/2$ (the triangular numbers) undirected edges, where $\binom{n}{k}$ is a binomial coefficient. In older literature, complete graphs are sometimes called universal graphs.

The complete graph on n nodes is implemented in the Wolfram Language as `CompleteGraph`[*n*] or `CompleteGraph`[*n*] in the Wolfram Language package `Combinatorica`` , and precomputed properties are available using `GraphData`[{`"Complete"`, *n*}]. A graph may be tested to see if it is complete in the Wolfram Language using the function `CompleteGraphQ`[*g*].

The complete graph on 0 nodes is a trivial graph known as the null graph, while the complete graph on 1 node is a trivial graph known as the singleton graph.

In the 1890s, Walecki showed that complete graphs K_n admit a Hamilton decomposition for odd n, and decompositions into Hamiltonian cycles plus a perfect matching for even n (Lucas 1892, Bryant 2007, Alspach 2008). Alspach *et al.* (1990) give a construction for Hamilton decompositions of all K_n.

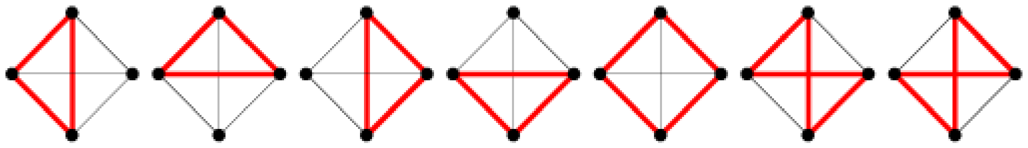
The graph complement of the complete graph K_n is the empty graph on n nodes. K_n has graph genus $\lceil (n-3)(n-4)/12 \rceil$ for $n \geq 3$ (Ringel and Youngs 1968; Harary 1994, p. 118), where $\lceil x \rceil$ is the ceiling function.

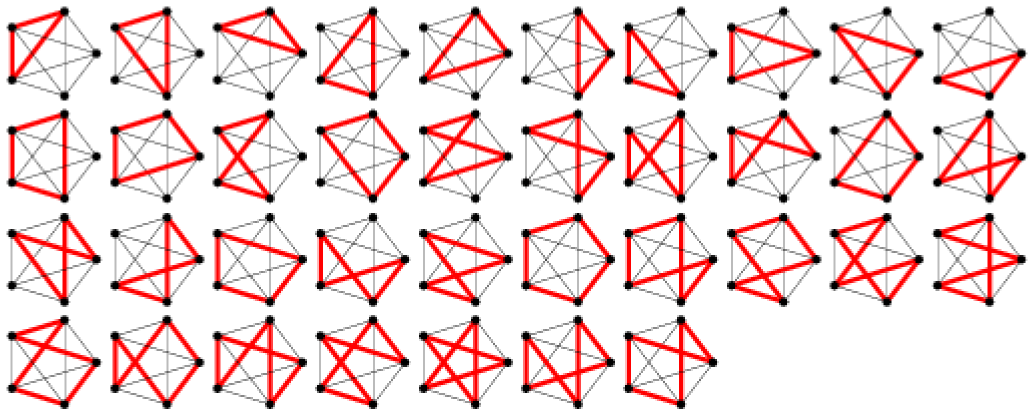
The adjacency matrix A of the complete graph G takes the particularly simple form of all 1s with 0s on the diagonal, i.e., the unit matrix minus the identity matrix,

$$\mathsf{A} = \mathsf{J} - \mathsf{I}. \qquad (1)$$

K_3 is the cycle graph C_3, as well as the odd graph O_2 (Skiena 1990, p. 162). K_4 is the tetrahedral graph, as well as the wheel graph W_4 , and is also a planar graph. K_5 is nonplanar, and is sometimes known as the pentatope graph or Kuratowski graph. Conway and Gordon (1983) proved that every embedding of K_6 is intrinsically linked with at least one pair of linked triangles, and K_6 is also a Cayley graph. Conway and Gordon (1983) also showed that any embedding of K_7 contains a knotted Hamiltonian cycle.

The complete graph K_n is the line graph of the star graph S_{n+1} .





The chromatic polynomial $\pi_n(z)$ of K_n is given by the falling factorial $(z)_n$. The independence polynomial is given by

$$I_n(x) = 1 + n\,x, \tag{2}$$

and the matching polynomial by

$$\mu(x) = \mathrm{He}_n(x) \tag{3}$$
$$= 2^{-n/2} H_n\left(\frac{x}{\sqrt{2}}\right), \tag{4}$$

where $\mathrm{He}_n(x)$ is a normalized version of the Hermite polynomial $H_n(x)$.

The chromatic number and clique number of K_n are n. The automorphism group of the complete graph $\mathrm{Aut}(K_n)$ is the symmetric group S_n (Holton and Sheehan 1993, p. 27). The numbers of graph cycles in the complete graph K_n for $n = 3$, 4, ... are 1, 7, 37, 197, 1172, 8018 ... (OEIS A002807). These numbers are given analytically by

$$C_n = \sum_{k=3}^{n} \frac{1}{2}\binom{n}{k}(k-1)! \tag{5}$$
$$= \frac{1}{4}\,n\,[2\,{}_3F_1(1, 1, 1-n; 2; -1) - n - 1], \tag{6}$$

where $\binom{n}{k}$ is a binomial coefficient and ${}_3F_1(a, b, c; d; z)$ is a generalized hypergeometric function (Char 1968, Holroyd and Wingate 1985).

It is not known in general if a set of trees with 1, 2, ..., $n-1$graph edges can always be packed into K_n. However, if the choice of trees is restricted to either the path or star from each family, then the packing can always be done (Zaks and Liu 1977, Honsberger 1985).

The bipartite double graph of the complete graph K_n is the n-crown graph.

GRAPH THEORY AND APPLICATIONS

G. Appasami, M.Sc., M.C.A., M.Phil., M.Tech., (Ph.D.)

Assistant Professor

Department of Computer Science and Engineering

Dr. Paul's Engineering Collage

Pauls Nagar, Villupuram

Tamilnadu, India

SARUMATHI PUBLICATIONS

Villupuram, Tamilnadu, India

First Edition: July 2016

Published By

SARUMATHI PUBLICATIONS

Price Rs. 101/-

Copies can be had from

SARUMATHI PUBLICATIONS

Villupuram, Tamilnadu, India.

Sarumathi.publications@gmail.com

Printed at

Meenam Offset

Pondicherry – 605001, India

CS6702 GRAPH THEORY AND APPLICATIONS L T P C 3 0 0 3

OBJECTIVES: The student should be made to:

- Be familiar with the most fundamental Graph Theory topics and results.
- Be exposed to the techniques of proofs and analysis.

UNIT I INTRODUCTION 9

Graphs – Introduction – Isomorphism – Sub graphs – Walks, Paths, Circuits – Connectedness – Components – Euler graphs – Hamiltonian paths and circuits – Trees – Properties of trees – Distance and centers in tree – Rooted and binary trees.

UNIT II TREES, CONNECTIVITY & PLANARITY 9

Spanning trees – Fundamental circuits – Spanning trees in a weighted graph – cut sets – Properties of cut set – All cut sets – Fundamental circuits and cut sets – Connectivity and separability – Network flows – 1-Isomorphism – 2-Isomorphism – Combinational and geometric graphs – Planer graphs – Different representation of a planer graph.

UNIT III MATRICES, COLOURING AND DIRECTED GRAPH 8

Chromatic number – Chromatic partitioning – Chromatic polynomial – Matching – Covering -Four color problem – Directed graphs – Types of directed graphs – Digraphs and binary relations – Directed paths and connectedness – Euler graphs.

UNIT IV PERMUTATIONS & COMBINATIONS 9

Fundamental principles of counting - Permutations and combinations - Binomial theorem - combinations with repetition - Combinatorial numbers - Principle of inclusion and exclusion - Derangements - Arrangements with forbidden positions.

UNIT V GENERATING FUNCTIONS 10

Generating functions - Partitions of integers - Exponential generating function – Summation operator - Recurrence relations - First order and second order – Non-homogeneous recurrence relations - Method of generating functions.

TOTAL: 45 PERIODS

OUTCOMES:

Upon Completion of the course, the students should be able to:

- Write precise and accurate mathematical definitions of objects in graph theory.
- Use mathematical definitions to identify and construct examples and to distinguish examples from non-examples.
- Validate and critically assess a mathematical proof.
- Use a combination of theoretical knowledge and independent mathematical thinking in creative investigation of questions in graph theory.
- Reason from definitions to construct mathematical proofs.

TEXT BOOKS:

1. Narsingh Deo, "Graph Theory: With Application to Engineering and Computer Science", Prentice Hall of India, 2003.
2. Grimaldi R.P. "Discrete and Combinatorial Mathematics: An Applied Introduction", Addison Wesley, 1994.

REFERENCES:

1. Clark J. and Holton D.A, "A First Look at Graph Theory", Allied Publishers, 1995.
2. Mott J.L., Kandel A. and Baker T.P. "Discrete Mathematics for Computer Scientists and Mathematicians" , Prentice Hall of India, 1996.
3. Liu C.L., "Elements of Discrete Mathematics", Mc Graw Hill, 1985.
4. Rosen K.H., "Discrete Mathematics and Its Applications", Mc Graw Hill, 2007.

Acknowledgement

I am very much grateful to the management of paul's educational trust, Respected principal **Dr. Y. R. M. Rao, M.E., Ph.D.,** cherished Dean **Dr. E. Mariappane, M.E., Ph.D.,** and helpful Head of the department **Mr. M. G. Lavakumar M.E., (Ph.D.)**.

I thank my colleagues and friends for their cooperation and their support in my career venture.

I thank my parents and family members for their valuable support in completion of the book successfully.

I express my special thanks to **SARUMATHI PUBLICATIONS** for their continued cooperation in shaping the work.

Suggestions and comments to improve the text are very much solicitated.

Mr. G. Appasami

TABLE OF CONTENTS

UNIT IV PERMUTATIONS & COMBINATIONS

CS6702 GRAPH THEORY AND APPLICATIONS

UNIT I INTRODUCTION

1.1 GRAPHS – INTRODUCTION

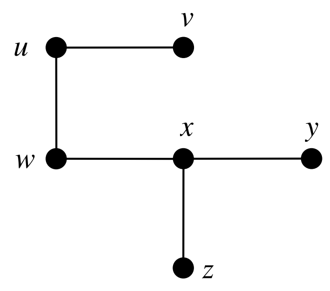
1.1.1 Introduction

A graph G = (V, E) consists of a set of objects V={v_1, v_2, v_3, … } called **vertices** (also called **points** or **nodes**) and other set E = {e_1, e_2, e_3, ...} whose elements are called **edges** (also called **lines** or **arcs**).

For example : A graph G is defined by the sets V(G) = {*u, v, w, x, y, z*} and

E(G) = {*uv, uw, wx, xy, xz*}.

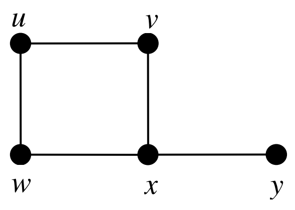
Graph G:



Graph G with 6 vertices and 5 edges

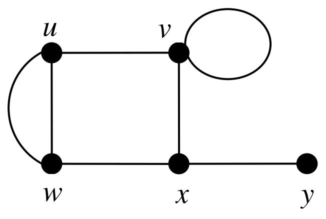
- The set V(G) is called the **vertex set** of G and E(G) is the **edge set** of G.
- A graph with p-vertices and q-edges is called a **(p, q) graph**.
- The (1, 0) graph is called **trivial graph**.
- An edge having the same vertex as its end vertices is called a **self-loop**.
- More than one edge associated a given pair of vertices called **parallel edges**.
- Intersection of any two edges is not a vertex.
- A graph that has neither self-loops nor parallel edges is called **simple graph**.

Graph G:



Simple Graph

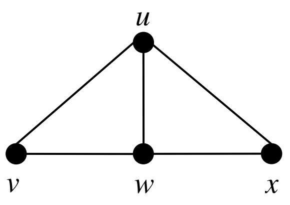
Graph H:



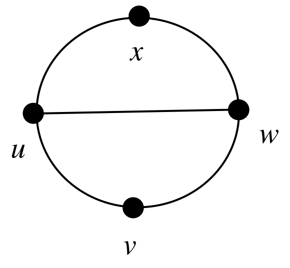
Pseudo Graph

- Same graph can be drawn in different ways.

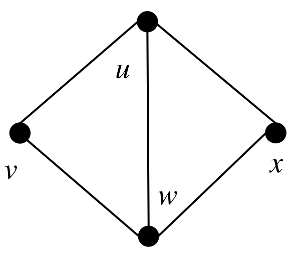
Graph G_1:



Graph G_2:



Graph G_3:



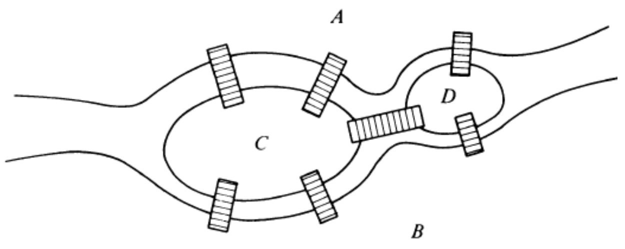
- A graph is also called a linear complex, a 1-complex, or a one-dimensional complex.
- A vertex is also referred to as a node, a junction, a point, O-cell, or an O-simplex.
- Other terms used for an edge are a branch, a line, an element, a 1-cell, an arc, and a 1-simplex.

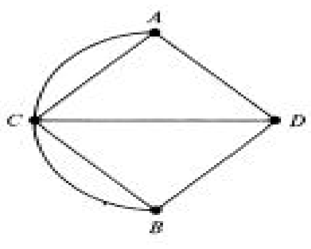
1.1.2 Applications of graph.

(i) Konigsberg bridge problem

The city of Königsberg in Prussia (now Kaliningrad, Russia) was set on *both sides* (A and B) of the Pregel River, and included two *large islands* (C and D) which were connected to each other and the mainland by *seven bridges*. The problem was to devise a walk through the city that would cross each bridge once and only once, with the provisos that: the islands could only be reached by the bridges and every bridge once accessed must be crossed to its other end. The starting and ending points of the walk need not be the same.

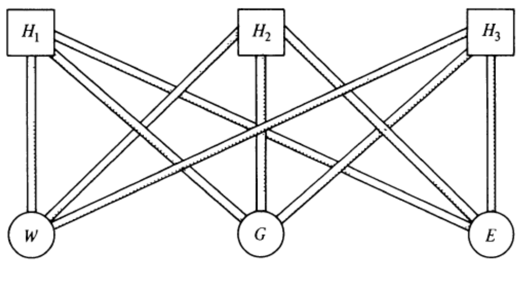
Euler proved that the problem has no solution. This problem can be represented by a graph as shown below.

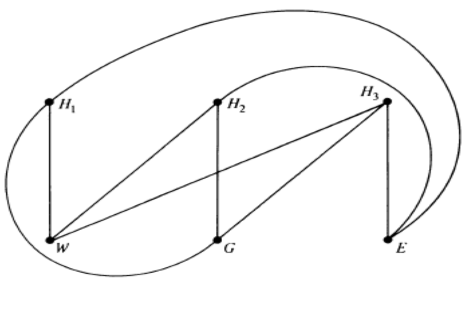




(ii) Utilities problem

There are three houses H1, H2 and H3, each to be connected to each of the three utilities water (W), gas (G) and electricity (E) by means of conduits. This problem can be represented by a graph as shown below.

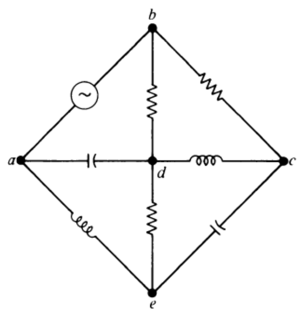


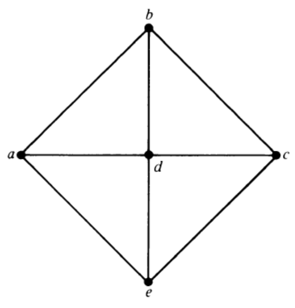


(iii) Electrical network problems

Every Electrical network has two factor.

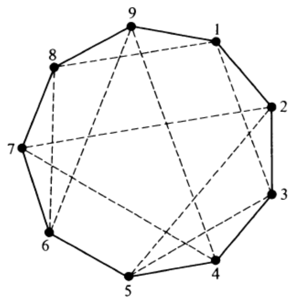
1. Elements such as resisters, inductors, transistors, and so on.
2. The way these elements are connected together (topology)





(iv) Seating problems

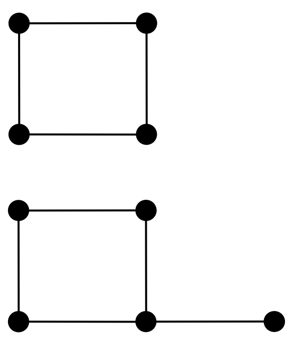
Nine members of a new club meet each day for lunch at a round table. They decide to sit such that every member has different neighbors at each lunch. How many days can this arrangement last?



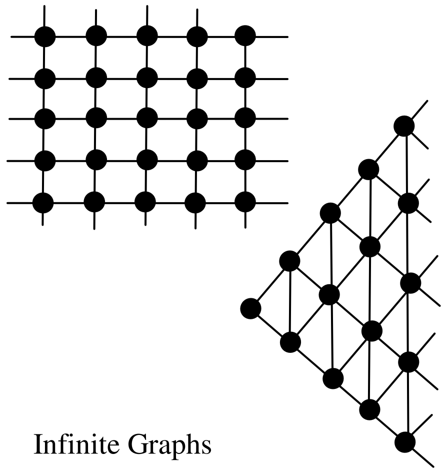
This situation can be represented by a graph with nine vertices such that each vertex represents a member, and an edge joining two vertices represents the relationship of sitting next to each other. Figure shows two possible seating arrangements—these are 1 2 3 4 5 6 7 8 9 1 (solid lines), and 1 3 5 2 7 4 9 6 8 1 (dashed lines). It can be shown by graph-theoretic considerations that there are more arrangements possible.

1.2.3 Finite and infinite graphs

A graph with a finite number off vertices as well as a finite number of edges is called a *finite* graph; otherwise, it is an *infinite* graph.



Finite Graphs

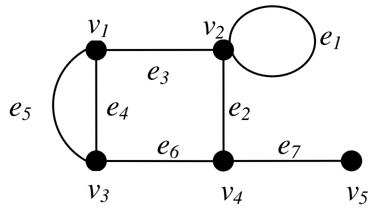


Infinite Graphs

1.1.4 Incidence, adjacent and degree.

When a vertex v_i is an end vertex of some edge e_j, v_i and e_j are said to be *incident* with each other. Two non parallel edges are said to be *adjacent* if they are incident on a common vertex. The number of edges incident on a vertex v_i, with self-loops counted twice, is called the *degree* (also called valency), $d(v_i)$, of the vertex v_i. A graph in which all vertices are of equal degree is called *regular graph.*

Graph G:



The edges e_2, e_6 and e_7 are incident with vertex v_4.

The edges e_2 and e_7 are adjacent.

The edges e_2 and e_4 are not adjacent.

The vertices v_4 and v_5 are adjacent.

The vertices v_1 and v_5 are not adjacent.

$d(v_1) = d(v_3) = d(v_4) = 3$. $d(v_2) = 4$. $d(v_5) = 1$.

Total degree $= d(v_1) + d(v_2) + d(v_3) + d(v_4) + d(v_5)$

$= 3 + 4 + 3 + 3 + 1 = 14 =$ Twice the number of edges.

Theorem 1-1

The number of vertices of odd degree in a graph is always even.

Proof: Let us now consider a graph G with *e* edges and *n* vertices $v_1, v_2, ..., v_n$. Since each edge contributes two degrees, the sum of the degrees of all vertices in G is twice the number of edges in G. That is,

$$\sum_{i=1}^{n} d(v_i) = 2e.$$

If we consider the vertices with odd and even degrees separately, the quantity in the left side of the above equation can be expressed as the sum of two sums, each taken over vertices of even and odd degrees, respectively, as follows:

$$\sum_{i=1}^{n} d(v_i) = \sum_{even} d(v_j) + \sum_{odd} d(v_k)$$

Since the left-hand side in the above equation is even, and the first expression on the right-hand side is even (being a sum of even numbers), the second expression must also be even:

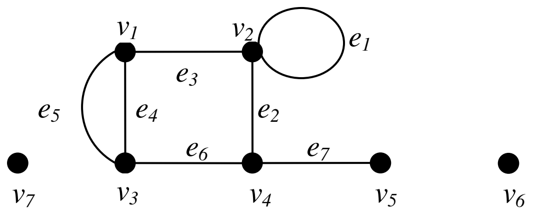
$$\sum_{odd} d(v_k) = an\ even\ number$$

Because in the above equation each *d(v$_k$)* is odd, the total number of terms in the sum must be even to make the sum an even number. Hence the theorem. ■

1.1.5 Define Isolated and pendent vertex.

A vertex having no incident edge is called an *isolated vertex*. In other words, isolated vertices are vertices with zero degree. A vertex of degree one is called a *pendant vertex* or an *end vertex*.

Graph G:



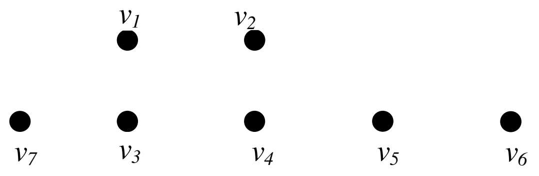
The vertices v_6 and v_7 are *isolated vertices*.

The vertex v_5 is a *pendant vertex*.

1.1.6 Null graph and Multigraph

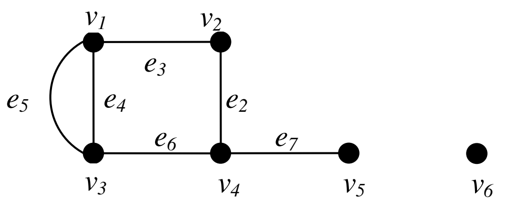
In a graph G=(V, E), If E is empty (Graph without any edges), then G is called a **null graph**.

Graph G:



In a multigraph, no loops are allowed but more than one edge can join two vertices, these edges are called **multiple edges** or parallel edges and a graph is called **multigraph.**

Graph G:



The edges e_5 and e_4 are **multiple** (parallel) edges.

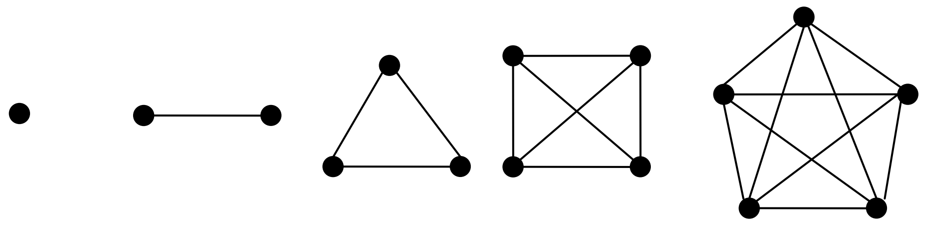
1.1.7 Complete graph and Regular graph

Complete graph

A simple graph G is said to be **complete** if every vertex in G is connected with every other vertex. *i.e.,* if G contains exactly one edge between each pair of distinct vertices.

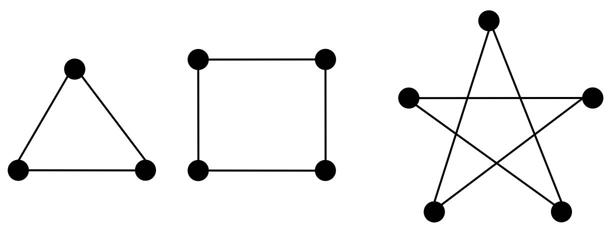
A complete graph is usually denoted by **K**$_n$**.** It should be noted that Kn has exactly $n(n-1)/2$ edges.

The complete graphs K$_n$ for n = 1, 2, 3, 4, 5 are show in the following Figure.



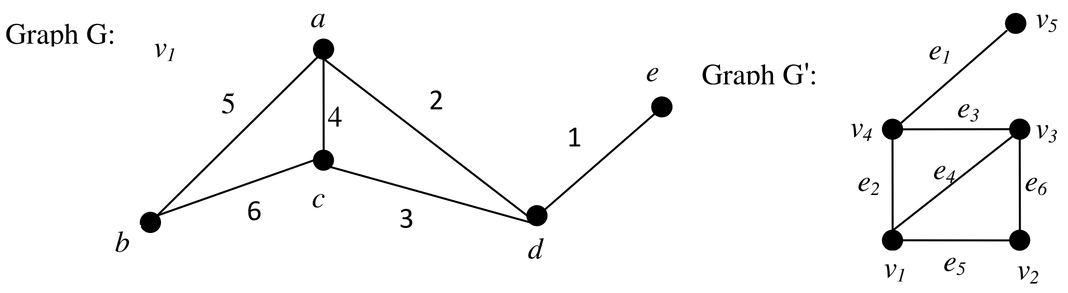
Regular graph

A graph, in which all vertices are of **equal degree,** is called a **regular graph.** If the degree of each vertex is r, then the graph is called a regular **graph of degree *r*.**



1.2 ISOMORPHISM

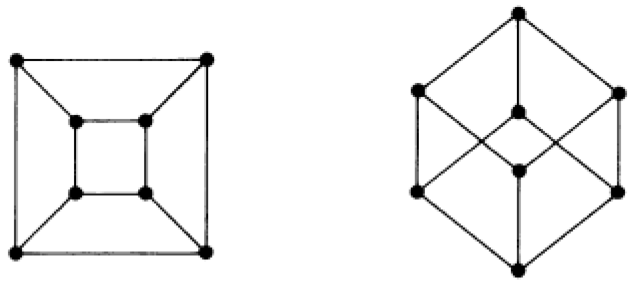
Two graphs G and G' are said to be **isomorphic** to each other if there is a one-to-one correspondence (bijection) between their vertices and between their edges such that the incidence relationship is preserved.



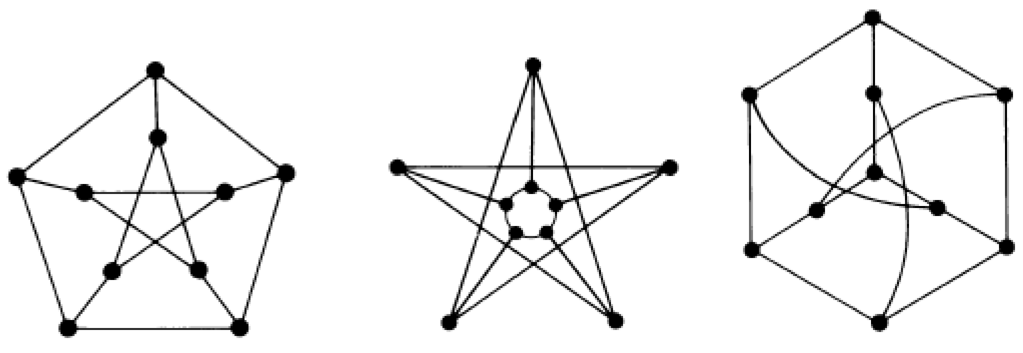
Correspondence of vertices	Correspondence of edges
$f(a) = v_1$	$f(1) = e_1$
$f(b) = v_2$	$f(2) = e_2$
$f(c) = v_3$	$f(3) = e_3$
$f(d) = v_4$	$f(4) = e_4$
$f(e) = v_5$	$f(5) = e_5$

Adjacency also preserved. Therefore G and G' are said to be isomorphic.

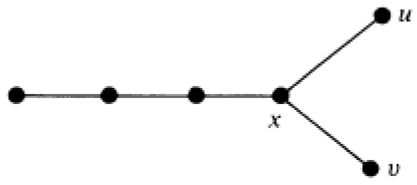
The following graphs are isomorphic to each other. i.e two different ways of drawing the same geaph.

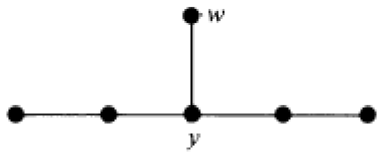


The following three graphs are isomorphic.



The following two graphs are not isomorphic, because x is adjacent to two pendent vertex is not preserved.

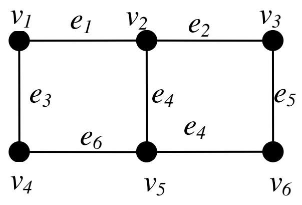




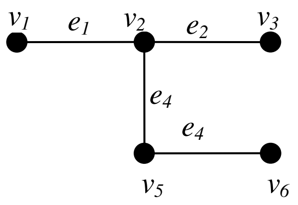
1.3 SUB GRAPHS

A graph G' is said to be a subgraph of a graph G, if all the vertices and all the edges of G' are in G, and each edge of G' has the same end vertices in G' as in G.

Graph G:



Subgraph G' of G:



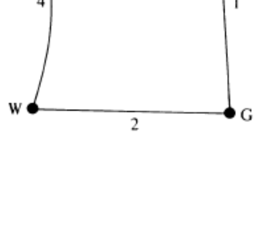
A subgraph can be thought of as being contained in (or a part of) another graph. The symbol from set theory, g ⊂ G, is used in stating "g is a subgraph of G".

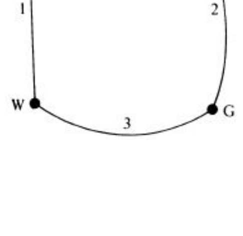
The following observations can be made immediately:

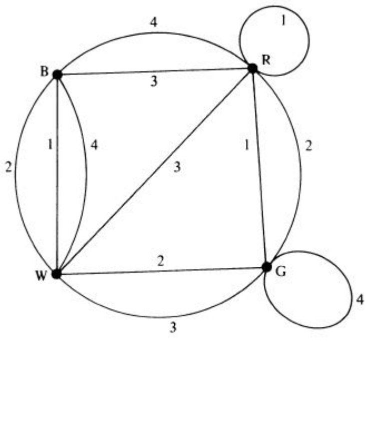
1. Every graph is its own subgraph.
2. A subgraph of a subgraph of G is a subgraph of G.
3. A single vertex in a graph C is a subgraph of G.
4. A single edge in G, together with its end vertices, is also a subgraph of G.

Edge-Disjoint Subgraphs: Two (or more) subgraphs g_1, and g_2 of a graph G are said to be edge disjoint if g_1, and g_2 do not have any edges in common.

For example, the following two graphs are edge-disjoint sub-graphs of the graph G.







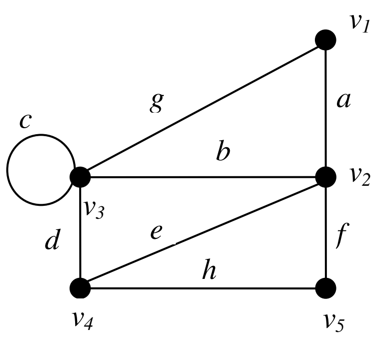
Note that although edge-disjoint graphs do not have any edge in common, they may have vertices in common. Sub-graphs that do not even have vertices in common are said to be vertex disjoint. (Obviously, graphs that have no vertices in common cannot possibly have edges in common.)

1.4 WALKS, PATHS, CIRCUITS

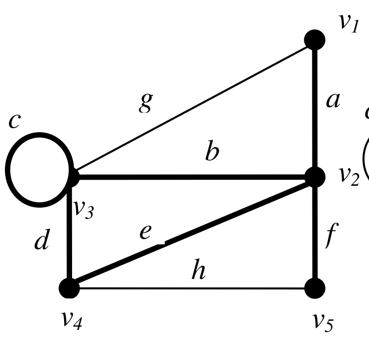
A **walk** is defined as a finite alternating sequence of vertices and edges, beginning and ending with vertices. No edge appears more than once. It is also called as an edge train or a chain.

An open walk in which no vertex appears more than once is called **path**. The number of edges in the path is called **length of a path**.

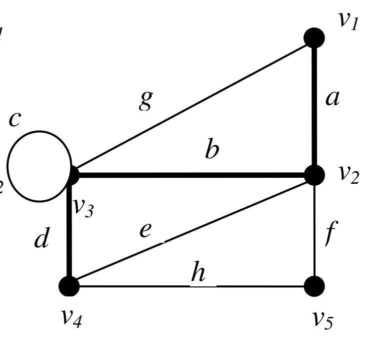
A closed walk in which no vertex (except initial and final vertex) appears more than once is called a **circuit**. That is, a circuit is a closed, nonintersecting walk.



Graph G:



Open walk



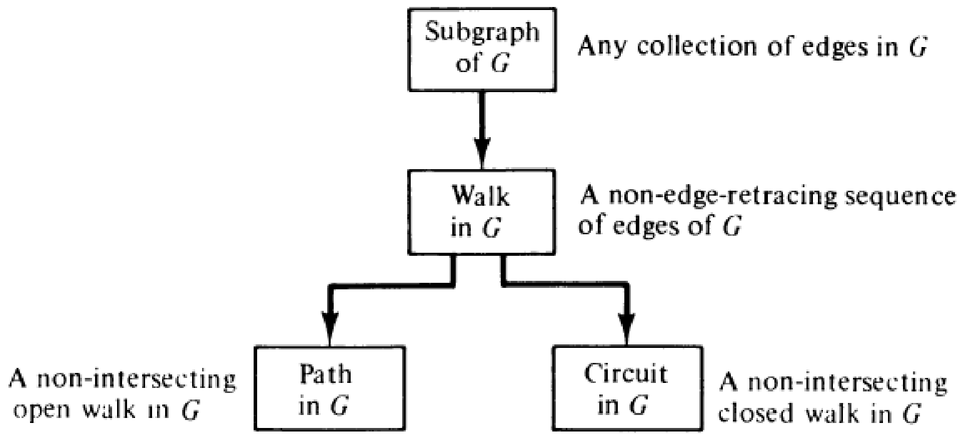
Path of length 3

$v_1\ a\ v_2\ b\ v_3\ c\ v_3\ d\ v_4\ e\ v_2 f\ v_5$ is a walk. v_1 *and* v_5 are terminals of walk.

$v_1\ a\ v_2\ b\ v_3\ d\ v_4$ is a path. $a\ v_2\ b\ v_3\ c\ v_3\ d\ v_4\ e\ v_2 f\ v_5$ is not a path.

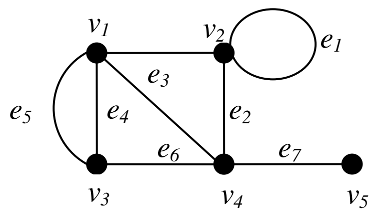
$v_2\ b\ v_3\ d\ v_4\ e\ v_2$ is a circuit.

The concept of walks, paths, and circuits are simple and tha relation is represented by the following figure.

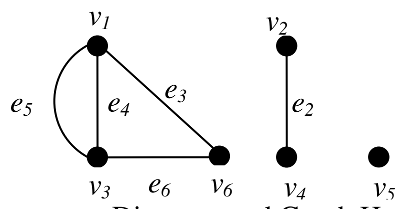

Subgraph of G
Any collection of edges in G
Walk in G
A non-edge-retracing sequence of edges of G
A non-intersecting open walk in G
Path in G
Circuit in G
A non-intersecting closed walk in G

1.5 CONNECTEDNESS

A graph G is said to be **connected** if there is at least one path between every pair of vertices in G. Otherwise, G is disconnected.



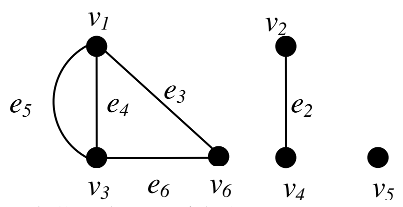
Connected Graph G



Disconnected Graph H

1.6 COMPONENTS

A disconnected graph consists of two or more connected graphs. Each of these connected subgraphs is called a component.



Disconnected Graph H with 3 components

THEOREM 1-2

A graph G is disconnected if and only if its vertex set V can be partitioned into two nonempty, disjoint subsets V_1 and V_2 such that there exists no edge in G whose one end vertex is in subset V_1 and the other in subset V_2.

Proof: Suppose that such a partitioning exists. Consider two arbitrary vertices a and b of G, such that $a \in V_1$ and $b \in V_2$. No path can exist between vertices a and b; otherwise, there would be at least one edge whose one end vertex would be in V_1 and the other in V_2. Hence, if a partition exists, G is not connected.

Conversely, let G be a disconnected graph. Consider a vertex a in G. Let V_1 be the set of all vertices that are joined by paths to a. Since G is disconnected, V_1 does not include all vertices of G. The remaining vertices will form a (nonempty) set V_2. No vertex in V_1 is joined to any in V_2 by an edge. Hence the partition. ■

THEOREM 1-3

If a graph (connected or disconnected) has exactly two vertices of odd degree, there must be a path joining these two vertices.

Proof: Let G be a graph with all even vertices except vertices v_1, and v_2, which are odd. From Theorem 1-2, which holds for every graph and therefore for every component of a disconnected graph, no graph can have an odd number of odd vertices. Therefore, in graph G, v_1 and v2 must belong to the same component, and hence must have a path between them. ■

THEOREM 1-4

A simple graph (i.e., a graph without parallel edges or self-loops) with n vertices and k components can have at most $(n — k)(n — k + 1)/2$ edges.

Proof: Let the number of vertices in each of the k components of a graph G be n_1, n_2, ..., n_k. Thus we have $n_1 + n_2 + \cdots + n_k = n, \quad n_k \geq 1$

The proof of the theorem depends on an algebraic inequality.

$$\sum_{i-1}^{k} n_i^2 \leq n^2 - (k-1)(2n-k)$$

Now the maximum number of edges in the ith component of G (which is a simple connected graph) is $\frac{1}{2}n_i(n_i - 1)$. Therefore, the maximum number of edges in G is

$$\frac{1}{2}\sum_{i-1}^{k}(n_i - 1)n_i = \frac{1}{2}\left(\sum_{i-1}^{k} n_i^2\right) - \frac{n}{2}$$

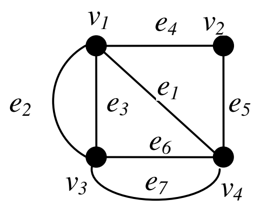
$$\leq \frac{1}{2}[n^2 - (k-1)(2n-k)] - \frac{n}{2}$$

$$= \frac{1}{2}(n-k)(n-k+1). \blacksquare$$

1.7 EULER GRAPHS

A path in a graph G is called Euler path if it includes every edges exactly once. Since the path
contains every edge exactly once, it is also called Euler trail / Euler line.

A closed Euler path is called Euler circuit. A graph which contains an Eulerian circuit is called an Eulerian graph.



v_4 e_1 v_1 e_2 v_3 e_3 v_1 e_4 v_2 e_5 v_4 e_6 v_3 e_7 v_4 is an Euler circuit. So the above graph is Euler graph.

THEOREM 1-4

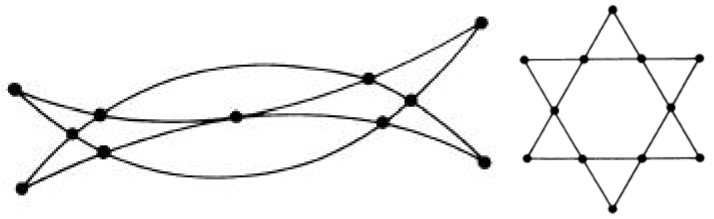
A given connected graph ***G*** is an Euler graph if and only if all vertices of ***G*** are of even degree.

Proof: Suppose that ***G*** is an Euler graph. It therefore contains an Euler line (which is a closed walk). In tracing this walk we observe that every time the walk meets a vertex *v* it goes through two "new" edges incident on *v* - with one we "entered" *v* and with the other "exited." This is true not only of all intermediate vertices of the walk but also of the terminal vertex, because we "exited" and "entered" the same vertex at the beginning and end of the walk, respectively. Thus if G is an Euler graph, the degree of every vertex is even.

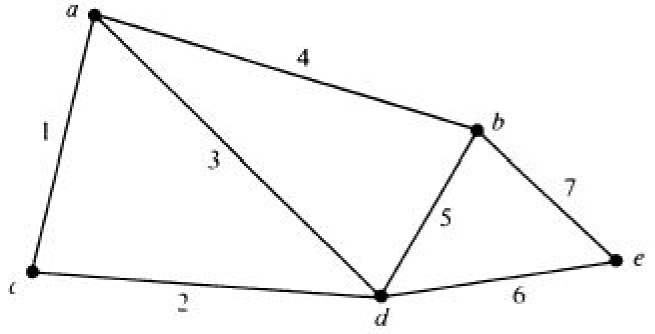
To prove the sufficiency of the condition, assume that all vertices of ***G*** are of even degree. Now we construct a walk starting at an arbitrary vertex *v* and going through the edges of ***G*** such that no edge is traced more than once. We continue tracing as far as possible. Since every vertex is of even degree, we can exit from every vertex we enter; the tracing cannot stop at any vertex but *v*. And since *v* is also of even degree, we shall eventually reach *v* when the tracing comes to an end. If this closed walk *h* we just traced includes all the edges of ***G, G*** is an Euler graph. If not, we remove from ***G*** all the edges in ***h*** and obtain a subgraph ***h'*** of ***G*** formed by the remaining edges. Since both ***G*** and ***h*** have all their vertices of even degree, the degrees of the vertices of ***h'*** are also even. Moreover, ***h'*** must touch ***h*** at least at one vertex ***a,*** because ***G*** is connected. Starting from ***a,*** we can again construct a new walk in graph ***h'***. Since all the vertices of ***h'*** are of even degree, this walk in ***h'*** must terminate at vertex ***a;*** but this walk in ***h'*** can be combined with ***h*** to form a new walk, which starts and ends at vertex *v* and has more edges than ***h.*** This process can be repeated until we obtain a closed walk that traverses all the edges of ***G.*** Thus ***G*** is an Euler graph. ■

Unicursal graph

An open walk that includes all the edges of a graph without retracing any edge is called ***unicrusal line*** or an ***open Euler line***. A (connected) graph that has a unicrusal line will be called a ***unicursal graph.***



Euler graphs (i) Mohammed's scimitars (ii) Star of david.



unicursal graph with a walk a 1 c 2 d 3 a 4 b 5 d 6 e 7 b.

THEOREM 1-5

In a connected graph ***G*** with exactly ***2k*** odd vertices, there exist ***k*** edge-disjoint subgraphs such that they together contain all edges of ***G*** and that each is a unicursal graph.

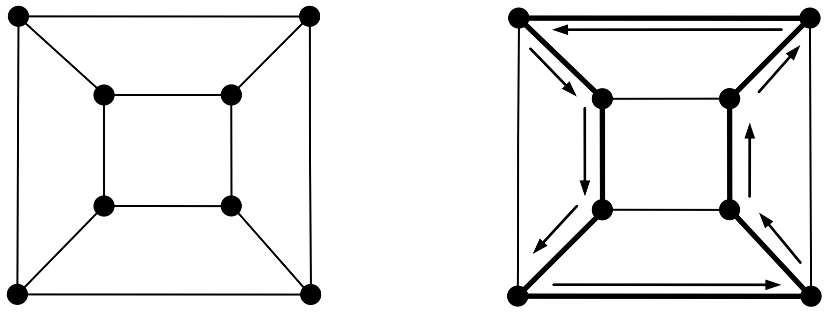
Proof: Let the odd vertices of the given graph ***G*** be named v_1, v_2, …, v_k; w_1, w_2, …, w_k in any arbitrary order. Add ***k*** edges to ***G*** between the vertex pairs (v_1, w_1), (v_2, w_2), …, (v_k, w_k) to form a new graph ***G'.***

Since every vertex of ***G'*** is of even degree, ***G'*** consists of an Euler line ***p.*** Now if we remove from ***p*** the ***k*** edges we just added (no two of these edges are incident on the same vertex), ***p*** will be split into ***k*** walks, each of which is a unicursal line: The first removal will leave a single unicursal line; the second removal will split that into two unicursal lines; and each successive removal will split a unicursal line into two unicursal lines, until there are ***k*** of them. Thus the theorem. ■

1.8 HAMILTONIAN PATHS AND CIRCUITS

A **Hamiltonian circuit** in a connected graph is defined as a closed walk that traverses every vertex of graph G exactly once except starting and terminal vertex.

Removal of any one edge from a Hamiltonian circuit generates a path. This path is called **Hamiltonian path**.

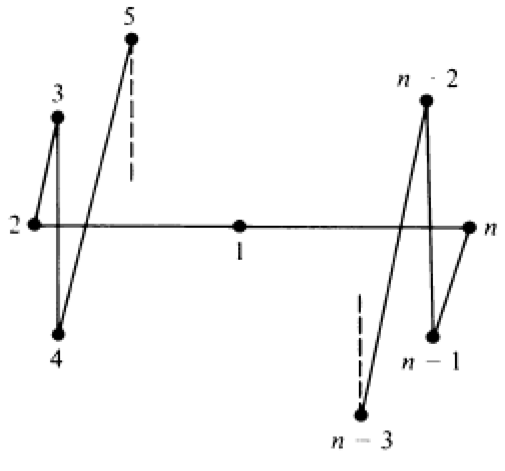


THEOREM 1-6

In a complete graph with $\boldsymbol{n}$ vertices there are $(\boldsymbol{n}-1)/2$ edge-disjoint Hamiltonian circuits, if $\boldsymbol{n}$ is an odd number ≥ 3.

Proof: A complete graph $\boldsymbol{G}$ of $\boldsymbol{n}$ vertices has $\boldsymbol{n}(\boldsymbol{n}-1)/2$ edges, and a Hamiltonian circuit in $\boldsymbol{G}$ consists of $\boldsymbol{n}$ edges. Therefore, the number of edge-disjoint Hamiltonian circuits in $\boldsymbol{G}$ cannot exceed $(\boldsymbol{n}-1)/2$. That there are $(\boldsymbol{n}-1)/2$ edge-disjoint Hamiltonian circuits, when $\boldsymbol{n}$ is odd, can be shown as follows:

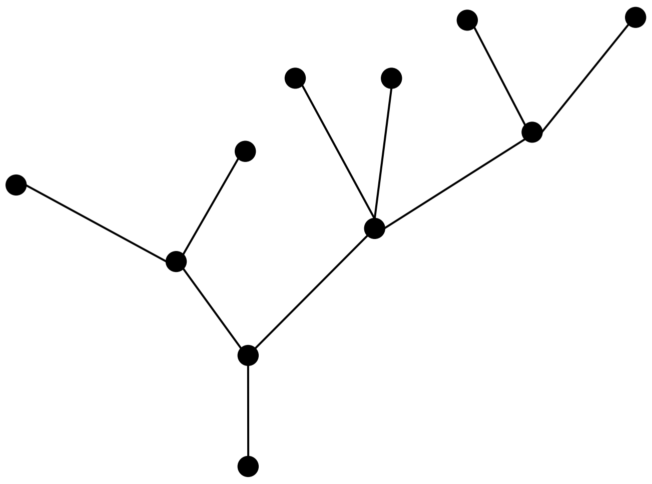
The subgraph (of a complete graph of $\boldsymbol{n}$ vertices) in Figure is a Hamiltonian circuit. Keeping the vertices fixed on a circle, rotate the polygonal pattern clockwise



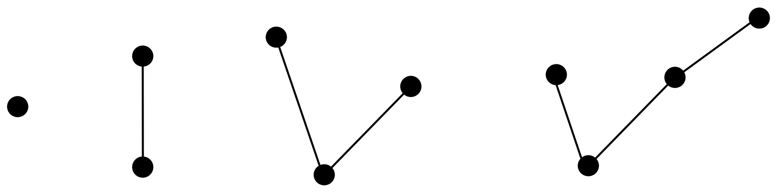
by $360/(n-1), 2\cdot 360/(n-1), 3\cdot 360/(n-1) \ldots\ldots (\boldsymbol{n}-3)/2\cdot 360/(n-1)$ degrees. Observe that each rotation produces a Hamiltonian circuit that has no edge in common with any of the previous ones. Thus we have ***(n* - **3)/2 new Hamiltonian circuits, all edge disjoint from the one in Figure and also edge disjoint among themselves. Hence the theorem. ■

1.9 TREES

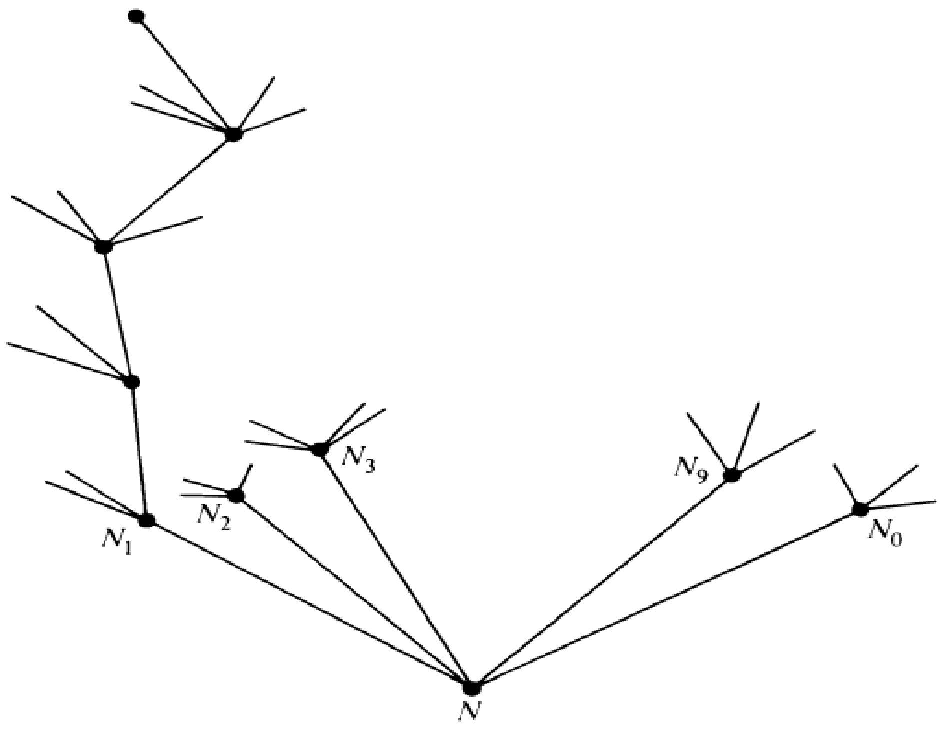
A tree is a connected graph without any circuits.



Trees with 1, 2, 3, and 4 vertices are shown in figure.



Decision tree shown in figure



1.10 PROPERTIES OF TREES

1. There is one and only one path between every pair of vertices in a tree T.
2. In a graph G there is one and only one path between every pair of vertices, G is a tree.
3. A tree with *n* vertices has *n*-1 edges.
4. Any connected graph with *n* vertices has *n*-1 edges is a tree.
5. A graph is a tree if and only if it is minimally connected.
6. A graph G with *n* vertices has *n*-1 edges and no circuits are connected.

THEOREM 1-7

There is one and only one path between every pair of vertices in a tree, *T*.

Proof: Since *T* is a connected graph, there must exist at least one path between every pair of vertices in *T*. Now suppose that between two vertices *a* and *b* of *T* there are two distinct paths. The union of these two paths will contain a circuit and *T* cannot be a tree. ■

Conversely:

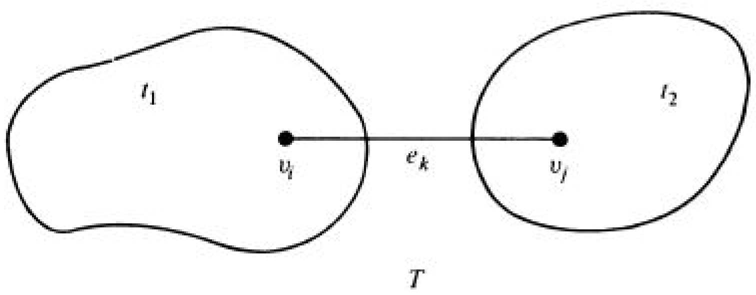
THEOREM 1-8

If in a graph G there is one and only one path between every pair of vertices, *G* is a tree.

Proof: Existence of a path between every pair of vertices assures that *G* is connected. A circuit in a graph (with two or more vertices) implies that there is at least one pair of vertices a, b such that there are two distinct paths between *a* and *b*. Since *G* has one and only one path between every pair of vertices, G can have no circuit. Therefore, *G* is a tree. Prepared by G. Appasami, Assistant professor, Dr. pauls Engineering College. ■

THEOREM 1-9

A tree with *n* vertices has $n - 1$ edges.



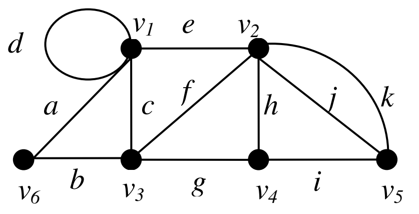
Proof: The theorem will be proved by induction on the number of vertices. It is easy to see that the theorem is true for n = 1, 2, and 3 (see Figure). Assume that the theorem holds for all trees with fewer than *n* vertices.

Let us now consider a tree *T* with *n* vertices. In *T* let e_k be an edge with end vertices v_i and v_j. According to Theorem 1-9, there is no other path between v_i and v_j, except e_k. Therefore, deletion of e_k from *T* will disconnect the graph, as shown in Figure. Furthermore, $T — e_k$ consists of exactly two components, and since there were no circuits in no begin with, each of these components is a tree. Both these trees, t_1 and t_2, have fewer than *n* vertices each, and therefore, by the induction hypothesis, each contains one less edge than the number of vertices in it. Thus $T — e_k$ consists of n — 2 edges (and *n* vertices). Hence *T* has exactly *n* — 1 edges. ■

1.11 DISTANCE AND CENTERS IN TREE

In a connected graph G, the distance $d(v_i, v_j)$ between two of its vertices v_i and v_j is the length of the shortest path.

Graph G:



Paths between vertices v_6 and v_2 are (a, e), (a, c, f), (b, c, e), (b, f), (b, g, h), and (b, g, i, k). The shortest paths between vertices v_6 and v_2 are (a, e) and (b, f), each of length two.

Hence $d(v_6, v_2) = 2$

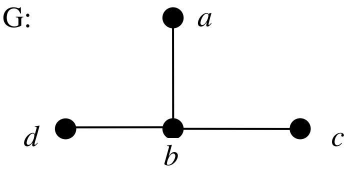
Define eccentricity and center.

The eccentricity E(v) of a vertex v in a graph G is the distance from v to the vertex farthest from v in G; that is,

$$E(v) = \max_{v_i \in G} d(v, v_i)$$

A vertex with minimum eccentricity in graph G is called a center of G

Graph G:

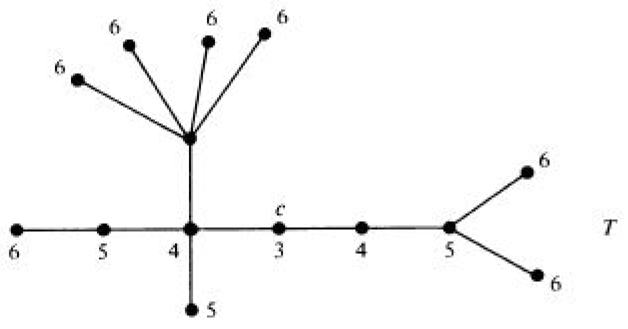


Distance d(a, b) = 1, d(a, c) =2, d(c, b)=1, and so on.

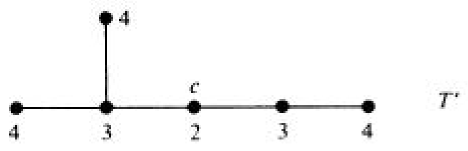
Eccentricity E(a) =2, E(b) =1, E(c) =2, and E(d) =2.

Center of G = A vertex with minimum eccentricity in graph G = b.

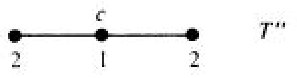
Finding Center of graph.



(a)



(b)



(c)

Distance metric.

The function *f* (*x, y*) of two variables defines the distance between them. These function must satisfy certain requirements. They are

1. Non-negativity: $f(x, y) \geq 0$, and $f(x, y) = 0$ if and only if $x = y$.
2. Symmetry: $f(x, y) = f(x, y)$.
3. Triangle inequality: $f(x, y) \leq f(x, z) + f(z, y)$ for any z.

Radius and Diameter in a tree.

The eccentricity of a center in a tree is defined as the radius of tree.

The length of the longest path in a tree is called the diameter of tree.

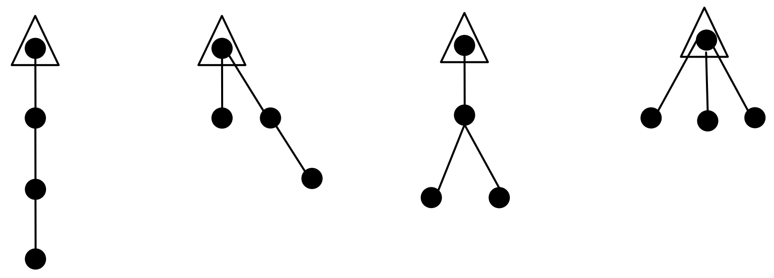
1.12 ROOTED AND BINARY TREES

Rooted tree

A tree in which one vertex (called the root) is distinguished from all the others is called a **rooted tree**.

In general tree means without any root. They are sometimes called as **free trees** (non rooted trees).

The root is enclosed in a small triangle. All rooted trees with four vertices are shown below.



Rooted binary tree

There is exactly one vertex of degree two (root) and each of remaining vertex of degree one or three. A binary rooted tree is special kind of rooted tree. Thus every binary tree is a rooted tree. A non pendent vertex in a tree is called an internal vertex.

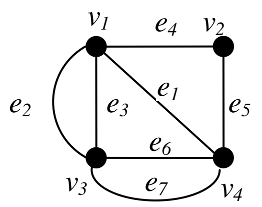
UNIT II TREES, CONNECTIVITY & PLANARITY

2. 1 SPANNING TREES

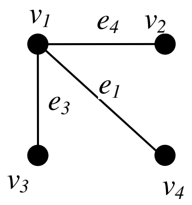
2.1.1 Spanning trees.

A tree T is said to be a spanning tree of a connected graph G if T is a subgraph of G and T contains all vertices (maximal tree subgraph).

Graph G:



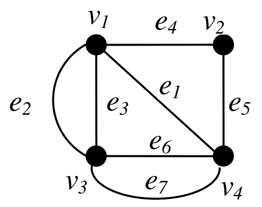
Spanning Tree T:



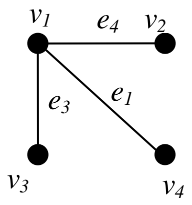
2.1.2. Branch and chord.

An edge in a spanning tree *T* is called a *branch* of *T*. An edge of G is not in a given spanning tree *T* is called a *chord* (*tie* or *link*).

Graph G:



Spanning Tree T:

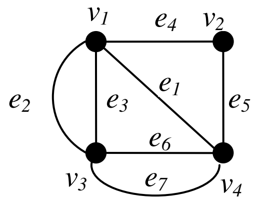


Edge e_1 is a branch of *T* *Edge e_5 is a chord* of *T*

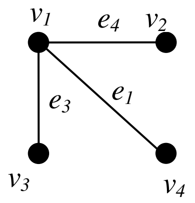
2.1.3. Complement of tree.

If T is a spanning tree of graph G, then the complement of T of G denoted by $\bar{T}$ is the collection of chords. It also called as *chord set* (*tie* set or *cotree*) of T

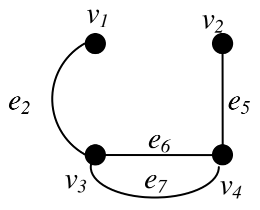
Graph G:



Spanning Tree T:



$\bar{\boldsymbol{T}}$:Complement of Tree T



$$\boldsymbol{T} \cup \bar{\boldsymbol{T}} = \boldsymbol{G}$$

2.1.4. Rank and Nullity:

A graph G with n number of vertices, e number of edges, and k number of components with the following constraints $n - k \geq 0$ and $e - n + k \geq 0$.

Rank $r = n - k$

Nullity $\mu = e - n + k$ (Nullity also called as *Cyclomatic number* or *first betti number*)

Rank of G = number of branches in any spanning tree of G

Nullity of G = number of chords in G

Rank + Nullity = e = number of edges in G

2. 2 FUNDAMENTAL CIRCUITS

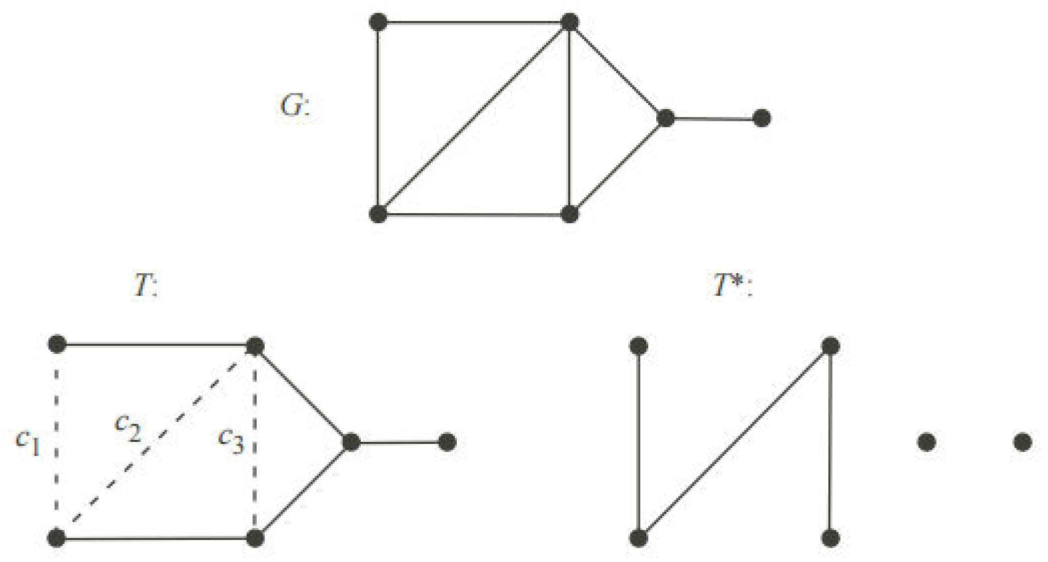
Addition of an edge between any two vertices of a tree creates a circuit. This is because there already exists a path between any two vertices of a tree.

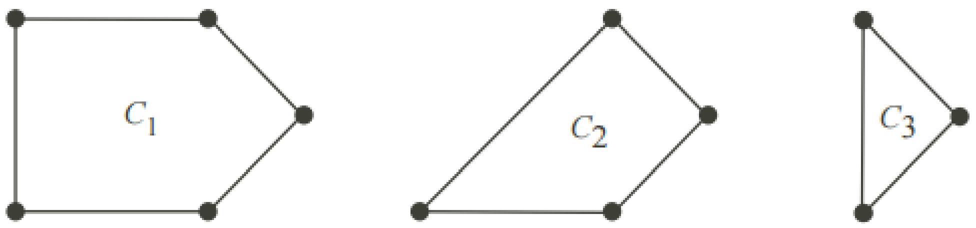
If the branches of the spanning tree T of a connected graph G are $b_1, \dots, b_{n-1}$ and the corresponding links of the co spanning tree T $*$ are $c_1, \dots, c_m$−n+1, then there exists one and only one circuit Ci in T + c_i (which is the subgraph of G induced by the branches of T and c_i)

Theorem: We call this circuit a fundamental circuit. Every spanning tree defines m − n + 1 fundamental circuits $C_1, \dots, C_{m-n+1}$, which together form a fundamental set of circuits. Every fundamental circuit has exactly one link which is not in any other fundamental circuit in the fundamental set of circuits.

Therefore, we can not write any fundamental circuit as a ring sum of other fundamental circuits in the same set. In other words, the fundamental set of circuits is linearly independent under the ring sum operation.

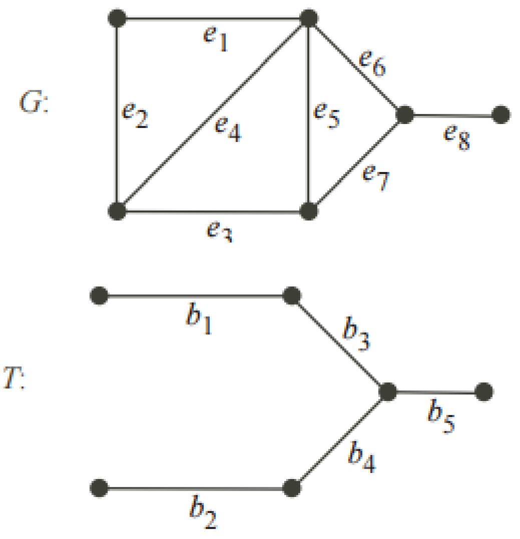
Example:





The graph T − b_i has two components T_1 and T_2. The corresponding vertex sets are V_1 and V_2. Then, (v_1,v_2) is a cut of G. It is also a cut set of G if we treat it as an edge set because G − hV1, V2i has two components . Thus, every branch bi of T has a corresponding cut set I_i . The cut sets I_1, . . . , I_{n-1} are also known as fundamental cut sets and they form a fundamental set of cut sets. Every fundamental cut set includes exactly one branch of T and every branch of T belongs to exactly one fundamental cut set. Therefore, every spanning tree defines a unique fundamental set of cut sets for G.

Example. (Continuing from the previous example) .



The graph has the spanning tree that defines these fundamental cut sets:

b1 : {e1, e2} b2 : {e2, e3, e4} b3 : {e2, e4, e5, e6} b4 : {e2, e4, e5, e7}b5 : {e8}

Next, we consider some properties of circuits and cut sets:

(a) Every cut set of a connected graph G includes at least one branch from every spanning tree of G. (Counter hypothesis: Some cut set F of G does not include any branches of a spanning tree T. Then, T is a subgraph of G − F and G − F is connected.

(b) Every circuit of a connected graph G includes at least one link from every co spanning tree of G. (Counter hypothesis: Some circuit C of G does not include any link of a cos panning tree T ∗ . Then, T = G − T ∗ has a circuit and T is not a tree.

2. 3 SPANNING TREES IN A WEIGHTED GRAPH

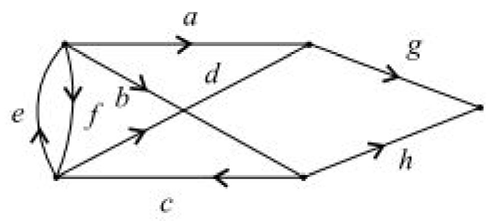
A spanning tree in a graph G is a minimal subgraph connecting all the vertices of G. If G is a weighted graph, then the weight of a spanning tree *T* of G is defined as the sum of the weights of all the branches in *T*.

A spanning tree with the smallest weight in a weighted graph is called a *shortest spanning tree (shortest-distance spanning tree* or *minimal spanning tree).*

A shortest spanning tree T for a weighted connected graph G with a constraint $d(v_i) \leq k$ for all vertices in T. for k=2, the tree will be Hamiltonian path.

A spanning tree is an n-vertex connected digraph analogous to a spanning tree in an undirected graph and consists of n − 1 directed arcs.

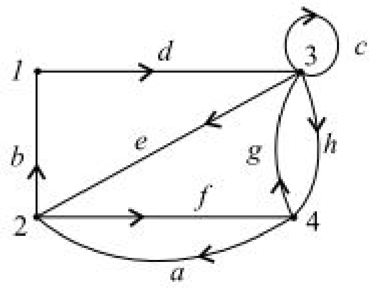
A spanning arborescence in a connected digraph is a spanning tree that is an arborescence. For example, {a, b, c, g} is a spanning arborescence in Figure .



Theorem: In a connected isograph D of n vertices and m arcs, let W = (a_1, a_2,..., a_m) be an Euler line, which starts and ends at a vertex v (that is, v is the initial vertex of a_1 and the terminal vertex of a_m). Among the m arcs in W there are n − 1 arcs that enter each of n−1 vertices, other than v, for the first time. The sub digraph D_1 of these n−1 arcs together with the n vertices is a spanning arborescence of D, rooted at vertex v. Prepared by G. Appasami, Assistant professor, Dr. pauls Engineering College.

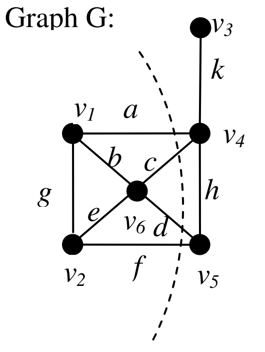
Proof : In the sub digraph D_1, vertex v is of in degree zero, and every other vertex is of indegree one, for D_1 includes exactly one arc going to each of the n−1 vertices and no arc going to v. Further, the way D_1 is defined in W, implies that D_1 is connected and contains n−1 arcs. Therefore D1 is a spanning arborescence in D and is rooted at v.

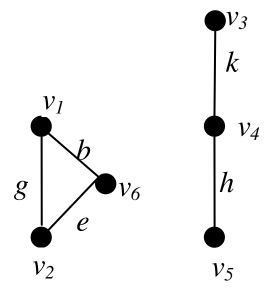
Illustration: In Figure, W = (b d c e f g h a) is an Euler line, starting and ending at vertex 2. The sub digraph {b, d, f } is a spanning arborescence rooted at vertex 2.



2. 4 CUT SETS

In a connected graph G, a cut-set is a set of edges whose removal from G leave the graph G disconnected.





Disconnected graph G with 2 components after removing cut set {a, c, d, f}

Possible cut sets are {a, c, d, f}, {a, b, e, f}, {a, b, g}, {d, h, f}, {k}, and so on.

{a, c, h, d} is not a cut set, because its proper subset {a, c, h} is a cut set.

{g, h} is not a cut set.

A minimal set of edges in a connected graph whose removal reduces the rank by one is called minimal cut set (simple cut-set or cocycle). Every edge of a tree is a cut set.

2. 5 PROPERTIES OF CUT SET

- Every cut-set in a connected graph G must contain at least one branch of every spanning tree of G.
- In a connected graph G, any minimal set of edges containing at least one branch of every spanning tree of G is a cut-set.
- Every circuit has an even number of edges in common with any cut set.

Properties of circuits and cut sets:

Every cut set of a connected graph G includes at least one branch from every spanning tree of G. (Counter hypothesis: Some cut set F of G does not include any branches of a spanning tree T. Then, T is a subgraph of G − F and G − F is connected.)

(b) Every circuit of a connected graph G includes at least one link from every co-spanning tree of G. (Counter hypothesis: Some circuit C of G does not include any link of a co-spanning tree T∗. Then, T = G − T∗ has a circuit and T is not a tree.)

Theorem : The edge set F of the connected graph G is a cut set of G if and only if

(i) F includes at least one branch from every spanning tree of G, and

(ii) if H ⊂ F, then there is a spanning tree none of whose branches is in H.

Proof. Let us first consider the case where F is a cut set. Then, (i) is true (previous proposition

(a). If H ⊂ F then G − H is connected and has a spanning tree T. This T is also a spanning tree of G. Hence, (ii) is true.

Let us next consider the case where both (i) and (ii) are true. Then G − F is disconnected.

If H ⊂ F there is a spanning tree T none of whose branches is in H. Thus T is a subgraph of G − H and G − H is connected. Hence, F is a cut set.

2. 6 ALL CUT SETS

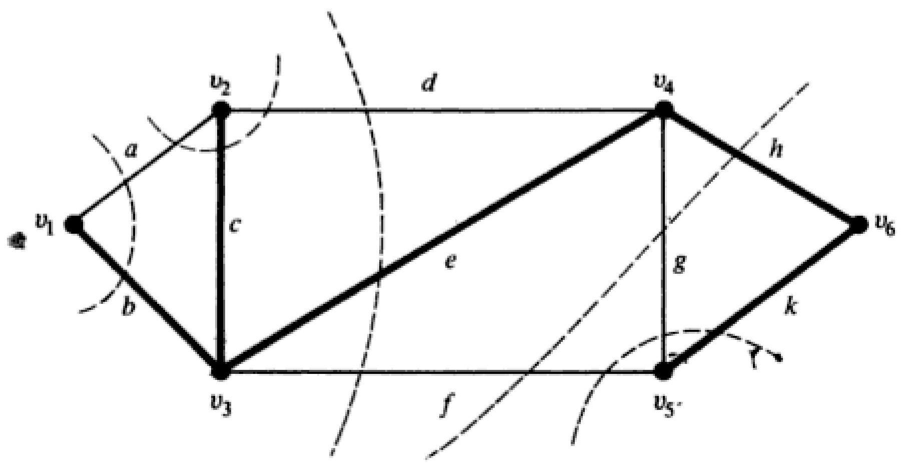
It was shown how cut-sets are used to identify weak spots in a communication net. For this purpose we list all cut-sets of the corresponding graph, and find which ones have the smallest number of edges. It must also have become apparent to you that even in a simple example, such as in Fig.

There is a large number of cut-sets, and we must have a systematic method of generating all relevant cut-sets. In the case of circuits, we solved a similar problem by the simple technique of finding a set of fundamental circuits and then realizing that other circuits in a graph are just combinations of two or more fundamental circuits.

We shall follow a similar strategy here. Just as a spanning tree is essential for defining a set of fundamental circuits, so is a spanning tree essential for a set of fundamental cut-sets. It will be beneficial for the reader to look for the parallelism between circuits and cut-sets. Fundamental Cut-Sets: Consider a spanning tree T of a connected graph G.

Take any branch b in T. Since {b) is a cut-set in T, (b) partitions all vertices of T into two disjoint sets—one at each end of b. Consider the same partition of vertices in G, and the cut set S in G that corresponds to this partition. Cut-set S will contain only one branch b of T, and the rest (if any) of the edges in S are chords with respect to T.

Such a cut-set S containing exactly one branch of a tree T is called a fundamental cut-set with respect to T. Sometimes a fundamental cut-set is also called a basic cut-set. Prepared by G. Appasami, Assistant professor, Dr. pauls Engineering College.



Fundamental cut sets of graph

T (in heavy lines) and all five of the fundamental cut-sets with respect to T are shown (broken lines "cutting" through each cut-set). Just as every chord of a spanning tree defines a unique fundamental cir-cuit, every branch of a spanning tree defines a unique fundamental cut-set.

It must also be kept in mind that the term fundamental cut-set (like the term fundamental circuit) has meaning only with respect to a given spanning tree. Now we shall show how other cut-sets of a graph can be obtained from a given set of cut-sets.

2. 7 FUNDAMENTAL CIRCUITS AND CUT SETS

Adding just one edge to a spanning tree will create a cycle; such a cycle is called a **fundamental cycle** (**Fundamental circuits**). There is a distinct fundamental cycle for each edge; thus, there is a one-to-one correspondence between fundamental cycles and edges not in the spanning tree. For a connected graph with V vertices, any spanning tree will have $V - 1$ edges, and thus, a graph of E edges and one of its spanning trees will have $E - V + 1$ fundamental cycles.

Dual to the notion of a fundamental cycle is the notion of a **fundamental cutset**. By deleting just one edge of the spanning tree, the vertices are partitioned into two disjoint sets. The fundamental cutset is defined as the set of edges that must be removed from the graph G to accomplish the same partition. Thus, each spanning tree defines a set of $V - 1$ fundamental cutsets, one for each edge of the spanning tree.

Consider a spanning tree T in a given connected graph G. Let c, be a chord with respect to T, and let the fundamental circuit made by e_i be called r, con-sisting of k branches b_1, b_2, . . . , b_4 in addition to the chord 4; that is, r = {b_1, b_2, b_3 ..... B_4} is a fundamental circuit with respect to T. Every branch of any spanning tree has a fundamental cut-set associated with it.

Let S_i be the fundamental cut-set associated with by consisting of q chords in addition to the branch b_1; that is,

S_i = {b_1, c_1 c_2 ..... C_4} is a fundamental cut-set with respect to T.

Because of above Theorem , there must be an even number of edges common to and S_i. Edge b, is in both Γ and S_i and there is only one other edge in Γ (which is c,) that can possibly also be in S_i. Therefore, we must have two edges b, and c, common to S, and r. Thus the chord c, is one of the chords c_1, , C_4.

Exactly the same argument holds for fundamental cut-sets associated with b_2,b_3, . , and b_k. Therefore, the chord c, is contained in every fundamental cut-set associated with branches in Γ . Is it possible for the chord c, to be in any other fundamental cut-set S' (with respect to T, of course) besides those associated with b_2,b_3, . and b_k. The answer is no.

Otherwise (since none of the branches in r are in S'), there would be only one edge c, common to S' and Γ , a contradiction to Theorem. Thus we have an important result.

THEOREM With respect to a given spanning tree T, a chord c_i that determines a fundamental circuit Γ occurs in every fundamental cut-set associated with the branches in Γ and in no other.

As an example, consider the spanning tree (b, c, e, h, k}, shown in heavy lines, in Fig. The fundamental circuit made by chord f is {f, e, h, k}.

The three fundamental cut-sets determined by the three branches e, h, and k are

determined by branch e: {d, e, f},

determined by branch h: {f, g, h},

determined by branch k: {f, g, k}.

Chord f occurs in each of these three fundamental cut-sets, and there is no other fundamental cut-set that contains f. The converse of above Theorem is also true.

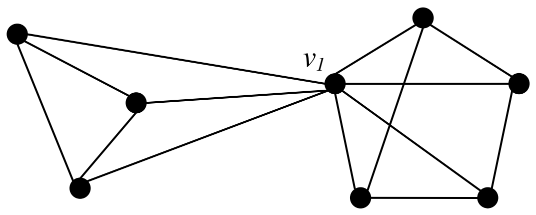
2. 8 CONNECTIVITY AND SEPARABILITY

edge Connectivity.

Each cut-set of a connected graph G consists of certain number of edges. The number of edges in the smallest cut-set is defined as the **edge Connectivity of G.**

The **edge Connectivity** of a connected graph G is defined as the minimum number of edges whose removal reduces the rank of graph by one.

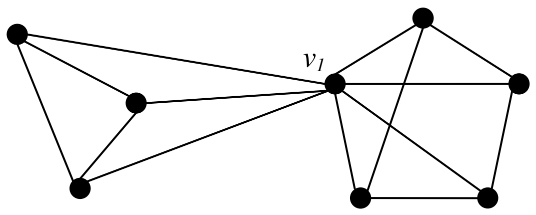
The edge Connectivity of a tree is one.



The edge Connectivity of the above graph G is three.

vertex Connectivity

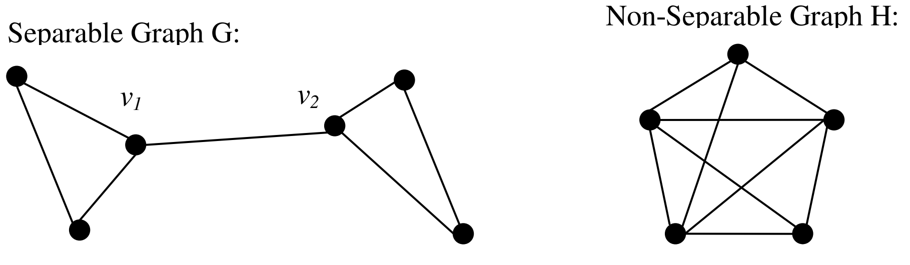
The **vertex Connectivity** of a connected graph G is defined as the minimum number of vertices whose removal from G leaves the remaining graph disconnected. The vertex Connectivity of a tree is one.



The vertex Connectivity of the above graph G is one.

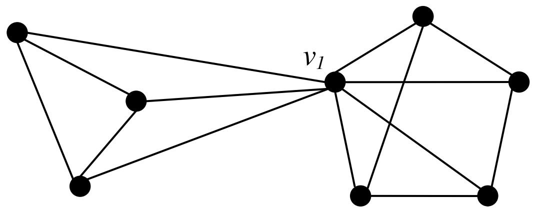
separable and non-separable graph.

A connected graph is said to be separable graph if its vertex connectivity is one. All other connected graphs are called non-separable graph.



articulation point.

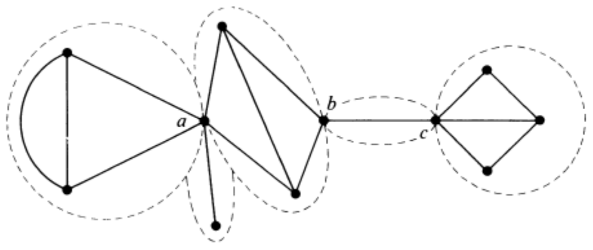
In a separable graph a vertex whose removal disconnects the graph is called *a cut-vertex, a cut-node, or an articulation point.*



v_1 is an articulation point.

component (or block) of graph.

A separable graph consists of two or more non separable subgraphs. Each of the largest nonseparable is called a block (or component).



The above graph has 5 blocks.

Lemmas:

If $S \subseteq V_G$ separates u and v, then every path P: u $\xrightarrow{\star}$ v visits a vertex of S.

If a connected graph G has no separating sets, then it is a complete graph.

Proof. If $v_G \leq 2$, then the claim is clear. For $V_G \geq 3$, assume that G is not complete,and let uv $\notin$ G. Now VG \ {u, v} is a separating set.

DEFINITION. The (vertex) connectivity number k(G) of G is defined as $k(G) = \min\{k \mid k = |S|, G{-}S$ disconnected or trivial, $S \subseteq V_G\}$.

A graph G is k-connected, if $k(G) \geq k$.

In other words,

• k(G) = 0, if G is disconnected,

• $k(G) = v_G - 1$, if G is a complete graph, and

• otherwise k(G) equals the minimum size of a separating set of G. Clearly, if G is connected, then it is 1-connected.

2. 9 NETWORK FLOWS

A **flow network** (also known as a transportation **network**) is a **graph** where each edge has a capacity and each edge receives a **flow**. The amount of **flow** on an edge cannot exceed the capacity of the edge.

The max. flow between two vertices = Min. of the capacities of all cut-sets.

Various transportation networks or water pipelines are conveniently represented by weighted directed graphs. These networks usually possess also some additional requirements.

Goods are transported from specific places (warehouses) to final locations (marketing places) through a network of roads.

In modeling a transportation network by a digraph, we must make sure that the number of goods remains the same at each crossing of the roads.

The problem setting for such networks was proposed by T.E. Harris in the 1950s. The connection to Kirchhoff's Current Law (1847) is immediate.

According to this law, in every electrical network the amount of current flowing in a vertex equals the amount flowing out that vertex.

Flows

A network N consists of

1) An **underlying digraph** D = (V, E),

2) Two distinct vertices s and r, called the source and the sink of N, and

3) A **capacity function** $\alpha : V \times V \to R_+$ (nonnegative real numbers), for which $\alpha(e) = 0$, if $e \notin E$.

Denote $V_N = V$ and $E_N = E$.

Let $A \subseteq V_N$ be a set of vertices, and $f : V_N \times V_N \to R$ any function such that $f(e) = 0$, if $e \notin N$. We adopt the following notations:

$[A, \overline{A}] = \{e \in D \mid e = uv, u \in A, v \notin A\}$,

$$f^+(A) = \sum_{e \in [A,\overline{A}]} f(e) \quad \text{and} \quad f^-(A) = \sum_{e \in [\overline{A},A]} f(e) .$$

In particular,

$$f^+(u) = \sum_{v \in N} f(uv) \quad \text{and} \quad f^-(u) = \sum_{v \in N} f(vu) .$$

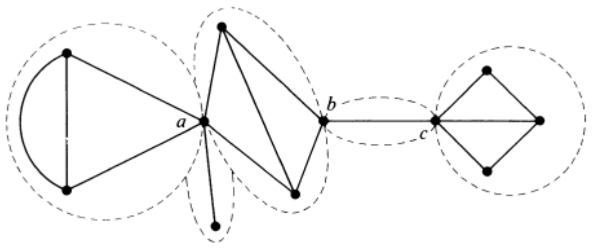
2. 10 1-ISOMORPHISM

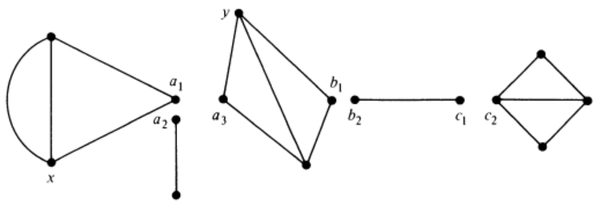
A graph G_1 was 1-Isomorphic to graph G_2 if the blocks of G_1 were isomorphic to the blocks of G_2.

Two graphs G_1 and G_2 are said to be 1-Isomorphic if they become isomorphic to each other under repeated application of the following operation.

Operation 1: "Split" a cut-vertex into two vertices to produce two disjoint subgraphs.

Graph G_1: Graph G_2:





Graph G_1 is 1-Isomorphism with Graph G_2.

A separable graph consists of two or more non separable subgraphs. Each of the largest non separable subgraphs is called a block.

(Some authors use the term component, but to avoid confusion with components of a disconnected graph, we shall use the term block.) The graph in Fig has two blocks. The graph in Fig has five blocks (and three cut-vertices a, b, and c); each block is shown enclosed by a broken line.

Note that a non separable connected graph consists of just one block. Visually compare the disconnected graph in Fig. with the one in Fig. These two graphs are certainly not isomorphic (they do not have the same number of vertices), but they are related by the fact that the blocks of the graph in Fig.are isomorphic to the components of the graph in above Fig. Such graphs are said to be I-isomorphic.

More formally: Two graphs G_1 and G_2 are said to be I-isomorphic if they become isomorphic to each other under repeated application of the following operation. Operation I: "Split" a cut-vertex into two vertices to produce two disjoint subgraphs. From this definition it is apparent that two nonseparable graphs are 1-isomorphic if and only if they are isomorphic.

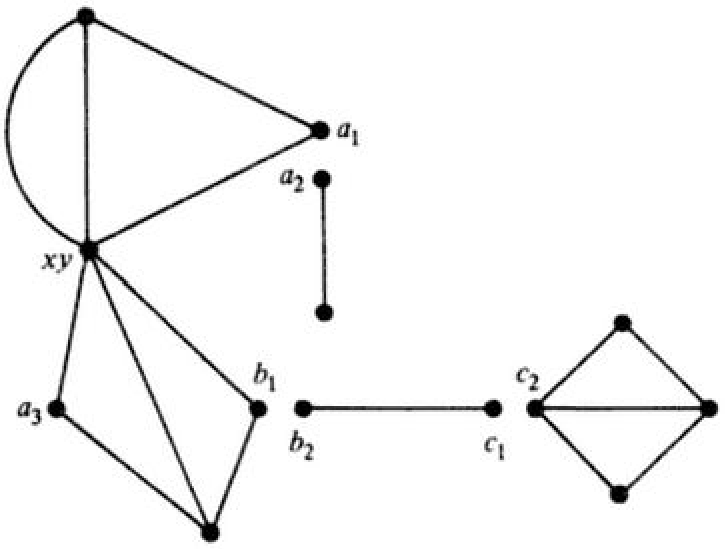
THEOREM If G_I and G_2 arc two 1-isomorphic graphs, the rank of GI equals the rank of C2 and the nullity of CI equals the nullity of G_2.

Proof: Under operation 1, whenever a cut-vertex in a graph G is "split" into two vertices, the number of components in G increases by one. Therefore, the rank of G which is number of vertices in C - number of components in G remains invariant under operation 1.

Also, since no edges are destroyed or new edges created by operation I, two 1-isomorphic graphs have the same number of edges.

Two graphs with equal rank and with equal numbers of edges must have the same nullity, because nullity = number of edges -- rank. What if we join two components of Fig by "gluing" together two vertices (say vertex x to y)? We obtain the graph shown in Fig. Clearly, the graph in Fig is 1-isomorphic to the graph in Fig.

Since the blocks of the graph in Fig are isomorphic to the blocks of the graph in Fig, these two graphs are also 1-isomorphic. Thus the three graphs in above Figs. are 1-isomorphic to one another.

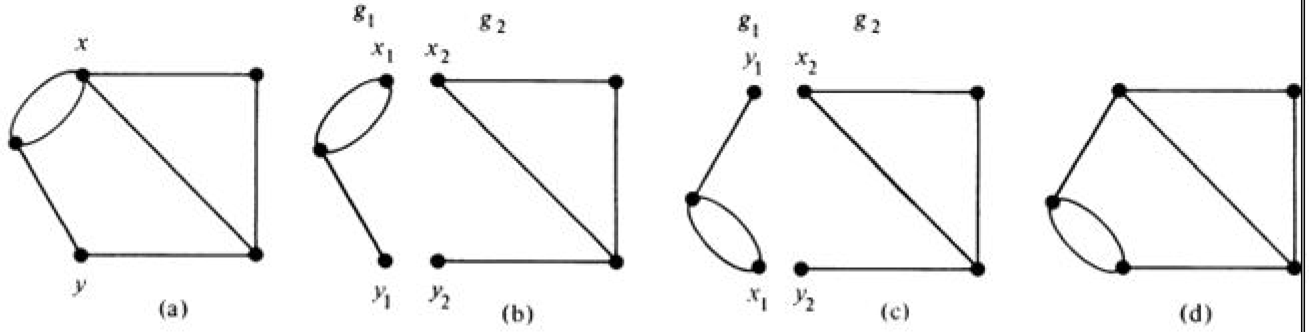


2. 11 2-ISOMORPHISM

Two graphs G_1 and G_2 are said to be **2-Isomorphic** if they become isomorphic after undergoing *operation 1* or *operation 2*, or both operations any number of times.

Operation 1: "Split" a cut-vertex into two vertices to produce two disjoint subgraphs.

Operation 2: "Split" the vertex x into x_1 and x_2 and the vertex y into y_1 and y_2 such that G is split into g_1 and g_2. Let vertices x_1 and y_1 go with g_1 and vertices x_2 and y_2 go with g_2. Now rejoin the graphs g_1 and g_2 by merging x_1 with y_2 and x_2 with y_1.



2. 12 COMBINATIONAL AND GEOMETRIC GRAPHS

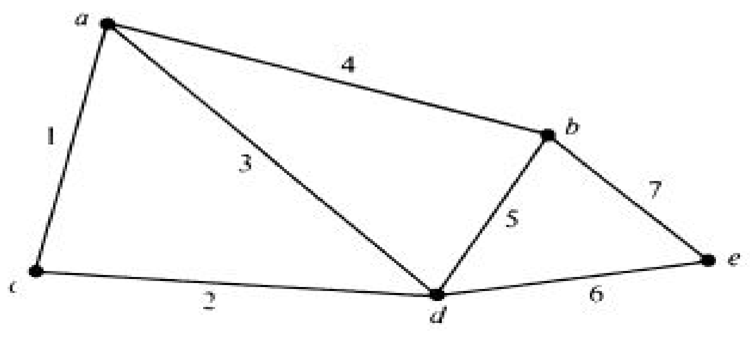
An abstract graph G can be defined as G = (V, E, Ψ)

Where the set V consists of five objects named *a, b, c, d*, and *e*, that is, $V = \{ a, b, c, d, e \}$ and the set E consist of seven objects named 1, 2, 3, 4, 5, 6, and 7, that is, $E = \{ 1, 2, 3, 4, 5, 6, 7\}$, and the relationship between the two sets is defined by the mapping Ψ, which consist of

Ψ= [1→(a, c), 2→(c, d) , 3→(a, d) , 4→(a, b) , 5→(b, d) , 6→(d, e) , 7→(b, e)].

Here the symbol 1→*(a, c)*, says that object 1 from set E is mapped onto the pair (a, c) of objects from set V.

This combinatorial abstract object G can also be represented by means of a geometric figure.



The figure is one such geometric representation of this graph G.

Any graph can be geometrically represented by means of such configuration in three dimensional Euclidian space.

2. 13 PLANER GRAPHS

A graph is said to be planar if it can be drawn in the plane in such a way that no two edges intersect each other. Drawing a graph in the plane without edge crossing is called embedding the graph in the plane (or planar embedding or planar representation).

Given a planar representation of a graph G, a face (also called a region) is a maximal section of the plane in which any two points can be joint by a curve that does not intersect any part of G.

When we trace around the boundary of a face in G, we encounter a sequence of vertices and edges, finally returning to our final position. Let $v_1, e_1, v_2, e_2, \ldots, v_d, e_d, v_1$ be the sequence obtained by tracing around a face, then d is the degree of the face.

Some edges may be encountered twice because both sides of them are on the same face. A tree is an extreme example of this: each edge is encountered twice.

The following result is known as Euler's Formula.

PLANAR GRAPHS:

It has been indicated that a graph can be represented by more than one geometrical drawing. In some drawing representing graphs the edges intersect (cross over) at points which are not vertices of the graph and in some others the edges meet only at the vertices.

A graph which can be represented by at least one plane drawing in which the edges meet only at vertices is called a 'planar graph'.

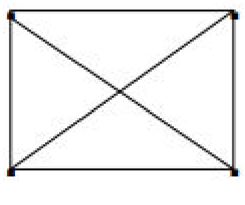
On the other hand, a graph which cannot be represented by a plane drawing in which the edges meet only at the vertices is called a non planar graph.

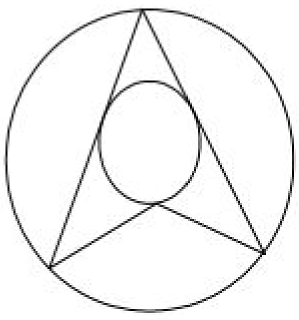
In other words, a non planar graph is a graph whose every possible plane drawing contains at least two edges which intersect each other at points other than vertices.

Example 1

Show that (a) a graph of order 5 and size 8, and (b) a graph of order 6 and size 12, are planar graphs.

Solution: A graph of order 5 and size 8 can be represented by a plane drawing.





Graph(a)Graph(b)

In which the edges of the graph meet only at the vertices, as shown in fig a. therefore, this graph is a planar graph.

Similarly, fig. b shows that a graph of order 6 and size 12 is a planar graph.

The plane representations of graphs are by no means unique. Indeed, a graph G can be drawn in arbitrarily many different ways.

Also, the properties of a graph are not necessarily immediate from one representation, but may be apparent from another.There are, however, important families of graphs, the surface graphs, that rely on the (topological or geometrical) properties of the drawings of graphs.

We restrict ourselves in this chapter to the most natural of these, the planar graphs. The geometry of the plane will be treated intuitively.

A planar graph will be a graph that can be drawn in the plane so that no two edges intersect with each other.

Such graphs are used, e.g., in the design of electrical (or similar) circuits, where one tries to (or has to) avoid crossing the wires or laser beams.

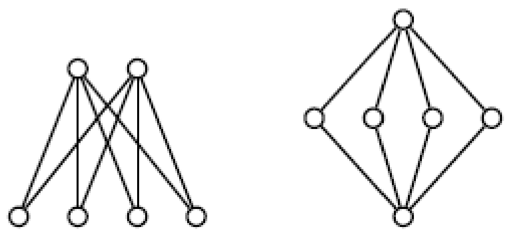
Planar graphs come into use also in some parts of mathematics, especially in group theory and topology.

There are fast algorithms (linear time algorithms) for testing whether a graph is planar or not. However, the algorithms are all rather difficult to implement. Most of them are based on an algorithm.

Definition

A graph G is a **planar graph**, if it has a plane figure P(G), called the **plane embedding** of G,where the lines (or continuous curves) corresponding to the edges do not intersect each other except at their ends.

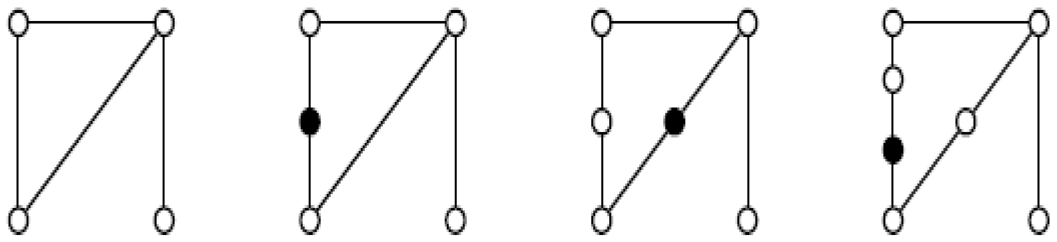
The complete bipartite graph $K_{2,4}$ is a planar graph.



Bipartite graph

An edge e = uv ∈ G is **subdivided**, when it is replaced by a path u −→ x −→ v of length two by introducing a new vertex x.

A **subdivision** H of a graph G is obtained from G by a sequence of subdivisions.



Graph

The following result is clear.

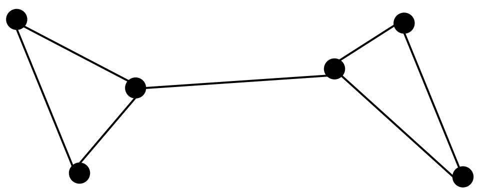
2. 14 DIFFERENT REPRESENTATION OF A PLANER GRAPH

between Planar and non-planar graphs

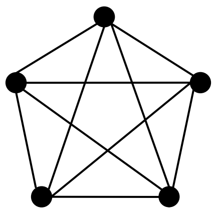
A graph G is said to be *planar* if there exists some geometric representation of G which can be drawn on a plan such that no two of its edges intersect.

A graph that cannot be drawn on a plan without crossover its edges is called *non-planar.*

Planar Graph G:



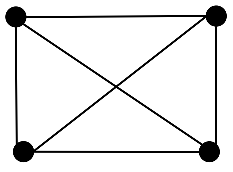
Non-planar Graph H:



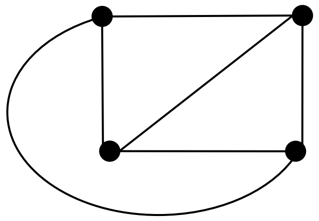
embedding graph.

A drawing of a geometric representation of a graph on any surface such that no edges intersect is called embedding.

Graph G:

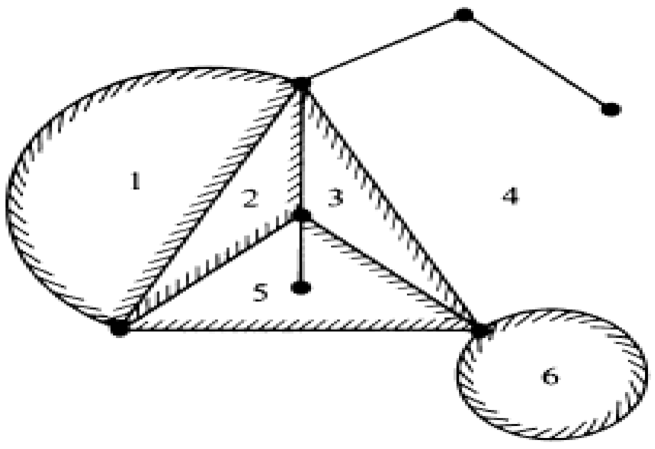


Embedded Graph G:



region in graph.

In any planar graph, drawn with no intersections, the edges divide the planes into different **regions (windows, faces, or meshes)**. The regions enclosed by the planar graph are called **interior faces** of the graph. The region surrounding the planar graph is called the **exterior** (or infinite or unbounded) face of the graph. Prepared by G. Appasami, Assistant professor, Dr. pauls Engineering College.



The graph has 6 regions.

graph embedding on sphere.

To eliminate the distinction between finite and infinite regions, a planar graph is often embedded in the surface of sphere. This is done by stereographic projection.

UNIT III MATRICES, COLOURING AND DIRECTED GRAPH

3. 1 CHROMATIC NUMBER

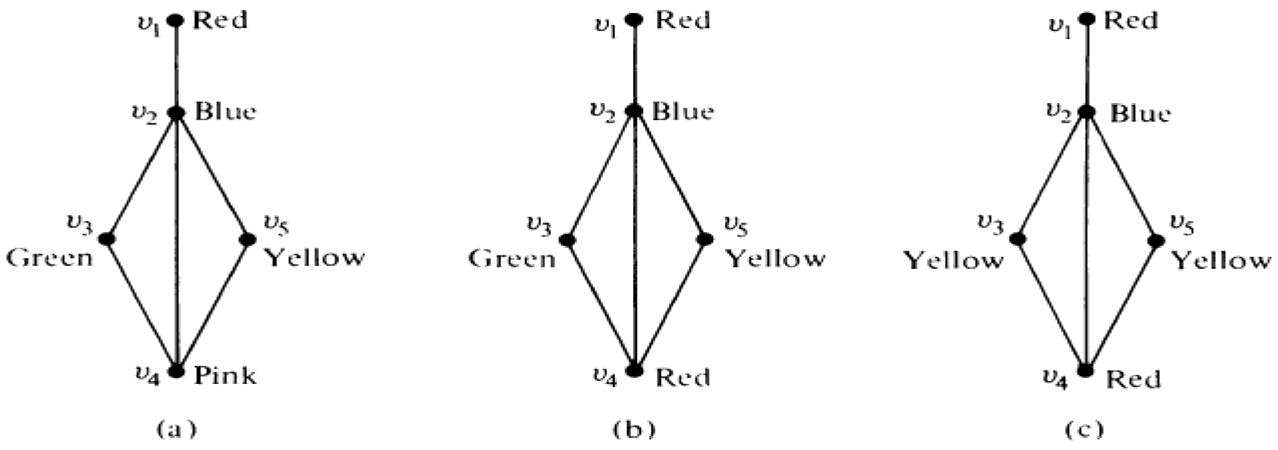
1. proper coloring

Painting all the vertices of a graph with colors such that no two adjacent vertices have the same color is called the *proper coloring* (simply *coloring*) of a graph. A graph in which every vertex has been assigned a color according to a proper coloring is called a *properly colored graph.*

2. Chromatic number

A graph G that requires k different colors for its proper coloring, and no less, is called *k-chromatic* graph, and the number k is called the *chromatic number* of G.

The minimum number of colors required for the proper coloring of a graph is called *Chromatic number.*



The above graph initially colored with 5 different colors, then 4, and finally 3. So the chromatic number is 3. i.e., The graph is 3-chromatic

3. properties of chromatic numbers (observations).

- A graph consisting of only isolated vertices is 1-chromatic.
- Every tree with two or more vertices is 2-chromatic.
- A graph with one or more vertices is at least 2-chromatic.
- A graph consisting of simply one circuit with n ≥ 3 vertices is 2-chromatic if n is even and 3-chromatic if n is odd.
- A complete graph consisting of n vertices is n-chromatic.

SOME RESULTS:

i) A graph consisting of only isolated vertices (ie., Null graph) is 1–Chromatic (Because no two vertices of such a graph are adjacent and therefore we can assign the same color to all vertices).

ii) A graph with one or more edges is at least 2 -chromatic (Because such a graph has at least one pair of adjacent vertices which should have different colors).

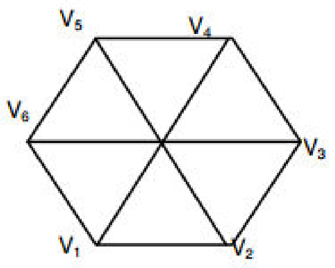
iii) If a graph G contains a graph G1 as a subgraph, then $\chi(G) \geq \chi(G_1)$.

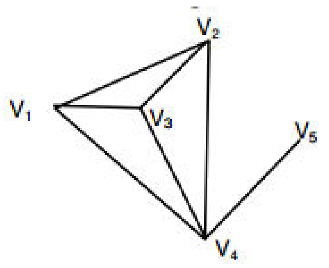
iv. If G is a graph of n vertices, then $\chi(G) \leq n$.

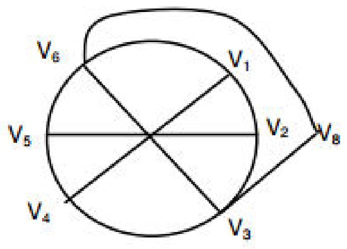
v. (Kn) = n, for all n 1. (Because, in Kn, every two vertices are adjacent and as such all the n vertices should have different colors)

vi. If a graph G contains Kn as a subgraph, then $\chi(G) \geq n$.

Example 1: Find the chromatic number of each of the following graphs.







Solution

i) For the graph (a), let us assign a color a to the vertex V_1, then for a proper coloring, we have to assign a different color to its neighbors V_2,V_4,V_6, since V_2, V_4, V6are mutually non-adjacent vertices, they can have the same color as V_1, namely α.

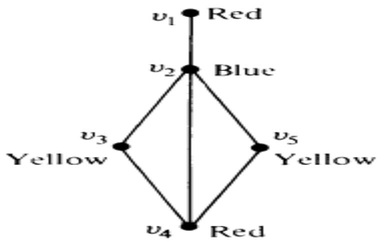
Thus, the graph can be properly colored with at lest two colors, with the vertices V_1,V_3,V_5 having one color α and V_2,V_4,V_6 having a different color β. Hence, the chromatic number of the graph is 2.

ii) For the graph (b) , let us assign the color α to the vertex V_1. Then for a proper coloring its neighbors V_2,V_3 & V_4 cannot have the color α. Further more, V_2, V_3,V_4 must have different colors, say β, γ , δ .Thus, at least four colors are required for a proper coloring of the graph. Hence the chromatic number of the graph is 4.

iii) For the graph (c) , we can assign the same color, say α, to the non-adjacent vertices V_1,V_3, V_5. Then the vertices V_2,V_4,V_6consequently $V_7$and V_8 can be assigned the same color which is different from both α and β. Thus, a minimum of three colors are needed for a proper coloring of the graph. Hence its chromatic number is 3. Prepared by G. Appasami, Assistant professor, Dr. pauls Engineering College.

3. 2 CHROMATIC PARTITIONING

A proper coloring of a graph naturally induces a partitioning of the vertices into different subsets based on colors.



For example, the coloring of the above graph produces the portioning $\{v_1, v_4\}$, $\{v_2\}$, and $\{v_3, v_5\}$.

A proper coloring of a graph naturally induces a partitioning of the vertices into different subsets. For example, the coloring in Fig. produces the partitioning

$\{v_1, v_4\}$, $\{v_2\}$, and $\{v_3, v_5\}$.

No two vertices in any of these three subsets are adjacent. Such a subset of vertices is called an independent set; more formally:

A set of vertices in a graph is said to be an independent set of vertices or simply an independent set (or an internally stable set) if no two vertices in the set arc adjacent.

For example: in Fig., {a, c, d} is an independent set. A single vertex in any graph constitutes an independent set. A maximal independent set (or maximal internally stable set) is an independent set to which no other vertex can be added without destroying its independence property.

The set {a, c, d, f} in Fig. is a maximal independent set.

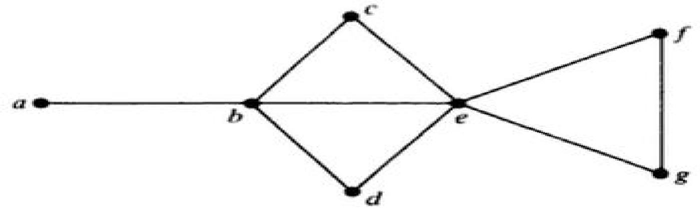
The set lb,/ I is another maximal independent set.

The set {b, g} is a third one. From the preceding example, it is clear that a graph, in general, has many maximal independent sets; and they may be of different sizes.

Among all maximal independent sets, one with the largest number of vertices is often of particular interest . Suppose that the graph in Fig. describes the following problem.

Each of the seven vertices of the graph is a possible code word to be used in some communication. Some words are so close (say, in sound) to others that they might be confused for each other. Pairs of such words that may be mistaken for one another are joined by edges. Find a largest set of code words for a reliable communication.

This is a problem of finding a maximal independent set with largest number of vertices. In this simple example, {a, c, d, f} is an answer.



Chromatic partitioning graph

The number of vertices in the largest independent set of a graph G is called the independence number (or coefficient of internal stability), β(G).

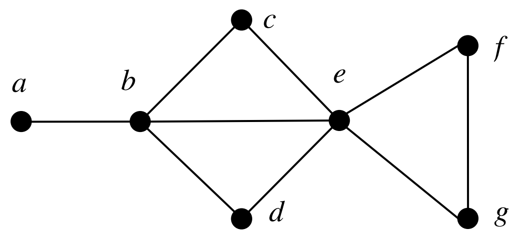
Consider a K-chromatic graph G of n vertices properly colored with different colors.

Since the largest number of vertices in G with the same color cannot exceed the independence number β(G), we have the inequality

$$\beta(G) \geq \frac{n}{K}.$$

3. 3 CHROMATIC POLYNOMIAL

A set of vertices in a graph is said to be an *independent set* of vertices or simply independent set (or an internally stable set) if two vertices in the set are adjacent.



For example, in the above graph produces {a, c, d} is an independent set.

A single vertex in any graph constitutes an independent set.

A maximal independent set is an independent set to which no other vertex can be added without destroying its independence property.

{a, c, d, f} is one of the maximal independent set. {b, f} is one of the maximal independent set.

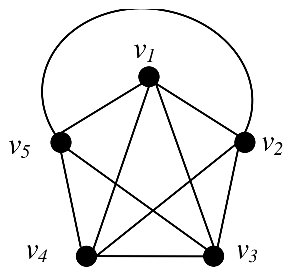
The number of vertices in the largest independent set of a graph G is called the *independence number* (or coefficients of internal stability), denoted by β(G).

For a K-chromatic graph of n vertices, the independence number β(G) $\geq\frac{n}{k}$.

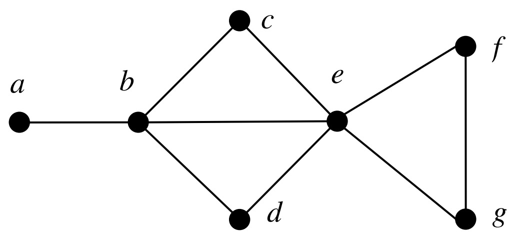
Uniquely colorable graph.

A graph that has only one chromatic partition is called a uniquely colorable graph. For example,

Uniquely colorable graph G:

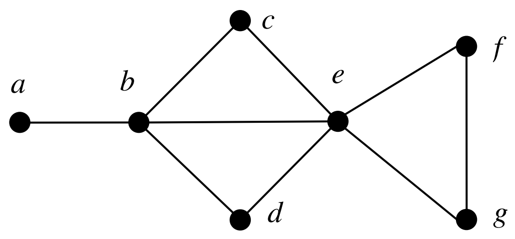


Not uniquely colorable graph H:



Dominating set.

A dominating set (or an externally stable set) in a graph G is a set of vertices that dominates every vertex v in G in the following sense: Either v is included in the dominating set or is adjacent to one or more vertices included in the dominating set.



{b, g} is a dominating set, {a, b, c, d, f} is a dominating set. A is a dominating set need not be independent set. Set of all vertices is a dominating set.

A minimal dominating set is a dominating set from which no vertex can be removed without destroying its dominance property.

{b, e} is a minimal dominating set.

Chromatic polynomial.

A graph G of n vertices can be properly colored in many different ways using a sufficiently large number of colors. This property of a graph is expressed elegantly by means of polynomial. This polynomial is called the Chromatic polynomial of G.

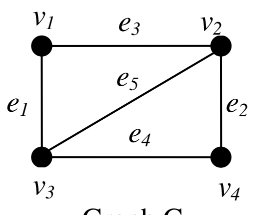
The value of the Chromatic polynomial $P_n(\lambda)$ of a graph with n vertices the number of ways of properly coloring the graph , using λ or fewer colors.

3. 4 MATCHING

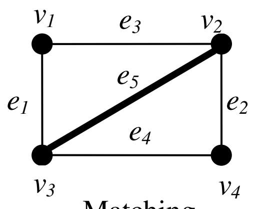
A *matching* in a graph is a subset of edges in which no two edges are adjacent. A single edge in a graph is a matching.

A *maximal matching* is a matching to which no edge in the graph can be added.

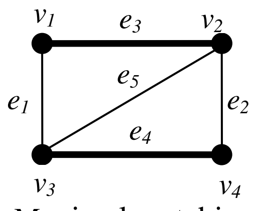
The maximal matching with the largest number of edges are called the *largest maximal matching*.



Graph G



Matching



Maximal matching

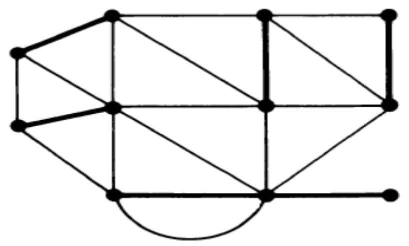
3. 5 COVERING

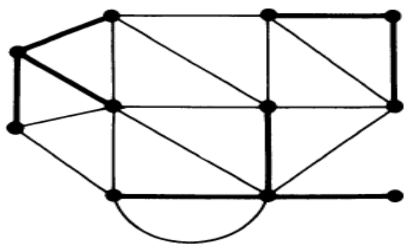
A set g of edges in a graph G is said to be cover og G if every vertex in G is incident on at least one edge in g. A set of edges that covers a graph G is said to be a covering (or an edge covering, or a coverring subgraph) of G.

Every graph is its own covering.

A spanning tree in a connected graph is a covering.

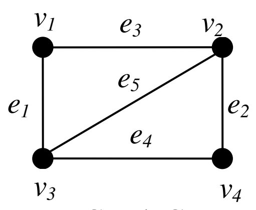
A Hamiltonian circuit in a graph is also a covering.



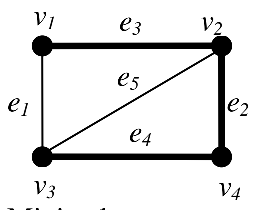


Minimal cover.

A **minimal covering** is a covering from which no edge can be removed without destroying it ability to cover the graph G.



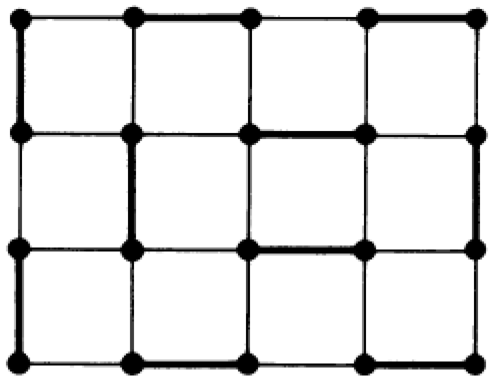
Graph G

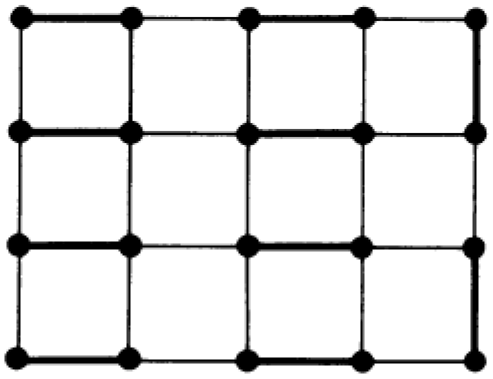


Minimal cover

Dimer covering

A covering in which every vertex is of degree one is called a *dimer covering* or a *1-factor*. A dimmer covering is a maximal matching because no two edges in it are adjacent.


(a)


(b)

Two dimmer coverings.

Let G = (V, E) be a graph. A stable set is a subset C of V such that e ⊆ C for each edge e of G. A vertex cover is a subset W of V such that e ∩ W = ∅ for each edge e of G. It is not difficult to show that for each U ⊆ V :

U is a stable set ⇐⇒ V \ U is a vertex cover.

A matching is a subset M of E such that e∩e ′ = ∅ for all e, e′ ∈ M with e = e ′ . A matching is called perfect if it covers all vertices (that is, has size 1 2 |V |). An edge cover is a subset F of E such that for each vertex v there exists e ∈ F satisfying v ∈ e. Note that an edge cover can exist only if G has no isolated vertices.

Define:

α(G) := max{|C| | C is a stable set},

τ (G) := min{|W| | W is a vertex cover},

ν(G) := max{|M| | M is a matching},

ρ(G) := min{|F| | F is an edge cover}.

These numbers are called the stable set number, the vertex cover number, the matching number, and the edge cover number of G, respectively.

It is not difficult to show that: (3) $\alpha(G) \leq \rho(G)$ and $\nu(G) \leq \tau(G)$. The triangle K3 shows that strict inequalities are possible.

Theorem 1(Gallai's theorem). If G = (V, E) is a graph without isolated vertices, then $\alpha(G) + \tau(G) = |V| = \nu(G) + \rho(G)$.

Proof.

The first equality follows directly from (1).

To see the second equality, first let M be a matching of size $\nu(G)$. For each of the $|V| - 2|M|$ vertices v missed by M, add to M an edge covering v. We obtain an edge cover F of size $|M|+(|V|-2|M|) = |V|-|M|$. Hence $\rho(G) \leq |F| = |V|-|M| = |V|-\nu(G)$.

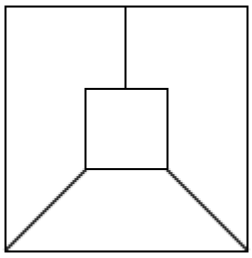
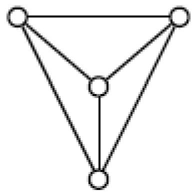
Second, let F be an edge cover of size $\rho(G)$. Choose from each component of the graph (V, F) one edge, to obtain a matching M. As (V, F) has at least $|V| - |F|$ components , we have $\nu(G) \geq |M| \geq |V| - |F| = |V| - \rho(G)$.

This proof also shows that if we have a matching of maximum cardinality in any graph G, then we can derive from it a minimum cardinality edge cover, and conversely.

3. 6 FOUR COLOR PROBLEM

- Every planar graph has a chromatic number of four or less.
- Every triangular planar graph has a chromatic number of four or less.
- The regions of every planar, regular graph of degree three can be colored properly with four colors.
- **4-Colour Theorem:**
- If G is a planar graph, then $\chi(G) \leq 4$. By the following theorem, each planar graph can be decomposed into two bipartite graphs.
- Let G = (V, E) be a 4-chromatic graph, $\chi(G) \leq 4$.
- Then the edges of G can be partitioned into two subsets E_1 and E_2 such that (V, E_1) and (V, E_2) are both bipartite.
- **Proof.** Let $V_i = \alpha^{-1}(i)$ be the set of vertices coloured by i in a proper 4-colouring a of G.
- The define E_1 as the subset of the edges of G that are between the sets $V_1$and V_2;V_1 and V_4; V_3 and V_4.
- Let E_2 be the rest of the edges, that is, they are between the sets V_1 and V_3; V_2 and V_3; V_2 and V_4. It is clear that (V, E_1) and (V, E_2) are bipartite, since the sets V_i are stable.

- **Map colouring***
- The 4-Colour Conjecture was originally stated for maps.
- In the map-colouring problem we are given several countries with common borders and we wish to colour each country so that no neighboring countries obtain the same colour.
- How many colors are needed?
- A border between two countries is assumed to have a positive length in particular,countries that have only one point in common are not allowed in the map colouring.
- Formally, we define a map as a connected planar (embedding of a) graph with no bridges. The edges of this graph represent the boundaries between countries.
- Hence a country is a face of the map, and two neighbouring countries share a common edge (not just a single vertex). We deny bridges, because a bridge in such a map would be a boundary inside a country.
- The map-colouring problem is restated as follows:
- How many colours are needed for the faces of a plane embedding so that no adjacent faces obtain the same colour.
- The illustrated map can be 4-coloured, and it cannot be coloured using only 3 colours, because every two faces have a common border.

-
- Colour map
- Let $F_1, F_2, \dots, F_n$ be the countries of a map M, and define a graph G with $V_G = \{v_1, v_2, \dots, v_n\}$ such that $v_iv_j \in G$ if and only if the countries F_i and $F_j$are neighbour's.
- It is easy to see that G is a planar graph. Using this notion of a dual graph, we can state the map-colouring problem in new form:
- What is the chromatic number of a planar graph? By the 4-Colour Theorem it is at most four.
- Map-colouring can be used in rather generic topological setting, where the maps are defined by curves in the plane.

- As an example, consider finitely many simple closed curves in the plane. These curves divide the plane into regions. The regions are 2-colourable.
- That is, the graph where the vertices correspond to the regions, and the edges correspond to the neighbourhood relation, is bipartite.
- To see this, colour a region by 1, if the region is inside an odd number of curves, and, otherwise, colour it by 2.

State five color theorem

Every planar map can be properly colored with five colors.

i.e., the vertices of every plannar graph can be properly colored with five colors.

Vertex coloring and region coloring.

A graph has a dual if and only if it is planar. Therefore, coloring the regions of a planar graph G is equivalent to coloring the vertices of its dual G* and vice versa.

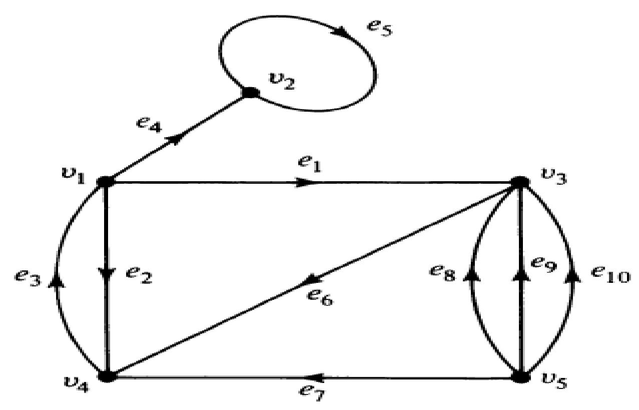
Regularization of a planar graph

- Remove every vertex of degree one from the graph G does not affect the regions of a plannar graph.
- Remove every vertex of degree two and merge the two edges in series from the graph G.
- Such a transformation may be called regularization of a planar graph.

3. 7 DIRECTED GRAPHS

A *directed graph* (or a *digraph, or an oriented graph*) G consists of a set of vertices $V = \{ v_1, v_2, \ldots \}$, a set of edges $E = \{ e_1, e_2, \ldots \}$, and a mapping Ψ that maps every edge onto some ordered pair of vertices (v_i, v_j).

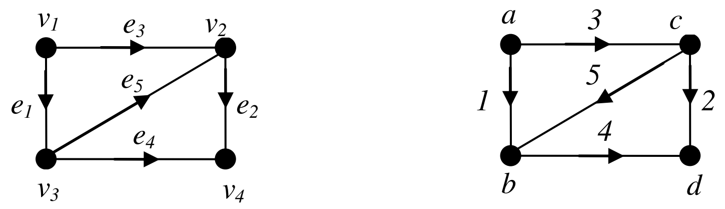
For example,



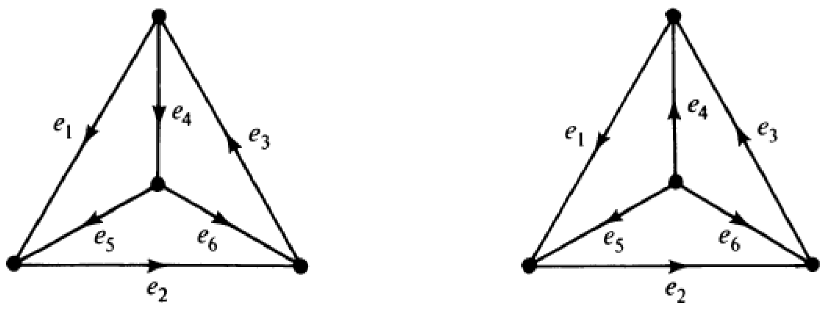
isomorphic digraph.

Among directed graphs, if their labels are removed, two isomorphic graphs are indistinguishable then these graphs are **isomorphic digraph**.

For example,



Two isomorphic digraphs.



Two non-isomorphic digraphs.

3. 8 TYPES OF DIRECTED GRAPHS

Like undirected graphs , digraphs are also has so many verities. In fact, due to the choice of assigning a direction to each edge, directed graphs have more varieties than undirected ones.

Simple Digraphs:

A digraph that has no self-loop or parallel edges is called a simple digraph .

Asymmetric Digraphs:

Digraphs that have at most one directed edge between a pair of vertices, but are allowed to have self-loops, are called asymmetric or antisymmetric.

Symmetric Digraphs:

Digraphs in which for every edge (a, b) (i.e., from vertex a to b) there is also an edge (b, a). A digraph that is both simple and symmetric is called a simple symmetric digraph. Similarly, a digraph that is both simple and asymmetric is simple asymmetric.

The reason for the terms symmetric and asymmetric will be apparent in the context of binary relations.

Complete Digraphs:

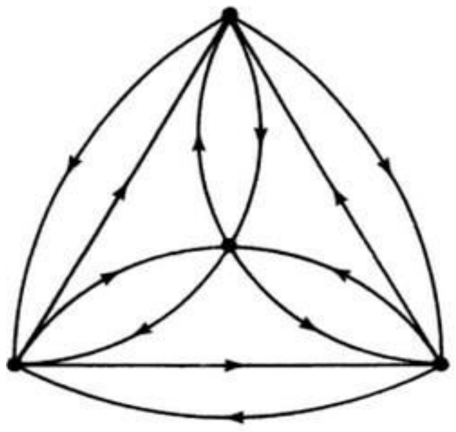
A complete undirected graph was defined as a simple graph in which every vertex is joined to every other vertex exactly by one edge.

For digraphs we have two types of complete graphs.

A complete symmetric digraph is a simple digraph in which there is exactly one edge directed from every vertex to every other vertex, and a complete asymmetric digraph is an asymmetric digraph in which there is exactly one edge between every pair of vertices.

A complete asymmetric digraph of n vertices contains n(n - 1)/2 edges, but a complete symmetric digraph of n vertices contains n(n - 1) edges. A complete asymmetric digraph is also called a tournament or a complete tournament (the reason for this term will be made clear).

A digraph is said to be balanced if for every vertex v, the in-degree equals the out-degree; that is, d+(vi) = div,). (A balanced digraph is also referred to as a pseudo symmetric digraph. or an isograph.) A balanced digraph is said to be regular if every vertex has the same in-degree and out-degree as every other vertex.



Complete symmetric digraph of four vertices

3. 9 DIGRAPHS AND BINARY RELATIONS

In a set of objects, X, where X={x_1, x_2, …}, A *binary relation R* between pairs (x_i, x_j) can be written as $x_i\ R\ x_j$ and say that x_i has relation *R* to x_j.

If the binary relation *R* is reflexive, symmetric, and transitive then *R* is an equivalence relation. This produces equivalence classes.

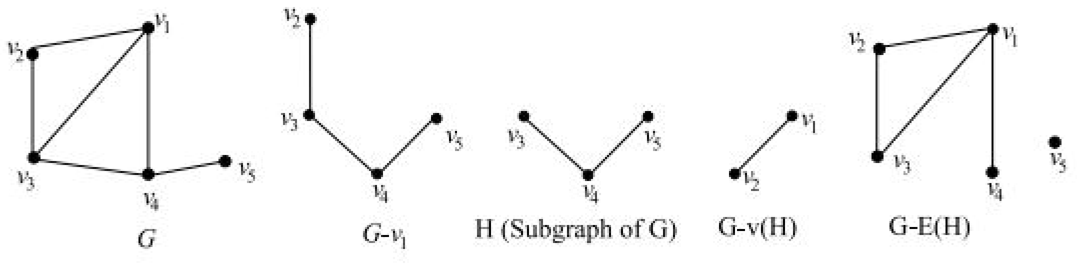
Let A and B be nonempty sets. A (binary) relation R from A to B is a subset of A x B. If R $\subseteq$ AxB and(a,b) $\in$ R, where a $\in$ A, b $\in$ B, we say a "is related to" b by R, and we write aRb. If a is not related to b by R, we write a 0 b. A relation R defined on a set X is a subset of X x X.

For example, less than, greater than and equality are the relations in the set of real numbers. The property "is congruent to" defines a relation in the set of all triangles in a plane. Also, parallelism defines a relation in the set of all lines in a plane.

Let R define a relation on a non empty set X. If R relates every element of X to itself, the relation R is said to be reflexive. A relation R is said to be symmetric if for all $x_i\, x_j$ $\in$ X, x_i R xj implies x_j R x_i. A relation R is said to be transitive if for any three elements x_i, x_j and x_k in X, x,Rx, and x_j R x_k imply xi /2 x_k. A binary relation is called an equivalence relation if it is reflexive, symmetric and transitive.

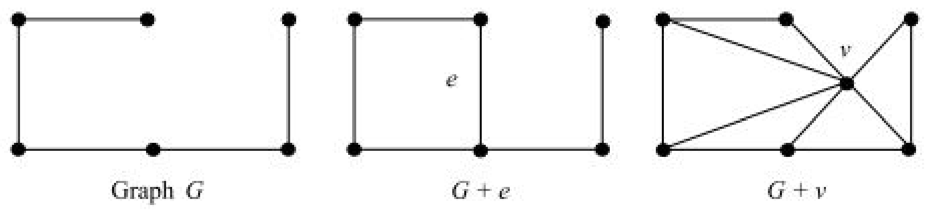
A binary relation Ron a set X can always be represented by a digraph. In such a representation, each x_j E X is represented by a vertex xi and whenever there is a relation R from xi to x_j, an arc is drawn from xi to x_j, for every pair (x_i, x_j). The digraph in Figure represents the relation is less than, on a set consisting of four numbers 2, 3, 4, 6.

We note that every binary relation on a finite set can be represented by a digraph without parallel edges and vice versa. Clearly, the digraph of a reflexive relation contains a loop at every vertex Fig. A digraph representing a reflexive binary relation is called a reflexive digraph.



Example

The digraph of a symmetric relation is a symmetric digraph because for every arc from x_i to x_j , there is an arc from x_j to x_i . Figure shows the digraph of an irreflexive and symmetric relation on a set of three elements.

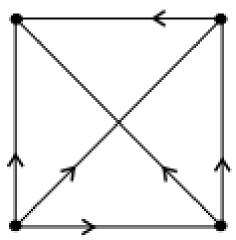


Graph G $G+e$ $G+v$

(a) (b) (c)

Example

A digraph representing a transitive relation on its vertex set is called a transitive digraph. Figure shows the digraph of a transitive, which is neither reflexive, nor symmetric.



Example

A binary relation R on a set M can also be represented by a matrix, called a relation matrix. This is a (0, 1), n×n matrix $M_R = [m_{i\,j}]$, where n is the number of elements in M, and is defined by

$$m_{ij} = \begin{cases} 1 & if\ x_i\ R\ x_j\ is\ true, \\ 0, & otherwise. \end{cases}$$

In some problems the relation between the objects is not symmetric. For these cases we need directed graphs, where the edges are oriented from one vertex to another.

Definitions

A digraph (or a directed graph) $D = (V_D, E_D)$ consists of the vertices V_D and (directed) edges $E_D \subseteq V_D \times V_D$ (without loops vv).

We still write uv for (u, v), but note that now uv $\neq$ vu.

For each pair e = uv define the inverse of e as e^{-1} = vu (= (v, u)).

Note that $e \in D$ does not imply $e^{-1} \in D$.

Let D be a digraph. Then A is its further classified into:

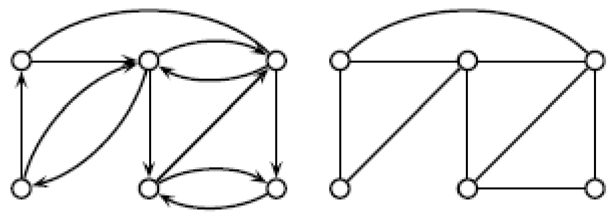
Subdigraph, if $V_A \subseteq V_D$ and $E_A \subseteq E_D$,

Induced subdigraph, A = D[X], if VA = X and $E_A = E_D \cap (X \times X)$.

The **underlying graph** U(D) of a digraph D is the graph on V_D such that if e ∈ D, then the undirected edge with the same ends is in U(D).

A digraph D is an **orientation** of a graph G, if G = U(D) and e ∈ D implies e^{-1} /∈ D.

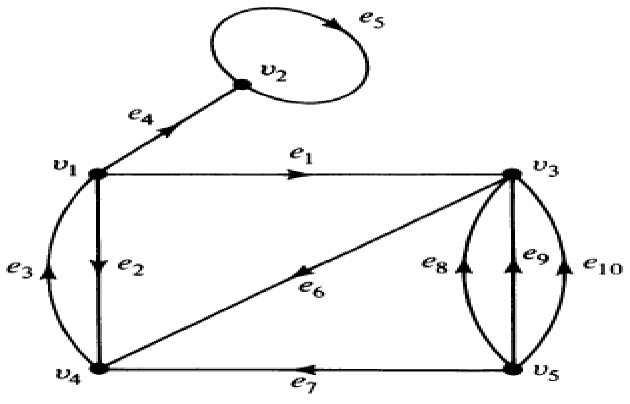
In this case, D is said to be an **oriented graph**.



Example

3. 10 DIRECTED PATHS AND CONNECTEDNESS

A path in a directed graph is called Directed path.



v_5 e_8 v_3 e_6 v_4 e_3 v_1 is a directed path from v_5 to v_1.

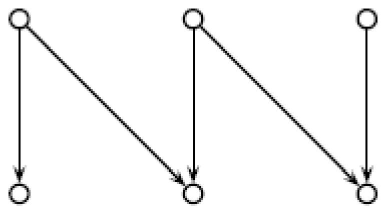
Whereas v_5 e_7 v_4 e_6 v_3 e_1 v_1 is a semi-path from v_5 to v.

- **Strongly connected digraph:** A digraph G is said to be strongly connected if there is at least one directed path from every vertex to every other vertex.

Weakly connected digraph: A digraph G is said to be weakly connected if its corresponding undirected graph is connected. But G is not strongly connected.

The relationship between paths and directed paths is in general rather complicated. This digraph has a path of length five, but its directed paths are of length one.

There is a nice connection between the lengths of directed paths and the chromatic number χ(D) = χ(U(D)).



Example

Theorem : A digraph D has a directed path of length χ(D) −1.

Proof. Let A ⊆ E_D be a minimal set of edges such that the subdigraph D−A contains no directed cycles.

Let k be the length of the longest directed path in D−A.

For each vertex v ∈ D, assign a colour α(v) = i, if a longest directed path from v has length i −1 in D−A. Here 1 ≤ i ≤ k +1.

First we observe that if P = e_1e_2 . . . e_r (r ≥ 1) is any directed path u $\overset{\star}{\longrightarrow}$ v in D−A, then α(u) $\neq$ α(v).

Indeed, if a(v) = i, then there exists a directed path Q: v $\overset{\star}{\longrightarrow}$ w of length i − 1, and PQ is a directed path, since D−A does not contain directed cycles.

Since PQ: u $\overset{\star}{\longrightarrow}$ w, a(u) $\neq$ i = a(v). In particular, if e = uv ∈ D−A, then α(u) $\neq$ α(v).

Consider then an edge e = vu ∈ A. By the minimality of A, (D−A)+ e contains a directed cycle C: u $\overset{\star}{\longrightarrow}$ v −→ u, where the part u $\overset{\star}{\longrightarrow}$ v is a directed path in D−A, and hence a(u) $\neq$ a(v).

This shows that a is a proper colouring of U(D), and therefore χ(D) ≤ k + 1, that is, k ≥ χ (D) − 1.

The bound χ(D) − 1 is the best possible in the following sense.

Connectedness:

A digraph is said to be disconnected if it is not even weak. A digraph is said to be strictly weak if it is weak, but not unilateral.

It is strictly unilateral, if it is unilateral but not strong. Two vertices of a digraph D are said to be

i. 0-connected if there is no semi path joining them,

ii. 1-connected if there is a semi path joining them, but there is no u−v path or v−u path,

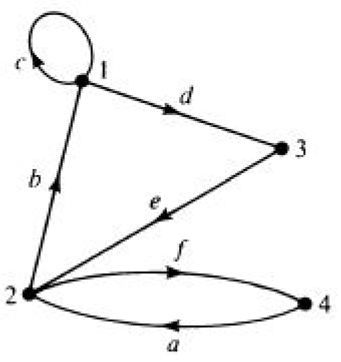
iii. 2-connected if there is a u−v or a v−u path, but not both,

iv. 3-connected if there is u−v path and a v−u path.

3. 11 EULER GRAPHS

In a digraph G, a closed directed walk which traverses every edge of G exactly once is called a *directed Euler line*. A digraph containing a directed Euler line is called an **Euler digraphs**

For example,



It contains directed Euler line **a b c d e f.**

teleprinter's problem.

Constructing a longest circular sequence of 1's and 0's such that no subsequence of r bits appears more than once in the sequence.

Teleprinter's problem was solved in 1940 by I.G. Good using digraph.

Theorem:

A connected graph G is an Euler graph if and only if all vertices of G are of even degree.

Proof : ***Necessity*** Let G(V, E) be an Euler graph.

Thus G contains an Euler line Z, which is a closed walk. Let this walk start and end at the vertex u ∈ V. Since each visit of Z to an intermediate vertex v of Z contributes two to the degree of v and since Z traverses each edge exactly once, d(v) is even for every such vertex. Each intermediate visit to u contributes two to the degree of u, and also the initial and final edges of Z contribute one each to the degree of u. So the degree d(u) of u is also even.

Sufficiency Let G be a connected graph and let degree of each vertex of G be even.

Assume G is not Eulerian and let G contain least number of edges. Since $\delta \geq 2$, G has a cycle. Let Z be a closed walk in G of maximum length.

Clearly, G−E(Z) is an even degree graph. Let C1 be one of the components of G−E(Z). As C1 has less number of edges than G, it is Eulerian and has a vertex v in common with Z.

Let Z' be an Euler line in C1. Then Z' ∪Z is closed in G, starting and ending at v. Since it is longer than Z, the choice of Z is contradicted. Hence G is Eulerian.

Second proof for sufficiency Assume that all vertices of G are of even degree.

We construct a walk starting at an arbitrary vertex v and going through the edges of G such that no edge of G is traced more than once. The tracing is continued as far as possible.

Since every vertex is of even degree, we exit from the vertex we enter and the tracing clearly cannot stop at any vertex but v. As v is also of even degree, we reach v when the tracing comes to an end.

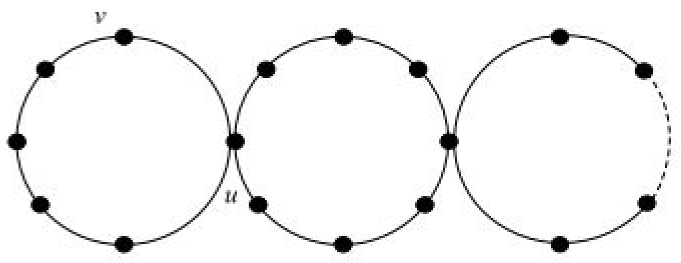
If this closed walk Z we just traced includes all the edges of G, then G is an Euler graph.

If not, we remove from G all the edges in Z and obtain a subgraph Z' of G formed by the remaining edges. Since both G and Z have all their vertices of even degree, the degrees of the vertices of Z' are also even.

Also, Z' touches Z' at least at one vertex say u, because G is connected. Starting from u, we again construct a new walk in Z' .

As all the vertices of Z' are of even degree, therefore this walk in Z' terminates at vertex u. This walk in Z'combined with Z forms a new walk, which starts and ends at the vertex v and has more edges than Z. Prepared by G. Appasami, Assistant professor, Dr. pauls Engineering College.

This process is repeated till we obtain a closed walk that traces all the edges of G. Hence G is an Euler graph.



Euler graph

UNIT IV PERMUTATIONS & COMBINATIONS

4. 1 FUNDAMENTAL PRINCIPLES OF COUNTING

The Fundamental Counting Principle is a way to figure out the total number of ways different events can occur.

1. rule of sum.

If the first task can be performed in *m* ways, while a second task can be performed in *n* ways, and the two tasks cannot be performed simultaneously, then performing either task can be accomplished in any one of $m + n$ ways.

Example: A college library has 40 books on C++ and 50 books on Java. A student at this college can select 40+50=90 books to learn programming language.

2. Define rule of Product

If a procedure can be broken into first and second stages, and if there are *m* possible outcomes for the first stage and if, for each of these outcomes, there are *n* possible outcomes for the second stage, then the total procedure can be carried out, in the designed order, in *mn* ways.

Example: A drama club with six men and eight can select male and female role in 6 x 8 = 48 ways.

4. 2 PERMUTATIONS AND COMBINATIONS

Permutations

For a given collection of *n* objects, any linear arrangement of these objects is called a permutation of the collection. Counting the linear arrangement of objects can be done by rule of product.

For a given collection of *n* distinct objects, and *r* is an integer, with $1 \le r \le n$, then by rule of product, the number of permutations of size r for the n objects is

$$P(n,r) = n \times (n-1) \times (n-2) \times \dots \times (n-r+1) = \frac{n!}{(n-r)!}, \quad 0 \le r \le n$$

Example: In a class of 10 students, five are to be chosen and seated in a row for a picture.

The total number of arrangements = 10 x 9 x 8 x 7 x 6 = 30240.

Define combinations

For a given collection of *n* objects, each selection, or combination, of *r* of these objects, with no reference to order, corresponds to *r*! (Permutations of size *r* from the *n* objects). Thus the number of combinations of size *r* from a collection of size *n* is

$$C(n,r) = \frac{P(n,r)}{r!} = \frac{n!}{r!\,(n-r)!}, \quad 0 \le r \le n$$

Example: In a test, students are directed to answer 7 questions out of 10. The student can answer the examination in

$$C(n,r) = C(10,7) = \frac{10!}{7!\,(10-7)!} = \frac{10 \times 9 \times 8}{3 \times 2 \times 1} = 120\, ways$$

4. 3 BINOMIAL THEOREM

The Binomial theorem: If *x* and *y* are variables and n is a positive integer, then

$$(x+y)^n = \binom{n}{0} x^0 y^n + \binom{n}{1} x^1 y^{n-1} + \binom{n}{2} x^2 y^{n-2} + \cdots$$

$$\binom{n}{n-1} x^{n-1} y^1 + \binom{n}{n} x^n y^0 = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k}$$

$\binom{n}{k}$ is referred as Binomial coefficient.

Application of permutation and combinations:

$$(1+x)^n = \sum_{k=0}^{n} \binom{n}{k} x^k$$

$$(1+x)^3 = \overbrace{(1+x)}^{1}\overbrace{(1+x)}^{2}\overbrace{(1+x)}^{3} = 1 + 3x + 3x^2 + x^3 = \binom{3}{0} + \binom{3}{1}x + \binom{3}{2}x^2 + \binom{3}{3}x^3$$

Proof.

$$(1+x)^n = \overbrace{(1+x)}^{1}\overbrace{(1+x)}^{2}\cdots\overbrace{(1+x)}^{n}$$

In order to get x ^{k}, we need to choose x in k of {1, . . . , n}. There are $\binom{n}{k}$ ways of doing this.

Theorem: 1

$$(x+y)^n = \binom{n}{0}x^n + \binom{n}{1}x^{n-1}y + \binom{n}{2}x^{n-2}y^2 + \cdots + \binom{n}{n}y^n = \sum_{i=0}^{n}\binom{n}{i}x^{n-i}y^i$$

Proof.

We prove this by induction on n. It is easy to check the first few, say for n=0,1,2, which form the base case. Now suppose the theorem is true for n−1, that is,

$$(x+y)^{n-1} = \sum_{i=0}^{n-1}\binom{n-1}{i}x^{n-1-i}y^i.$$

Then

$$(x+y)^n = (x+y)(x+y)^{n-1} = (x+y)\sum_{i=0}^{n-1}\binom{n-1}{i}x^{n-1-i}y^i.$$

Using the distributive property,this becomes

$$x\sum_{i=0}^{n-1}\binom{n-1}{i}x^{n-1-i}y^i + y\sum_{i=0}^{n-1}\binom{n-1}{i}x^{n-1-i}y^i$$
$$= \sum_{i=0}^{n-1}\binom{n-1}{i}x^{n-i}y^i + \sum_{i=0}^{n-1}\binom{n-1}{i}x^{n-1-i}y^{i+1}.$$

These two sums have much in common, but it is slightly disguised by an "offset": the first sum starts with an x^ny^0 term and ends with an x^1y^n-1 term, while the corresponding terms in the second sum are $x^{n-1}y^1$ and x^0y^n.

Let's rewrite the second sum so that they match:

$$\sum_{i=0}^{n-1}\binom{n-1}{i}\,x^{n-i}y^i + \sum_{i=0}^{n-1}\binom{n-1}{i}x^{n-1-i}y^{i+1}$$
$$= \sum_{i=0}^{n-1}\binom{n-1}{i}x^{n-i}y^i + \sum_{i=1}^{n}\binom{n-1}{i-1}x^{n-i}y^i$$
$$\binom{n-1}{0}x^n + \sum_{i=1}^{n-1}(\binom{n-1}{i} + \binom{n-1}{i-1})x^{n-i}y^i + \binom{n-1}{n-1}y^n$$
$$= \binom{n-1}{0}x^n + \sum_{i=1}^{n-1}\binom{n}{i}x^{n-i}y^i + \binom{n-1}{n-1}y^n.$$
$$= \binom{n}{0}x^n + \sum_{i=1}^{n-1}\binom{n}{i}x^{n-i}y^i + \binom{n}{n}y^n$$
$$= \sum_{i=0}^{n}\binom{n}{i}x^{n-i}y^i.$$

At the next to last step we used the facts that $\binom{n-1}{0} = \binom{n}{0}$ and $\binom{n-1}{n-1} = \binom{n}{n}$.

Here is an interesting consequence of this theorem: Consider

$$(x + y)^n = (x + y)(x + y) \cdots (x + y).$$

One way we might think of attempting to multiply this out is this: Go through the n factors (x+y) and in each factor choose either the x or the y; at the end, multiply your choices together, getting some term like xxyxyy⋯yx=$x^i y^j$, where of course i+j=n.

If we do this in all possible ways and then collect like terms, we will clearly get

$$\sum_{i=0}^{n} a_i x^{n-i} y^i.$$

We know that the correct expansion has $\binom{n}{i}$=a_i; is that in fact what we will get by this method? Yes: consider $x^{n-i}y^i$.

How many times will we get this term using the given method? It will be the number of times we end up with i y-factors.

Since there are n factors (x+y), the number of times we get i y-factors must be the number of ways to pick i of the (x+y) factors to contribute a y, namely $\binom{n}{i}$. This is probably not a useful method in practice, but it is interesting and occasionally useful.

4. 4 COMBINATIONS WITH REPETITION

If there is a selection with repetition, r of n distinct objects, then the combinations with of n objects taken r at a time with repetition is $C(n + r - 1, r)$.

$$C(n + r - 1, r) = \frac{(n + r - 1)!}{r!\,(n - 1)!} = \binom{n + r - 1}{r}$$

Example: A donut shop offers 20 kinds of donuts. Assuming that there are at least a dozen of each kind when we enter the shop. We can select a dozen donuts in $C(20 + 12 - 1, 12)$= $C(31, 12)$ = 141120525 ways

We can also have an r-combination of n items with repetition.

Same as other combinations: order doesn't matter.

Same as permutations with repetition: we can select the same thing multiple times.

Example: You walk into a candy store and have enough money for 6 pieces of candy. The store has chocolate (C), gummies (G), and horrible Chinese candy (H). How many different selections can you make?

Here are some possible selections you might make:

C C C G G H

C G G G G H

C C C C G G

H H H H H H

Since order doesn't matter, we'll list all of our selections in the same order: C then G then H.

We don't want our candy to mix: let's separate the types.

C C C | G G | H

C | G G G G | H

C C C C | G G |

| | H H H H H H

Now we don't need the actual identities in the diagram to know what's there:

- - - | - - | -

- | - - - - | -

- - - - | - - |

| | - - - - - -

Now the answer becomes obvious: we have 8 slots there and just have to decide where to put the two dividers.

There are C(8,2) ways to do that, so C(8,2)=28 possible selections. Or equivalently, there are C(8,6)=28 ways to place the candy selections.

If we are selecting an r-combination from n elements with repetition, there are C(n+r−1,r)=C(n+r−1,n−1) ways to do so.

Proof: like with the candy, but not specific to r=6 and n=3.

Example: How many solutions does this equation have in the non-negative integers?a+b+c=100

In order to satisfy the equation, we have to select 100 "ones", some that will contribute to a, some to b, some to c. In other words, we have balls labeled a, b, and c. We select 100 of them (with repetition) and that gives us a solution to the equation. C(102,2) solutions.

In summary we have these ways to select r things from n possibilities: Prepared by G. Appasami, Assistant professor, Dr. pauls Engineering College.

Order?	Repetition?	Formula
Yes (permutation)	No	$P(n,r) = \frac{n!}{(n-r)!}$
No (combination)	No	$C(n,r) = \frac{n!}{r!(n-r)!}$
Yes (permutation)	Yes	n^r
No (combination)	Yes	$C(n+r-1,r) = \frac{(n+r-1)!}{r!(n-1)!}$

4. 5 COMBINATORIAL NUMBERS

The Catalan numbers form a sequence of natural numbers that occur in various counting problems, often involving recursively-defined objects. They are named after the Belgian mathematician Eugène Charles Catalan. the nth Catalan number is given directly in terms of binomial coefficients by

$$C_n = \binom{2n}{n} - \binom{2n}{n+1} = \frac{(2n)!}{(n+1)!\,n!} = \frac{1}{n+1}\binom{2n}{n} \qquad \text{for } n \ge 0$$

4. 6 PRINCIPLE OF INCLUSION AND EXCLUSION

Very often, we need to calculate the number of elements in the union of certain sets.
Assuming that we know the sizes of these sets, and their mutual intersections, the principle of inclusion and exclusion allows us to do exactly that.
Suppose that we have two sets A, B.
The size of the union is certainly at most |A| + |B|.
This way, however, we are counting twice all elements in A ∩ B, the intersection of the two sets. To correct for this, we subtract |A ∩ B| to obtain the following formula:
|A ∪ B| = |A| + |B| − |A ∩ B|.
In general, the formula gets more complicated because we have to take into account intersections of multiple sets. The following formula is what we call the principle of inclusion and exclusion

$$\begin{aligned} N(\bar{c}_1\bar{c}_2\bar{c}_3\bar{c}_4) = N &- [N(c_1) + N(c_2) + N(c_3) + N(c_4)] \\ &+ [N(c_1c_2) + N(c_1c_3) + N(c_1c_4) + N(c_2c_3) + N(c_2c_4) + N(c_3c_4)] \\ &- [N(c_1c_2c_3) + N(c_1c_2c_4) + N(c_1c_3c_4) + N(c_2c_3c_4)] \\ &+ N(c_1c_2c_3c_4). \end{aligned}$$

4. 7 DERANGEMENTS

A derangement is a permutation of the elements of a set, such that no element appears in its original position.

The number of derangements of a set of size n, usually written D_n, d_n, or !n, is called the "derangement number" or "de Montmort number".

Example: The number of derangements of 1, 2, 3, 4 is

$d_4 = 4!\ [1 - 1 + (1/2!)-(1/3!)+(1/4!)] = 9.$

4. 8 ARRANGEMENTS WITH FORBIDDEN POSITIONS

The number of acceptable assignments is equal to the number of ways of placing nontaking rooks on this chessboard so that none of the rooks is in a forbidden position. The key to determining this number of arrangements is the inclusion- exclusion principle.

Suppose we shuffle a deck of cards; what is the probability that no card is in its original location?

More generally, how many permutations of $[n]=\{1,2,3,\ldots,n\}$ have none of the integers in their "correct" locations? That is, 1 is not first, 2 is not second, and so on. Such a permutation is called a derangement of [n].

Let S be the set of all permutations of [n] and Ai be the permutations of [n] in which i is in the correct place. Then we want to know $|\bigcap n_i=1A^c{}_i|$.

For any i, $|A_i|=(n-1)!$: once i is fixed in position i, the remaining $n-1$ integers can be placed in any locations.

What about $|A_i \cap A_j|$? If both i and j are in the correct position, the remaining $n-2$integers can be placed anywhere, so $|A_i \cap A_j|=(n-2)!$.

In the same way, we see that $|A_{i1} \cap A_{i2} \cap \cdots \cap A_{ik}|=(n-k)!$.

$$\begin{aligned}
|\bigcap_{i=1}^{n} A_i^c| &= |S| + \sum_{k=1}^{n}(-1)^k \binom{n}{k}(n-k)! \\
&= n! + \sum_{k=1}^{n}(-1)^k \frac{n!}{k!(n-k)!}(n-k)! \\
&= n! + \sum_{k=1}^{n}(-1)^k \frac{n!}{k!} \\
&= n! + n!\sum_{k=1}^{n}(-1)^k \frac{1}{k!} \\
&= n!\left(1 + \sum_{k=1}^{n}(-1)^k \frac{1}{k!}\right) \\
&= n!\sum_{k=0}^{n}(-1)^k \frac{1}{k!}.
\end{aligned}$$

The last sum should look familiar:

$$e^x = \sum_{k=0}^{\infty} \frac{1}{k!} x^k$$

Substituting x=−1 gives

$$e^{-1} = \sum_{k=0}^{\infty} \frac{1}{k!}(-1)^k.$$

The probability of getting a derangement by chance is then

$$\frac{1}{n!} n! \sum_{k=0}^{n}(-1)^k \frac{1}{k!} = \sum_{k=0}^{n}(-1)^k \frac{1}{k!},$$

and when n is bigger than 6, this is quite close toe^{-1}≈0.3678.

So in the case of a deck of cards, the probability of a derangement is about 37%.

Let $D_n = n! \sum_{k=0}^{n}(-1)^k \frac{1}{k!}$ These derangement numbers have some interesting properties.

The derangements of [n] may be produced as follows: For each i∈{2,3,…,n}, put i in position 1 and 1 in position i.

Then permute the numbers {2,3,…,i−1,i+1,…n} in all possible ways so that none of these n−2 numbers is in the correct place.There are D_{n-2} ways to do this.

Then, keeping 1 in position i, derange the numbers {i,2,3,…,i−1,i+1,…n}, with the "correct" position of i now considered to be position 1.

There are D_{n-1} ways to do this. Thus, $D_n=(n-1)(D_{n-1}+D_{n-2})$.

We explore this recurrence relation a bit:

$D_n=nD_{n-1}-D_{n-1}+(n-1)D_{n-2}$

$=nD_{n-1}-(n-2)(D_{n-2}+D_{n-3})+(n-1)D_{n-2}$

$=nD_{n-1}-(n-2)D_{n-2}-(n-2)D_{n-3}+(n-1)D_{n-2}$

$=nD_{n-1}+D_{n-2}-(n-2)D_{n-3}$

$=nD_{n-1}+(n-3)(D_{n-3}+D_{n-4})-(n-2)D_{n-3}$

$=nD_{n-1}+(n-3)D_n-3+(n-3)D_{n-4}-(n-2)D_{n-3}$

$=nD_{n-1}-D_{n-3}+(n-3)D_{n-4}$

$=nD_{n-1}-(n-4)(D_n-4+D_n-5)+(n-3)D_{n-4}$

$=nD_{n-1}-(n-4)D_{n-4}-(n-4)D_{n-5}+(n-3)D_{n-4}$

$=nD_{n-1}+D_{n-4}-(n-4)D_{n-5}$.

It appears from the starred lines that the pattern here is that

$D_n=nD_{n-1}+(-1)^kD_{n-k}+(-1)^{k+1}(n-k)D_{n-k-1}$.

If this continues, we should get to

$D_n=nD_{n-1}+(-1)^{n-2}D_2+(-1)^{n-1}(2)D_1$.

Since $D_2=1$ and $D_1=0$, this would give $D_n=nD_{n-1}+(-1)^n$,

Since $(-1)^n=(-1)^{n-2}$. Indeed this is true, and can be proved by induction. This gives a somewhat simpler recurrence relation, making it quite easy to compute D_n.

UNIT V GENERATING FUNCTIONS

5. 1 GENERATING FUNCTIONS

A **generating function** describes an infinite sequence of numbers (a_n) by treating them like the coefficients of a series expansion. The sum of this infinite series is the generating function. Unlike an ordinary series, this formal series is allowed to diverge, meaning that the generating function is not always a true function and the "variable" is actually an indeterminate.

The generating function for 1, 1, 1, 1, 1, 1, 1, 1, 1, ..., whose ordinary generating function is

$$\sum_{n=0}^{\infty}(x)^n = \frac{1}{1-x}$$

The generating function for the geometric sequence 1, a, a^2, a^3, ... for any constant a:

$$\boldsymbol{\sum_{n=0}^{\infty}(ax)^n = \frac{1}{1-ax}}$$

5. 2 PARTITIONS OF INTEGERS

Partitioning a positive n into positive summands and seeking the number of such partitions without regard to order is called Partitions of integer.

This number is denoted by $p(n)$. For example

P(1) = 1:	1
P(2) = 2:	2 = 1 + 1
P(3) = 3:	3 = 2 +1 = 1 + 1 +1
P(4) = 5:	4 = 3 + 1 = 2 + 2 = 2 + 1 + 1 = 1 + 1 + 1 + 1
P(5) = 7:	5 = 4 + 1 = 3 + 2 = 3 + 1 + 1 = 2 + 2 + 1 = 2 + 1 + 1+ 1 = 1 + 1 + 1 + 1 + 1

There is no simple formula for p_n, but it is not hard to find a generating function for them. As with some previous examples, we seek a product of factors so that when the factors are multiplied out, the coefficient of x_n is p_n.

We would like each x_n term to represent a single partition, before like terms are collected. A partition is uniquely described by the number of 1s, number of 2s, and so on, that is, by the repetition numbers of the multi-set. We devote one factor to each integer:

$$(1+x+x^2+x^3+\cdots)(1+x^2+x^4+x^6+\cdots)\cdots(1+x^k+x^{2k}+x^{3k}+\cdots)\cdots=\prod_{k=1}^{\infty}\sum_{i=0}^{\infty}x^{ik}$$

When this product is expanded, we pick one term from each factor in all possible ways, with the further condition that we only pick a finite number of "non-1" terms. For example, if we pick x^3 from the first factor, x^3 from the third factor, x^{15} from the fifth factor, and 1s from all other factors, we get x^{21}.

In the context of the product, this represents 3.1+1.3+3.5, corresponding to the partition 1+1+1+3+5+5+5, that is, three 1s, one 3, and three 5s. Each factor is a geometric series; the kth factor is

$$1+x^k+(x^k)^2+(x^k)^3+\cdots=\frac{1}{1-x^k},$$ .so the generating function can be written $$\prod_{k=1}^{\infty}\frac{1}{1-x^k}.$$

Note that if we are interested in some particular pn, we do not need the entire infinite product, or even any complete factor, since no partition of n can use any integer greater than n, and also cannot use more than n/k copies of k.

Example 2:Find p8

We expand

$(1+x(1+x^2+x^3+x^4+x^5+x^6+x^7+x^8)(1+x^2+x^4+x^6+x^8)(1+x^3+x^6)+x^4+x^8)(1+x^5)(1+x^6)(1+x^7)(1+x^8$
$)$

$=1+x+2x^2+3x^3+5x^4+7x^5+11x^6+15x^7+22x^8+\cdots+x^{56}$,

so $p_8=22$. Note that all of the coefficients prior to this are also correct, but the following coefficients are not necessarily the corresponding partition numbers.

Partitions of integers have some interesting properties. Let pd(n) be the number of partitions of n into distinct parts; let po(n) be the number of partitions into odd parts.

5. 3 EXPONENTIAL GENERATING FUNCTION

For a sequence $a_0, a_1, a_2, a_3,, \ldots$ of real numbers.

$$f(x) = a_0 + a_1 x + a_2 \frac{x^2}{2!} + a_3 \frac{x^3}{3!} + \cdots = \sum_{i=0}^{\infty} a_i \frac{x^{i^i}}{i!}$$

is called the exponential generating function for the given sequence.

Ordinary generating functions arise when we have a (finite or countably in- finite) set of objects S and a weight function ω : S → N ^{r} .

Then the ordinary generating function Φω S (x) is defined and we can proceed with calculations. Exponential generating functions arise in a somewhat more complicated situation. The basic idea is that they are used to enumerate "combinatorial structures on finite sets".

Definition 1:(Exponential Generating Functions). Let A be a class of structures. The exponential generating function of A is

$$A(x) := \sum_{n=0}^{\infty} (\#\mathcal{A}_n) \frac{x^n}{n!}.$$

Let's illustrate this with a few examples for which we already know the answer.

Example:

First, consider the class S of permutations: to each finite set X it associates the finite set SX of all bijections σ : X → X from X to X.

Condition (i) is easy, and condition (ii) follows from Examples, so that S satisfies above definition . Way back in Theorem 2 we saw that #Sn = n! for all n ∈ N, so that the exponential generating function for the class of permutations is

$$S(x) := \sum_{n=0}^{\infty} (\#\mathcal{S}_n) \frac{x^n}{n!} = \sum_{n=0}^{\infty} n! \frac{x^n}{n!} = \frac{1}{1-x}.$$

5. 4 SUMMATION OPERATOR

1. aclaurin series expansion of e^x and e^{-x}.

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots$$

$$e^{-x} = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \frac{x^4}{4!} - \cdots$$

Adding these two series together, we get,

$$e^x + e^{-x} = 2(1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \cdots)$$

$$\frac{e^x + e^{-x}}{2} = 1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \cdots$$

Generating function for a sequence a_0, $a_0 + a_1$, $a_0 + a_1 + a_2$, $a_0 + a_1 + a_2 + a_3$,, … .

For $(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots$,, consider the function *f(x)/(1-x)*

$$\frac{f(x)}{1-x} = f(x).\frac{1}{1-x} = [a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots][1 + x + x^2 + x^3 + \cdots]$$

$$= a_0 + (a_0 + a_1)x + (a_0 + a_1 + a_2)x^2 + +(a_0 + a_1 + a_2 + a_3)x^3 + \cdots$$

So *f(x)/(1-x)* generates the sequence of sums a_0, $a_0 + a_1$, $a_0 + a_1 + a_2$, $a_0 + a_1 + a_2 + a_3$,,

1/(1-*x*) is called the summation operator

5. 5 RECURRENCE RELATIONS

A **recurrence relation** is an equation that recursively defines a sequence or multidimensional array of values, once one or more initial terms are given: each further term of the sequence or array is defined as a function of the preceding terms.

The term **difference equation** sometimes (and for the purposes of this article) refers to a specific type of recurrence relation. However, "difference equation" is frequently used to refer to *any* recurrence relation.

Fibonacci numbers and relation

The recurrence satisfied by the Fibonacci numbers is the archetype of a homogeneous linear recurrence relation with constant coefficients (see below). The Fibonacci sequence is defined using the recurrence

$F_n = F_{n-1} + F_{n-2}$

with seed values $F_0 = 0$ and $F_1 = 1$

We obtain the sequence of Fibonacci numbers, which begins

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

Recurrence Relation (Epp)

A **recurrence relation** for a sequence a_0, a_1, a_2, … is a formula that relates each term a_k to certain of its predecessors a_{k-1}, …, a_{k-i}, where i is fixed and $k \geq i$. The initial conditions specify the fixed values of a_0, …, a_{i-1}.

Most of the time, though, there is only one fixed value or base value.

However, nothing prevents us from defining a sequence with multiple base values (consider the two 1s in the Fibonacci sequence), hence the generality with i in the definition above.

There are also two forms of induction to handle this generality. The one we looked at earlier works when we have one base case. There's another form of induction called *strong induction* that proves claims where there are multiple base cases.

We could spend quite a bit of time studying recurrences by themselves. We'll just scratch the surface and use them a few times in the remainder of the course.

Question: If we have a recurrence relation for a sequence, is it possible to express the sequence in a way that does *not* use recursion?

Answer: Sometimes. When we are able to do so, we find what is called the **closed form** of the recurrence. It is an algebraic formula or a definition that tells us how to find the nth term without needing to know any of the preceding terms. The process of finding the closed form is called **solving a recurrence**.

There are various methods to "solving" recurrence that are used in practice. Each has its place, each has a difference sort of output.

Three Methods to Solving Recurrences:

Iteration: Start with the recurrence and keep applying the recurrence equation until we get a pattern. The result is a *guess* at the closed form.

Substitution: Guess the solution; prove it using induction. The result here is a proven closed form. It's often difficult to come up the guess so, in practice, iteration and substitution are used hand-in-hand.

Master Theorem: Plugging into a formula that gives an approximate bound on the solution. The result here is only a *bound* on the closed form. It is not an exact solution.

5. 6 FIRST ORDER

The general form of First order linear homogeneous recurrence relation can be written as

$a_{n+1} = d\, a_n$, n ≥ 0, where d is a constant. The relation is first order since a_{n+1} depends on a_n.

a_0 *or* a_1 are called boundary conditions.

5. 7 SECOND ORDER

Let $k \in \mathbf{Z}^+$ and C_0 ($\neq 0$), $C_1, C_2, \ldots, C_k$ ($\neq 0$) be real numbers. If a_n, for $n \geq 0$, is a discrete function, then

$$C_0a_n + C_1a_{n-1} + C_2a_{n-2} + \cdots + C_ka_{n-k} = f(n), \qquad n \geq k,$$

is a linear recurrence relation (with constant coefficients) of *order k*. When $f(n) = 0$ for all $n \geq 0$, the relation is called *homogeneous*; otherwise, it is called *nonhomogeneous*.

In this section we shall concentrate on the homogeneous relation of order two:

$$C_0a_n + C_1a_{n-1} + C_2a_{n-2} = 0, \qquad n \geq 2.$$

On the basis of our work in Section 10.1, we seek a solution of the form $a_n = cr^n$, where $c \neq 0$ and $r \neq 0$.

Substituting $a_n = cr^n$ into $C_0a_n + C_1a_{n-1} + C_2a_{n-2} = 0$, we obtain

$$C_0cr^n + C_1cr^{n-1} + C_2cr^{n-2} = 0.$$

With $c, r \neq 0$, this becomes $C_0r^2 + C_1r + C_2 = 0$, a quadratic equation which is called the *characteristic equation*. The roots r_1, r_2 of this equation determine the following three cases: (a) r_1, r_2 are distinct real numbers; (b) r_1, r_2 form a complex conjugate pair; or (c) r_1, r_2 are real, but $r_1 = r_2$. In all cases, r_1 and r_2 are called the *characteristic roots*.

5. 8 NON-HOMOGENEOUS RECURRENCE RELATIONS

We now turn to the recurrence relations

$$a_n + C_1 a_{n-1} = f(n), \qquad n \geq 1, \tag{1}$$

$$a_n + C_1 a_{n-1} + C_2 a_{n-2} = f(n), \qquad n \geq 2, \tag{2}$$

where C_1 and C_2 are constants, $C_1 \neq 0$ in Eq. (1), $C_2 \neq 0$, and $f(n)$ is not identically 0. Although there is no general method for solving all nonhomogeneous relations, for certain functions $f(n)$ we shall find a successful technique.

We start with the special case for Eq. (1), when $C_1 = -1$. For the nonhomogeneous relation $a_n - a_{n-1} = f(n)$, we have

$$\begin{aligned}
a_1 &= a_0 + f(1) \\
a_2 &= a_1 + f(2) = a_0 + f(1) + f(2) \\
a_3 &= a_2 + f(3) = a_0 + f(1) + f(2) + f(3) \\
&\vdots \\
a_n &= a_{n-1} + f(n) = a_0 + f(1) + \cdots + f(n) = a_0 + \sum_{i=1}^{n} f(i).
\end{aligned}$$

We can solve this type of relation in terms of n, if we can find a suitable summation formula for $\sum_{i=1}^{n} f(i)$.

Definition 1. A sequence $\{a_n\}$ is given by a linear non homogeneous recurrence relation of order k if an = $c_1a_{n-1} + c_2a_{n-2} + c_3a_{n-3} + \cdot\cdot\cdot + c_ka_{n-k} + p(n)$ for all $n \geq k$. The recurrence relation $b_n = c_1b_{n-1} + c_2b_{n-2} + c_3b_{n-3} + \cdot\cdot\cdot + c_kb_{n-k}$ is referred to as the associated linear homogeneous recurrence relation One result is as easy to show for LNRRs as for LHRRs; the following can be proven as a very slight variation of the similar proof for LHRRs

Proposition 1. A sequence is uniquely determined by an LNRR of order k and the initial values $a_0, a_1, a_2, \ldots, a_{k-1}$. However, there is one very important difference between LNRRs and LHRRs: linear combinations of LNRR-satisfying sequences do no, in general, satisfy the LNRR. However, we do have the result:

Proposition 2. If $\{a_n\}$ satisfies an LNRR, and $\{b_n\}$ satisfies the associated LHRR, then $\{a_n+b_n\}$ satisfies the LNRR.

Proof:We know that

$a_n = c_1a_{n-1} + c_2a_{n-2} + c_3a_{n-3} + \cdot\cdot\cdot + c_ka_{n-k} + p(n)$

And $b_n = c_1b_{n-1} + c_2b_{n-2} + c_3b_{n-3} + \cdot\cdot\cdot + c_kb_{n-k}$

Adding these two equations will give $(a_n + b_n) = c_1(a_{n-1} + b_{n-1}) + c_2(a_{n-2} + b_{n-2}) + \cdot\cdot\cdot + c_k(a_{n-k} + b_{n-k}) + p(n)$ and thus $\{a_n + b_n\}$ satisfies the LNRR.

This means that if we have a specific LNRR solution, then we can get a wide range of LNRR solutions simply by adding the associated LHRR solution. Note that this is very similar to the method used to solve non homogeneous linear differential equations.

The tricky part of this is, of course, coming up with a solution to the LNRR in the first place. Let's try doing that for the example above, where $a_n = 3a_{n-1} + 5^{n-1}$. We might do this by inspired guesswork: since the in homogeneous term is 5^{n-1} , we might think some multiple of 5^n will do the trick, so suppose $a_n = C5^{\,n}$. Then, the recurrence relation gives us $C5^{\,n} = 3C \cdot 5^{\,n-1} + 5^{n-1} = (3C + 1)5^{n-1}$

The solution method for solving an LNRR with initial conditions is a very minor variation on the LHRR solution method. Given a LNRR an = $c_1a_{n-1}+c_2a_{n-2} +c_3a_{n-3} +\cdot\cdot\cdot+c_ka_{n-k} +p(n)$ with initial conditions a0, . . . , a_{k-1}, this is our process:

1.Find a single sequence $\{a_n{}^P\}$ to the LNRR.

Find the general solution $\{b_n\}$ to the associated LHRR. By the nature of its construction, {bn} will have k undetermined constants.

The general solution to the LNRR will be $\{a_n\} = \{a_n{}^P + b_n\}$. Like $\{b_n\}$, the sequence $\{a_n\}$ will have k undetermined constants in its expression.

4. Setting the known values of $a_0, a_1, . . . , a_k$ equal to the general-form expressions will yield k equations in k unknowns. Solve for the unknowns to determine the constants in the formula for $\{a_n\}$.

5. 9 METHOD OF GENERATING FUNCTIONS.

On solving a recurrence relation, we have the solution in the form of a sequence. Instead of solving in the form of a sequence, we can also determine the generating function of the sequence from the recurrence relation. One of the uses of generating function method is to find the closed form formula for a recurrence relation.

Once the generating function is known, an expression for the value of sequence can easily be obtained. Before using this method, ensure that the given recurrence equation is in linear form. A non-linear recurrence equation cannot be solved by the generating method.

We use substitution of variable technique to convert a non-linear recurrence relation into linear equation. We explain this method by means of the examples.

Example: Solve the recurrence relation

$$a_r - 3a_{r-1} + 2a_{r-2} = 0 \quad r \geq 2$$

by the generating function with the initial conditions $a_0 = 2$ and $a_1 = 3$.

Solution. Let $A(Z)$ be the generating function of the sequence $<a_n>$ that is

$$A(Z) = \sum_{r=0}^{\infty} a_r Z^r$$

Multiply the given recurrence relation by Z^r, we get

$$a_r Z^r - 3a_{r-1} Z^r + 2a_{r-2} Z^r = 0$$

Summing from $r = 2$ to ∞, we obtain

$$\sum_{r=2}^{\infty} a_r Z^r - 3\sum_{r=2}^{\infty} a_{r-1} Z^r + 2\sum_{r=2}^{\infty} a_{r-2} Z^r = 0$$

$$(A(Z) - a_0 - a_1 Z) - 3Z(A(Z) - a_0) + 2Z^2 A(Z) = 0$$

$$(2Z^2 - 3Z + 1)\, A(Z) - a_0 - a_1 Z + 3a_0 Z = 0$$

Now using the given conditions *i.e.*, $a_0 = 2$, $a_1 = 3$, we get

$$(2Z^2 - 3Z + 1)\, A(Z) - 2 - 3Z + 6Z = 0$$

$$A(Z) = \frac{2-3Z}{2Z^2 - 3Z + 1}$$

$$= \frac{1}{(1-Z)} + \frac{1}{(1-2z)}$$

Thus, $\quad a_r = 1 + 2^r$

CS6702 GRAPH THEORY AND APPLICATIONS
2 MARKS QUESTIONS AND ANSWERS

UNIT I INTRODUCTION

1. Define Graph.

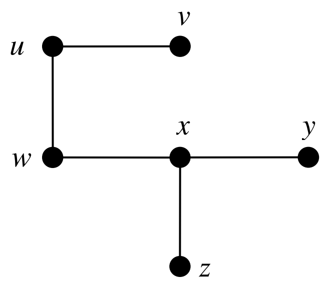
A graph G = (V, E) consists of a set of objects V={v_1, v_2, v_3, … } called **vertices** (also called **points** or **nodes**) and other set E = {e_1, e_2, e_3, ...} whose elements are called **edges** (also called **lines** or **arcs**).

The set V(G) is called the **vertex set** of G and E(G) is the **edge set** of G.

For example :

A graph G is defined by the sets V(G) = {*u, v, w, x, y, z*} and E(G) = {*uv, uw, wx, xy, xz*}.

Graph G:

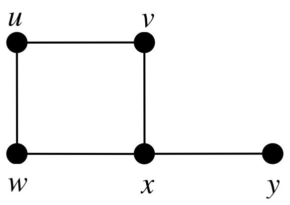


A graph with *p*-vertices and *q*-edges is called a **(*p, q*) graph.** The (1, 0) graph is called **trivial graph.**

2. Define Simple graph.

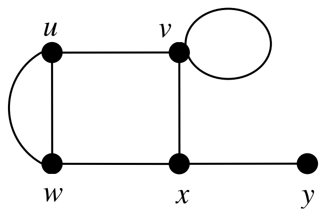
- An edge having the same vertex as its end vertices is called a self-loop.
- More than one edge associated a given pair of vertices called parallel edges.
- A graph that has neither self-loops nor parallel edges is called simple graph.

Graph G:



Simple Graph

Graph H:



Pseudo Graph

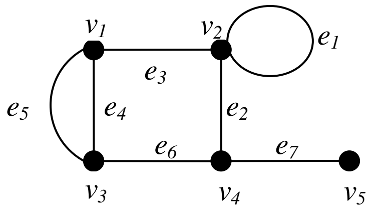
3. Write few problems solved by the applications of graph theory.

Konigsberg bridge problem
Utilities problem
Electrical network problems
Seating problems

4. Define incidence, adjacent and degree.

When a vertex v_i is an end vertex of some edge e_j, v_i and e_j are said to be *incident* with each other. Two non parallel edges are said to be *adjacent* if they are incident on a common vertex. The number of edges incident on a vertex v_i, with self-loops counted twice, is called the *degree* (also called valency), *d*(v_i), of the vertex v_i. A graph in which all vertices are of equal degree is called *regular graph.*

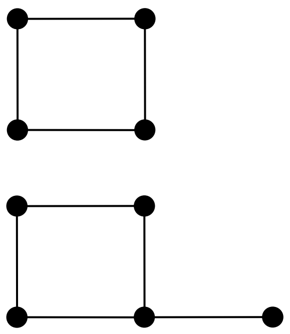
Graph G:

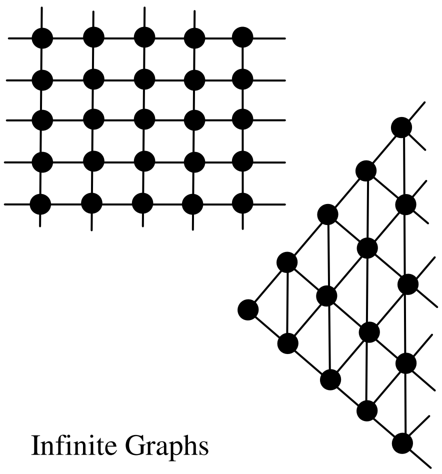


The edges e_2, e_6 and e_7 are incident with vertex v_4.
The edges e_2 and e_7 are adjacent.
The edges e_2 and e_4 are not adjacent.
The vertices v_4 and v_5 are adjacent.
The vertices v_1 and v_5 are not adjacent.
$d(v_1) = d(v_3) = d(v_4) = 3$. $d(v_2) = 4$. $d(v_5) = 1$.

5. What are finite and infinite graphs?

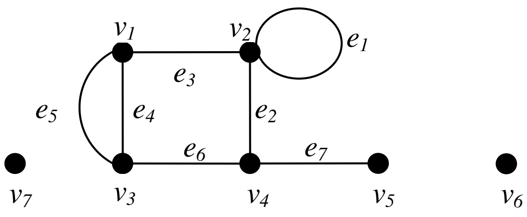
A graph with a finite number off vertices as well as a finite number of edges is called a *finite* graph; otherwise, it is an *infinite* graph.


Finite Graphs


Infinite Graphs

6. Define Isolated and pendent vertex.

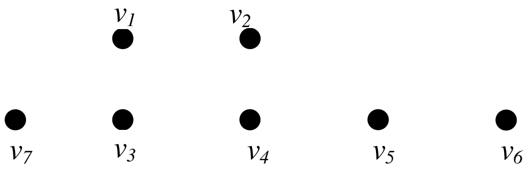
A vertex having no incident edge is called an *isolated vertex*. In other words, isolated vertices are vertices with zero degree. A vertex of degree one is called a *pendant vertex* or an *end vertex*.

Graph G:


The vertices v_6 and v_7 are *isolated vertices*.
The vertex v_5 is a *pendant vertex.*

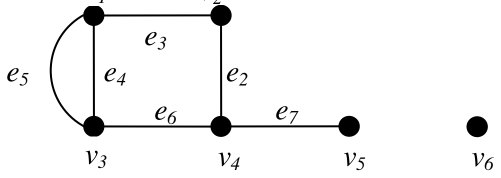
7. Define null graph.

In a graph G=(V, E), If E is empty (Graph without any edges) Then G is called a *null graph.*

Graph G:


8. Define Multigraph

In a multigraph, no loops are allowed but more than one edge can join two vertices, these edges are called **multiple edges** or parallel edges and a graph is called **multigraph.**

Graph G:


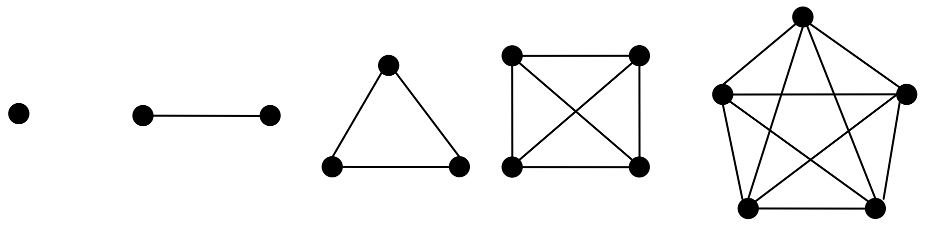
The edges e_5 and e_4 are **multiple** (parallel) edges.

9. Define complete graph

A simple graph G is said to be **complete** if every vertex in G is connected with every other vertex. *i.e.,* if G contains exactly one edge between each pair of distinct vertices.

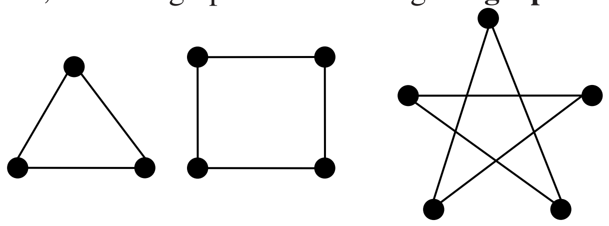
A complete graph is usually denoted by $\mathbf{K_n}$**.** It should be noted that K*n* has exactly *n(n-1)/2* edges.

The complete graphs K_n for $n = 1, 2, 3, 4, 5$ are show in the following Figure.



10. Define Regular graph

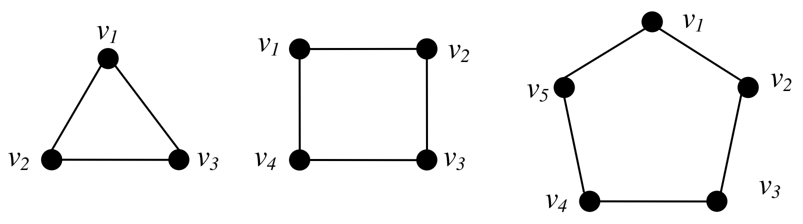
A graph in which all vertices are of **equal degree,** is called a **regular graph.** If the degree of each vertex is *r*, then the graph is called a regular **graph of degree *r*.**



11. Define Cycles

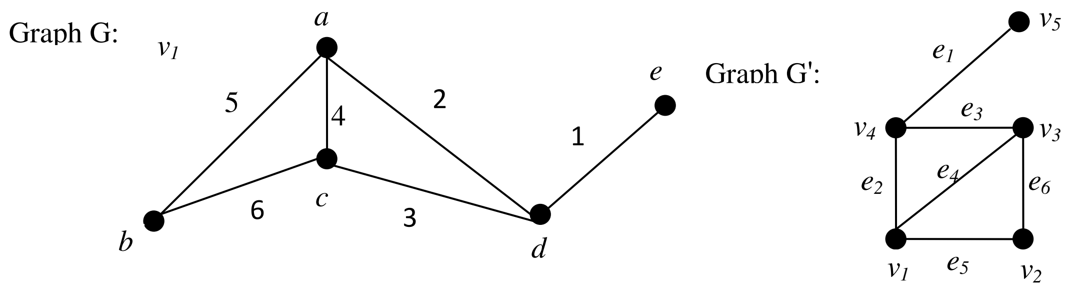
The cycle C_n, $n \geq 3$, consists of *n* vertices $v_1, v_2, ..., v_n$ and edges $\{v_1, v_2\}$, $\{v_2, v_3\}$, ..., $\{v_{n-1}, v_n\}$, and $\{v_n, v_1\}$.

The cyles c_3, c_4 and c_5 are shown in the following Figures



12. Define Isomorphism.

Two graphs G and G' are said to be **isomorphic** to each other if there is a one-to-one correspondence between their vertices and between their edges such that the incidence relationship is preserved.



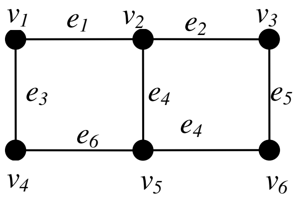
Correspondence of vertices	Correspondence of edges
$f(a) = v_1$	$f(1) = e_1$
$f(b) = v_2$	$f(2) = e_2$
$f(c) = v_3$	$f(3) = e_3$
$f(d) = v_4$	$f(4) = e_4$
$f(e) = v_5$	$f(5) = e_5$

Adjacency also preserved. Therefore G and G' are said to be isomorphic.

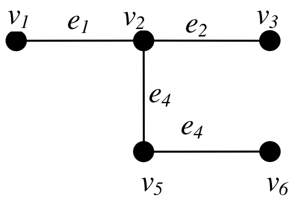
13. What is Subgraph?

A graph G' is said to be a subgraph of a graph G, if all the vertices and all the edges of G' are in G, and each edge of G' has the same end vertices in G' as in G.

Graph G:



Subgraph G' of G:

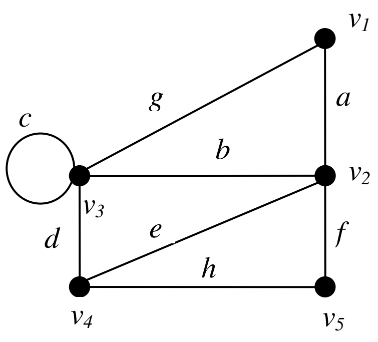


14. Define Walk, Path and Circuit.

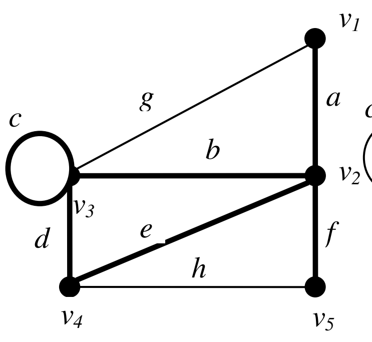
A **walk** is defined as a finite alternating sequence of vertices and edges, beginning and ending with vertices. No edge appears more than once. It is also called as an edge train or a chain.

An open walk in which no vertex appears more than once is called **path**. The number of edges in the path is called **length of a path**.

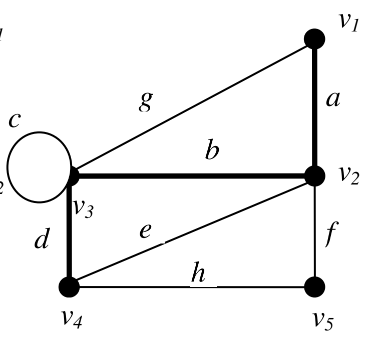
A closed walk in which no vertex (except initial and final vertex) appears more than once is called a circuit. That is, a circuit is a closed, nonintersecting walk.



Graph G:



Open walk



Path of length 3

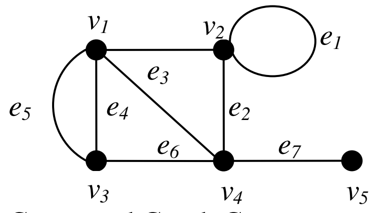
$v_1\ a\ v_2\ b\ v_3\ c\ v_3\ d\ v_4\ e\ v_2 f\ v_5$ is a walk. v_1 *and* v_5 are terminals of walk.

$v_1\ a\ v_2\ b\ v_3\ d\ v_4$ is a path. $a\ v_2\ b\ v_3\ c\ v_3\ d\ v_4\ e\ v_2 f\ v_5$ is not a path.

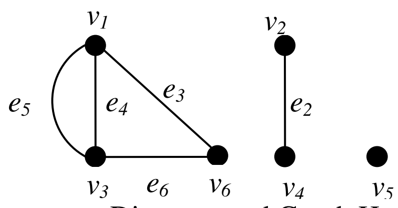
$v_2\ b\ v_3\ d\ v_4\ e\ v_2$ is a circuit.

15. Define connected graph. What is Connectedness?

A graph G is said to be **connected** if there is at least one path between every pair of vertices in G. Otherwise, G is disconnected.



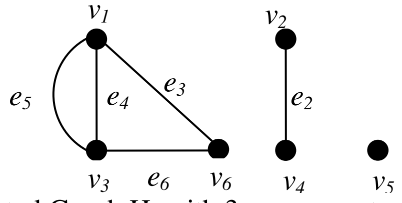
Connected Graph G



Disconnected Graph H

16. Define Components of graph.

A disconnected graph consists of two or more connected graphs. Each of these connected subgraphs is called a component.

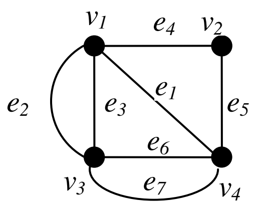


Disconnected Graph H with 3 components

17. Define Euler graph.

A path in a graph G is called Euler path if it includes every edges exactly once. Since the path contains every edge exactly once, it is also called Euler trail / Euler line.

A closed Euler path is called Euler circuit. A graph which contains an Eulerian circuit is called an Eulerian graph.

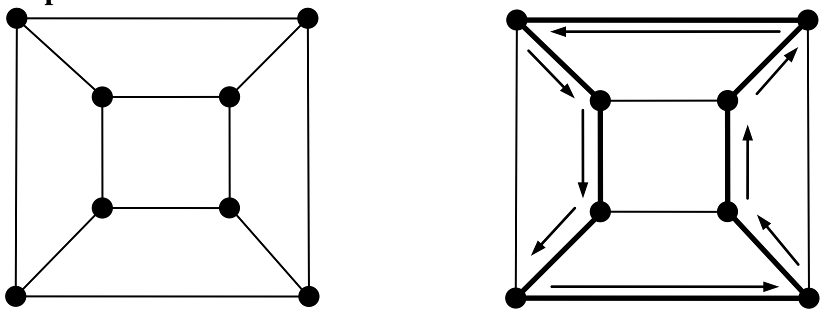


$v_4\ e_1\ v_1\ e_2\ v_3\ e_3\ v_1\ e_4\ v_2\ e_5\ v_4\ e_6\ v_3\ e_7\ v_4$ is an Euler circuit. So the above graph is Euler graph.

18. Define Hamiltonian circuits and paths

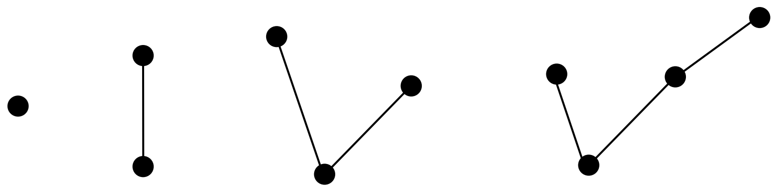
A **Hamiltonian circuit** in a connected graph is defined as a closed walk that traverses every vertex of graph G exactly once except starting and terminal vertex.

Removal of any one edge from a Hamiltonian circuit generates a path. This path is called **Hamiltonian path**.



19. Define Tree

A tree is a connected graph without any circuits. Trees with 1, 2, 3, and 4 vertices are shown in figure.

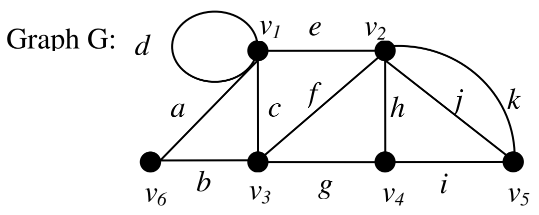


20. List out few Properties of trees.

1. There is one and only one path between every pair of vertices in a tree T.
2. In a graph G there is one and only one path between every pair of vertices, G is a tree.
3. A tree with *n* vertices has *n*-1 edges.
4. Any connected graph with *n* vertices has *n*-1 edges is a tree.
5. A graph is a tree if and only if it is minimally connected.
6. A graph G with *n* vertices has *n*-1 edges and no circuits are connected.

21. What is Distance in a tree?

In a connected graph G, the distance $d(v_i, v_j)$ between two of its vertices v_i and v_j is the length of the shortest path.



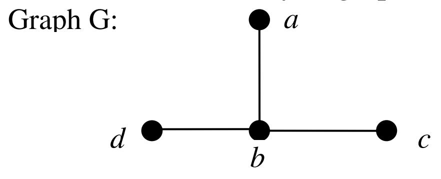
Paths between vertices v_6 and v_2 are (a, e), (a, c, f), (b, c, e), (b, f), (b, g, h), and (b, g, i, k).
The shortest paths between vertices v_6 and v_2 are (a, e) and (b, f), each of length two.
Hence $d(v_6, v_2) = 2$

22. Define eccentricity and center.

The eccentricity E(v) of a vertex v in a graph G is the distance from v to the vertex farthest from v in G; that is,

$$E(v) = \max_{v_i \in G} d(v, v_i)$$

A vertex with minimum eccentricity in graph G is called a center of G



Distance d(a, b) = 1, d(a, c) =2, d(c, b)=1, and so on.
Eccentricity E(a) =2, E(b) =1, E(c) =2, and E(d) =2.
Center of G = A vertex with minimum eccentricity in graph G = b.

23. Define distance metric.

The function $f(x, y)$ of two variables defines the distance between them. These function must satisfy certain requirements. They are

1. Non-negativity: $f(x, y) \geq 0$, and $f(x, y) = 0$ if and only if $x = y$.
2. Symmetry: $f(x, y) = f(x, y)$.
3. Triangle inequality: $f(x, y) \leq f(x, z) + f(z, y)$ for any z.

24. What are the Radius and Diameter in a tree.

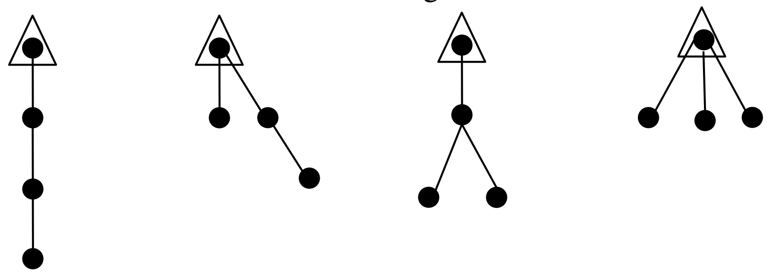
The eccentricity of a center in a tree is defined as the radius of tree.
The length of the longest path in a tree is called the diameter of tree.

25. Define Rooted tree

A tree in which one vertex (called the root) is distinguished from all the others is called a **rooted tree**.

In general tree means without any root. They are sometimes called as **free trees** (non rooted trees).

The root is enclosed in a small triangle. All rooted trees with four vertices are shown below.



26. Define Rooted binary tree

There is exactly one vertex of degree two (root) and each of remaining vertex of degree one or three.

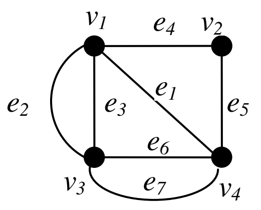
A binary rooted tree is special kind of rooted tree. Thus every binary tree is a rooted tree. A non pendent vertex in a tree is called an internal vertex. Prepared by G. Appasami, Assistant professor, Dr. pauls Engineering College.

UNIT II TREES, CONNECTIVITY & PLANARITY

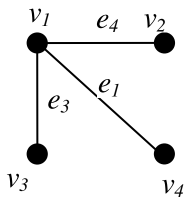
1. Define Spanning trees.

A tree T is said to be a spanning tree of a connected graph G if T is a subgraph of G and T contains all vertices (maximal tree subgraph).

Graph G:



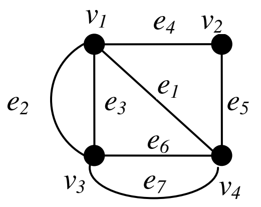
Spanning Tree T:



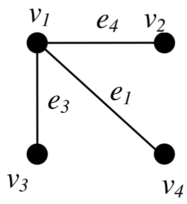
2. Define Branch and chord.

An edge in a spanning tree *T* is called a *branch* of *T*. An edge of G is not in a given spanning tree *T* is called a *chord* (*tie* or *link*).

Graph G:



Spanning Tree T:

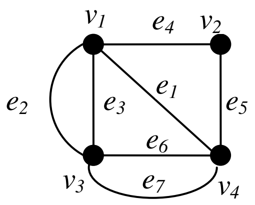


Edge e_1 is a branch of *T* *Edge e_5 is a chord* of *T*

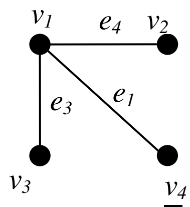
3. Define complement of tree.

If T is a spanning tree of graph G, then the complement of T of G denoted by $\bar{T}$ is the collection of chords. It also called as *chord set* (*tie* set or *cotree*) of T

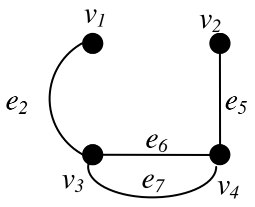
Graph G:



Spanning Tree T:



$\bar{T}$:Complement of Tree T



$$T \cup \bar{T} = G$$

4. Define Rank and Nullity:

A graph G with *n* number of vertices, *e* number of edges, and *k* number of components with the following constraints $n - k \geq 0$ and $e - n + k \geq 0$.

Rank $r = n - k$

Nullity $\mu = e - n + k$ (Nullity also called as *Cyclomatic number* or *first betti number*)

Rank of G = number of branches in any spanning tree of G

Nullity of G = number of chords in G

Rank + Nullity = e = number of edges in G

5. How Fundamental circuits created?

Addition of an edge between any two vertices of a tree creates a circuit. This is because there already exists a path between any two vertices of a tree.

6. Define Spanning trees in a weighted graph

A spanning tree in a graph G is a minimal subgraph connecting all the vertices of G. If G is a weighted graph, then the weight of a spanning tree T of G is defined as the sum of the weights of all the branches in *T*.

A spanning tree with the smallest weight in a weighted graph is called a *shortest spanning tree (shortest-distance spanning tree* or *minimal spanning tree).*

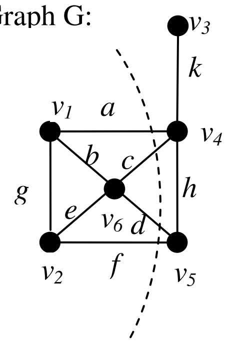
7. Define degree-constrained shortest spanning tree.

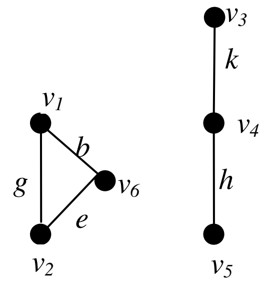
A shortest spanning tree T for a weighted connected graph G with a constraint $d(v_i) \leq k$ for all vertices in T. for k=2, the tree will be Hamiltonian path.

8. Define cut sets and give example.

In a connected graph G, a cut-set is a set of edges whose removal from G leave the graph G disconnected.

Graph G:





Disconnected graph G with 2 components after removing cut set {a, c, d, f}

Possible cut sets are {a, c, d, f}, {a, b, e, f}, {a, b, g}, {d, h, f}, {k}, and so on.
{a, c, h, d} is not a cut set, because its proper subset {a, c, h} is a cut set.
{g, h} is not a cut set.

A minimal set of edges in a connected graph whose removal reduces the rank by one is called minimal cut set (simple cut-set or cocycle). Every edge of a tree is a cut set.

9. Write the Properties of cut set

- Every cut-set in a connected graph G must contain at least one branch of every spanning tree of G.
- In a connected graph G, any minimal set of edges containing at least one branch of every spanning tree of G is a cut-set.
- Every circuit has an even number of edges in common with any cut set.

10. Define Fundamental circuits

Adding just one edge to a spanning tree will create a cycle; such a cycle is called a **fundamental cycle (Fundamental circuits)**. There is a distinct fundamental cycle for each edge; thus, there is a one-to-one correspondence between fundamental cycles and edges not in the spanning tree. For a connected graph with V vertices, any spanning tree will have $V - 1$ edges, and thus, a graph of E edges and one of its spanning trees will have $E - V + 1$ fundamental cycles.

11. Define Fundamental cut sets

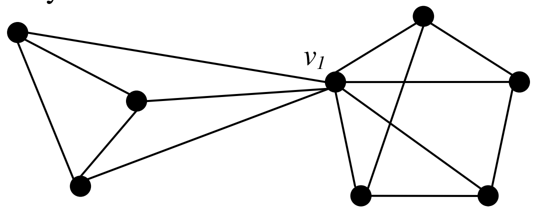
Dual to the notion of a fundamental cycle is the notion of a **fundamental cutset**. By deleting just one edge of the spanning tree, the vertices are partitioned into two disjoint sets. The fundamental cutset is defined as the set of edges that must be removed from the graph G to accomplish the same partition. Thus, each spanning tree defines a set of $V - 1$ fundamental cutsets, one for each edge of the spanning tree.

12. Define edge Connectivity.

Each cut-set of a connected graph G consists of certain number of edges. The number of edges in the smallest cut-set is defined as the **edge Connectivity of G.**

The **edge Connectivity** of a connected graph G is defined as the minimum number of edges whose removal reduces the rank of graph by one.

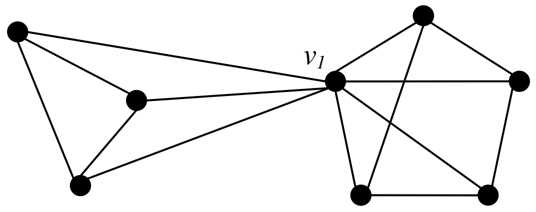
The edge Connectivity of a tree is one.



The edge Connectivity of the above graph G is three.

13. Define vertex Connectivity

The **vertex Connectivity** of a connected graph G is defined as the minimum number of vertices whose removal from G leaves the remaining graph disconnected. The vertex Connectivity of a tree is one.

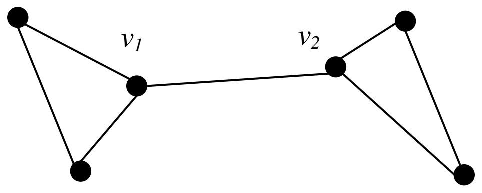


The vertex Connectivity of the above graph G is one.

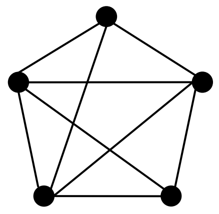
14. Define separable and non-separable graph.

A connected graph is said to be separable graph if its vertex connectivity is one. All other connected graphs are called non-separable graph.

Separable Graph G:

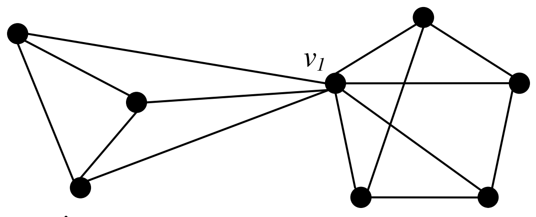


Non-Separable Graph H:



15. Define articulation point.

In a separable graph a vertex whose removal disconnects the graph is called *a cut-vertex, a cut-node, or an articulation point.*



v_1 is an articulation point.

16. What is Network flows

A **flow network** (also known as a transportation **network**) is a **graph** where each edge has a capacity and each edge receives a **flow**. The amount of **flow** on an edge cannot exceed the capacity of the edge.

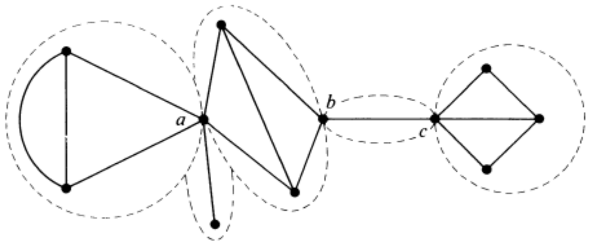
17. Define max-flow and min-cut theorem (equation).

The maximum flow between two vertices a and b in a flow network is equal to the minimum of the capacities of all cut-sets with respect to a and b.

The max. flow between two vertices = Min. of the capacities of all cut-sets.

18. Define component (or block) of graph.

A separable graph consists of two or more non separable subgraphs. Each of the largest nonseparable is called a block (or component).



The above graph has 5 blocks.

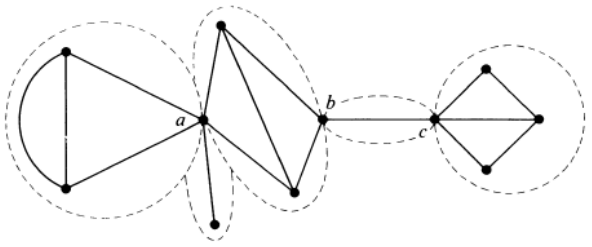
19. Define 1-Isomorphism

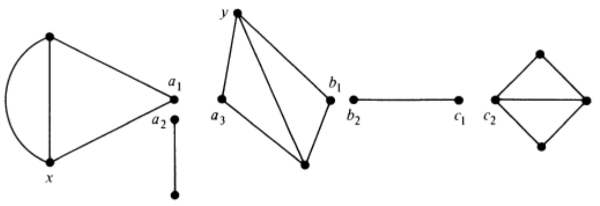
A graph G_1 was 1-Isomorphic to graph G_2 if the blocks of G_1 were isomorphic to the blocks of G_2.

Two graphs G_1 and G_2 are said to be 1-Isomorphic if they become isomorphic to each other under repeated application of the following operation.

Operation 1: "Split" a cut-vertex into two vertices to produce two disjoint subgraphs.

Graph G_1: Graph G_2:





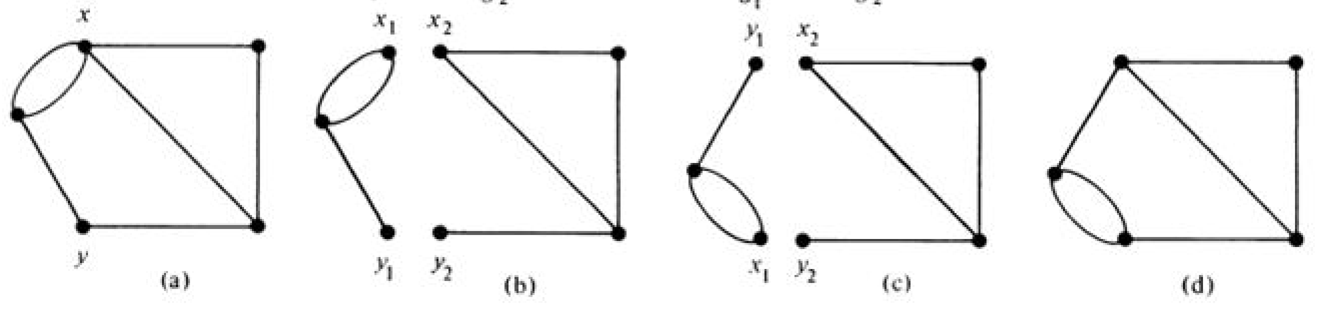
Graph G_1 is 1-Isomorphism with Graph G_2.

20. Define 2-Isomorphism

Two graphs G_1 and G_2 are said to be **2-Isomorphic** if they become isomorphic after undergoing *operation 1* or *operation 2*, or both operations any number of times.

Operation 1: "Split" a cut-vertex into two vertices to produce two disjoint subgraphs.

Operation 2: "Split" the vertex x into x_1 and x_2 and the vertex y into y_1 and y_2 such that G is split into g_1 and g_2. Let vertices x_1 and y_1 go with g_1 and vertices x_2 and y_2 go with g_2. Now rejoin the graphs g_1 and g_2 by merging x_1 with y_2 and x_2 with y_1.



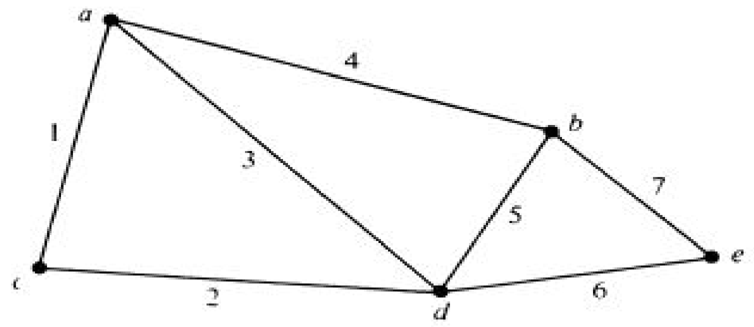
21. Briefly explain Combinational and geometric graphs

An abstract graph G can be defined as G = (V, E, Ψ)

Where the set V consists of five objects named *a, b, c, d*, and *e*, that is, $V = \{ a, b, c, d, e \}$ and the set E consist of seven objects named 1, 2, 3, 4, 5, 6, and 7, that is, $E = \{ 1, 2, 3, 4, 5, 6, 7\}$, and the relationship between the two sets is defined by the mapping Ψ, which consist of

Ψ= [1→(a, c), 2→(c, d) , 3→(a, d) , 4→(a, b) , 5→(b, d) , 6→(d, e) , 7→(b, e)].
Here the symbol 1→(*a, c)*, says that object 1 from set E is mapped onto the pair (a, c) of objects from set V.

This combinatorial abstract object G can also be represented by means of a geometric figure.



The figure is one such geometric representation of this graph G.

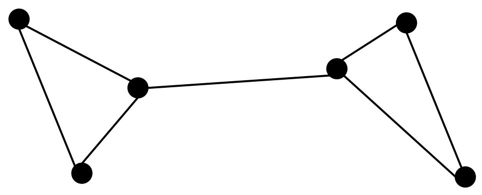
Any graph can be geometrically represented by means of such configuration in three dimensional Euclidian space. Prepared by G. Appasami, Assistant professor, Dr. pauls Engineering College.

22. Distinguish between Planar and non-planar graphs

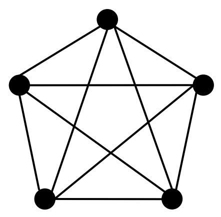
A graph G is said to be *planar* if there exists some geometric representation of G which can be drawn on a plan such that no two of its edges intersect.

A graph that cannot be drawn on a plan without crossover its edges is called *non-planar.*

Planar Graph G:



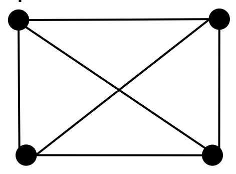
Non-planar Graph H:



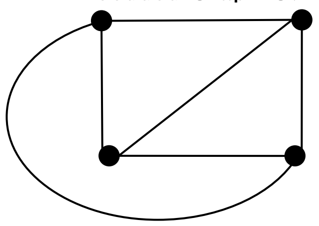
23. Define embedding graph.

A drawing of a geometric representation of a graph on any surface such that no edges intersect is called embedding.

Graph G:

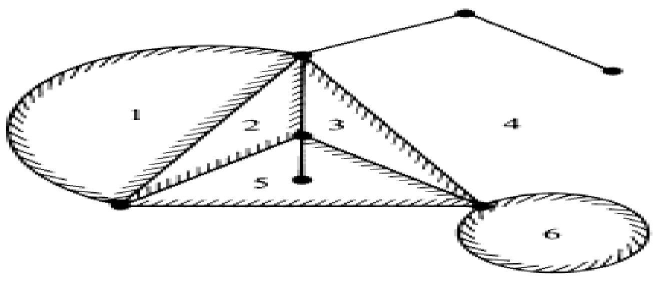


Embedded Graph G:



24. Define region in graph.

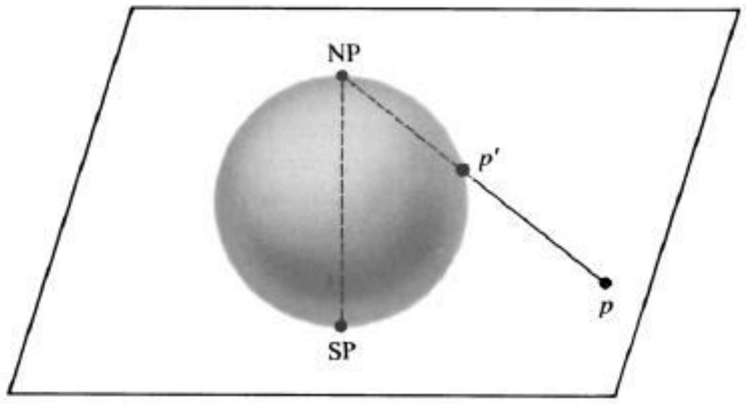
In any planar graph, drawn with no intersections, the edges divide the planes into different **regions (windows, faces, or meshes)**. The regions enclosed by the planar graph are called **interior faces** of the graph. The region surrounding the planar graph is called the **exterior** (or infinite or unbounded) face of the graph. Prepared by G. Appasami, Assistant professor, Dr. pauls Engineering College.



The graph has 6 regions.

25. Why the graph is embedding on sphere.

To eliminate the distinction between finite and infinite regions, a planar graph is often embedded in the surface of sphere. This is done by stereographic projection.



UNIT III MATRICES, COLOURING AND DIRECTED GRAPH

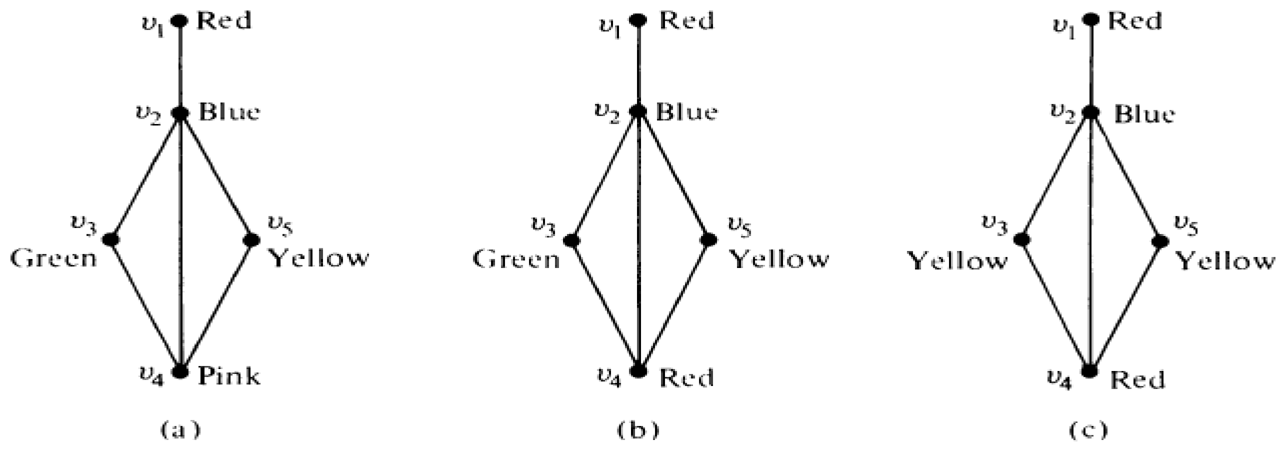
1. What is proper coloring?

Painting all the vertices of a graph with colors such that no two adjacent vertices have the same color is called the *proper coloring* (simply *coloring*) of a graph. A graph in which every vertex has been assigned a color according to a proper coloring is called a *properly colored graph.*

2. Define Chromatic number

A graph G that requires k different colors for its proper coloring, and no less, is called *k-chromatic* graph, and the number k is called the *chromatic number* of G.

The minimum number of colors required for the proper coloring of a graph is called *Chromatic number.*



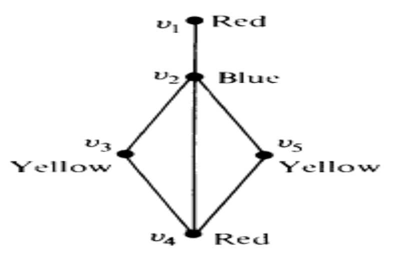
The above graph initially colored with 5 different colors, then 4, and finally 3. So the chromatic number is 3. i.e., The graph is 3-chromatic.

3. Write the properties of chromatic numbers (observations).

- A graph consisting of only isolated vertices is 1-chromatic.
- Every tree with two or more vertices is 2-chromatic.
- A graph with one or more vertices is at least 2-chromatic.
- A graph consisting of simply one circuit with $n \geq 3$ vertices is 2-chromatic if n is even and 3-chromatic if n is odd.
- A complete graph consisting of n vertices is n-chromatic.

4. Define Chromatic partitioning

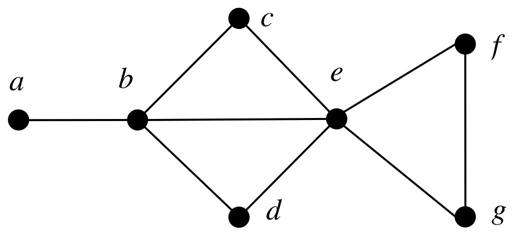
A proper coloring of a graph naturally induces a partitioning of the vertices into different subsets based on colors.



For example, the coloring of the above graph produces the portioning $\{v_1, v_4\}$, $\{v_2\}$, and $\{v_3, v_5\}$.

5. Define independent set and maximal independent set.

A set of vertices in a graph is said to be an *independent set* of vertices or simply independent set (or an internally stable set) if two vertices in the set are adjacent.



For example, in the above graph produces {a, c, d} is an independent set.

A single vertex in any graph constitutes an independent set.

A maximal independent set is an independent set to which no other vertex can be added without destroying its independence property.

{a, c, d, f} is one of the maximal independent set. {b, f} is one of the maximal independent set.

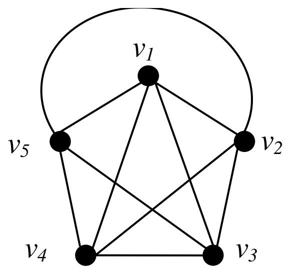
The number of vertices in the largest independent set of a graph G is called the *independence number* (or coefficients of internal stability), denoted by β(G).

For a K-chromatic graph of n vertices, the independence number β(G) $\geq \frac{n}{k}$.

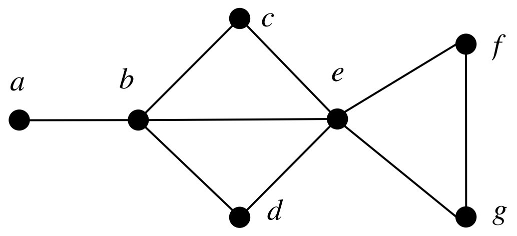
6. Define uniquely colorable graph.

A graph that has only one chromatic partition is called a uniquely colorable graph. For example,

Uniquely colorable graph G:

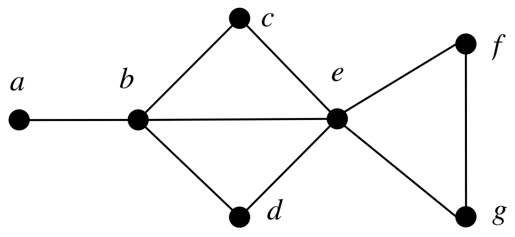


Not uniquely colorable graph H:



7. Define dominating set.

A dominating set (or an externally stable set) in a graph G is a set of vertices that dominates every vertex v in G in the following sense: Either v is included in the dominating set or is adjacent to one or more vertices included in the dominating set.



{b, g} is a dominating set, {a, b, c, d, f} is a dominating set. A is a dominating set need not be independent set. Set of all vertices is a dominating set.

A minimal dominating set is a dominating set from which no vertex can be removed without destroying its dominance property.

{b, e} is a minimal dominating set.

8. Define Chromatic polynomial.

A graph G of n vertices can be properly colored in many different ways using a sufficiently large number of colors. This property of a graph is expressed elegantly by means of polynomial. This polynomial is called the Chromatic polynomial of G.

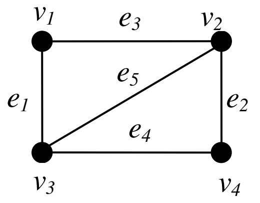
The value of the Chromatic polynomial $P_n(\lambda)$ of a graph with n vertices the number of ways of properly coloring the graph , using λ or fewer colors.

9. Define Matching (Assignment).

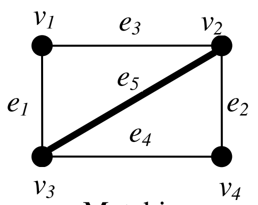
A *matching* in a graph is a subset of edges in which no two edges are adjacent. A single edge in a graph is a matching.

A *maximal matching* is a matching to which no edge in the graph can be added.

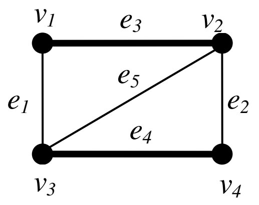
The maximal matching with the largest number of edges are called the *largest maximal matching*.



Graph G



Matching



Maximal matching

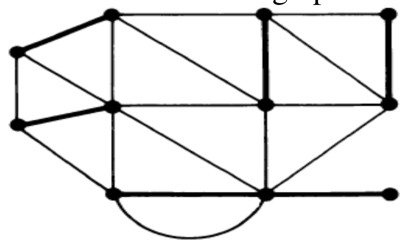
10. What is Covering?

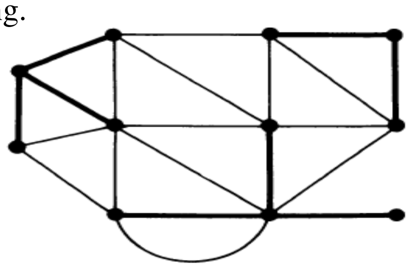
A set g of edges in a graph G is said to be cover og G if every vertex in G is incident on at least one edge in g. A set of edges that covers a graph G is said to be a covering (or an edge covering, or a coverring subgraph) of G.

Every graph is its own covering.

A spanning tree in a connected graph is a covering.

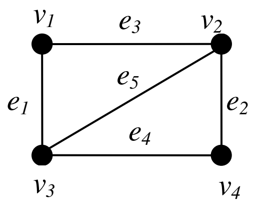
A Hamiltonian circuit in a graph is also a covering.



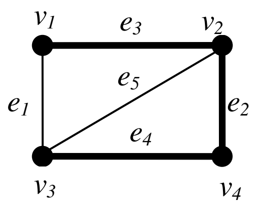


11. Define minimal cover.

A **minimal covering** is a covering from which no edge can be removed without destroying it ability to cover the graph G.



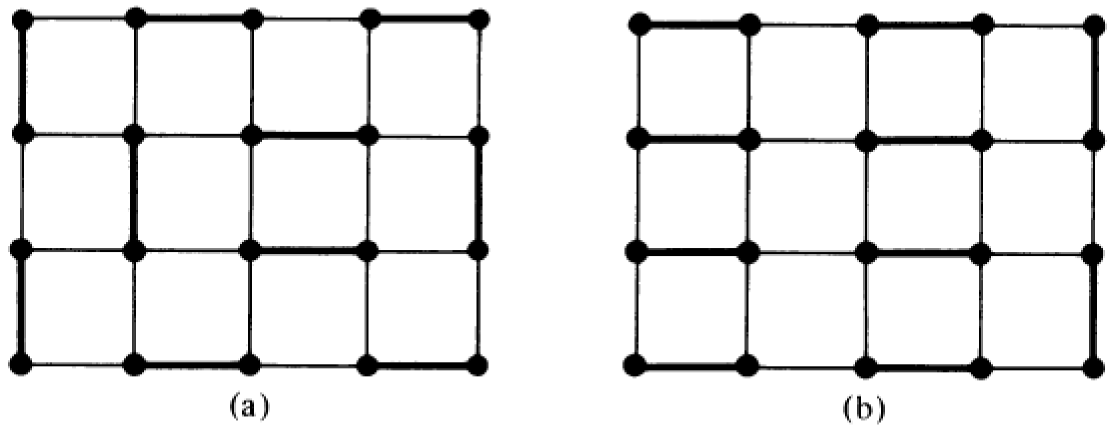
Graph G



Minimal cover

12. What is dimer covering?

A covering in which every vertex is of degree one is called a *dimer covering* or a *1-factor*. A dimmer covering is a maximal matching because no two edges in it are adjacent. Prepared by G. Appasami, Assistant professor, Dr. pauls Engineering College.



Two dimmer coverings.

13. Define four color problem / conjecture.

- Every planar graph has a chromatic number of four or less.
- Every triangular planar graph has a chromatic number of four or less.
- The regions of every planar, regular graph of degree three can be colored properly with four colors.

14. State five color theorem

Every planar map can be properly colored with five colors.
i.e., the vertices of every plannar graph can be properly colored with five colors.

15. Write about vertex coloring and region coloring.

A graph has a dual if and only if it is planar. Therefore, coloring the regions of a planar graph G is equivalent to coloring the vertices of its dual G* and vice versa.

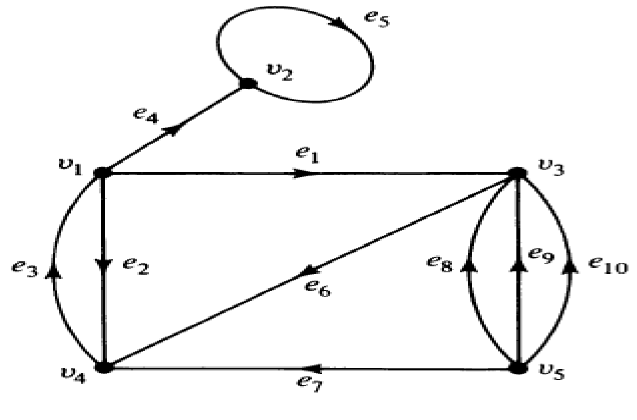
What is meant by regularization of a planar graph?

- Remove every vertex of degree one from the graph G does not affect the regions of a plannar graph.
- Remove every vertex of degree two and merge the two edges in series from the graph G.
- Such a transformation may be called regularization of a planar graph.

16. Directed graphs

A *directed graph* (or a *digraph, or an oriented graph*) G consists of a set of vertices $V = \{ v_1, v_2, \ldots \}$, a set of edges $E = \{ e_1, e_2, \ldots \}$, and a mapping Ψ that maps every edge onto some ordered pair of vertices (v_i, v_j).

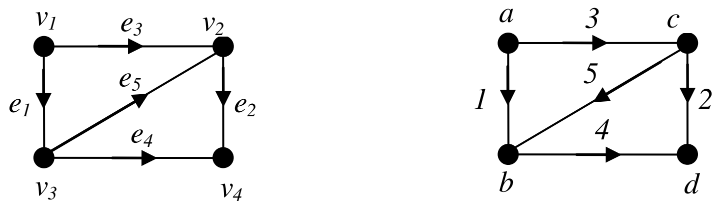
For example,



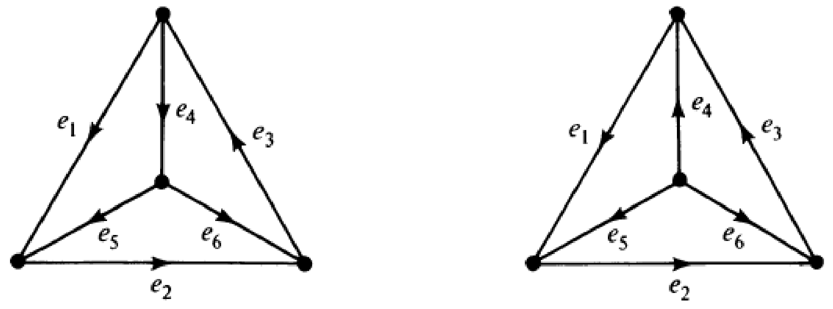
17. Define isomorphic digraph.

Among directed graphs, if their labels are removed, two isomorphic graphs are indistinguishable then these graphs are **isomorphic digraph**.

For example,



Two isomorphic digraphs.



Two non-isomorphic digraphs.

18. List out some types of directed graphs

- Simple Digraphs
- Asymmetric Digraphs (Anti-symmetric)
- Symmetric Digraphs
- Simple Symmetric Digraphs
- Simple Asymmetric Digraphs
- Complete Digraphs
- Complete Symmetric Digraphs
- Complete Asymmetric Digraphs (tournament)
- Balance digraph (a pseudo symmetric digraph or an isograph)

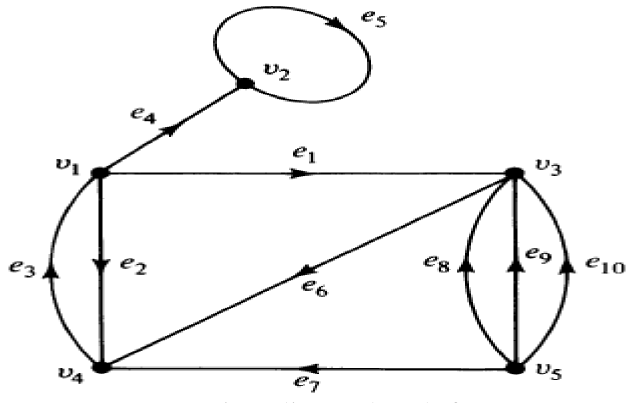
19. Define binary relations.

In a set of objects, X, where X={x_1, x_2, …}, A *binary relation R* between pairs (x_i, x_j) can be written as x_i R x_j and say that x_i has relation R to x_j.

If the binary relation R is reflexive, symmetric, and transitive then R is an equivalence relation. This produces equivalence classes.

20. What is Directed path?

A path in a directed graph is called Directed path.



v_5 e_8 v_3 e_6 v_4 e_3 v_1 is a directed path from v_5 to v_1.
Whereas v_5 e_7 v_4 e_6 v_3 e_1 v_1 is a semi-path from v_5 to v.

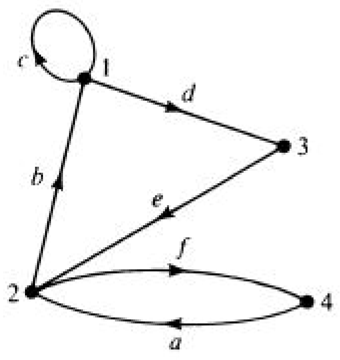
21. Write the types of connected digraphs

- **Strongly connected digraph:** A digraph G is said to be strongly connected if there is at least one directed path from every vertex to every other vertex.
- **Weakly connected digraph**: A digraph G is said to be weakly connected if its corresponding undirected graph is connected. But G is not strongly connected.

22. Define Euler digraphs

In a digraph G, a closed directed walk which traverses every edge of G exactly once is called a *directed Euler line*. A digraph containing a directed Euler line is called an **Euler digraphs**

For example,



It contains directed Euler line **a b c d e f.**

23. What is teleprinter's problem.

Constructing a longest circular sequence of 1's and 0's such that no subsequence of r bits appears more than once in the sequence.

Teleprinter's problem was solved in 1940 by I.G. Good using digraph.

UNIT IV PERMUTATIONS & COMBINATIONS

1. Define Fundamental principles of counting

The Fundamental Counting Principle is a way to figure out the total number of ways different events can occur.

If the first task can be performed in m ways, while a second task can be performed in n ways, and the two tasks cannot be performed simultaneously, then performing either task can be accomplished in any one of $m + n$ ways.

If a procedure can be broken into first and second stages, and if there are *m* possible outcomes for the first stage and if, for each of these outcomes, there are *n* possible outcomes for the second stage, then the total procedure can be carried out, in the designed order, in *mn* ways.

2. Define rule of sum.

If the first task can be performed in m ways, while a second task can be performed in n ways, and the two tasks cannot be performed simultaneously, then performing either task can be accomplished in any one of $m + n$ ways.

Example: A college library has 40 books on C++ and 50 books on Java. A student at this college can select 40+50=90 books to learn programming language.

3. Define rule of Product

If a procedure can be broken into first and second stages, and if there are *m* possible outcomes for the first stage and if, for each of these outcomes, there are *n* possible outcomes for the second stage, then the total procedure can be carried out, in the designed order, in *mn* ways.

Example: A drama club with six men and eight can select male and female role in 6 x 8 = 48 ways.

4. Define Permutations

For a given collection of n objects, any linear arrangement of these objects is called a permutation of the collection. Counting the linear arrangement of objects can be done by rule of product.

For a given collection of *n* distinct objects, and *r* is an integer, with 1 ≤ r ≤ n, then by rule of product, the number of permutations of size r for the n objects is

$$P(n,r) = n \times (n-1) \times (n-2) \times \dots \times (n-r+1) = \frac{n!}{(n-r)!}, \quad 0 \le r \le n$$

Example: In a class of 10 students, five are to be chosen and seated in a row for a picture.

The total number of arrangements = 10 x 9 x 8 x 7 x 6 = 30240.

5. Define combinations

For a given collection of *n* objects, each selection, or combination, of *r* of these objects, with no reference to order, corresponds to *r*! (Permutations of size *r* from the *n* objects). Thus the number of combinations of size *r* from a collection of size *n* is

$$C(n,r) = \frac{P(n,r)}{r!} = \frac{n!}{r!\,(n-r)!}, \quad 0 \le r \le n$$

Example: In a test, students are directed to answer 7 questions out of 10. The student can answer the examination in

$$C(n,r) = C(10,7) = \frac{10!}{7!\,(10-7)!} = \frac{10 \times 9 \times 8}{3 \times 2 \times 1} = 120\ ways$$

6. State Binomial theorem

The Binomial theorem: If *x* and *y* are variables and n is a positive integer, then

$$(x+y)^n = \binom{n}{0} x^0 y^n + \binom{n}{1} x^1 y^{n-1} + \binom{n}{2} x^2 y^{n-2} + \cdots$$
$$\binom{n}{n-1} x^{n-1} y^1 + \binom{n}{n} x^n y^0 = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k}$$

$\binom{n}{k}$ is referred as Binomial coefficient.

7. Define combinations with repetition

If there is a selection with repetition, r of n distinct objects, then the combinations with of n objects taken r at a time with repetition is $C(n+r-1,r)$.

$$C(n+r-1,r) = \frac{(n+r-1)!}{r!\,(n-1)!} = \binom{n+r-1}{r}$$

Example: A donut shop offers 20 kinds of donuts. Assuming that there are at least a dozen of each kind when we enter the shop. We can select a dozen donuts in $C(20+12-1,12)$= $C(31,12)$ = 141120525 ways

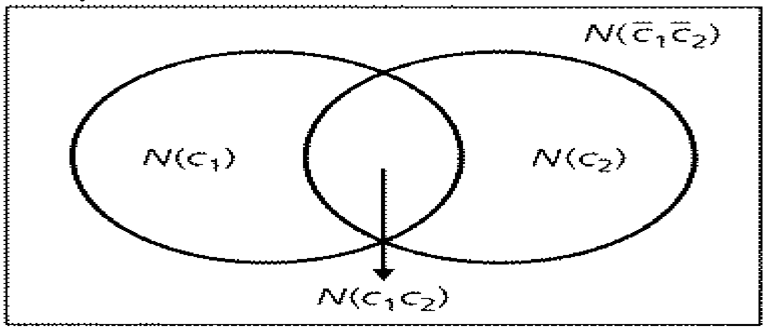
8. Define Catalan numbers

The Catalan numbers form a sequence of natural numbers that occur in various counting problems, often involving recursively-defined objects. They are named after the Belgian mathematician Eugène Charles Catalan. the nth Catalan number is given directly in terms of binomial coefficients by Prepared by G. Appasami, Assistant professor, Dr. pauls Engineering College.

$$C_n = \binom{2n}{n} - \binom{2n}{n+1} = \frac{(2n)!}{(n+1)!\,n!} = \frac{1}{n+1}\binom{2n}{n} \qquad \text{for } n \geq 0$$

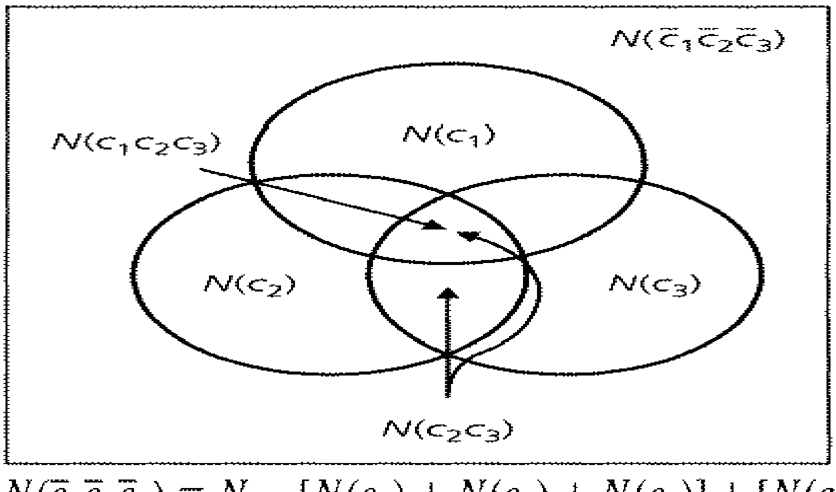
9. Write the Principle of inclusion and exclusion formula.

For any 2 sets, C_1 and C_2,



$$N(\bar{c}_1\bar{c}_2) = N - [N(c_1) + N(c_2)] + N(c_1c_2)$$

For any 3 sets, C_1, C_2 and C_3,



$$N(\bar{c}_1\bar{c}_2\bar{c}_3) = N - [N(c_1) + N(c_2) + N(c_3)] + [N(c_1c_2) + N(c_1c_3) + N(c_2c_3)] - N(c_1c_2c_3).$$

For any 4 sets, C_1, C_2 , C_2 and C_4,

$$\begin{aligned} N(\bar{c}_1\bar{c}_2\bar{c}_3\bar{c}_4) = N &- [N(c_1) + N(c_2) + N(c_3) + N(c_4)] \\ &+ [N(c_1c_2) + N(c_1c_3) + N(c_1c_4) + N(c_2c_3) + N(c_2c_4) + N(c_3c_4)] \\ &- [N(c_1c_2c_3) + N(c_1c_2c_4) + N(c_1c_3c_4) + N(c_2c_3c_4)] \\ &+ N(c_1c_2c_3c_4). \end{aligned}$$

10. Define Derangements

A derangement is a permutation of the elements of a set, such that no element appears in its original position.

The number of derangements of a set of size n, usually written D_n, d_n, or !n, is called the "derangement number" or "de Montmort number".

Example: The number of derangements of 1, 2, 3, 4 is

$d_4 = 4!\ [1 – 1 + (1/2!)-(1/3!)+(1/4!)] = 9$.

11. What is meant by Arrangements with forbidden (banned) positions.

The number of acceptable assignments is equal to the number of ways of placing nontaking rooks on this chessboard so that none of the rooks is in a forbidden position. The key to determining this number of arrangements is the inclusion- exclusion principle.

UNIT V GENERATING FUNCTIONS

1. Define Generating function.

A **generating function** describes an infinite sequence of numbers (a_n) by treating them like the coefficients of a series expansion. The sum of this infinite series is the generating function. Unlike an ordinary series, this formal series is allowed to diverge, meaning that the generating function is not always a true function and the "variable" is actually an indeterminate.

The generating function for 1, 1, 1, 1, 1, 1, 1, 1, 1, ..., whose ordinary generating function is

$$\sum_{n=0}^{\infty} (x)^n = \frac{1}{1-x}$$

The generating function for the geometric sequence 1, *a*, a^2, a^3, ... for any constant *a*:

$$\sum_{n=0}^{\infty} (ax)^n = \frac{1}{1-ax}$$

2. What is Partitions of integer?

Partitioning a positive n into positive summands and seeking the number of such partitions without regard to order is called Partitions of integer.

This number is denoted by *p(n)*. For example

P(1) = 1:	1
P(2) = 2:	2 = 1 + 1
P(3) = 3:	3 = 2 +1 = 1 + 1 +1
P(4) = 5:	4 = 3 + 1 = 2 + 2 = 2 + 1 + 1 = 1 + 1 + 1 + 1
P(5) = 7:	5 = 4 + 1 = 3 + 2 = 3 + 1 + 1 = 2 + 2 + 1 = 2 + 1 + 1+ 1 = 1 + 1 + 1 + 1 + 1

3. Define Exponential generating function

For a sequence $a_0, a_1, a_2, a_3,, \ldots$ of real numbers.

$$f(x) = a_0 + a_1 x + a_2 \frac{x^2}{2!} + a_3 \frac{x^3}{3!} + \cdots = \sum_{i=0}^{\infty} a_i \frac{x^{i^i}}{i!}$$

is called the exponential generating function for the given sequence.

4. Define Maclaurin series expansion of e^x and e^{-x}.

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots$$
$$e^{-x} = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \frac{x^4}{4!} - \cdots$$

Adding these two series together, we get,

$$e^x + e^{-x} = 2(1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \cdots)$$
$$\frac{e^x + e^{-x}}{2} = 1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \cdots$$

5. Define Summation operator

Generating function for a sequence $a_0, a_0 + a_1, a_0 + a_1 + a_2, a_0 + a_1 + a_2 + a_3,, \ldots$.

For $(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots$,, consider the function *f(x)/*(1-*x)*

$$\frac{f(x)}{1-x} = f(x).\frac{1}{1-x} = [a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots][1 + x + x^2 + x^3 + \cdots]$$
$$= a_0 + (a_0 + a_1)x + (a_0 + a_1 + a_2)x^2 + +(a_0 + a_1 + a_2 + a_3)x^3 + \cdots$$

So *f(x)/*(1-*x)* generates the sequence of sums $a_0, a_0 + a_1, a_0 + a_1 + a_2, a_0 + a_1 + a_2 + a_3,,$

1/(1-*x*) is called the summation operator.

6. What is Recurrence relation?

A **recurrence relation** is an equation that recursively defines a sequence or multidimensional array of values, once one or more initial terms are given: each further term of the sequence or array is defined as a function of the preceding terms.

The term **difference equation** sometimes (and for the purposes of this article) refers to a specific type of recurrence relation. However, "difference equation" is frequently used to refer to *any* recurrence relation.

7. Write Fibonacci numbers and relation

The recurrence satisfied by the Fibonacci numbers is the archetype of a homogeneous linear recurrence relation with constant coefficients (see below). The Fibonacci sequence is defined using the recurrence

$F_n = F_{n-1} + F_{n-2}$

with seed values $F_0 = 0$ and $F_1 = 1$

We obtain the sequence of Fibonacci numbers, which begins

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

8. Define First order linear recurrence relation

The general form of First order linear homogeneous recurrence relation can be written as

$a_{n+1} = d\,a_n$, $n \geq 0$, where d is a constant. The relation is first order since a_{n+1} depends on a_n.

a_0 *or* a_1 are called boundary conditions.

9. Define Second order recurrence relation

Let $k \in \mathbf{Z}^+$ and $C_0\ (\neq 0), C_1, C_2, \ldots, C_k\ (\neq 0)$ be real numbers. If a_n, for $n \geq 0$, is a discrete function, then

$$C_0a_n + C_1a_{n-1} + C_2a_{n-2} + \cdots + C_ka_{n-k} = f(n), \qquad n \geq k,$$

is a linear recurrence relation (with constant coefficients) of *order k*. When $f(n) = 0$ for all $n \geq 0$, the relation is called *homogeneous*; otherwise, it is called *nonhomogeneous*.

In this section we shall concentrate on the homogeneous relation of order two:

$$C_0a_n + C_1a_{n-1} + C_2a_{n-2} = 0, \qquad n \geq 2.$$

On the basis of our work in Section 10.1, we seek a solution of the form $a_n = cr^n$, where $c \neq 0$ and $r \neq 0$.

Substituting $a_n = cr^n$ into $C_0a_n + C_1a_{n-1} + C_2a_{n-2} = 0$, we obtain

$$C_0cr^n + C_1cr^{n-1} + C_2cr^{n-2} = 0.$$

With $c, r \neq 0$, this becomes $C_0r^2 + C_1r + C_2 = 0$, a quadratic equation which is called the *characteristic equation*. The roots r_1, r_2 of this equation determine the following three cases: (a) r_1, r_2 are distinct real numbers; (b) r_1, r_2 form a complex conjugate pair; or (c) r_1, r_2 are real, but $r_1 = r_2$. In all cases, r_1 and r_2 are called the *characteristic roots*.

10. Briefly explain Non-homogeneous recurrence relation.

We now turn to the recurrence relations

$$a_n + C_1 a_{n-1} = f(n), \qquad n \geq 1, \tag{1}$$

$$a_n + C_1 a_{n-1} + C_2 a_{n-2} = f(n), \qquad n \geq 2, \tag{2}$$

where C_1 and C_2 are constants, $C_1 \neq 0$ in Eq. (1), $C_2 \neq 0$, and $f(n)$ is not identically 0. Although there is no general method for solving all nonhomogeneous relations, for certain functions $f(n)$ we shall find a successful technique.

We start with the special case for Eq. (1), when $C_1 = -1$. For the nonhomogeneous relation $a_n - a_{n-1} = f(n)$, we have

$$\begin{aligned}
a_1 &= a_0 + f(1) \\
a_2 &= a_1 + f(2) = a_0 + f(1) + f(2) \\
a_3 &= a_2 + f(3) = a_0 + f(1) + f(2) + f(3) \\
&\vdots \\
a_n &= a_{n-1} + f(n) = a_0 + f(1) + \cdots + f(n) = a_0 + \sum_{i=1}^{n} f(i).
\end{aligned}$$

We can solve this type of relation in terms of n, if we can find a suitable summation formula for $\sum_{i=1}^{n} f(i)$.

CS6702 GRAPH THEORY AND APPLICATIONS
QUESTION BANK

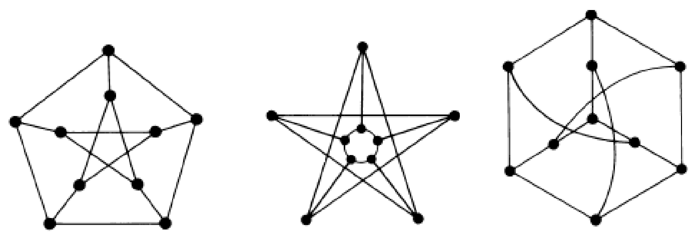
UNIT I INTRODUCTION

PART – A

1. Define Graph.
2. Define Simple graph.
3. Write few problems solved by the applications of graph theory.
4. Define incidence, adjacent and degree.
5. What are finite and infinite graphs?
6. Define Isolated and pendent vertex.
7. Define null graph.
8. Define Multigraph
9. Define complete graph
10. Define Regular graph
11. Define Cycles
12. Define Isomorphism.
13. What is Subgraph?
14. Define Walk, Path and Circuit.
15. Define connected graph. What is Connectedness?
16. Define Euler graph.
17. Define Hamiltonian circuits and paths
18. Define Tree
19. List out few Properties of trees.
20. What is Distance in a tree?
21. Define eccentricity and center.
22. Define distance metric.
23. What are the Radius and Diameter in a tree.
24. Define Rooted tree
25. Define Rooted binary tree

PART – B

1. Explain various applications of graph.
2. Define the following kn, cn, kn,n, dn, trail, walk, path, circuit with an example.
3. Show that a connected graph G is an Euler graph iff all vertices are even degree.
4. Prove that a simple graph with n vertices and k components can have at most (n-k)(n-k+1)/2 edges.



5. Are they isomorphic?
6. Prove that in a complete graph with n vertices there are (n-1)/2 edges-disjoint Hamiltonian circuits, if n is odd number ≥3.
7. Prove that, there is one and only one path between every pair of vertices in a tree T.
8. Prove the given statement, "A tree with *n* vertices has *n*-1 edges".
9. Prove that, any connected graph with *n* vertices has *n*-1 edges is a tree.
10. Show that a graph is a tree if and only if it is minimally connected.
11. Prove that, a graph G with *n* vertices has *n*-1 edges and no circuits are connected.

UNIT II TREES, CONNECTIVITY & PLANARITY

PART – A

1. Define Spanning trees.
2. Define Branch and chord.
3. Define complement of tree.
4. Define Rank and Nullity.
5. How Fundamental circuits created?
6. Define Spanning trees in a weighted graph.
7. Define degree-constrained shortest spanning tree.
8. Define cut sets and give example.
9. Write the Properties of cut set
10. Define Fundamental circuits
11. Define Fundamental cut sets
12. Define edge Connectivity.
13. Define vertex Connectivity.
14. Define separable and non-separable graph.
15. Define articulation point.
16. What is Network flows.
17. Define max-flow and min-cut theorem (equation).
18. Define component (or block) of graph.
19. Define 1-Isomorphism.
20. Define 2-Isomorphism.
21. Briefly explain Combinational and geometric graphs.
22. Distinguish between Planar and non-planar graphs.
23. Define embedding graph.
24. Define region in graph.
25. Why the graph is embedding on sphere.

PART – B

1. Find the shortest spanning tree for the following graph.

$$\begin{array}{c|cccccc|}
 & v_1 & v_2 & v_3 & v_4 & v_5 & v_6 \\
\hline
v_1 & - & 10 & 16 & 11 & 10 & 17 \\
v_2 & 10 & - & 9.5 & \infty & \infty & 19.5 \\
v_3 & 16 & 9\ 5 & - & 7 & \infty & 12 \\
v_4 & 11 & \infty & 7 & - & 8 & 7 \\
v_5 & 10 & \infty & \infty & 8 & - & 9 \\
v_6 & 17 & 19.5 & 12 & 7 & 9 & -
\end{array}$$

2. Explain 1 - isomarphism and 2 - isomarphism of graphs with your own example.
3. Prove that a connected graph G with *n* vertices and *e* edges has *e-n+2* regions.
4. Write all possible spanning tree for K5.
5. Prove that every cut-set in a connected graph G must contain at least one branch of every spanning tree of G.
6. Prove that the every circuit which has even number of edges in common with any cut-set.
7. Show that the ring sum of any two cut-sets in a graph is either a third cut set or en edge disjoint union of cut sets.
8. Explain network flow problem in detail.
9. If G_1 and G_2 are two 1-isomorphic graphs, the rank of G_1 equals the rank of G_2 and the nullity of G_1 equals the nullity of G_2, prove this.
10. Prove that any two graphs are 2-isomorphic if and only if they have circuit correspondence.

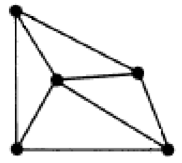
UNIT III MATRICES, COLOURING AND DIRECTED GRAPH

PART – A

1. What is proper coloring?
2. Define Chromatic number
3. Write the properties of chromatic numbers (observations).
4. Define Chromatic partitioning
5. Define independent set and maximal independent set.
6. Define uniquely colorable graph.
7. Define dominating set.
8. Define Chromatic polynomial.
9. Define Matching (Assignment).
10. What is Covering?
11. Define minimal cover.
12. What is dimer covering?
13. Define four color problem / conjecture.
14. State five color theorem
15. Write about vertex coloring and region coloring.
16. What is meant by regularization of a planar graph?
17. Define Directed graphs .
18. Define isomorphic digraph.
19. List out some types of directed graphs.
20. Define Simple Digraphs.
21. Define Asymmetric Digraphs (Anti-symmetric).
22. What is meant by Symmetric Digraphs?
23. Define Simple Symmetric Digraphs.
24. Define Simple Asymmetric Digraphs.
25. Give example for Complete Digraphs.
26. Define Complete Symmetric Digraphs.
27. Define Complete Asymmetric Digraphs (tournament).
28. Define Balance digraph (a pseudo symmetric digraph or an isograph).
29. Define binary relations.
30. What is Directed path?
31. Write the types of connected digraphs.
32. Define Euler graphs.

PART – B

1. Prove that any simple planar graph can be embedded in a plane such that every edge is drawn as a straight line.
2. Show that a connected planar graph with n vertices and e edges has e-n+2 regions.
3. Define chromatic polynomial. Find the chromatic polynomial for the following graph.



4. Explain matching and bipartite graph in detail.
5. Write the observations of minimal covering of a graph.
6. Prove that the vertices of every planar graph can be properly colored with five colors.
7. Explain matching in detail.
8. Prove that a covering g of graph G is minimal iff g contains no path of length three or more.
9. Illustrate four-color problem.
10. Explain Euler digraphs in detail.

UNIT IV PERMUTATIONS & COMBINATIONS

PART – A

1. Define Fundamental principles of counting
2. Define rule of sum.
3. Define rule of Product
4. Define Permutations
5. Define combinations
6. State Binomial theorem
7. Define combinations with repetition
8. Define Catalan numbers
9. Write the Principle of inclusion and exclusion formula.
10. Define Derangements
11. What is meant by Arrangements with forbidden (banned) positions.

PART – B

1. Explain the Fundamental principles of counting.
2. Find the number of ways of ways of arranging the word APPASAMIAP and out of it how many arrangements have all A's together.
3. Discuss the rules of sum and product with example.
4. Determine the number of (staircase) paths in the *xy*-plane from (2, 1) to (7, 4), where each path is made up of individual steps going 1 unit to the right (R) or one unit upward (U). iv. Find the coefficient of a^5b^2 in the expansion of $(2a - 3b)^7$.
5. State and prove binomial theorem.
6. How many times the print statement executed in this program segment?

```
for i := 1 to 20 do
  for j := 1 to i do
    for k := 1 to j do
      print (i * j + k)
```

7. Discuss the Principle of inclusion and exclusion.
8. How many integers between 1 and 300 (inc.) are not divisible by at least one of 5, 6, 8?
9. How 32 bit processors address the content? How many address are possible?
10. Explain the Arrangements with forbidden positions.

UNIT V GENERATING FUNCTIONS

PART – A

1. Define Generating function.
2. What is Partitions of integer?
3. Define Exponential generating function
4. Define Maclaurin series expansion of e^x and e^{-x}.
5. Define Summation operator
6. What is Recurrence relation?
7. Write Fibonacci numbers and relation
8. Define First order linear recurrence relation
9. Define Second order recurrence relation
10. Briefly explain Non-homogeneous recurrence relation.

PART – B

1. Explain Generating functions
2. Find the convolution of the sequences 1, 1, 1, 1, ….. and 1,-1,1,-1,1,-1.
3. Find the number of non negative & positive integer solutions of for x1+x2+x3+x4=25.
4. Find the coefficient of x5 in(1-2x)7.
5. The number of virus affected files in a system is 1000 and increases 250% every 2 hours.
6. Explain Partitions of integers
7. Use a recurrence relation to find the number of viruses after one day.
8. Explain First order homogeneous recurrence relations.
9. Solve the recurrence relation an+2-4an+1+3an=-200 with a0=3000 and a1=3300.
10. Solve the Fibonacci relation Fn = Fn-1+Fn-2.
11. Find the recurrence relation from the sequence 0, 2, 6, 12, 20, 30, 42, … .
12. Determine (1+√3i)10.
13. Discuss Method of generating functions.

All the Best – No substitution for hard work.

Reg. No.

MODEL EXAMINATION

Department of Computer Science and Engineering

Subject Name	**: Graph Theory and Applications**	**Date**	**: 30/09/2016**
Subject Code	**: CS6702**	**Duration**	**: 3 Hours**
Year / Sem.	**: IV / VII**	**Marks**	**: 100**

Part-A (10x2=20)
Answer All Questions

1. What are finite and infinite graphs? Give an example.
2. Differentiate regular and complete graph.
3. Define cut sets and give example.
4. Briefly explain Combinational and geometric graphs.
5. Define Chromatic number. Find Chromatic number for K_5.
6. Define Matching (Assignment).
7. A donut shop offers 20 kinds of donuts. Assuming that there is at least a dozen of each kind when we enter the shop. How many ways we can select a dozen donuts?
8. Define Derangements. Find the derangements of 1, 2, 3, 4.
9. What is Partitions of integer?
10. Define Second order recurrence relation. Give an example.

Part-B (5x16=80)

Answer All Questions

11. a. i. Explain various applications of graph. (8)
ii. Define the following k_n, c_n, $k_{n,n}$, d_n, trail, walk, path, circuit with an example. (8)

Or

b. i. Show that a connected graph G is an Euler graph iff all vertices are even degree. (8)
ii. Prove that a simple graph with *n* vertices and *k* components can have at most $(n-k)(n-k+1)/2$ edges. (8)

12. a. i. Find the shortest spanning tree for the following graph. (8)

$$\begin{array}{c|cccccc|}
 & v_1 & v_2 & v_3 & v_4 & v_5 & v_6 \\
v_1 & - & 10 & 16 & 11 & 10 & 17 \\
v_2 & 10 & - & 9.5 & \infty & \infty & 19.5 \\
v_3 & 16 & 9\ 5 & - & 7 & \infty & 12 \\
v_4 & 11 & \infty & 7 & - & 8 & 7 \\
v_5 & 10 & \infty & \infty & 8 & - & 9 \\
v_6 & 17 & 19.5 & 12 & 7 & 9 & -
\end{array}$$

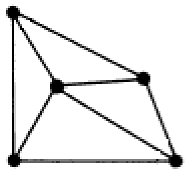
ii. Prove that the every circuit which has even number of edges in common with any cut-set. (8)

Or

b. i. Explain 1 - isomarphism and 2 - isomarphism of graphs with your own example. (8)

ii. Prove that a connected graph G with *n* vertices and *e* edges has e-n+2 regions (8)

13. a. i. Define chromatic polynomial. Find the chromatic polynomial for the following graph. (8)



ii. Explain matching and bipartite graph in detail. (8)

Or

b. i. Write the observations of minimal covering of a graph. (8)

ii. Prove that the vertices of every planar graph can be properly colored with five colors. (8)

14. a. i. Find the number of ways of ways of arranging the word MASSASAUGA and out of it how many arrangements have all A's together. (4)

ii. Discuss the rules of sum and product with example. (4)

iii. Determine the number of (staircase) paths in the *xy*-plane from (2, 1) to (7, 4), where each path is made up of individual steps going 1 unit to the right (R) or one unit upward (U). (4)

iv. Find the coefficient of a^5b^2 in the expansion of $(2a - 3b)^7$. (4)

Or

b. i. State and prove binomial theorem. (8)

ii. How many times the print statement executed in this program segment? (4)

```
for i := 1 to 20 do
   for j := 1 to i do
      for k := 1 to j do
         print (i * j + k)
```

iii. How many integers between 1 and 300 (inc.) are not divisible by at least one of 5, 6, 8? (4)

15. a. i. Find the convolution of the sequences 1, 1, 1, 1, ….. and 1,-1,1,-1,1,-1. (4)

ii. Find the number of non negative & positive integer solutions of for $x_1+x_2+x_3+x_4=25$. (4)

iii. Find the coefficient of x^5 in$(1-2x)^7$. (4)

iv. The number of virus affected files in a system is 1000 and increases 250% every 2 hours. (4) Use a recurrence relation to find the number of viruses after one day.

Or

b. i. Solve the recurrence relation $a_{n+2}-4a_{n+1}+3a_n=-200$ with $a_0=3000$ and $a_1=3300$. (4)

ii. Solve the Fibonacci relation $F_n = F_{n-1}+F_{n-2}$. (4)

iii. Find the recurrence relation from the sequence 0, 2, 6, 12, 20, 30, 42, … . (4)

iv. Determine $(1+\sqrt{3}i)^{10}$. (4)

REFERENCES:

1. Narsingh Deo, “Graph Theory: With Application to Engineering and Computer Science”, Prentice Hall of India, 2003.
2. Grimaldi R.P. “Discrete and Combinatorial Mathematics: An Applied Introduction”, Addison Wesley, 1994.
3. Clark J. and Holton D.A, “A First Look at Graph Theory”, Allied Publishers, 1995.
4. Mott J.L., Kandel A. and Baker T.P. “Discrete Mathematics for Computer Scientists and Mathematicians” , Prentice Hall of India, 1996.
5. Liu C.L., “Elements of Discrete Mathematics”, Mc Graw Hill, 1985.
6. Rosen K.H., “Discrete Mathematics and Its Applications”, Mc Graw Hill, 2007.

Differential Equations

A Differential Equation is an equation with a function and one or more of its derivatives:

$$y + \overset{\substack{\text{differential} \\ \text{(derivative)}}}{\frac{dy}{dx}} \overset{\text{equation}}{=} 5x$$

Example: an equation with the function **y** and its derivative $\frac{dy}{dx}$

Solving

We **solve** it when we discover **the function y** (or set of functions y).

There are many "tricks" to solving Differential Equations (if they can be solved!), but first: why?

Why Are Differential Equations Useful?

In our world things change, and **describing how they change** often ends up as a Differential Equation:

Example: Rabbits!

The more rabbits we have the more baby rabbits we get. Then those rabbits grow up and have babies too! The population will grow faster and faster.

The important parts of this are:

- the population **N** at any time **t**
- the growth rate **r**
- the population's rate of change $\frac{d}{dt}$**N**

Let us imagine some actual values:

- the population **N** is **1000**

- the growth rate **r** is **0.01** new rabbits per week **for every current rabbit**

The population's rate of change $\frac{d}{dt}$**N** is then 1000×0.01 = **10 new rabbits** per week.

But that is only true at a **specific time**, and doesn't include that the population is constantly increasing.

Remember: the bigger the population, the more new rabbits we get!

So it is better to say the rate of change (at any instant) is the growth rate times the population at that instant:

$$\frac{dN}{dt} = rN$$

And it is a **Differential Equation**, because it has a function **N(t)** and its derivative.

And how powerful mathematics is! That short equation says "the rate of change of the population over time equals the growth rate times the population".

Differential Equations can describe how populations change, how heat moves, how springs vibrate, how radioactive material decays and much more. They are a very natural way to describe many things in the universe.

What To Do With Them?

On its own, a Differential Equation is a wonderful way to express something, but is hard to use.

So we try to **solve** them by turning the Differential Equation into a simpler Algebra-style equation (without the differential bits) so we can do calculations, make graphs, predict the future, and so on.

Example: Compound Interest

Money earns interest. The interest can be calculated at fixed times, such as yearly, monthly, etc. and added to the original amount.

This is called compound interest.

But when it is compounded continuously then at any time the interest gets added in proportion to the current value of the loan (or investment).

And the bigger the loan the more interest it earns.

Using **t** for time, **r** for the interest rate and **V** for the current value of the loan:

$$\frac{dV}{dt} = rV$$

And here is a cool thing: it is the same as the equation we got with the Rabbits! It just has different letters. So mathematics shows us these two things behave the same.

Solving

The Differential Equation says it well, but is hard to use.

But don't worry, it can be solved (using a special method called Separation of Variables) and results in:

$$V = Pe^{rt}$$

Where **P** is the Principal (the original loan).

So a continuously compounded loan of $1,000 for 2 years at an interest rate of 10% becomes:

$$V = 1000 \times e^{(2\times0.1)}$$

$$V = 1000 \times 1.22140...$$
$$= \$1{,}221.40 \text{ (to nearest cent)}$$

So Differential Equations are great at describing things, but need to be solved to be useful.

More Examples of Differential Equations

The Verhulst Equation

Example: Rabbits Again!

Remember our growth Differential Equation:

$$\frac{dN}{dt} = rN$$

Well, that growth can't go on forever as they will soon run out of available food.

So let's improve it by including:

- the maximum population that the food can support **k**

A guy called Verhulst figured it all out and got this Differential Equation:

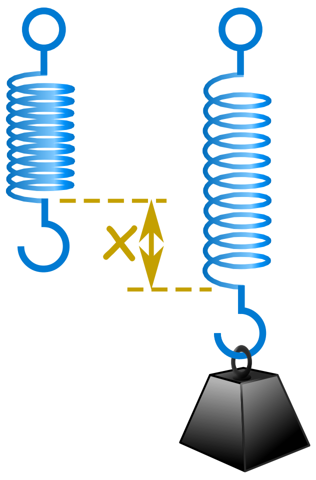
$$\frac{dN}{dt} = rN(1-N/k)$$

The Verhulst Equation

Simple harmonic motion

In Physics, Simple Harmonic Motion is a type of periodic motion where the restoring force is directly proportional to the displacement. An example of this is given by a mass on a spring.

Example: Spring and Weight



A spring gets a weight attached to it:

- the weight is pulled down by gravity,
- the tension in the spring increases as it stretches,
- then the spring bounces back up,
- then back down, up and down, again and again.

Describe this with mathematics!

The weight is pulled down by gravity, and we know from Newton's Second Law that force equals mass times acceleration:

$$\mathbf{F} = m\mathbf{a}$$

And acceleration is the second derivative of position with respect to time, so:

$$\mathbf{F} = m\,\frac{d^2x}{dt^2}$$

The spring pulls it back up based on how stretched it is (**k** is the spring's stiffness, and **x** is how stretched it is): **F = -kx**

The two forces are always equal:

$$m\,\frac{d^2x}{dt^2} = -kx$$

We have a differential equation!

It has a function **x(t)**, and it's second derivative $\mathbf{\frac{d^2x}{dt^2}}$

Note: we haven't included "damping" (the slowing down of the bounces due to friction), that is just a little more complicated.

OK, now we want to **solve** it to find how the spring bounces up and down over time.

Classify Before Trying To Solve

OK, we want to **solve** them, but how?

Over the years wise people have worked out **special methods** to solve **some types** of Differential Equations.

So we need to know **what type** of Differential Equation it is first.

It is like travel: different kinds of transport have solved how to get to certain places. Is it near, so we can just walk? Is there a road so we can take a car? Is it over water so we need a ship? Or is it in another galaxy and we just can't get there yet?

So let us **classify the Differential Equation**.

Ordinary or Partial

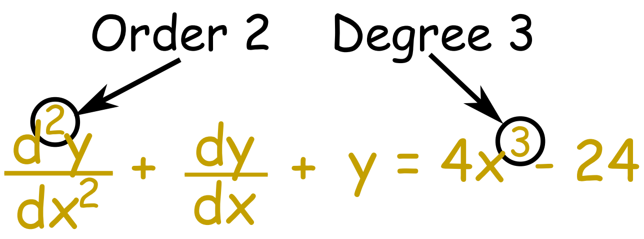
The first major grouping is:

- "Ordinary Differential Equations" (ODEs) have **a single independent variable** (like **y**)
- "Partial Differential Equations" (PDEs) have two or more independent variables.

We are learning about **Ordinary Differential Equations** here!

Order and Degree

Next we work out the Order and the Degree:



Order

The Order is the **highest derivative** (is it a first derivative? a second derivative? etc):

Example:

$$\frac{dy}{dx} + y^2 = 5x$$

It has only the first derivative $\frac{dy}{dx}$, so is "First Order"

Example:

$$\frac{d^2y}{dx^2} + xy = \sin(x)$$

This has a second derivative $\frac{d^2y}{dx^2}$, so is "Order 2"

Example:

$$\frac{d^3y}{dx^3} + x\frac{dy}{dx} + y = e^x$$

This has a third derivative $\frac{d^3y}{dx^3}$ which outranks the $\frac{dy}{dx}$, so is "Order 3"

Degree

The degree is the exponent of the highest derivative.

Example:

$$\left(\frac{dy}{dx}\right)^2 + y = 5x^2$$

The highest derivative is just dy/dx, and it has an exponent of 2, so this is "Second Degree"

In fact it is a **First Order Second Degree Ordinary Differential Equation**

Example:

$$\frac{d^3y}{} + \left(\frac{dy}{dx}\right)^2 + y = 5x^2$$

dx^3

The highest derivative is d^3y/dx^3, but it has no exponent (well actually an exponent of 1 which is not shown), so this is "First Degree".

(The exponent of 2 on dy/dx does not count, as it is not the highest derivative).

So it is a **Third Order First Degree Ordinary Differential Equation**

Be careful not to confuse order with degree. Some people use the word order when they mean degree!

Linear

It is **Linear** when the variable (and its derivatives) has no exponent or other function put on it.

So **no** y^2, y^3, $\sqrt{y}$, sin(y), ln(y) etc, **just plain y** (or whatever the variable is).

More formally a **Linear Differential Equation** is in the form:

$$\frac{dy}{dx} + P(x)y = Q(x)$$

Solving

OK, we have classified our Differential Equation, the next step is solving.

This is not a complete list of how to solve differential equations, but it should get you started:

- Separation of Variables
- Solving First Order Linear Differential Equations
- Homogeneous Differential Equations

Distinguishable Permutations

Example

Suppose we toss a gold dollar coin 8 times. What is the probability that the sequence of 8 tosses yields 3 heads (H) and 5 tails (T)?

Solution. Two such sequences, for example, might look like this:

H H H T T T T T or this **H T H T H T T T**

Assuming the coin is fair, and thus that the outcomes of tossing either a head or tail are equally likely, we can use the classical approach to assigning the probability. The Multiplication Principle tells us that there are:

$$2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2$$

or 256 possible outcomes in the sample space of 8 tosses. (Can you imagine enumerating all 256 possible outcomes?) Now, when counting the number of sequences of 3 heads and 5 tosses, we need to recognize that we are dealing with arrangements or permutations of the letters, since order matters, but in this case not all of the objects are distinct. We can think of *choosing* (note that choice of word!) $r = 3$ positions for the heads (H) out of the $n = 8$ possible tosses. That would, of course, leave then $n - r = 8 - 3 = 5$ positions for the tails (T). Using the formula for a combination of n objects taken r at a time, there are therefore:

$$\binom{8}{3} = \frac{8!}{3!5!} = 56$$

distinguishable permutations of 3 heads (H) and 5 tails (T). The probability of tossing 3 heads (H) and 5 tails (T) is thus 56/256 = 0.22.

Let's formalize our work here!

Definition. Given n objects with:

- r of one type, and
- $n - r$ of another type

there are:

$$_nC_r = \binom{n}{r} = \frac{n!}{r!(n-r)!}$$

distinguishable permutations of the n objects.

Let's take a look at another example that involves counting distinguishable permutations of objects of two types.

Example

Suppose Dr. DoesBadThings conducts research which involves infecting 20 people with the swine flu virus. He is interested in studying how many actually end up ill (I) and how many remain healthy (H).

How many arrangements are there of the 20 people that involve 0 people ill?

Solution. In this case, we can readily determine that there is just 1 way. The sequence would look like this:

HHHHHHHHHHHHHHHHHHHH

Alternatively, we could use the formula for counting distinguishable permutations. The formula yields:

$$\binom{20}{0} = \frac{20!}{0!20!} = 1$$

How many arrangements are there of the 20 people that involve 1 person ill?

Solution. I think we can readily convince ourselves that there are 20 ways. The first possible sequence might look like this:

I H H H H H H H H H H H H H H H H H H H

The second possible sequence might look like this:

H I H H H H H H H H H H H H H H H H H H

And, the last possible sequence might look like this:

H H H H H H H H H H H H H H H H H H H I

That must mean that there are 20 possible positions for the one I. Alternatively, we could use the formula for counting distinguishable permutations. The formula yields:

$$\binom{20}{1} = \frac{20!}{1!19!} = 20$$

How may arrangements are there of the 20 people that involve 2 people ill?

Solution. I'm going to leave it to you to decide if you want to try to count this one out by hand! I can tell you that the first possible sequence might look like this:

I I H H H H H H H H H H H H H H H H H H

The formula for counting distinguishable permutations yields:

$$\binom{20}{2} = \frac{20!}{2!18!} = 190$$

possible arrangements.

Example

How many ordered arrangements are there of the letters in MISSISSIPPI?

Solution. Well, there are 11 letters in total:

1 M, 4 I, 4 S and 2 P

We are again dealing with arranging objects that are not all distinguishable. We couldn't distinguish among the 4 I's in any one arrangement, for example. In this case, however, we don't have just two, but rather four, different types of objects. In trying to solve this problem, let's see if we can come up with some kind of a general formula for the number of distinguishable permutations of *n* objects when there are more than two different types of objects.

Let's formalize our work.

Definition. The **number of distinguishable permutations** of n objects, of which:

- n_1 are of one type
- n_2 are of a second type
- ... and ...
- n_k are of the last type

and $n = n_1 + n_2 + ... + n_k$ is given by:

$$\binom{n}{n_1 n_2 n_3 \ldots n_k} = \frac{n!}{n_1!n_2!n_3! \ldots n_k!}$$

Let's take a look at a few more examples involving distinguishable permutations of objects of more than two types.

Example

How many ordered arrangements are there of the letters in the word PHILIPPINES?

Solution. The number of ordered arrangements of the letters in the word PHILIPPINES is:

$$\frac{11!}{3!1!3!1!1!1!1!} = 1,108,800$$

Example

Fifteen (15) pigs are available to use in a study to compare three (3) different diets. Each of the diets (let's say, A, B, C) is to be used on five randomly selected pigs. In how many ways can the diets be assigned to the pigs?

Solution. Well, one possible assignment of the diets to the pigs would be for the first five pigs to be placed on diet A, the second five pigs to be placed on diet B, and the last 5 pigs to be placed on diet C. That is:

A A A A A B B B B B C C C C C

Another possible assignment might look like this:

A B C A B C A B C A B C A B C

Upon studying these possible assignments, we see that we need to count the number of distinguishable permutations of 15 objects of which 5 are of type A, 5 are of type B, and 5 are of type C. Using the formula, we see that there are:

$$\frac{15!}{5!5!5!} = 756756$$

ways in which 15 pigs can be assigned to the 3 diets. That's a lot of ways!

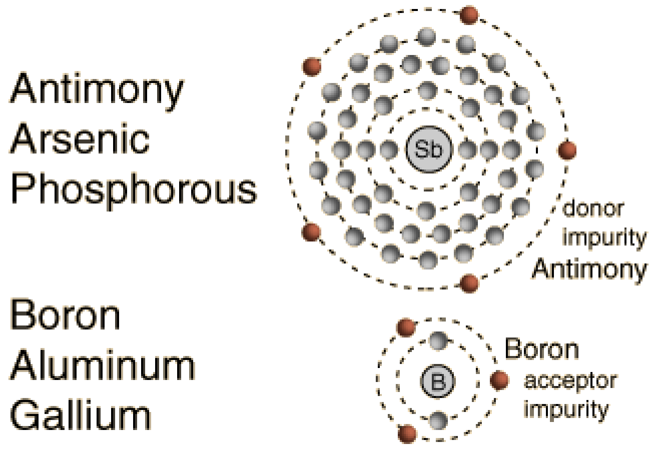
Source URL: https://onlinecourses.science.psu.edu/stat414/node/31

The Doping of Semiconductors

The addition of a small percentage of foreign atoms in the regular crystal lattice of silicon or germanium produces dramatic changes in their electrical properties, producing n-type and p-type semiconductors.

Pentavalent impurities

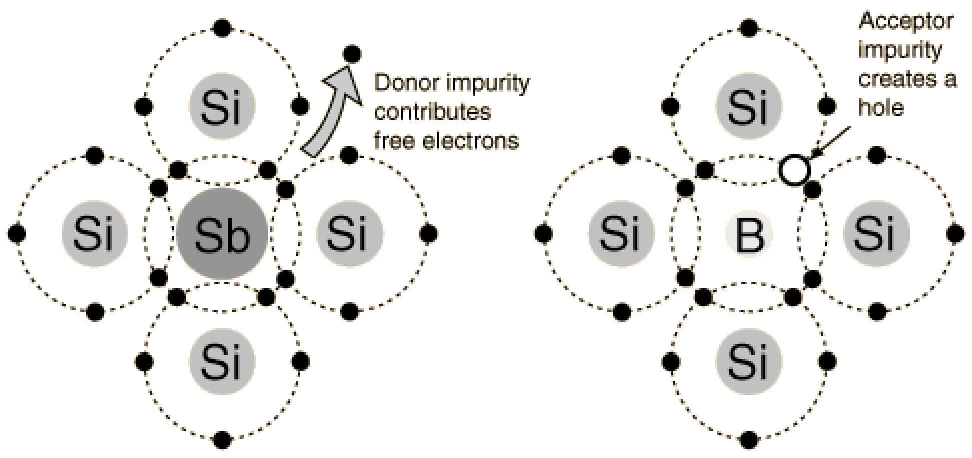
Impurity atoms with 5 valence electrons produce n-type semiconductors by contributing extra electrons.



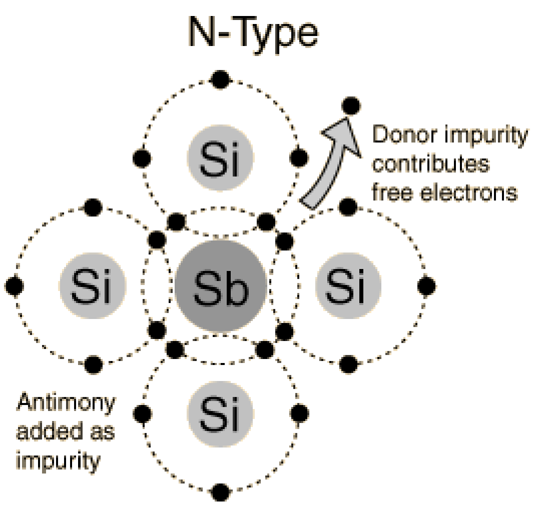
Trivalent impurities

Impurity atoms with 3 valence electrons produce p-type semiconductors by producing a "hole" or electron deficiency.

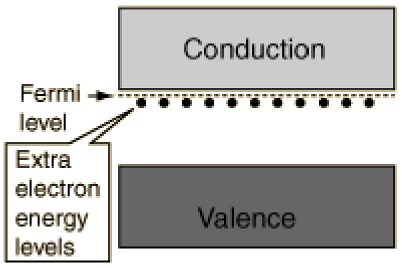
P- and N- Type Semiconductors



N-Type Semiconductor

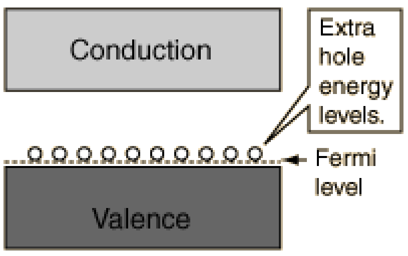

N-Type
Si
Sb
Donor impurity contributes free electrons
Antimony added as impurity

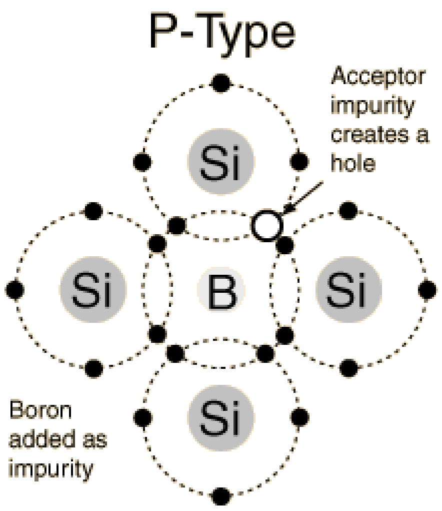
The addition of pentavalent impurities such as antimony, arsenic or phosphorous contributes free electrons, greatly increasing the conductivity of the intrinsic semiconductor. Phosphorous may be added by diffusion of phosphine gas (PH3).



P-Type Semiconductor

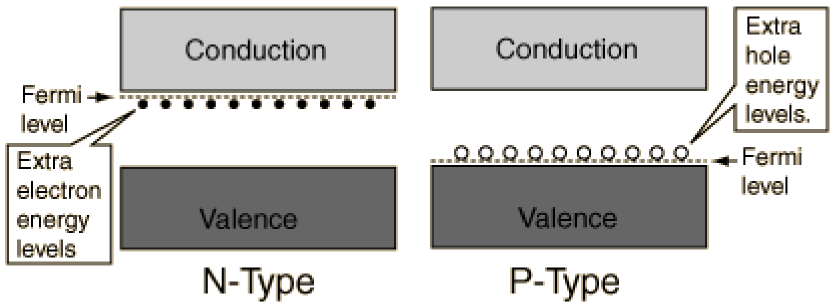
The addition of trivalent impurities such as boron, aluminum or gallium to an intrinsic semiconductor creates deficiencies of valence electrons,called "holes". It is typical to use B_2H_6 diborane gas to diffuse boron into the silicon material.





Bands for Doped Semiconductors

The application of band theory to n-type and p-type semiconductors shows that extra levels have been added by the impurities. In n-type material there are electron energy levels near the top of the band gap so that they can be easily excited into the conduction band. In p-type material, extra holes in the band gap allow excitation of valence band electrons, leaving mobile holes in the valence band.



Semiconductor Diodes

electronicshub.org/semiconductor-diodes/

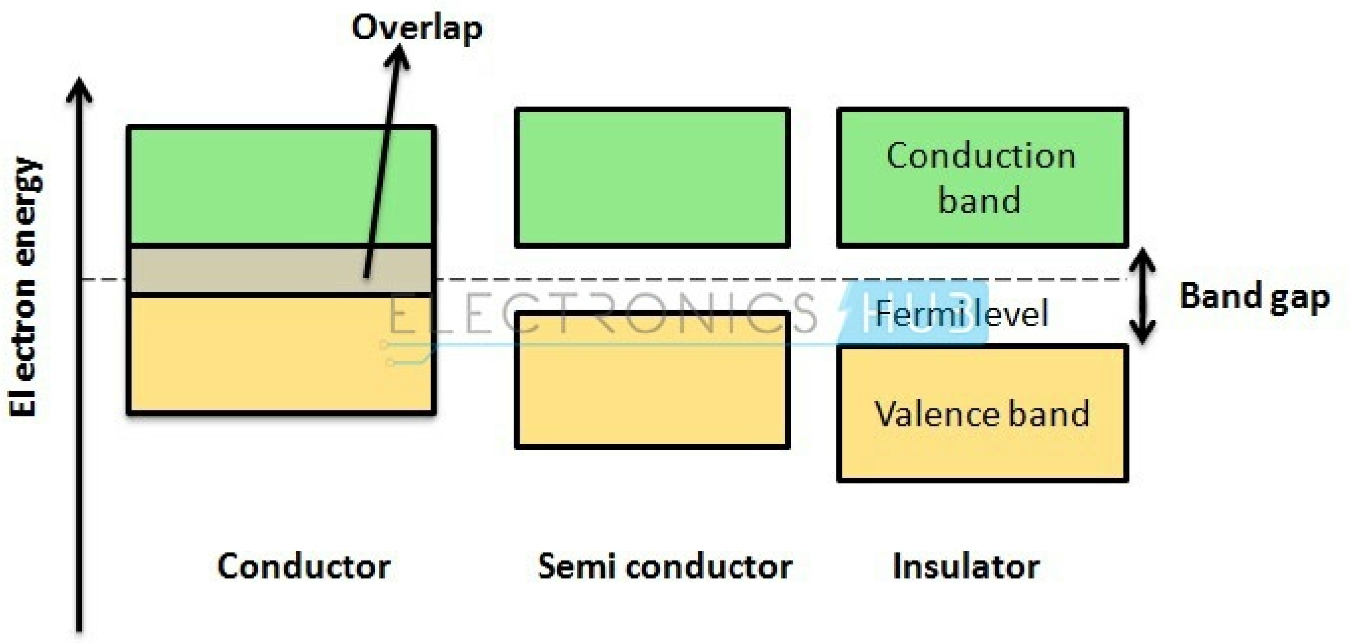
Contents [hide]

Introduction

There are two types of semiconductor components in electronic and electrical circuits. They are active and passive components. Diodes are the foremost active components and resistors are the foremost passive components in electronic design circuits. Diodes are essentially unidirectional devices having exponential relationship for the current-voltage characteristics are made from semiconductor materials.

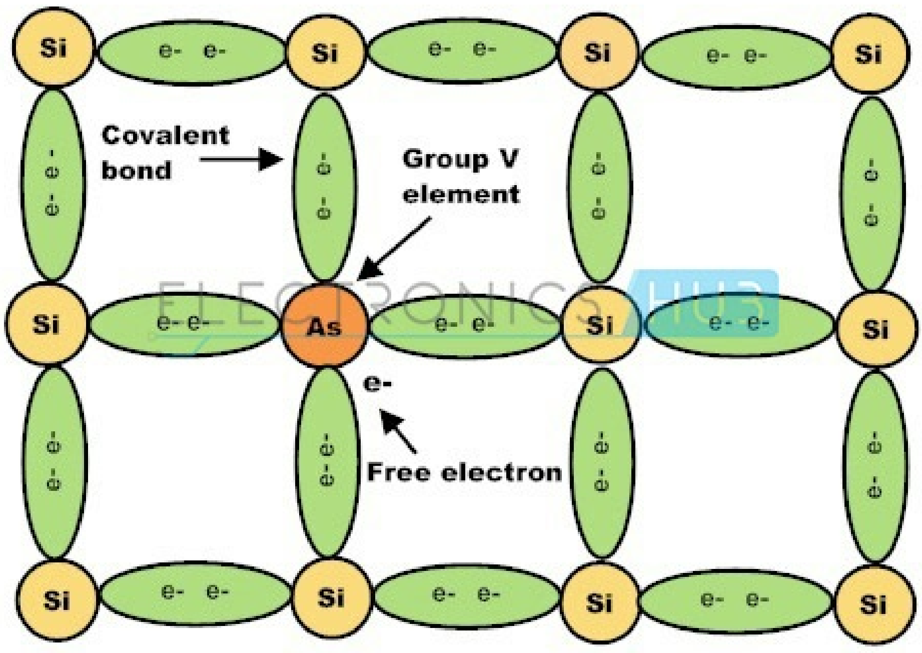
The three necessary materials that are utilized in electronics are insulators, semiconductors and conductors. These materials are classified in terms of electrical phenomenon. Electrical resistivity conjointly known as electrical resistance is a measure of how efficiently a material refuses the electrical current to flow through it. The quality unit of the electrical resistivity is the ohm meter [Ω m]. A material with low electrical resistivity indicates the effective movement of electrical charge throughout the semiconductor.

Semiconductors are the materials whose resistivity values are in between insulators and conductors. These materials are neither smart insulators nor smart conductors. They have only a few free electrons because their atoms are tightly bonded in an exceedingly crystalline form are referred to as a “crystal lattice”. Samples of semiconductors are silicon and germanium.



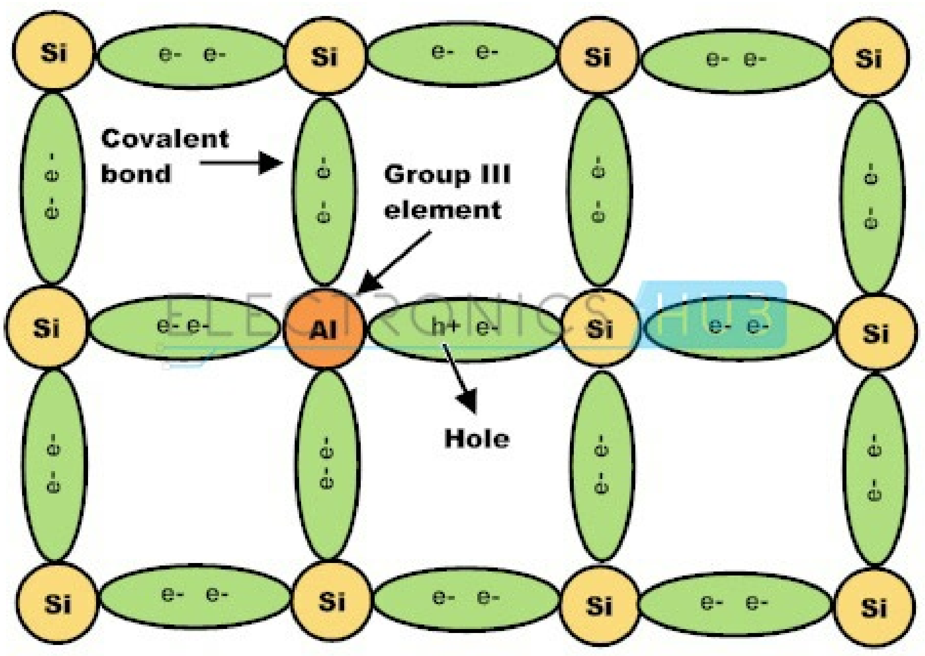
Semiconductors have high importance in the manufacture of electronic circuits and integrated devices. The conductivity of semiconductors can be altered easily by varying the temperature and concentration of doping in the fabrication process. The capability to conduct electricity in semiconductor materials is considerably increased by a adding definite quantity of impurities to the crystalline lattice producing additional free electrons than holes.

The properties of semiconductor materials change considerably by adding small amounts of impurities to it. The process of shifting the balance between electrons and holes by incorporating impurity atoms in the silicon crystal lattice is called as doping. These impurity atoms are known as dopants. Based on the type of doping material incorporated, semiconductor crystals are classified into two types particularly n-type semiconductors and p-type semiconductors.



Group –V elements such as phosphorus, antimony and arsenic are usually classified as N-type impurities. These elements have five valence electrons. When N-type impurities are doped into silicon crystal, four of the five valence electrons form four strong covalent bonds with adjacent crystal atoms leaving one free electron. Likewise, every N-type impurity atom produces a free electron in the conduction band which will drift to conduct electric current if a potential is applied to the material. N-type semiconductors can also be referred as Donors.

Group–III elements such as boron, aluminium, gallium and indium are usually classified as P-type impurities. These elements have three valence electrons. When P-type impurities are doped into silicon crystal, all the three valence electrons form three strong covalent bonds with adjacent crystal atoms. There is a deficit of electrons to form the fourth covalent bond and this deficiency is termed as holes. Likewise, every P-type impurity atom produces a hole in the valence band which will drift to conduct electric current if a potential is applied to the material. P-type semiconductors can also be referred as Acceptors.

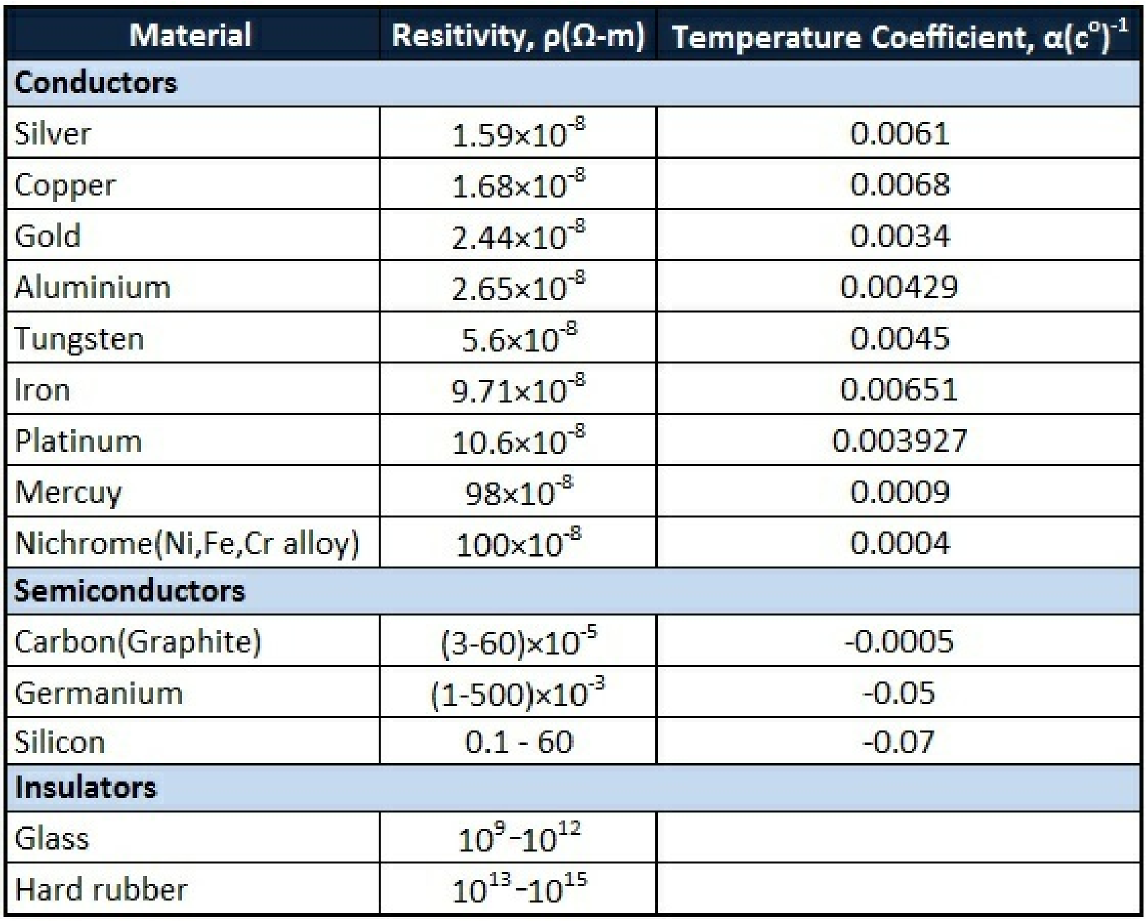


BACK TO TOP

Resistivity:

A characteristic property of every material that is helpful in comparing different materials on the basis of their ability to conduct electric current is known as electrical resistivity. Resistivity can be approximated by multiplying the resistance R of an electrical wire and the cross sectional area A, divided by the length of the wire L. Conductivity that is reciprocal of the electrical resistivity conjointly characterizes the materials how well they permit the electrical current to pass through them. Sensible conductors have lowest electrical resistivity and high conductivity. Electrical resistivity depends strongly on the presence of impurity atoms within the material and on the temperature of the material, i.e., at room temperature (20ºC).

For various conductors, semiconductors and insulators, the resistivity values vary linearly with variations in temperature. The change in electrical resistance per degree Celsius of temperature change is called the temperature coefficient of resistance. This factor is represented by the letter “alpha” (α). A positive temperature coefficient for a material means that its resistance increases with an increase in temperature. Pure conductors will typically have a positive temperature coefficient of resistance. A negative coefficient for a material means that its resistance decreases with an increase in temperature. Semiconductor materials (carbon, silicon, and germanium) typically have a negative temperature coefficient of resistance. Different materials with their resistivity values and temperature coefficients are given in the table below.



Material	Resitivity, $\rho(\Omega\text{-m})$	Temperature Coefficient, $\alpha(c^{o})^{-1}$
Conductors		
Silver	1.59×10^{-8}	0.0061
Copper	1.68×10^{-8}	0.0068
Gold	2.44×10^{-8}	0.0034
Aluminium	2.65×10^{-8}	0.00429
Tungsten	5.6×10^{-8}	0.0045
Iron	9.71×10^{-8}	0.00651
Platinum	10.6×10^{-8}	0.003927
Mercuy	98×10^{-8}	0.0009
Nichrome(Ni,Fe,Cr alloy)	100×10^{-8}	0.0004
Semiconductors		
Carbon(Graphite)	$(3\text{-}60)\times10^{-5}$	-0.0005
Germanium	$(1\text{-}500)\times10^{-3}$	-0.05
Silicon	0.1 - 60	-0.07
Insulators		
Glass	$10^{9}\text{-}10^{12}$	
Hard rubber	$10^{13}\text{-}10^{15}$	

BACK TO TOP

Conductors:

Conductors are built with low resistive materials having resistivity values in the order of micro-ohms per meter (µΩ/m). Metals with terribly low electrical resistivity of the order of 1 x ohm meters are called as conductors. These metals have a large number of free electrons. These free electrons leave the valence layer of their parental atom and form a drift of electrons known as an electric current. Therefore, metals are superb conductors of electricity.

Metals like copper, aluminium, gold and silver and other non metals such as carbon are ancient conducting materials. Most of the metallic conductors are good conductors of electricity, having smaller resistance values and high conduction values. Throughout the process of conduction, heat flows throughout the body. During conduction this heat flow may be considered as a loss of energy and the loss increases with increase in temperature after it reaches the room temperature i.e., 25°C.

BACK TO TOP

Insulators:

In distinction to conductors, insulators are made up of non-metals having resistivity values in the order of 1 x ohm meters. Non-metals have only a few or no free electrons flowing through it or within the parental atomic structure as the outermost electrons are tightly bonded in covalent bonds between a pair of atoms. Since the electrons are negatively charged, the free electrons within the valence layer are easily attracted by the positively charged particles within the nucleus. Since there are no free electrons, when a positive potential is applied, there will be no electrical current to flow through the material giving insulating properties. Therefore insulators (non-metals) are very poor conductors of electricity.

Non-metals like glass, plastic, rubber, wood, sand, quartz and Teflon are sensible examples of insulators. Glass insulators are used for prime voltage power transmission. Insulators are used as protectors of warmth, sound and electricity.

BACK TO TOP

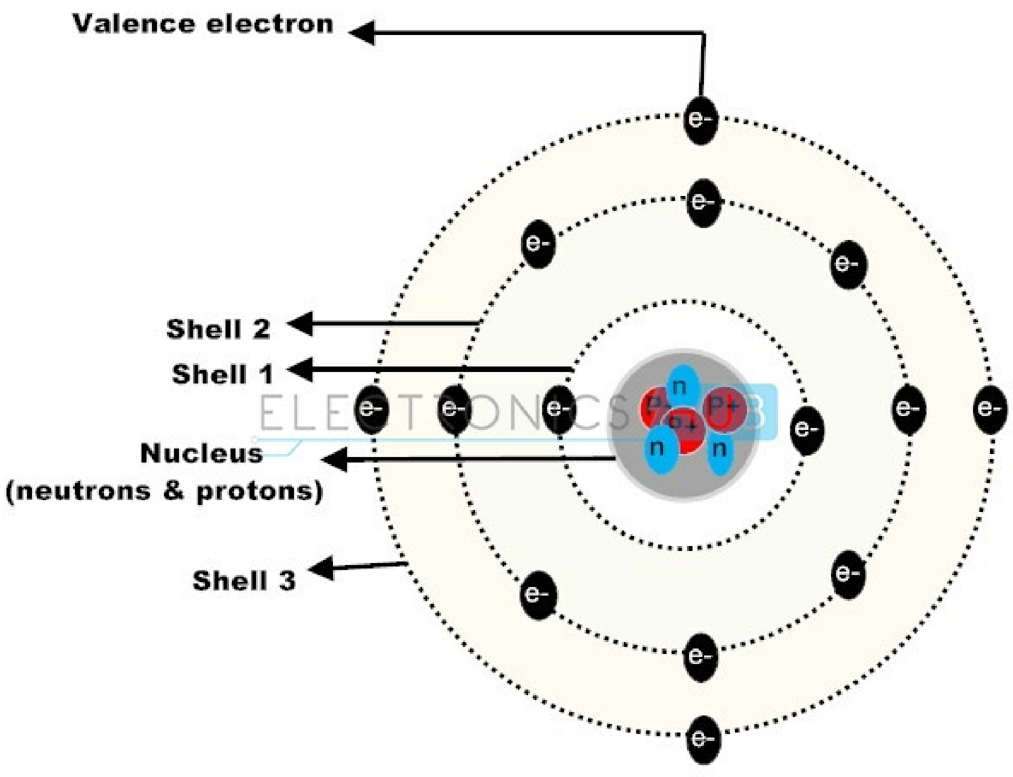
Semiconductors:

Semiconductors have the electrical properties in between insulators and conductors. Smart examples of perfect semiconductors are silicon (Si), germanium (Ge) and gallium arsenide (GaAs). These elements have only a few electrons within the parental atomic structure that form a crystal lattice. Silicon, the foremost basic semiconductor material contains four valence electrons within the outer shell forming four strong covalent bonds with four adjacent silicon atoms, such that each atom shares an electron with the neighbouring atom creating a strong covalent bond. The silicon atoms are organized in a lattice form, creating them a crystalline structure.

Conducting electric current is feasible with silicon semiconductor crystal by supplying external potential to the semiconductor and incorporating the impurity dopants into the semiconductor crystal thereby creating positive and negative charged holes.

BACK TO TOP

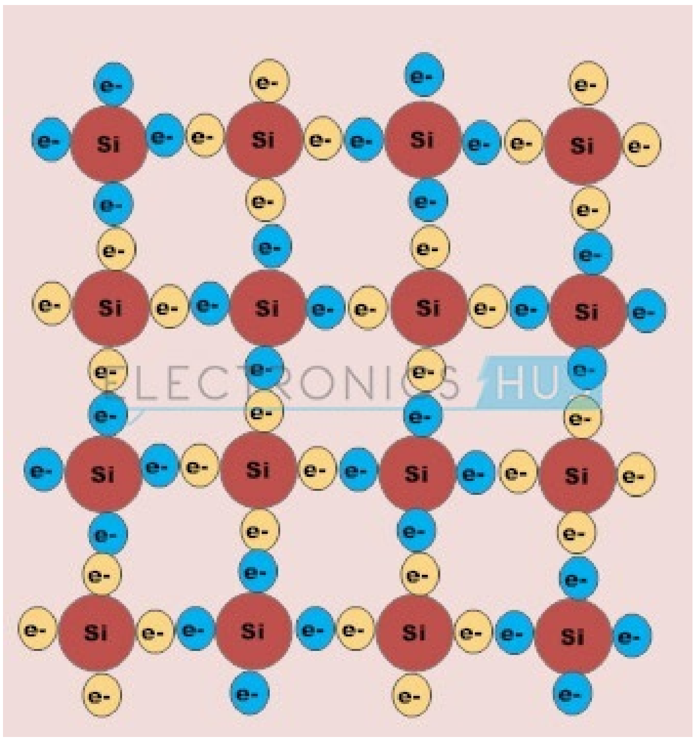
Pure Silicon Atom Structure:



The silicon atom has 14 electrons; however the orbital arrangement has solely 4 valence electrons to be shared by alternative atoms. These valence electrons play a crucial role in photo voltaic effect. Large number of silicon atoms bond together to make a crystalline structure. In this structure, each silicon atom shares one of its four valence electrons with their neighbouring silicon atoms. The solid silicon crystal composed of a regular series of units of five silicon atoms. This regular and fixed arrangement of silicon atoms are unit is referred to as a crystal lattice.

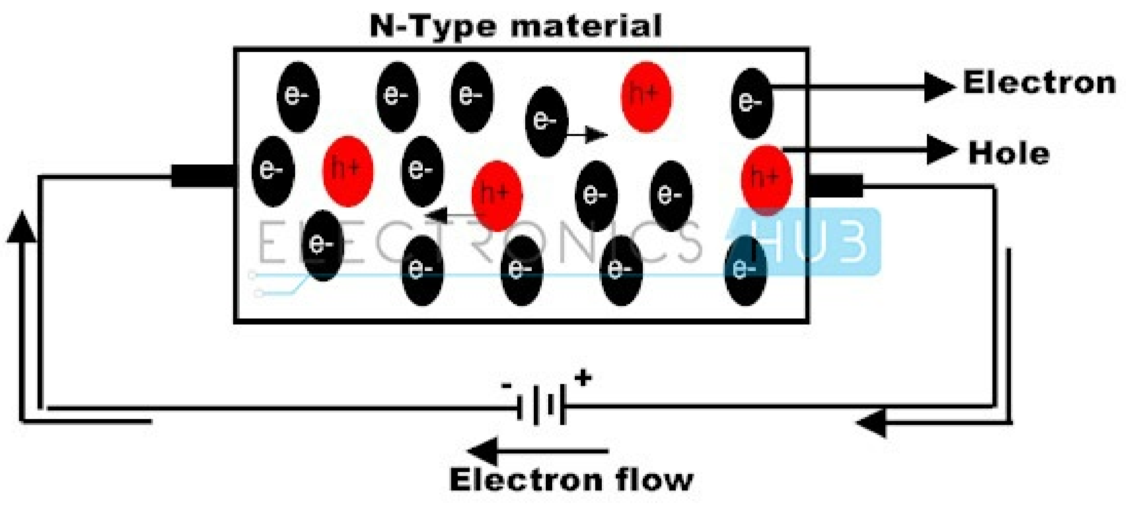
BACK TO TOP

N-Type Semiconductor:



Impurities like phosphorous, arsenic and antimony are added to the silicon crystalline structure, to transform intrinsic semiconductor into extrinsic semiconductor. These impurity atoms are known as pentavalent impurities as a result of the five valence electrons in the outermost shell to share the free electrons with the neighbouring atoms.

Pentavalent impurity atoms are also known as donors because the five valence electrons in the impurity atom bond with the four valence electrons of silicon forming four covalent bonds, leaving one free electron. Each impurity atom produces a free electron within the conduction band. Once a positive potential is applied to the N-type semiconductor, the remaining free electrons form a drift to produce an electrical current.



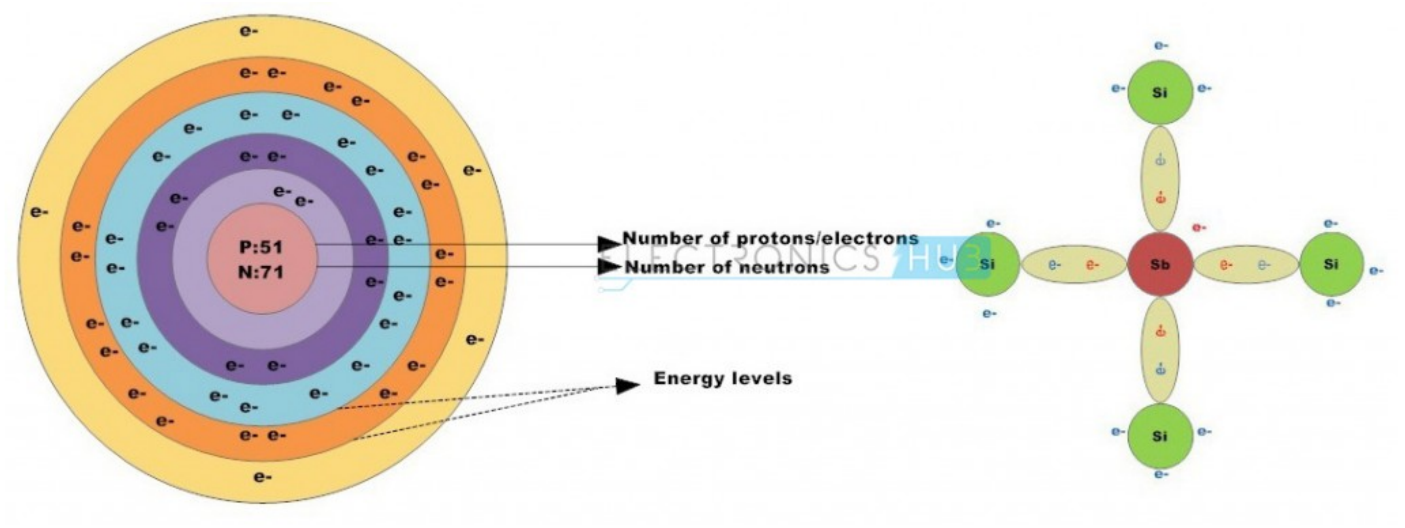
An N-type semiconductor is a better conductor than the intrinsic semiconductor material. The majority charge carriers in N-type semiconductors are electrons and minority charge carriers are holes. The N-type semiconductors are not negatively charged, because the negative charge of donor impurity atoms is balanced by the positive charge within the nucleus.

The major contribution to the electric current flow is negatively charged electrons though there is some amount of contribution by the positively charged holes due to electron-hole pair.

BACK TO TOP

N-type Semiconductor Doping:

If group 5 element, such as Antimony impurity is added to the silicon crystal, the Antimony atom builds four covalent bonds with four silicon atoms by bonding the valence electrons of antimony with the valence electrons within the silicon outermost shell, leaving one free electron. Therefore the impurity atom has donated a free electron to the structure so these impurities are referred to as donor atoms.

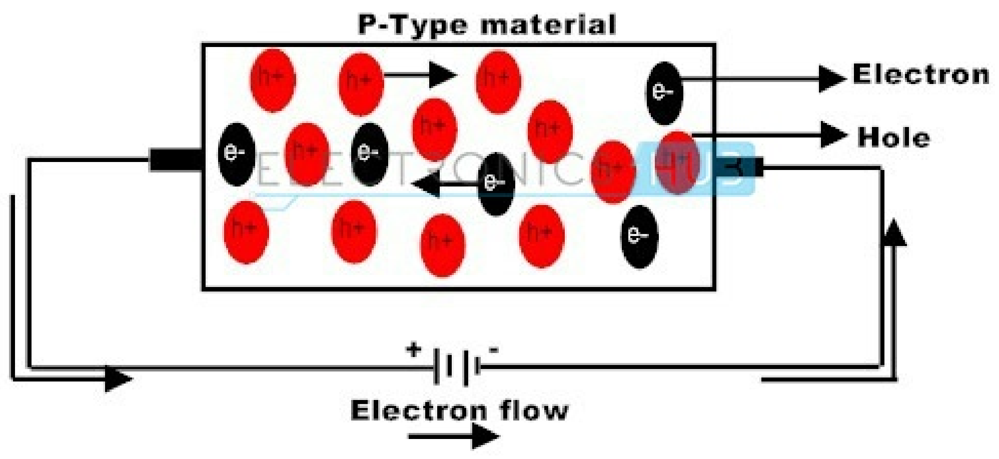


BACK TO TOP

P-Type Semiconductor:

The group 3 elements such as boron, aluminium and indium are supplementary to the silicon crystalline structure having solely three electrons within the outermost shell, form three closed covalent bonds, leaving the hole in the covalent bond structure and therefore a hole in the valence band of the energy level diagram. This action leaves an abundant number of positively charged carriers referred to as holes in the crystalline structure when there is electron deficiency. These group 3 elements are called as trivalent impurity atoms.

The presence of abundant holes attracts the neighboring electrons to sit in it. As long as the electron fills the holes in the silicon crystal there will be new holes behind the electron as it goes far from it. The newly created holes successfully attract the electrons, creating other new holes leads to the movement of holes, creating a standard electric current flow in the semiconductor.

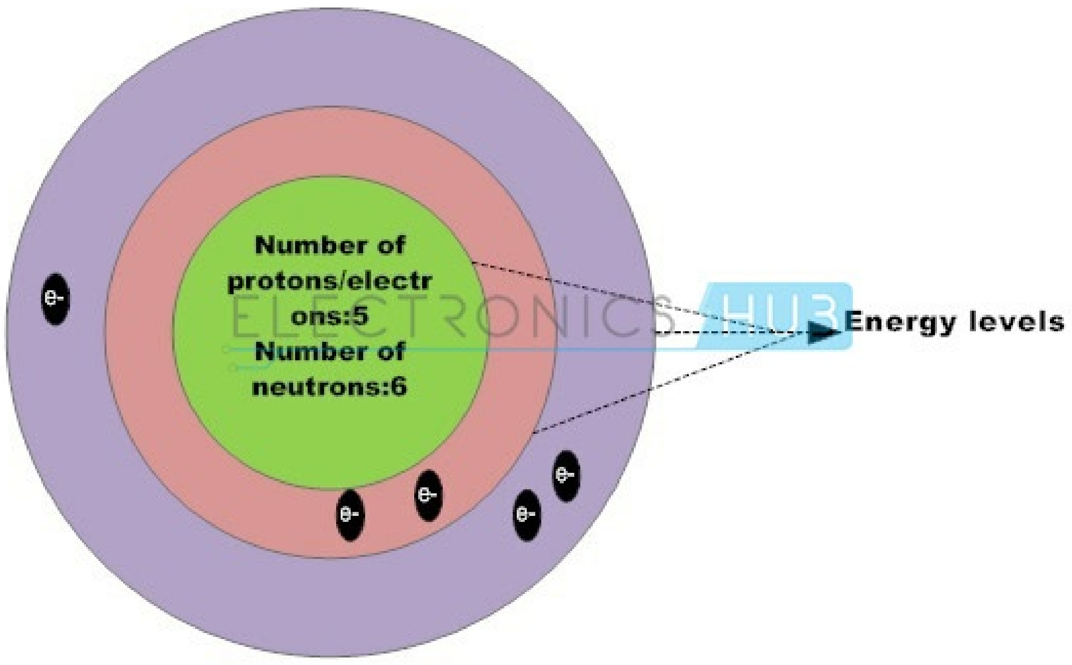


The movement of holes in the silicon crystal seems the silicon crystal as a positive pole. As long as the impurity atoms invariably generate holes, group 3 elements are referred to as acceptors as a result of the impurity atoms are continually accepting the free electrons.

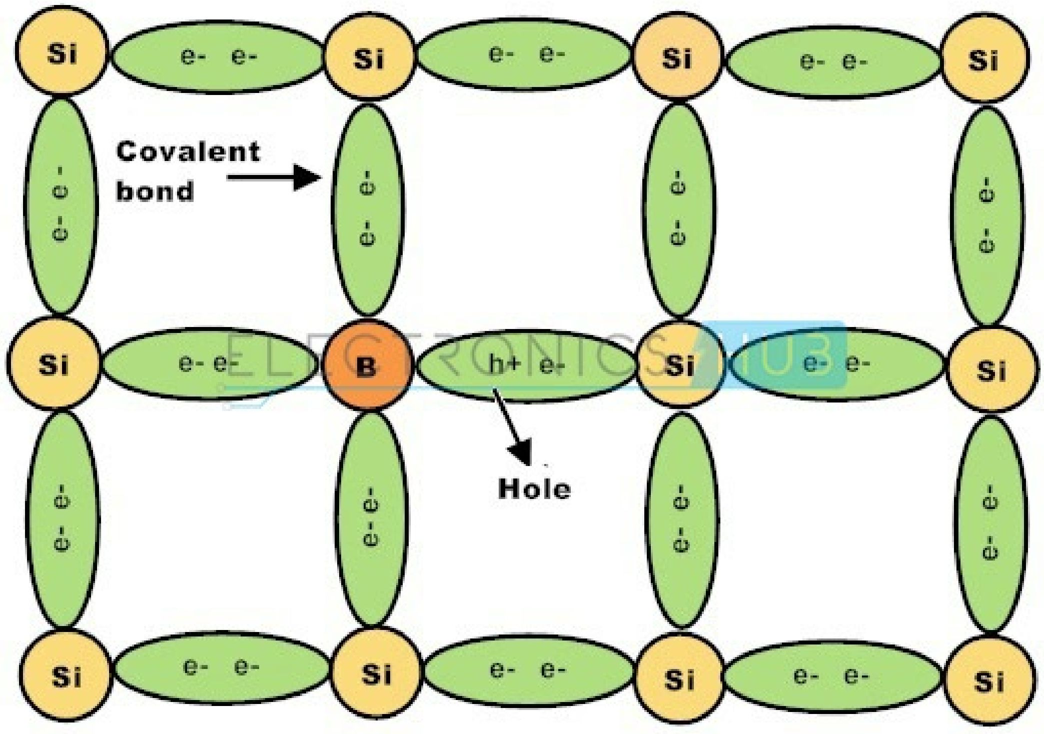
The doping of group 3 elements in silicon crystal leads to P-type semiconductor. In this P-type semiconductor holes are the majority charge carriers and electrons are the minority charge carriers.

BACK TO TOP

P-Type Semiconductor Doping:



If group 3 elements such as such as boron, gallium and indium are added to the semiconductor crystal, the impurity atoms having three valence electrons form three strong covalent bonds with the silicon crystal valence electrons leaving one vacancy. This vacancy is called as a hole and it is diagrammatically represented by a small circle or positive sign due to the absence of a negative charge.



BACK TO TOP

Semiconductor Basics Summary:

N-type materials are type of materials formed by adding group 5 elements (pentavalent impurity atoms) to the semiconductor crystals and conduct the electric current by movement of electrons.

In N-type Semiconductors:

- The impurity atoms are pentavalent elements.
- Impurity elements with solid crystal give a large number of free electrons.
- Pentavalent impurities are also called as donors.
- Doping gives the less number of holes in relation to the number of free electrons.
- Doping with group 5 elements results in positively charged donors and negatively charged free electrons.

P-type materials are a type of materials formed when group 3 elements (trivalent impurity atoms) are added to the solid crystal. In these semiconductors the current flow is mainly due to the holes.

In P-type Semiconductors:

1. The impurity atoms are trivalent elements.
2. Trivalent elements results in excess number of holes which always accepts electrons. Hence trivalent impurities are called as acceptors.
3. Doping gives the less number of free electrons in relation to the number of holes.
4. Doping results in negatively charged acceptors and positively charged holes

Both p-type and N-type are electrically neural on their own because the contribution of electrons and holes required for conducting electrical current are equal due to electron-hole pair. Both boron (B) and antimony (Sb) are called metalloids because they are the most commonly used doping agents for the intrinsic semiconductor to improve the properties of conductivity.

Aviation

Satellite communications provide ultimate connectivity for a consistent global experience for the whole aircraft, from safety communications to high-speed broadband and live TV in the cabin. Aviation safety is enabled through data communications with control centres on the ground using the C-band. Communications with the aircraft themselves are enabled using the L-band. They enable a wide range of uses in the cockpit and the cabin. These include safety communications, weather and flight-plan updates, as well as passenger connectivity for email, Internet access, VoIP telephone calls, GSM and SMS messaging.

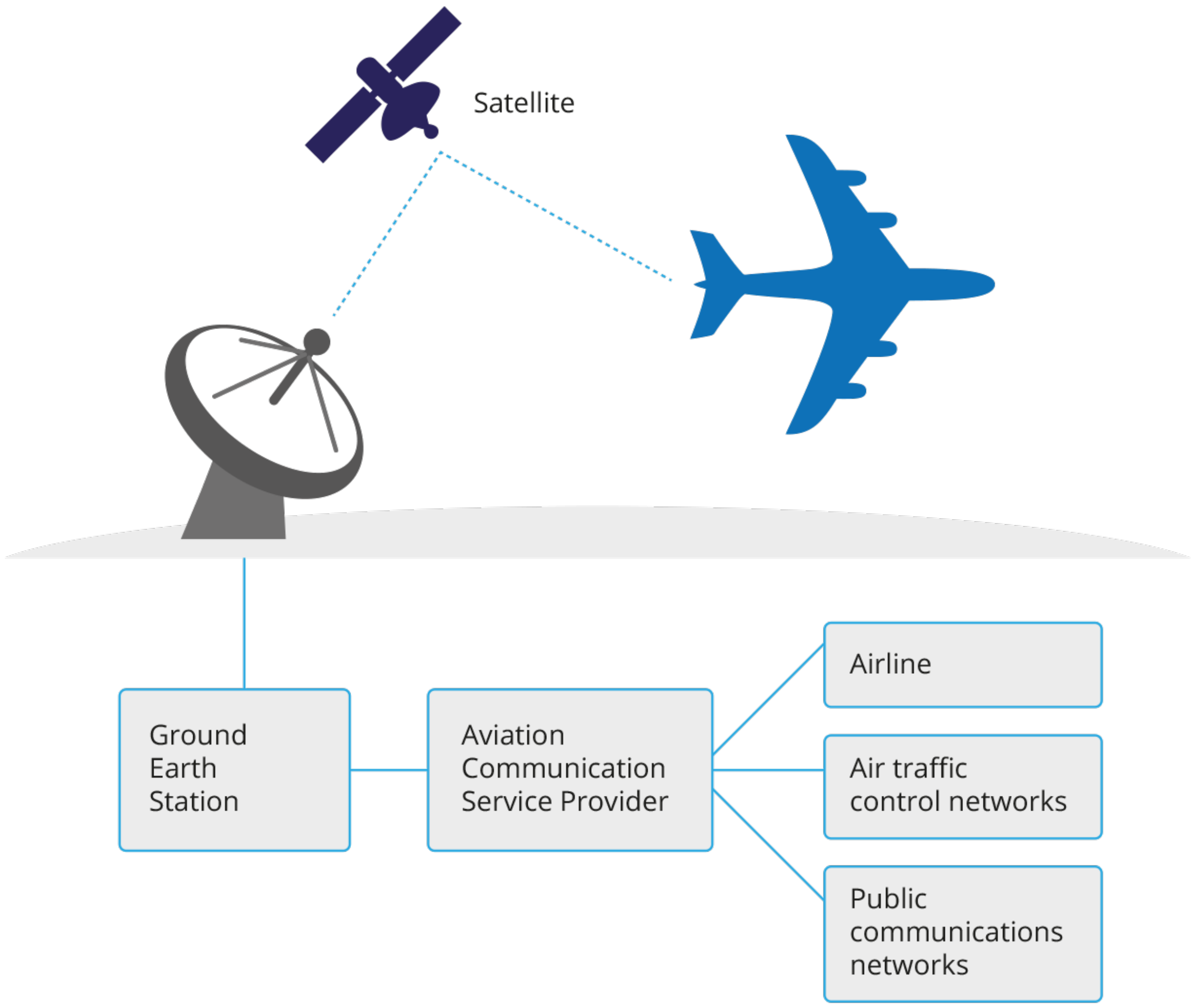
Satellite communication enables safe and efficient passage of the aircraft to its destination with up-to-date information, including route, air traffic and airline operational information, supporting all key cockpit applications.

How satellites safeguard your safe passage

- Satellite-aided ATC
  When an aircraft is out of range of VHF/UHF radio, such as in oceanic airspace, satellite services enable satellite-aided air traffic control (ATC). The ability to have reliable communications with ground-based controllers at all times, on all major oceanic routes, is an important safety feature.
- Automatic dependent surveillance (ADS)
  Satellite communication facilitates the automatic reporting of an aircraft's real-time position, including altitude, speed and heading, via satellite to air traffic control centres, helping controllers know where an aircraft is at all times.
- Controller / pilot datalink communications (CPDLC)
  Satellite communication also enables the Aircraft Communications Addressing and Reporting System (ACARS) routing instructions, clearances and other messages that need to be sent directly to the cockpit as electronic data messages. The benefits are increased flight safety and efficiency through more effective communications.
- Voice services
  Satellite services also support air traffic control and critical airline voice communications when the aircraft is out of VHF radio range. The aircraft systems are set up so that cockpit communications have priority over any of the passenger communications, so maximising safety and reliability.

Always available, always reliable.

How it works


Satellite
Ground Earth Station
Aviation Communication Service Provider
Airline
Air traffic control networks
Public communications networks

Air Traffic Management

On a typical day, in the height of summer, around 30,000 flights travel through European airspace. In the world, that means over 100,000 flights every day!

Satellite networks play an indispensable role in efficient air traffic management. Ka-band satellites using the Gigahertz frequency spectrum can reach user terminals across most of the populated world. As a result, ATM based satellite networks can be effectively used to provide real time as well as non-real time communications services to remote areas. Satellites offer wide geographic coverage including interconnection of “ATM islands”, multipoint to multipoint communications facilitated by the inherent broadcasting ability of satellites, bandwidth on demand (Demand Assignment Multiple Access (DAMA) capabilities), and an alternative to fiber optic networks for disaster recovery options.

Satellites facilitate safe and efficient ATM

Essential C

By Nick Parlante

This Stanford CS Education document tries to summarize all the basic features of the C language. The coverage is pretty quick, so it is most appropriate as review or for someone with some programming background in another language. Topics include variables, int types, floating point types, promotion, truncation, operators, control structures (if, while, for), functions, value parameters, reference parameters, structs, pointers, arrays, the pre-processor, and the standard C library functions.

The most recent version is always maintained at its Stanford CS Education Library URL `http://cslibrary.stanford.edu/101/`. Please send your comments to nick.parlante@cs.stanford.edu.

I hope you can share and enjoy this document in the spirit of goodwill in which it is given away -- Nick Parlante, 4/2003, Stanford California.

Table of Contents

The C Language

C is a professional programmer's language. It was designed to get in one's way as little as possible. Kernighan and Ritchie wrote the original language definition in their book, *The C Programming Language* (below), as part of their research at AT&T. Unix and C++ emerged from the same labs. For several years I used AT&T as my long distance carrier in appreciation of all that CS research, but hearing "thank you for using AT&T" for the millionth time has used up that good will.

Some languages are forgiving. The programmer needs only a basic sense of how things work. Errors in the code are flagged by the compile-time or run-time system, and the programmer can muddle through and eventually fix things up to work correctly. The C language is not like that.

The C programming model is that the programmer knows exactly what they want to do and how to use the language constructs to achieve that goal. The language lets the expert programmer express what they want in the minimum time by staying out of their way.

C is "simple" in that the number of components in the language is small-- If two language features accomplish more-or-less the same thing, C will include only one. C's syntax is terse and the language does not restrict what is "allowed" -- the programmer can pretty much do whatever they want.

C's type system and error checks exist only at compile-time. The compiled code runs in a stripped down run-time model with no safety checks for bad type casts, bad array indices, or bad pointers. There is no garbage collector to manage memory. Instead the programmer mangages heap memory manually. All this makes C fast but fragile.

Analysis -- Where C Fits

Because of the above features, C is hard for beginners. A feature can work fine in one context, but crash in another. The programmer needs to understand how the features work and use them correctly. On the other hand, the number of features is pretty small.

Like most programmers, I have had some moments of real loathing for the C language. It can be irritatingly obedient -- you type something incorrectly, and it has a way of compiling fine and just doing something you don't expect at run-time. However, as I have become a more experienced C programmer, I have grown to appreciate C's straight-to-the point style. I have learned not to fall into its little traps, and I appreciate its simplicity.

Perhaps the best advice is just to be careful. Don't type things in you don't understand. Debugging takes too much time. Have a mental picture (or a real drawing) of how your C code is using memory. That's good advice in any language, but in C it's critical.

Perl and Java are more "portable" than C (you can run them on different computers without a recompile). Java and C++ are more structured than C. Structure is useful for large projects. C works best for small projects where performance is important and the progammers have the time and skill to make it work in C. In any case, C is a very popular and influential language. This is mainly because of C's clean (if minimal) style, it's lack of annoying or regrettable constructs, and the relative ease of writing a C compiler.

Other Resources

- *The C Programming Language*, 2nd ed., by Kernighan and Ritchie. The thin book which for years was the bible for all C programmers. Written by the original designers of the language. The explanations are pretty short, so this book is better as a reference than for beginners.
- http://cslibrary.stanford.edu/102/ Pointers and Memory -- Much more detail about local memory, pointers, reference parameters, and heap memory than in this article, and memory is really the hardest part of C and C++.
- http://cslibrary.stanford.edu//103/ Linked List Basics -- Once you understand the basics of pointers and C, these problems are a good way to get more practice.

Section 1
Basic Types and Operators

C provides a standard, minimal set of basic data types. Sometimes these are called "primitive" types. More complex data structures can be built up from these basic types.

Integer Types

The "integral" types in C form a family of integer types. They all behave like integers and can be mixed together and used in similar ways. The differences are due to the different number of bits ("widths") used to implement each type -- the wider types can store a greater ranges of values.

`char` ASCII character -- at least 8 bits. Pronounced "car". As a practical matter `char` is basically always a byte which is 8 bits which is enough to store a single ASCII character. 8 bits provides a signed range of -128..127 or an unsigned range is 0..255. `char` is also required to be the "smallest addressable unit" for the machine -- each byte in memory has its own address.

`short` Small integer -- at least 16 bits which provides a signed range of -32768..32767. Typical size is 16 bits. Not used so much.

`int` Default integer `--` at least 16 bits, with 32 bits being typical. Defined to be the "most comfortable" size for the computer. If you do not really care about the range for an integer variable, declare it `int` since that is likely to be an appropriate size (16 or 32 bit) which works well for that machine.

`long` Large integer -- at least 32 bits. Typical size is 32 bits which gives a signed range of about -2 billion ..+2 billion. Some compilers support "long long" for 64 bit ints.

The integer types can be preceded by the qualifier `unsigned` which disallows representing negative numbers, but doubles the largest positive number representable. For example, a 16 bit implementation of `short` can store numbers in the range -32768..32767, while `unsigned short` can store 0..65535. You can think of pointers as being a form of `unsigned long` on a machine with 4 byte pointers. In my opinion, it's best to avoid using `unsigned` unless you really need to. It tends to cause more misunderstandings and problems than it is worth.

Extra: Portability Problems

Instead of defining the exact sizes of the integer types, C defines lower bounds. This makes it easier to implement C compilers on a wide range of hardware. Unfortunately it occasionally leads to bugs where a program runs differently on a 16-bit-int machine than it runs on a 32-bit-int machine. In particular, if you are designing a function that will be implemented on several different machines, it is a good idea to use typedefs to set up types like `Int32` for 32 bit int and `Int16` for 16 bit int. That way you can prototype a function `Foo(Int32)` and be confident that the typedefs for each machine will be set so that the function really takes exactly a 32 bit int. That way the code will behave the same on all the different machines.

char Constants

A `char` constant is written with single quotes (') like 'A' or 'z'. The `char` constant 'A' is really just a synonym for the ordinary integer value 65 which is the ASCII value for uppercase 'A'. There are special case char constants, such as '\t' for tab, for characters which are not convenient to type on a keyboard.

`'A'`	uppercase 'A' character
`'\n'`	newline character
`'\t'`	tab character
`'\0'`	the "null" character -- integer value 0 (different from the char digit '0')
`'\012'`	the character with value 12 in octal, which is decimal 10

int Constants
Numbers in the source code such as `234` default to type `int`. They may be followed by an 'L' (upper or lower case) to designate that the constant should be a `long` such as 42L. An integer constant can be written with a leading 0x to indicate that it is expressed in hexadecimal -- `0x10` is way of expressing the number 16. Similarly, a constant may be written in octal by preceding it with "0" -- `012` is a way of expressing the number 10.

Type Combination and Promotion
The integral types may be mixed together in arithmetic expressions since they are all basically just integers with variation in their width. For example, `char` and `int` can be combined in arithmetic expressions such as `('b' + 5)`. How does the compiler deal with the different widths present in such an expression? In such a case, the compiler "promotes" the smaller type (`char`) to be the same size as the larger type (`int`) before combining the values. Promotions are determined at compile time based purely on the **types** of the values in the expressions. Promotions do not lose information -- they always convert from a type to compatible, larger type to avoid losing information.

Pitfall -- int Overflow
I once had a piece of code which tried to compute the number of bytes in a buffer with the expression `(k * 1024)` where `k` was an `int` representing the number of kilobytes I wanted. Unfortunately this was on a machine where `int` happened to be 16 bits. Since `k` and 1024 were both `int`, there was no promotion. For values of k >= 32, the product was too big to fit in the 16 bit `int` resulting in an overflow. The compiler can do whatever it wants in overflow situations -- typically the high order bits just vanish. One way to fix the code was to rewrite it as `(k * 1024L)` -- the `long` constant forced the promotion of the `int`. This was not a fun bug to track down -- the expression sure looked reasonable in the source code. Only stepping past the key line in the debugger showed the overflow problem. "Professional Programmer's Language." This example also demonstrates the way that C only promotes based on the **types** in an expression. The compiler does not consider the values 32 or 1024 to realize that the operation will overflow (in general, the values don't exist until run time anyway). The compiler just looks at the compile time types, `int` and `int` in this case, and thinks everything is fine.

Floating point Types

`float`	Single precision floating point number	typical size: 32 bits
`double`	Double precision floating point number	typical size: 64 bits
`long double`	Possibly even bigger floating point number (somewhat obscure)	

Constants in the source code such as 3.14 default to type `double` unless the are suffixed with an 'f' (float) or 'l' (long double). Single precision equates to about 6 digits of precision and double is about 15 digits of precision. Most C programs use `double` for their computations. The main reason to use `float` is to save memory if many numbers need to be stored. The main thing to remember about floating point numbers is that they are inexact. For example, what is the value of the following `double` expression?

```
   (1.0/3.0 + 1.0/3.0 + 1.0/3.0)    // is this equal to 1.0 exactly?
```

The sum may or may not be 1.0 exactly, and it may vary from one type of machine to another. For this reason, you should never compare floating numbers to eachother for equality (==) -- use inequality (<) comparisons instead. Realize that a correct C program run on different computers may produce slightly different outputs in the rightmost digits of its floating point computations.

Comments

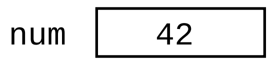
Comments in C are enclosed by slash/star pairs: `/* .. comments .. */` which may cross multiple lines. C++ introduced a form of comment started by two slashes and extending to the end of the line: `// comment until the line end`
The // comment form is so handy that many C compilers now also support it, although it is not technically part of the C language.

Along with well-chosen function names, comments are an important part of well written code. Comments should not just repeat what the code says. Comments should describe what the code **accomplishes** which is much more interesting than a translation of what each statement does. Comments should also narrate what is tricky or non-obvious about a section of code.

Variables

As in most languages, a variable declaration reserves and names an area in memory at run time to hold a value of particular type. Syntactically, C puts the type first followed by the name of the variable. The following declares an `int` variable named "`num`" and the 2nd line stores the value 42 into `num`.

```
   int num;
   num = 42;
```



A variable corresponds to an area of memory which can store a value of the given type. Making a drawing is an excellent way to think about the variables in a program. Draw each variable as box with the current value inside the box. This may seem like a "beginner" technique, but when I'm buried in some horribly complex programming problem, I invariably resort to making a drawing to help think the problem through.

Variables, such as `num`, do not have their memory cleared or set in any way when they are allocated at run time. Variables start with random values, and it is up to the program to set them to something sensible before depending on their values.

Names in C are case sensitive so "x" and "X" refer to different variables. Names can contain digits and underscores (_), but may not begin with a digit. Multiple variables can be declared after the type by separating them with commas. C is a classical "compile time" language -- the names of the variables, their types, and their implementations are all flushed out by the compiler at compile time (as opposed to figuring such details out at run time like an interpreter).

```
   float x, y, z, X;
```

Assignment Operator =

The assignment operator is the single equals sign (=).

```
   i = 6;
   i = i + 1;
```

The assignment operator copies the value from its right hand side to the variable on its left hand side. The assignment also acts as an expression which returns the newly assigned value. Some programmers will use that feature to write things like the following.

```
   y = (x = 2 * x);     // double x, and also put x's new value in y
```

Truncation

The opposite of promotion, truncation moves a value from a type to a smaller type. In that case, the compiler just drops the extra bits. It may or may not generate a compile time warning of the loss of information. Assigning from an integer to a smaller integer (e.g.. `long` to `int`, or `int` to `char`) drops the most significant bits. Assigning from a floating point type to an integer drops the fractional part of the number.

```
   char ch;
   int i;

   i = 321;
   ch = i;     // truncation of an int value to fit in a char
   // ch is now 65
```

The assignment will drop the upper bits of the `int` 321. The lower 8 bits of the number 321 represents the number 65 (321 - 256). So the value of `ch` will be (`char`)65 which happens to be 'A'.

The assignment of a floating point type to an integer type will drop the fractional part of the number. The following code will set `i` to the value 3. This happens when assigning a floating point number to an integer or passing a floating point number to a function which takes an integer.

```
   double pi;
   int i;

   pi = 3.14159;
   i = pi;     // truncation of a double to fit in an int
   // i is now 3
```

Pitfall -- int vs. float Arithmetic

Here's an example of the sort of code where `int` vs. `float` arithmetic can cause problems. Suppose the following code is supposed to scale a homework score in the range 0..20 to be in the range 0..100.

```
   {
      int score;
   ...// suppose score gets set in the range 0..20 somehow
```

```
      score = (score / 20) * 100;         // NO -- score/20 truncates to 0
   ...
```

Unfortunately, score will almost always be set to 0 for this code because the integer division in the expression `(score/20)` will be 0 for every value of score less than 20. The fix is to force the quotient to be computed as a floating point number...

```
   score = ((double)score / 20) * 100;    // OK -- floating point division from cast

   score = (score / 20.0) * 100;          // OK -- floating point division from 20.0

   score = (int)(score / 20.0) * 100;     // NO -- the (int) truncates the floating
                                          // quotient back to 0
```

No Boolean -- Use int

C does not have a distinct boolean type-- `int` is used instead. The language treats integer 0 as false and all non-zero values as true. So the statement...

```
   i = 0;
   while (i - 10) {
      ...
```

will execute until the variable `i` takes on the value 10 at which time the expression `(i - 10)` will become false (i.e. 0). (we'll see the while() statement a bit later)

Mathematical Operators

C includes the usual binary and unary arithmetic operators. See the appendix for the table of precedence. Personally, I just use parenthesis liberally to avoid any bugs due to a misunderstanding of precedence. The operators are sensitive to the type of the operands. So division (`/`) with two integer arguments will do integer division. If either argument is a float, it does floating point division. So `(6/4)` evaluates to 1 while `(6/4.0)` evaluates to 1.5 -- the 6 is promoted to 6.0 before the division.

`+`	Addition
`-`	Subtraction
`/`	Division
`*`	Multiplication
`%`	Remainder (mod)

Unary Increment Operators: ++ --

The unary ++ and -- operators increment or decrement the value in a variable. There are "pre" and "post" variants for both operators which do slightly different things (explained below)

`var``++`	increment	"post" variant
`++`*`var`*	increment	"pre" variant

`var--`	decrement	"post" variant
`--var`	decrement	"pre" variant

```
int i = 42;
i++;     // increment on i
// i is now 43
i--;     // decrement on i
// i is now 42
```

Pre and Post Variations

The Pre/Post variation has to do with nesting a variable with the increment or decrement operator inside an expression -- should the entire expression represent the value of the variable before or after the change? I **never** use the operators in this way (see below), but an example looks like...

```
int i = 42;
int j;

j = (i++ + 10);
// i is now 43
// j is now 52 (NOT 53)

j = (++i + 10)
// i is now 44
// j is now 54
```

C Programming Cleverness and Ego Issues

Relying on the difference between the pre and post variations of these operators is a classic area of C programmer ego showmanship. The syntax is a little tricky. It makes the code a little shorter. These qualities drive some C programmers to show off how clever they are. C invites this sort of thing since the language has many areas (this is just one example) where the programmer can get a complex effect using a code which is short and dense.

If I want j to depend on i's value before the increment, I write...

```
j = (i + 10);
i++;
```

Or if I want to j to use the value after the increment, I write...

```
i++;
j = (i + 10);
```

Now then, isn't that nicer? (editorial) Build programs that do something cool rather than programs which flex the language's syntax. Syntax -- who cares?

Relational Operators

These operate on integer or floating point values and return a 0 or 1 boolean value.

`==`	Equal

`!=`	Not Equal
`>`	Greater Than
`<`	Less Than
`>=`	Greater or Equal
`<=`	Less or Equal

To see if x equals three, write something like:

```
   if (x == 3) ...
```

Pitfall = ≠ ==

An absolutely classic pitfall is to write assignment (=) when you mean comparison (==). This would not be such a problem, except the incorrect assignment version compiles fine because the compiler assumes you mean to use the value returned by the assignment. This is rarely what you want

```
   if (x = 3) ...
```

This does not test if `x` is 3. This sets `x` to the value 3, and then returns the 3 to the `if` for testing. 3 is not 0, so it counts as "true" every time. This is probably the single most common error made by beginning C programmers. The problem is that the compiler is no help -- it thinks both forms are fine, so the only defense is extreme vigilance when coding. Or write "= ≠ ==" in big letters on the back of your hand before coding. This mistake is an absolute classic and it's a bear to debug. Watch Out! And need I say: "Professional Programmer's Language."

Logical Operators

The value 0 is false, anything else is true. The operators evaluate left to right and stop as soon as the truth or falsity of the expression can be deduced. (Such operators are called "short circuiting") In ANSI C, these are furthermore guaranteed to use 1 to represent true, and not just some random non-zero bit pattern. However, there are many C programs out there which use values other than 1 for true (non-zero pointers for example), so when programming, do not assume that a true boolean is necessarily 1 exactly.

`!`	Boolean not (unary)
`&&`	Boolean and
`||`	Boolean or

Bitwise Operators

C includes operators to manipulate memory at the bit level. This is useful for writing low-level hardware or operating system code where the ordinary abstractions of numbers, characters, pointers, etc... are insufficient -- an increasingly rare need. Bit manipulation code tends to be less "portable". Code is "portable" if with no programmer intervention it compiles and runs correctly on different types of computers. The bitwise operations are typically used with unsigned types. In particular, the shift operations are guaranteed to shift 0 bits into the newly vacated positions when used on unsigned values.

`~`	Bitwise Negation (unary) – flip 0 to 1 and 1 to 0 throughout
`&`	Bitwise And
`|`	Bitwise Or
`^`	Bitwise Exclusive Or
`>>`	Right Shift by right hand side (RHS) (divide by power of 2)
`<<`	Left Shift by RHS (multiply by power of 2)

Do not confuse the Bitwise operators with the logical operators. The bitwise connectives are one character wide (`&`, `|`) while the boolean connectives are two characters wide (`&&`, `||`). The bitwise operators have higher precedence than the boolean operators. The compiler will never help you out with a type error if you use `&` when you meant `&&.` As far as the type checker is concerned, they are identical-- they both take and produce integers since there is no distinct boolean type.

Other Assignment Operators

In addition to the plain = operator, C includes many shorthand operators which represents variations on the basic =. For example "+=" adds the right hand side to the left hand side. `x = x + 10;` can be reduced to `x += 10;`. This is most useful if x is a long expression such as the following, and in some cases it may run a little faster.

```
person->relatives.mom.numChildren += 2;      // increase children by 2
```

Here's the list of assignment shorthand operators...

`+=, -=`	Increment or decrement by RHS
`*=, /=`	Multiply or divide by RHS
`%=`	Mod by RHS
`>>=`	Bitwise right shift by RHS (divide by power of 2)
`<<=`	Bitwise left shift RHS (multiply by power of 2)
`&=, |=, ^=`	Bitwise and, or, xor by RHS

Section 2
Control Structures

Curly Braces {}

C uses curly braces ({}) to group multiple statements together. The statements execute in order. Some languages let you declare variables on any line (C++). Other languages insist that variables are declared only at the beginning of functions (Pascal). C takes the middle road -- variables may be declared within the body of a function, but they must follow a '{'. More modern languages like Java and C++ allow you to declare variables on any line, which is handy.

If Statement

Both an if and an if-else are available in C. The *<expression>* can be any valid expression. The parentheses around the expression are required, even if it is just a single variable.

```
if (<expression>) <statement>    // simple form with no {}'s or else clause


if (<expression>) {        // simple form with {}'s to group statements
   <statement>
   <statement>
}


if (<expression>) {        // full then/else form
   <statement>
}
else {
   <statement>
}
```

Conditional Expression -or- The Ternary Operator

The conditional expression can be used as a shorthand for some if-else statements. The general syntax of the conditional operator is:

```
<expression1> ? <expression2> : <expression3>
```

This is an expression, not a statement, so it represents a value. The operator works by evaluating expression1. If it is true (non-zero), it evaluates and returns expression2 . Otherwise, it evaluates and returns expression3.

The classic example of the ternary operator is to return the smaller of two variables. Every once in a while, the following form is just what you needed. Instead of...

```
if (x < y) {
   min = x;
}
else {
   min = y;
}
```

You just say...

```
min = (x < y) ? x : y;
```

Switch Statement

The `switch` statement is a sort of specialized form of `if` used to efficiently separate different blocks of code based on the value of an integer. The switch expression is evaluated, and then the flow of control jumps to the matching const-expression case. The case expressions are typically int or char constants. The `switch` statement is probably the single most syntactically awkward and error-prone features of the C language.

```
switch (<expression>) {
   case <const-expression-1>:
      <statement>
      break;

   case <const-expression-2>:
      <statement>
      break;

   case <const-expression-3>:    // here we combine case 3 and 4
   case <const-expression-4>:
      <statement>
      break;

   default:    // optional
      <statement>
}
```

Each constant needs its own `case` keyword and a trailing colon (:). Once execution has jumped to a particular case, the program will keep running through all the cases from that point down -- this so called "fall through" operation is used in the above example so that expression-3 and expression-4 run the same statements. The explicit `break` statements are necessary to exit the `switch`. Omitting the `break` statements is a common error -- it compiles, but leads to inadvertent fall-through behavior.

Why does the switch statement fall-through behavior work the way it does? The best explanation I can think of is that originally C was developed for an audience of assembly language programmers. The assembly language programmers were used to the idea of a jump table with fall-through behavior, so that's the way C does it (it's also relatively easy to implement it this way.) Unfortunately, the audience for C is now quite different, and the fall-through behavior is widely regarded as a terrible part of the language.

While Loop

The while loop evaluates the test expression before every loop, so it can execute zero times if the condition is initially false. It requires the parenthesis like the if.

```
while (<expression>) {
   <statement>
}
```

Do-While Loop
Like a while, but with the test condition at the bottom of the loop. The loop body will always execute at least once. The do-while is an unpopular area of the language, most everyone tries to use the straight `while` if at all possible.

```
   do {
      <statement>
   } while (<expression>)
```

For Loop
The for loop in C is the most general looping construct. The loop header contains three parts: an initialization, a continuation condition, and an action.

```
   for (<initialization>; <continuation>; <action>) {
      <statement>
   }
```

The initialization is executed once before the body of the loop is entered. The loop continues to run as long as the continuation condition remains true (like a `while`). After every execution of the loop, the action is executed. The following example executes 10 times by counting 0..9. Many loops look very much like the following...

```
   for (i = 0; i < 10; i++) {
      <statement>
   }
```

C programs often have series of the form 0..(some_number-1). It's idiomatic in C for the above type loop to start at 0 and use < in the test so the series runs up to but not equal to the upper bound. In other languages you might start at 1 and use <= in the test.

Each of the three parts of the `for` loop can be made up of multiple expressions separated by commas. Expressions separated by commas are executed in order, left to right, and represent the value of the last expression. (See the string-reverse example below for a demonstration of a complex for loop.)

Break
The `break` statement will move control outside a loop or switch statement. Stylistically speaking, `break` has the potential to be a bit vulgar. It's preferable to use a straight `while` with a single test at the top if possible. Sometimes you are forced to use a `break` because the test can occur only somewhere in the midst of the statements in the loop body. To keep the code readable, be sure to make the break obvious -- forgetting to account for the action of a break is a traditional source of bugs in loop behavior.

```
   while (<expression>) {
      <statement>
      <statement>

      if (<condition which can only be evaluated here>)
         break;

      <statement>
      <statement>
   }
   // control jumps down here on the break
```

The `break` does not work with `if`. It only works in loops and switches. Thinking that a break refers to an `if` when it really refers to the enclosing `while` has created some high quality bugs. When using a `break`, it's nice to write the enclosing loop to iterate in the most straightforward, obvious, normal way, and then use the `break` to explicitly catch the exceptional, weird cases.

Continue

The `continue` statement causes control to jump to the bottom of the loop, effectively skipping over any code below the continue. As with `break`, this has a reputation as being vulgar, so use it sparingly. You can almost always get the effect more clearly using an if inside your loop.

```
while (<expression>) {
   ...
   if (<condition>)
      continue;
   ...
   ...
   // control jumps here on the continue
}
```

Section 3
Complex Data Types

C has the usual facilities for grouping things together to form composite types-- arrays and records (which are called "structures"). The following definition declares a type called "struct fraction" that has two integer sub fields named "numerator" and "denominator". If you forget the semicolon it tends to produce a syntax error in whatever thing follows the struct declaration.

```
struct fraction {
   int numerator;
   int denominator;
};          // Don't forget the semicolon!
```

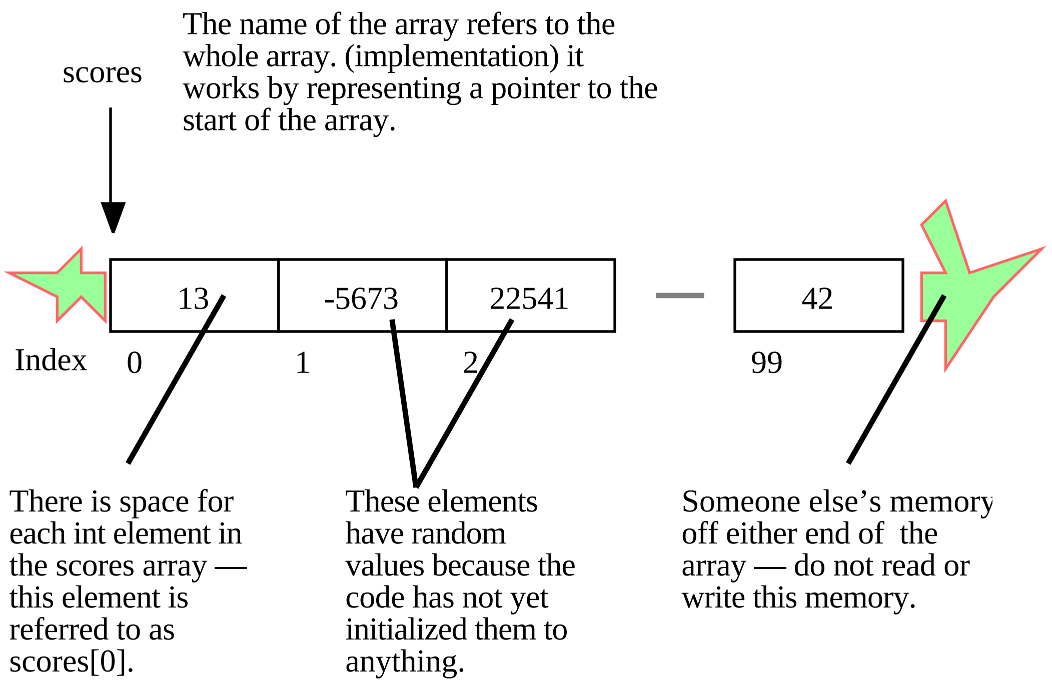
This declaration introduces the type `struct fraction` (both words are required) as a new type. C uses the period (.) to access the fields in a record. You can copy two records of the same type using a single assignment statement, however == does not work on structs.

```
struct fraction f1, f2;          // declare two fractions

f1.numerator = 22;
f1.denominator = 7;

f2 = f1;    // this copies over the whole struct
```

Arrays

The simplest type of array in C is one which is declared and used in one place. There are more complex uses of arrays which I will address later along with pointers. The following declares an array called `scores` to hold 100 integers and sets the first and last elements. C arrays are always indexed from 0. So the first `int` in `scores` array is `scores[0]` and the last is `scores[99]`.

```
int scores[100];

scores[0]  = 13;           // set first element
scores[99] = 42;           // set last element
```


scores
The name of the array refers to the whole array. (implementation) it works by representing a pointer to the start of the array.
13
-5673
22541
42
Index
0
1
2
99
There is space for each int element in the scores array — this element is referred to as scores[0].
These elements have random values because the code has not yet initialized them to anything.
Someone else's memory off either end of the array — do not read or write this memory.

It's a very common error to try to refer to non-existent `scores[100]` element. C does not do any run time or compile time bounds checking in arrays. At run time the code will just access or mangle whatever memory it happens to hit and crash or misbehave in some unpredictable way thereafter. "Professional programmer's language." The convention of numbering things `0..(number of things - 1)` pervades the language. To best integrate with C and other C programmers, you should use that sort of numbering in your own data structures as well.

Multidimensional Arrays

The following declares a two-dimensional 10 by 10 array of integers and sets the first and last elements to be 13.

```
int board [10][10];

board[0][0] = 13;
board[9][9] = 13;
```

The implementation of the array stores all the elements in a single contiguous block of memory. The other possible implementation would be a combination of several distinct one dimensional arrays -- that's not how C does it. In memory, the array is arranged with the elements of the rightmost index next to each other. In other words, `board[1][8]` comes right before `board[1][9]` in memory.

(highly optional efficiency point) It's typically efficient to access memory which is near other recently accessed memory. This means that the most efficient way to read through a chunk of the array is to vary the rightmost index the most frequently since that will access elements that are near each other in memory.

Array of Structs

The following declares an array named "numbers" which holds 1000 `struct fraction`'s.

```
   struct fraction numbers[1000];

   numbers[0].numerator = 22;       /* set the 0th struct fraction */
   numbers[0].denominator = 7;
```

Here's a general trick for unraveling C variable declarations: look at the right hand side and imagine that it is an expression. The type of that expression is the left hand side. For the above declarations, an expression which looks like the right hand side (`numbers[1000],` or really anything of the form `numbers[...]`) will be the type on the left hand side (`struct fraction`).

Pointers

A pointer is a value which represents a reference to another value sometimes known as the pointer's "pointee". Hopefully you have learned about pointers somewhere else, since the preceding sentence is probably inadequate explanation. This discussion will concentrate on the syntax of pointers in C -- for a much more complete discussion of pointers and their use see http://cslibrary.stanford.edu/102/, Pointers and Memory.

Syntax

Syntactically C uses the asterisk or "star" (*) to indicate a pointer. C defines pointer types based on the type pointee. A `char*` is type of pointer which refers to a single `char.` a `struct fraction*` is type of pointer which refers to a `struct fraction`.

```
   int* intPtr;   // declare an integer pointer variable intPtr

   char* charPtr; // declares a character pointer --
                  // a very common type of pointer


   // Declare two struct fraction pointers
   // (when declaring multiple variables on one line, the *
   //  should go on the right with the variable)
   struct fraction *f1, *f2;
```

The Floating "*"

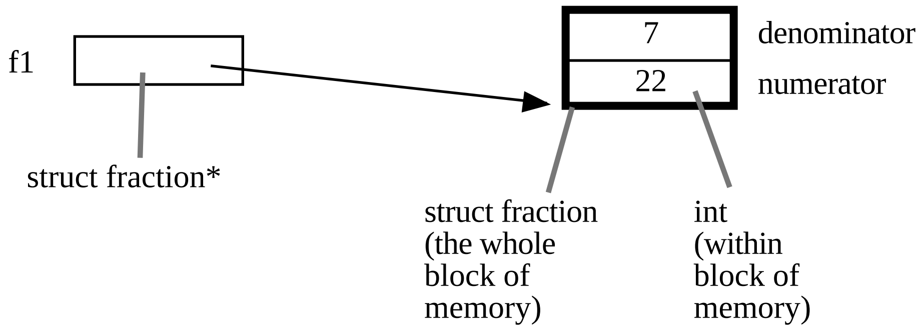
In the syntax, the star is allowed to be anywhere between the base type and the variable name. Programmer's have their own conventions-- I generally stick the * on the left with the type. So the above declaration of intPtr could be written equivalently...

```
   int  *intPtr;        // these are all the same
   int * intPtr;
   int*  intPtr;
```

Pointer Dereferencing

We'll see shortly how a pointer is set to point to something -- for now just assume the pointer points to memory of the appropriate type. In an expression, the unary * to the left of a pointer dereferences it to retrieve the value it points to. The following drawing shows the types involved with a single pointer pointing to a struct fraction.

```
struct fraction* f1;
```


f1
7
22
denominator
numerator
struct fraction*
struct fraction (the whole block of memory)
int (within block of memory)

```
Expression          Type
f1                  struct fraction*
*f1                 struct fraction
(*f1).numerator     int
```

There's an alternate, more readable syntax available for dereferencing a pointer to a struct. A "`->`" at the right of the pointer can access any of the fields in the struct. So the reference to the numerator field could be written `f1->numerator`.

Here are some more complex declarations...

```
struct fraction** fp;      // a pointer to a pointer to a struct fraction

struct fraction fract_array[20];       // an array of 20 struct fractions

struct fraction* fract_ptr_array[20];  // an array of 20 pointers to
                                       // struct fractions
```

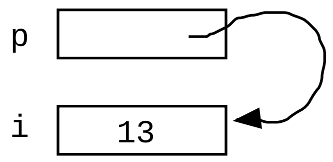
One nice thing about the C type syntax is that it avoids the circular definition problems which come up when a pointer structure needs to refer to itself. The following definition defines a node in a linked list. Note that no preparatory declaration of the node pointer type is necessary.

```
struct node {
   int data;
   struct node* next;
};
```

The & Operator

The & operator is one of the ways that pointers are set to point to things. The & operator computes a pointer to the argument to its right. The argument can be any variable which takes up space in the stack or heap (known as an "LValue" technically). So `&i` and `&(f1->numerator)` are ok, but `&6` is not. Use & when you have some memory, and you want a pointer to that memory.

```
void foo() {
   int* p;  // p is a pointer to an integer
   int i;   // i is an integer

   p = &i;  // Set p to point to i
   *p = 13; // Change what p points to -- in this case i -- to 13

   // At this point i is 13. So is *p. In fact *p is i.
}
```



When using a pointer to an object created with &, it is important to only use the pointer so long as the object exists. A local variable exists only as long as the function where it is declared is still executing (we'll see functions shortly). In the above example, `i` exists only as long as `foo()` is executing. Therefore any pointers which were initialized with `&i` are valid only as long as `foo()` is executing. This "lifetime" constraint of local memory is standard in many languages, and is something you need to take into account when using the & operator.

NULL

A pointer can be assigned the value 0 to explicitly represent that it does not currently have a pointee. Having a standard representation for "no current pointee" turns out to be very handy when using pointers. The constant NULL is defined to be 0 and is typically used when setting a pointer to NULL. Since it is just 0, a NULL pointer will behave like a boolean false when used in a boolean context. Dereferencing a NULL pointer is an error which, if you are lucky, the computer will detect at runtime -- whether the computer detects this depends on the operating system.

Pitfall -- Uninitialized Pointers

When using pointers, there are two entities to keep track of. The pointer and the memory it is pointing to, sometimes called the "pointee". There are three things which must be done for a pointer/pointee relationship to work...

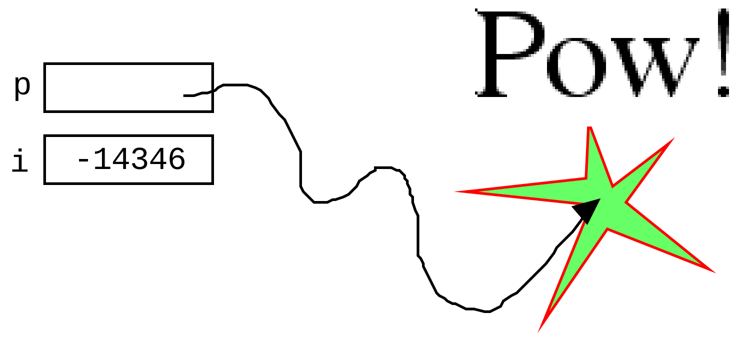
(1) The pointer must be declared and allocated

(2) The pointee must be declared and allocated

(3) The pointer (1) must be initialized so that it points to the pointee (2)

The most common pointer related error of all time is the following: Declare and allocate the pointer (step 1). Forget step 2 and/or 3. Start using the pointer as if it has been setup to point to something. Code with this error frequently compiles fine, but the runtime results are disastrous. Unfortunately the pointer does not point anywhere good unless (2) and (3) are done, so the run time dereference operations on the pointer with `*` will misuse and trample memory leading to a random crash at some point.

```
{
   int* p;

   *p = 13;     // NO NO NO p does not point to an int yet
                // this just overwrites a random area in memory
}
```



Of course your code won't be so trivial, but the bug has the same basic form: declare a pointer, but forget to set it up to point to a particular pointee.

Using Pointers

Declaring a pointer allocates space for the pointer itself, **but it does not allocate space for the pointee**. The pointer must be set to point to something before you can dereference it.

Here's some code which doesn't do anything useful, but which does demonstrate (1) (2) (3) for pointer use correctly...

```
int* p;     // (1) allocate the pointer
int i;      // (2) allocate pointee
struct fraction f1;  // (2) allocate pointee

p = &i;     // (3) setup p to point to i
*p = 42;    // ok to use p since it's setup

p = &(f1.numerator);       // (3) setup p to point to a different int
*p = 22;

p = &(f1.denominator);     // (3)
*p = 7;
```

So far we have just used the & operator to create pointers to simple variables such as `i`. Later, we'll see other ways of getting pointers with arrays and other techniques.

C Strings

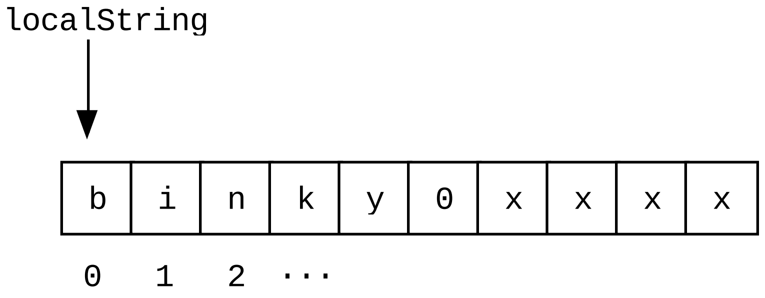
C has minimal support of character strings. For the most part, strings operate as ordinary arrays of characters. Their maintenance is up to the programmer using the standard facilities available for arrays and pointers. C does include a standard library of functions which perform common string operations, but the programmer is responsible for the managing the string memory and calling the right functions. Unfortunately computations involving strings are very common, so becoming a good C programmer often requires becoming adept at writing code which manages strings which means managing pointers and arrays.

A C string is just an array of `char` with the one additional convention that a "null" character ('\0') is stored after the last real character in the array to mark the end of the string. The compiler represents string constants in the source code such as "binky" as arrays which follow this convention. The string library functions (see the appendix for a partial list) operate on strings stored in this way. The most useful library function is `strcpy(char dest[], const char source[]);` which copies the bytes of one string over to another. The order of the arguments to strcpy() mimics the arguments in of '=' -- the right is assigned to the left. Another useful string function is `strlen(const char string[]);` which returns the number of characters in C string not counting the trailing '\0'.

Note that the regular assignment operator (=) does **not** do string copying which is why strcpy() is necessary. See Section 6, Advanced Pointers and Arrays, for more detail on how arrays and pointers work.

The following code allocates a 10 `char` array and uses strcpy() to copy the bytes of the string constant "binky" into that local array.

```
{
   char localString[10];

   strcpy(localString, "binky");
}
```



The memory drawing shows the local variable `localString` with the string "binky" copied into it. The letters take up the first 5 characters and the '\0' char marks the end of the string after the 'y'. The x's represent characters which have not been set to any particular value.

If the code instead tried to store the string "I enjoy languages which have good string support" into localString, the code would just crash at run time since the 10 character array can contain at most a 9 character string. The large string will be written passed the right hand side of localString, overwriting whatever was stored there.

String Code Example

Here's a moderately complex `for` loop which reverses a string stored in a local array. It demonstrates calling the standard library functions strcpy() and strlen() and demonstrates that a string really is just an array of characters with a '\0' to mark the effective end of the string. Test your C knowledge of arrays and `for` loops by making a drawing of the memory for this code and tracing through its execution to see how it works.

```
{

char string[1000];   // string is a local 1000 char array
int len;

strcpy(string, "binky");
len = strlen(string);

/*
 Reverse the chars in the string:
 i starts at the beginning and goes up
 j starts at the end and goes down
 i/j exchange their chars as they go until they meet
*/
int i, j;
char temp;
for (i = 0, j = len - 1; i < j; i++, j--) {
   temp = string[i];
   string[i] = string[j];
   string[j] = temp;
}

// at this point the local string should be "yknib"

}
```

"Large Enough" Strings

The convention with C strings is that the owner of the string is responsible for allocating array space which is "large enough" to store whatever the string will need to store. Most routines do not check that size of the string memory they operate on, they just assume its big enough and blast away. Many, many programs contain declarations like the following...

```
{
   char localString[1000];
   ...
}
```

The program works fine so long as the strings stored are 999 characters or shorter. Someday when the program needs to store a string which is 1000 characters or longer, then it crashes. Such array-not-quite-big-enough problems are a common source of bugs, and are also the source of so called "buffer overflow" security problems. This scheme has the additional disadvantage that most of the time when the array is storing short strings, 95% of the memory reserved is actually being wasted. A better solution allocates the string dynamically in the heap, so it has just the right size.

To avoid buffer overflow attacks, production code should check the size of the data first, to make sure it fits in the destination string. See the strlcpy() function in Appendix A.

char*

Because of the way C handles the types of arrays, the type of the variable `localString` above is essentially `char*`. C programs very often manipulate strings using variables of type `char*` which point to arrays of characters. Manipulating the actual chars in a string requires code which manipulates the underlying array, or the use of library functions such as strcpy() which manipulate the array for you. See Section 6 for more detail on pointers and arrays.

TypeDef

A typedef statement introduces a shorthand name for a type. The syntax is...

```
   typedef <type> <name>;
```

The following defines `Fraction` type to be the type (`struct fraction)`. C is case sensitive, so `fraction` is different from `Fraction`. It's convenient to use typedef to create types with upper case names and use the lower-case version of the same word as a variable.

```
   typedef struct fraction Fraction;

   Fraction fraction;   // Declare the variable "fraction" of type "Fraction"
                        //  which is really just a synonym for "struct fraction".
```

The following typedef defines the name Tree as a standard pointer to a binary tree node where each node contains some data and "smaller" and "larger" subtree pointers.

```
   typedef struct treenode* Tree;
   struct treenode {
      int data;
      Tree smaller, larger;   // equivalently, this line could say
   };                         // "struct treenode *smaller, *larger"
```

Section 4
Functions

All languages have a construct to separate and package blocks of code. C uses the "function" to package blocks of code. This article concentrates on the syntax and peculiarities of C functions. The motivation and design for dividing a computation into separate blocks is an entire discipline in its own.

A function has a name, a list of arguments which it takes when called, and the block of code it executes when called. C functions are defined in a text file and the names of all the functions in a C program are lumped together in a single, flat namespace. The special function called "main" is where program execution begins. Some programmers like to begin their function names with Upper case, using lower case for variables and parameters, Here is a simple C function declaration. This declares a function named `Twice` which takes a single `int` argument named `num`. The body of the function computes the value which is twice the `num` argument and returns that value to the caller.

```
/*
 Computes double of a number.
 Works by tripling the number, and then subtracting to get back to double.
*/
static int Twice(int num) {
   int result = num * 3;
   result = result - num;
   return(result);
}
```

Syntax

The keyword "`static`" defines that the function will only be available to callers in the file where it is declared. If a function needs to be called from another file, the function cannot be static and will require a prototype -- see prototypes below. The `static` form is convenient for utility functions which will only be used in the file where they are declared. Next , the "`int`" in the function above is the type of its return value. Next comes name of the function and its list of parameters. When referring to a function by name in documentation or other prose, it's a convention to keep the parenthesis () suffix, so in this case I refer to the function as "`Twice()`". The parameters are listed with their types and names, just like variables.

Inside the function, the parameter `num` and the local variable `result` are "local" to the function -- they get their own memory and exist only so long as the function is executing. This independence of "local" memory is a standard feature of most languages (See CSLibrary/102 for the detailed discussion of local memory).

The "caller" code which calls `Twice`() looks like...

```
int num = 13;
int a = 1;
int b = 2;
a = Twice(a);        // call Twice() passing the value of a
b = Twice(b + num);  // call Twice() passing the value b+num
// a == 2
// b == 30
// num == 13 (this num is totally independent of the "num" local to Twice()
```

Things to notice...

(vocabulary) The expression passed to a function by its caller is called the "actual parameter" -- such as "a" and "b + num" above. The parameter storage local to the function is called the "formal parameter" such as the "num" in "static int Twice(int num)".

Parameters are passed "by value" that means there is a single copying assignment operation (=) from each actual parameter to set each formal parameter. The actual parameter is evaluated in the caller's context, and then the value is copied into the function's formal parameter just before the function begins executing. The alternative parameter mechanism is "by reference" which C does not implement directly, but which the programmer can implement manually when needed (see below). When a parameter is a struct, it is copied.

The variables local to `Twice()`, `num` and `result`, only exist temporarily while `Twice()` is executing. This is the standard definition for "local" storage for functions.

The `return` at the end of `Twice()` computes the return value and exits the function. Execution resumes with the caller. There can be multiple return statements within a function, but it's good style to at least have one at the end if a return value needs to be specified. Forgetting to account of a `return` somewhere in the middle of a function is a traditional source of bugs.

C-ing and Nothingness -- void

`void` is a type formalized in ANSI C which means "nothing". To indicate that a function does not return anything, use `void` as the return type. Also, by convention, a pointer which does not point to any particular type is declared as `void*`. Sometimes `void*` is used to force two bodies of code to not depend on each other where `void*` translates roughly to "this points to something, but I'm not telling you (the client) the type of the pointee exactly because you do not really need to know." If a function does not take any parameters, its parameter list is empty, or it can contain the keyword `void` but that style is now out of favor.

```
void TakesAnIntAndReturnsNothing(int anInt);

int TakesNothingAndReturnsAnInt();
int TakesNothingAndReturnsAnInt(void); // equivalent syntax for above
```

Call by Value vs. Call by Reference

C passes parameters "by value" which means that the actual parameter values are copied into local storage. The caller and callee functions do not share any memory -- they each have their own copy. This scheme is fine for many purposes, but it has two disadvantages.

1) Because the callee has its own copy, modifications to that memory are not communicated back to the caller. Therefore, value parameters do not allow the callee to communicate back to the caller. The function's return value can communicate some information back to the caller, but not all problems can be solved with the single return value.

2) Sometimes it is undesirable to copy the value from the caller to the callee because the value is large and so copying it is expensive, or because at a conceptual level copying the value is undesirable.

The alternative is to pass the arguments "by reference". Instead of passing a copy of a value from the caller to the callee, pass a pointer to the value. In this way there is only one copy of the value at any time, and the caller and callee both access that one value through pointers.

Some languages support reference parameters automatically. C does not do this -- the programmer must implement reference parameters manually using the existing pointer constructs in the language.

Swap Example
The classic example of wanting to modify the caller's memory is a `swap()` function which exchanges two values. Because C uses call by value, the following version of Swap will not work...

```
void Swap(int x, int y) {        // NO does not work
   int temp;

   temp = x;
   x = y;      // these operations just change the local x,y,temp
   y = temp;   // -- nothing connects them back to the caller's a,b
}


// Some caller code which calls Swap()...
int a = 1;
int b = 2;
Swap(a, b);
```

`Swap()` does not affect the arguments `a` and `b` in the caller. The function above only operates on the copies of `a` and `b` local to `Swap()` itself. This is a good example of how "local" memory such as (x, y, temp) behaves -- it exists independent of everything else only while its owning function is running. When the owning function exits, its local memory disappears.

Reference Parameter Technique
To pass an object X as a reference parameter, the programmer must pass a pointer to X instead of X itself. The formal parameter will be a pointer to the value of interest. The caller will need to use & or other operators to compute the correct pointer actual parameter. The callee will need to dereference the pointer with * where appropriate to access the value of interest. Here is an example of a correct Swap() function.

```
static void Swap(int* x, int* y) {     // params are int* instead of int
   int temp;

   temp = *x;        // use * to follow the pointer back to the caller's memory
   *x = *y;
   *y = temp;
}
```

```
// Some caller code which calls Swap()...
int a = 1;
int b = 2;

Swap(&a, &b);
```

Things to notice...

- The formal parameters are int* instead of int.
- The caller uses & to compute pointers to its local memory (a,b).
- The callee uses * to dereference the formal parameter pointers back to get the caller's memory.

Since the operator & produces the address of a variable -- `&a` is a pointer to `a`. In `Swap()` itself, the formal parameters are declared to be pointers, and the values of interest (a,b) are accessed through them. There is no special relationship between the **names** used for the actual and formal parameters. The function call matches up the actual and formal parameters by their order -- the first actual parameter is assigned to the first formal parameter, and so on. I deliberately used different names (a,b vs x,y) to emphasize that the names do not matter.

const

The qualifier const can be added to the left of a variable or parameter type to declare that the code using the variable will not change the variable. As a practical matter, use of `const` is very sporadic in the C programming community. It does have one very handy use, which is to clarify the role of a parameter in a function prototype...

```
void foo(const struct fraction* fract);
```

In the foo() prototype, the `const` declares that foo() does not intend to change the struct fraction pointee which is passed to it. Since the fraction is passed by pointer, we could not know otherwise if foo() intended to change our memory or not. Using the `const`, foo() makes its intentions clear. Declaring this extra bit of information helps to clarify the role of the function to its implementor and caller.

Bigger Pointer Example

The following code is a large example of using reference parameters. There are several common features of C programs in this example...Reference parameters are used to allow the functions Swap() and IncrementAndSwap() to affect the memory of their callers. There's a tricky case inside of IncrementAndSwap() where it calls Swap() -- no additional use of & is necessary in this case since the parameters x, y inside InrementAndSwap() are already pointers to the values of interest. The names of the variables through the program(a, b, x, y, alice, bob) do not need to match up in any particular way for the parameters to work. The parameter mechanism only depends on the types of the parameters and their order in the parameter list -- not their names. Finally this is an example of what multiple functions look like in a file and how they are called from the main() function.

```
static void Swap(int* a, int* b) {
   int temp;
   temp = *a;
   *a = *b;
   *b = temp;
}

static void IncrementAndSwap(int* x, int* y) {
   (*x)++;
   (*y)++;
   Swap(x, y);       // don't need & here since a and b are already
                     // int*'s.
}


int main()
{
   int alice = 10;
   int bob = 20;

   Swap(&alice, &bob);
   // at this point alice==20 and bob==10

   IncrementAndSwap(&alice, &bob);
   // at this point alice==11 and bob==21

   return 0;
}
```

Section 5
Odds and Ends

main()
The execution of a C program begins with function named main(). All of the files and libraries for the C program are compiled together to build a single program file. That file must contain exactly one main() function which the operating system uses as the starting point for the program. Main() returns an int which, by convention, is 0 if the program completed successfully and non-zero if the program exited due to some error condition. This is just a convention which makes sense in shell oriented environments such as Unix or DOS.

Multiple Files
For a program of any size, it's convenient to separate the functions into several separate files. To allow the functions in separate files to cooperate, and yet allow the compiler to work on the files independently, C programs typically depend on two features...

Prototypes
A "prototype" for a function gives its name and arguments but not its body. In order for a caller, in any file, to use a function, the caller must have seen the prototype for that function. For example, here's what the prototypes would look like for `Twice()` and `Swap()`. The function body is absent and there's a semicolon (;) to terminate the prototype...

```
int Twice(int num);
void Swap(int* a, int* b);
```

In pre-ANSI C, the rules for prototypes where very sloppy -- callers were not required to see prototypes before calling functions, and as a result it was possible to get in situations where the compiler generated code which would crash horribly.

In ANSI C, I'll oversimplify a little to say that...

1) a function may be declared `static` in which case it can only be used in the same file where it is used below the point of its declaration. Static functions do not require a separate prototype so long as they are defined before or above where they are called which saves some work.

2) A non-static function needs a prototype. When the compiler compiles a function definition, it must have previously seen a prototype so that it can verify that the two are in agreement ("prototype before definition" rule). The prototype must also be seen by any client code which wants to call the function ("clients must see prototypes" rule).(The require-prototypes behavior is actually somewhat of a compiler option, but it's smart to leave it on.)

Preprocessor
The preprocessing step happens to the C source before it is fed to the compiler. The two most common preprocessor directives are #define and #include...

#define

The #define directive can be used to set up symbolic replacements in the source. As with all preprocessor operations, #define is extremely unintelligent -- it just does textual replacement without understanding. #define statements are used as a crude way of establishing symbolic constants.

```
#define MAX 100
#define SEVEN_WORDS that_symbol_expands_to_all_these_words
```

Later code can use the symbols MAX or SEVEN_WORDS which will be replaced by the text to the right of each symbol in its #define.

#include

The "#include" directive brings in text from different files during compilation. #include is a very unintelligent and unstructured -- it just pastes in the text from the given file and continues compiling. The #include directive is used in the .h/.c file convention below which is used to satisfy the various constraints necessary to get prototypes correct.

```
#include "foo.h"     // refers to a "user" foo.h file --
                     //    in the originating directory for the compile

#include <foo.h>     // refers to a "system" foo.h file --
                     //    in the compiler's directory somewhere
```

foo.h vs foo.c

The universally followed convention for C is that for a file named "foo.c" containing a bunch of functions...

- A separate file named `foo.h` will contain the prototypes for the functions in `foo.c` which clients may want to call. Functions in `foo.c` which are for "internal use only" and should never be called by clients should be declared `static`.

- Near the top of `foo.c` will be the following line which ensures that the function definitions in `foo.c` see the prototypes in `foo.h` which ensures the "prototype before definition" rule above.
  ```
  #include "foo.h"    // show the contents of "foo.h"
                      // to the compiler at this point
  ```

- Any `xxx.c` file which wishes to call a function defined in `foo.c` must include the following line to see the prototypes, ensuring the "clients must see prototypes" rule above.
  ```
  #include "foo.h"
  ```

#if
At compile time, there is some space of names defined by the #defines. The #if test can be used at compile-time to look at those symbols and turn on and off which lines the compiler uses. The following example depends on the value of the FOO #define symbol. If it is true, then the "aaa" lines (whatever they are) are compiled, and the "bbb" lines are ignored. If FOO were 0, then the reverse would be true.

```
#define FOO 1

...

#if FOO
   aaa
   aaa
#else
   bbb
   bbb
#endif
```

You can use `#if 0 ...#endif` to effectively comment out areas of code you don't want to compile, but which you want to keeep in the source file.

Multiple #includes -- #pragma once
There's a problem sometimes where a .h file is #included into a file more than one time resulting in compile errors. This can be a serious problem. Because of this, you want to avoid #including .h files in other .h files if at all possible. On the other hand, #including .h files in .c files is fine. If you are lucky, your compiler will support the #pragma once feature which automatically prevents a single file from being #included more than once in any one file. This largely solves multiple #include problems.

```
// foo.h
// The following line prevents problems in files which #include "foo.h"
#pragma once

<rest of foo.h ...>
```

Assert
Array out of bounds references are an extremely common form of C run-time error. You can use the assert() function to sprinkle your code with your own bounds checks. A few seconds putting in assert statements can save you hours of debugging.

Getting out all the bugs is the hardest and scariest part of writing a large piece of software. Assert statements are one of the easiest and most effective helpers for that difficult phase.

```
#include <assert.h>
#define MAX_INTS 100
{
   int ints[MAX_INTS];
   i = foo(<something complicated>);   // i should be in bounds,
                                       // but is it really?
   assert(i>=0);           // safety assertions
   assert(i<MAX_INTS);

   ints[i] = 0;
```

Depending on the options specified at compile time, the assert() expressions will be left in the code for testing, or may be ignored. For that reason, it is important to only put expressions in assert() tests which do not need to be evaluated for the proper functioning of the program...

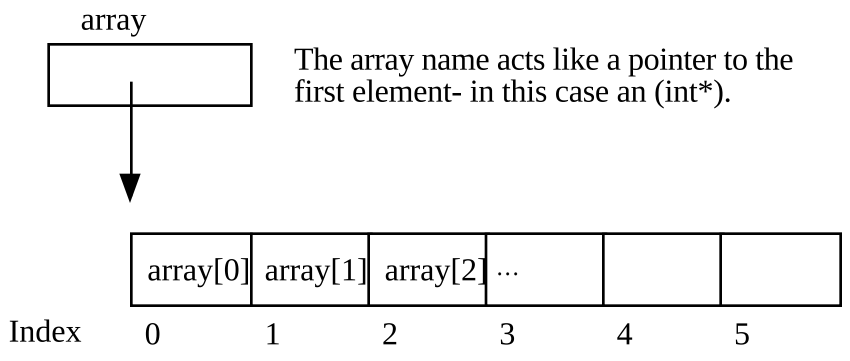
```
int errCode = foo();        // yes
assert(errCode == 0);


assert(foo() == 0);         // NO, foo() will not be called if
                            // the compiler removes the assert()
```

Section 6
Advanced Arrays and Pointers

Advanced C Arrays

In C, an array is formed by laying out all the elements contiguously in memory. The square bracket syntax can be used to refer to the elements in the array. The array as a whole is referred to by the address of the first element which is also known as the "base address" of the whole array.

```
{
   int array[6];

   int sum = 0;
   sum += array[0] + array[1];   // refer to elements using []
}
```



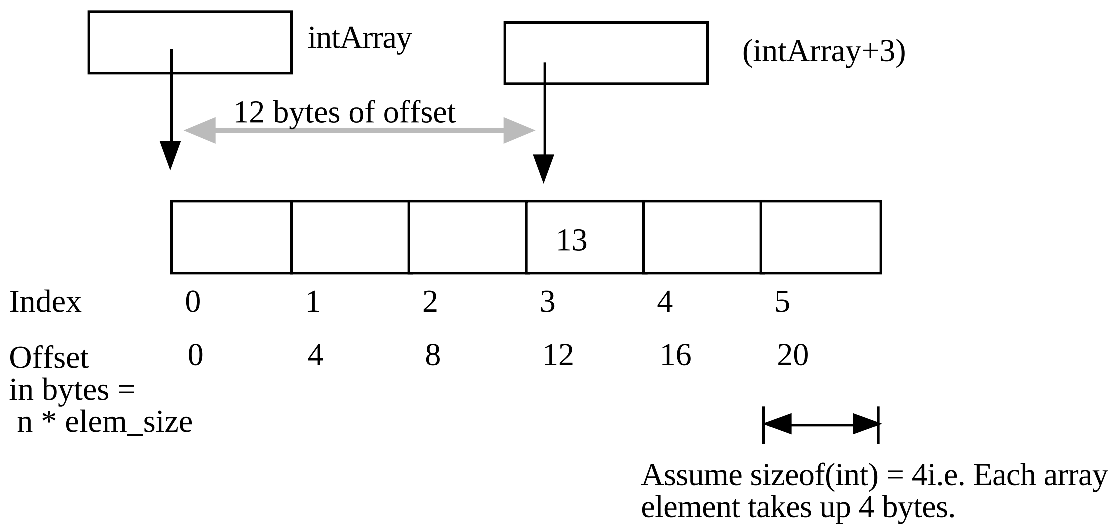
array
The array name acts like a pointer to the first element- in this case an (int*).
array[0]
array[1]
array[2]
...
Index
0
1
2
3
4
5

The programmer can refer to elements in the array with the simple [ ] syntax such as `array[1]`. This scheme works by combining the base address of the whole array with the index to compute the base address of the desired element in the array. It just requires a little arithmetic. Each element takes up a fixed number of bytes which is known at compile-time. So the address of element `n` in the array using 0 based indexing will be at an offset of (n * element_size) bytes from the base address of the whole array.

address of nth element = address_of_0th_element + (n * element_size_in_bytes)

The square bracket syntax [ ] deals with this address arithmetic for you, but it's useful to know what it's doing. The [ ] takes the integer index, multiplies by the element size, adds the resulting offset to the array base address, and finally dereferences the resulting pointer to get to the desired element.

```
{
   int intArray[6];

   intArray[3] = 13;
}
```



'+' Syntax
In a closely related piece of syntax, a `+` between a pointer and an integer does the same offset computation, but leaves the result as a pointer. The square bracket syntax gives the nth element while the `+` syntax gives a pointer to the nth element.

So the expression `(intArray + 3)` is a pointer to the integer `intArray[3]`. `(intArray + 3)` is of type `(int*)` while `intArray[3]` is of type `int`. The two expressions only differ by whether the pointer is dereferenced or not. So the expression `(intArray + 3)` is exactly equivalent to the expression `(&(intArray[3]))`. In fact those two probably compile to exactly the same code. They both represent a pointer to the element at index 3.

Any `[]` expression can be written with the `+` syntax instead. We just need to add in the pointer dereference. So `intArray[3]` is exactly equivalent to `*(intArray + 3)`. For most purposes, it's easiest and most readable to use the `[]` syntax. Every once in a while the + is convenient if you needed a pointer to the element instead of the element itself.

Pointer++ Style -- strcpy()
If `p` is a pointer to an element in an array, then `(p+1)` points to the next element in the array. Code can exploit this using the construct p++ to step a pointer over the elements in an array. It doesn't help readability any, so I can't recommend the technique, but you may see it in code written by others.

(This example was originally inspired by Mike Cleron) There's a library function called `strcpy(char* destination, char* source)` which copies the bytes of a C string from one place to another. Below are four different implementations of strcpy() written in order: from most verbose to most cryptic. In the first one, the normally straightforward while loop is actually sortof tricky to ensure that the terminating null character is copied over. The second removes that trickiness by moving assignment into the test. The last two are cute (and they demonstrate using ++ on pointers), but not really the sort of code you want to maintain. Among the four, I think strcpy2() is the best stylistically. With a smart compiler, all four will compile to basically the same code with the same efficiency.

```
// Unfortunately, a straight while or for loop won't work.
// The best we can do is use a while (1) with the test
// in the middle of the loop.
void strcpy1(char dest[], const char source[]) {
   int i = 0;

   while (1) {
      dest[i] = source[i];
      if (dest[i] == '\0') break;      // we're done
      i++;
   }
}


// Move the assignment into the test
void strcpy2(char dest[], const char source[]) {
   int i = 0;

   while ((dest[i] = source[i]) != '\0') {
      i++;
   }
}


// Get rid of i and just move the pointers.
// Relies on the precedence of * and ++.
void strcpy3(char dest[], const char source[])
{
   while ((*dest++ = *source++) != '\0') ;
}


// Rely on the fact that '\0' is equivalent to FALSE
void strcpy4(char dest[], const char source[])
{
   while (*dest++ = *source++) ;
}
```

Pointer Type Effects
Both [ ] and + implicitly use the compile time type of the pointer to compute the element_size which affects the offset arithmetic. When looking at code, it's easy to assume that everything is in the units of bytes.

```
int *p;

p = p + 12;    // at run-time, what does this add to p? 12?
```

The above code does not add the number 12 to the address in p-- that would increment p by 12 **bytes**. The code above increments p by 12 **ints**. Each int probably takes 4 bytes, so at run time the code will effectively increment the address in p by 48. The compiler figures all this out based on the type of the pointer.

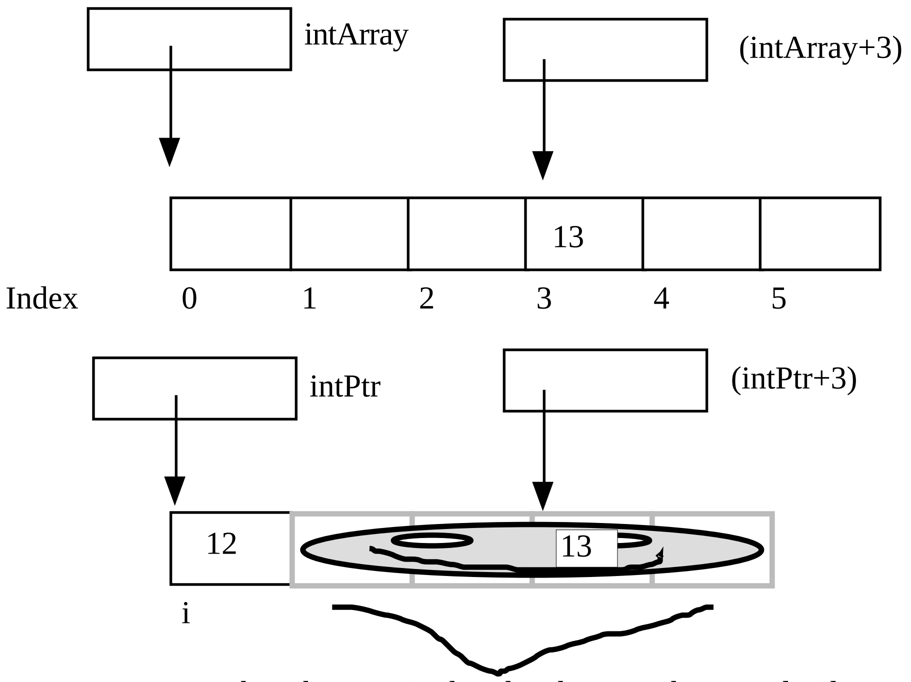
Using casts, the following code really does just add 12 to the address in the pointer p. It works by telling the compiler that the pointer points to `char` instead of `int`. The size of `char` is defined to be exactly 1 byte (or whatever the smallest addressable unit is on the computer). In other words, `sizeof(char)` is always 1. We then cast the resulting `(char*)` back to an `(int*)`. The programmer is allowed to cast any pointer type to any other pointer type like this to change the code the compiler generates.

```
   p = (int*) ( ((char*)p) + 12);
```

Arrays and Pointers

One effect of the C array scheme is that the compiler does not distinguish meaningfully between arrays and pointers-- they both just look like pointers. In the following example, the value of `intArray` is a pointer to the first element in the array so it's an `(int*)`. The value of the variable `intPtr` is also `(int*)` and it is set to point to a single integer `i`. So what's the difference between `intArray` and `intPtr`? Not much as far as the compiler is concerned. They are both just (`int*`) pointers, and the compiler is perfectly happy to apply the `[]` or `+` syntax to either. It's the programmer's responsibility to ensure that the elements referred to by a `[]` or `+` operation really are there. Really its' just the same old rule that C doesn't do any bounds checking. C thinks of the single integer `i` as just a sort of degenerate array of size 1.

```
   {
      int intArray[6];
      int *intPtr;
      int i;

      intPtr = &i;

      intArray[3] = 13;        // ok
      intPtr[0] = 12;          // odd, but ok. Changes i.
      intPtr[3] = 13;          // BAD! There is no integer reserved here!
   }
```



These bytes exist, but they have not been explicitly reserved. They are the bytes which happen to be adjacent to the memory for i. They are probably being used to store something already, such as a smashed looking smiley face. The 13 just gets blindly written over the smiley face. This error will only be apparent later when the program tries to read the smiley face data.

Array Names Are Const

One subtle distinction between an array and a pointer, is that the pointer which represents the base address of an array cannot be changed in the code. The array base address behaves like a `const` pointer. The constraint applies to the name of the array where it is declared in the code-- the variable `ints` in the example below.

```
{
   int ints[100]
   int *p;
   int i;

   ints = NULL;            // NO, cannot change the base addr ptr
   ints = &i;              // NO
   ints = ints + 1;        // NO
   ints++;                 // NO

   p = ints;         // OK, p is a regular pointer which can be changed
                     // here it is getting a copy of the ints pointer

   p++;              // OK, p can still be changed (and ints cannot)
   p = NULL;         // OK
   p = &i;           // OK

   foo(ints);        // OK (possible foo definitions are below)
}
```

Array parameters are passed as pointers. The following two definitions of foo look different, but to the compiler they mean exactly the same thing. It's preferable to use whichever syntax is more accurate for readability. If the pointer coming in really is the base address of a whole array, then use [ ].

```
void foo(int arrayParam[]) {
   arrayParam = NULL;      // Silly but valid. Just changes the local pointer
}

void foo(int *arrayParam) {
   arrayParam = NULL;      // ditto
}
```

Heap Memory

C gives programmers the standard sort of facilities to allocate and deallocate dynamic heap memory. A word of warning: writing programs which manage their heap memory is notoriously difficult. This partly explains the great popularity of languages such as Java and Perl which handle heap management automatically. These languages take over a task which has proven to be extremely difficult for the programmer. As a result Perl and Java programs run a little more slowly, but they contain far fewer bugs. (For a detailed discussion of heap memory see http://cslibrary.stanford.edu/102/, Pointers and Memory.)

C provides access to the heap features through library functions which any C code can call. The prototypes for these functions are in the file <stdlib.h>, so any code which wants to call these must #include that header file. The three functions of interest are...

`void* malloc(size_t size)` Request a contiguous block of memory of the given size in the heap. malloc() returns a pointer to the heap block or NULL if the request could not be satisfied. The type `size_t` is essentially an `unsigned long` which indicates how large a block the caller would like measured in bytes. Because the block pointer returned by malloc() is a void* (i.e. it makes no claim about the type of its pointee), a cast will probably be required when storing the void* pointer into a regular typed pointer.

`void free(void* block)` The mirror image of malloc() -- free takes a pointer to a heap block earlier allocated by malloc() and returns that block to the heap for re-use. After the free(), the client should not access any part of the block or assume that the block is valid memory. The block should not be freed a second time.

`void* realloc(void* block, size_t size);` Take an existing heap block and try to relocate it to a heap block of the given size which may be larger or smaller than the original size of the block. Returns a pointer to the new block, or NULL if the relocation was unsuccessful. Remember to catch and examine the return value of realloc() -- it is a common error to continue to use the old block pointer. Realloc() takes care of moving the bytes from the old block to the new block. Realloc() exists because it can be implemented using low-level features which make it more efficient than C code the client could write.

Memory Management

All of a program's memory is deallocated automatically when the it exits, so a program only needs to use free() during execution if it is important for the program to recycle its memory while it runs -- typically because it uses a lot of memory or because it runs for a long time. The pointer passed to free() must be exactly the pointer which was originally returned by malloc() or realloc(), not just a pointer into somewhere within the heap block.

Dynamic Arrays

Since arrays are just contiguous areas of bytes, you can allocate your own arrays in the heap using malloc(). The following code allocates two arrays of 1000 ints-- one in the stack the usual "local" way, and one in the heap using malloc(). Other than the different allocations, the two are syntactically similar in use.

```
{
   int a[1000];

   int *b;
   b = (int*) malloc( sizeof(int) * 1000);
   assert(b != NULL);      // check that the allocation succeeded

   a[123] = 13;      // Just use good ol' [] to access elements
   b[123] = 13;      // in both arrays.

   free(b);
}
```

Although both arrays can be accessed with [ ], the rules for their maintenance are very different....

Advantages of being in the heap

• Size (in this case 1000) can be defined at run time. Not so for an array like "a".

• The array will exist until it is explicitly deallocated with a call to free().

• You can change the size of the array at will at run time using realloc(). The following changes the size of the array to 2000. Realloc() takes care of copying over the old elements.

```
   ...
   b = realloc(b, sizeof(int) * 2000);
   assert(b != NULL);
```

Disadvantages of being in the heap

• You have to remember to allocate the array, and you have to get it right.

• You have to remember to deallocate it exactly once when you are done with it, and you have to get that right.

• The above two disadvantages have the same basic profile: if you get them wrong, your code still looks right. It compiles fine. It even runs for small cases, but for some input cases it just crashes unexpectedly because random memory is getting overwritten somewhere like the smiley face. This sort of "random memory smasher" bug can be a real ordeal to track down.

Dynamic Strings

The dynamic allocation of arrays works very well for allocating strings in the heap. The advantage of heap allocating a string is that the heap block can be just big enough to store the actual number of characters in the string. The common local variable technique such as `char string[1000];` allocates way too much space most of the time, wasting the unused bytes, and yet fails if the string ever gets bigger than the variable's fixed size.

```
#include <string.h>

/*
 Takes a c string as input, and makes a copy of that string
 in the heap. The caller takes over ownership of the new string
 and is responsible for freeing it.
*/
char* MakeStringInHeap(const char* source) {
   char* newString;

   newString = (char*) malloc(strlen(source) + 1); // +1 for the '\0'
   assert(newString != NULL);
   strcpy(newString, source);
   return(newString);
}
```

Section 7
Details and Library Functions

Precedence and Associativity

```
function-call() [] -> .                             L to R

! ~ ++ -- + - *(ptr deref) sizeof &(addr of)        R to L
(all unary ops are the same)

* / %                                               L to R
(the top tier arithmetic binary ops)

+ -                                                 L to R
(second tier arithmetic binary ops)

< <= > >=                                           L to R

== !=                                               L to R

in order: & ^ | && ||                               L to R
(note that bitwise comes before boolean)

= and all its variants                              R to L

, (comma)  .                                        L to R

A combinations which never works right without parens: *structptr.field
You have to write it as (*structptr).field or structptr->field
```

Standard Library Functions

Many basic housekeeping funcions are available to a C program in form of standard library functions. To call these, a program must #include the appropriate .h file. Most compilers link in the standard library code by default. The functions listed in the next section are the most commonly used ones, but there are many more which are not listed here.

stdio.h	file input and output
ctype.h	character tests
string.h	string operations
math.h	mathematical functions such as sin() and cos()
stdlib.h	utility functions such as malloc() and rand()
assert.h	the assert() debugging macro
stdarg.h	support for functions with variable numbers of arguments
setjmp.h	support for non-local flow control jumps
signal.h	support for exceptional condition signals
time.h	date and time

limits.h, float.h constants which define type range values such as INT_MAX

stdio.h
Stdio.h is a very common file to #include -- it includes functions to print and read strings from files and to open and close files in the file system.

`FILE* fopen(const char* fname, const char* mode);`
Open a file named in the filesystem and return a FILE* for it. Mode = "r" read,"w" write,"a"append, returns NULL on error. The standard files `stdout`, `stdin`, `stderr` are automatically opened and closed for you by the system.

`int fclose(FILE* file);`
Close a previously opened file. Returns EOF on error. The operating system closes all of a program's files when it exits, but it's tidy to do it beforehand. Also, there is typically a limit to the number of files which a program may have open simultaneously.

`int fgetc(FILE* in);`
Read and return the next unsigned char out of a file, or EOF if the file has been exhausted. (detail) This and other file functions return ints instead of a chars because the EOF constant they potentially is not a `char`, but is an `int`. getc() is an alternate, faster version implemented as a macro which may evaluate the FILE* expression more than once.

`char* fgets(char* dest, int n, FILE* in)`
Reads the next line of text into a string supplied by the caller. Reads at most n-1 characters from the file, stopping at the first '\n' character. In any case, the string is '\0' terminated. The '\n' is included in the string. Returns NULL on EOF or error.

`int fputc(int ch, FILE* out);`
Write the char to the file as an unsigned char. Returns ch, or EOF on err. putc() is an alternate, faster version implemented as a macro which may evaluate the FILE* expression more than once.

`int ungetc(int ch, FILE* in);`
Push the most recent fgetc() char back onto the file. EOF may not be pushed back. Returns ch or EOF on error.

`int printf(const char* format_string, ...);`
Prints a string with values possibly inserted into it to standard output. Takes a variable number of arguments -- first a format string followed by a number of matching arguments. The format string contains text mixed with % directives which mark things to be inserted in the output. %d = int, %Ld=long int, %s=string, %f=double, %c=char. Every % directive must have a matching argument of the correct type after the format string. Returns the number of characters written, or negative on error. If the percent directives do not match the number and type of arguments, printf() tends to crash or otherwise do the wrong thing at run time. fprintf() is a variant which takes an additional FILE* argument which specifies the file to print to. Examples...

```
printf("hello\n");                              prints:  hello
printf("hello %d there %d\n", 13, 1+1);         prints:  hello 13 there 2
printf("hello %c there %d %s\n", 'A', 42, "ok");    prints:  hello A there 42 ok
```

`int scanf(const char* format, ...)`

Opposite of printf() -- reads characters from standard input trying to match elements in the format string. Each percent directive in the format string must have a matching pointer in the argument list which scanf() uses to store the values it finds. scanf() skips whitespace as it tries to read in each percent directive. Returns the number of percent directives processed successfully, or EOF on error. scanf() is famously sensitive to programmer errors. If scanf() is called with anything but the correct pointers after the format string, it tends to crash or otherwise do the wrong thing at run time. sscanf() is a variant which takes an additional initial string from which it does its reading. fscanf() is a variant which takes an additional initial FILE* from which it does its reading. Example...

```
{
   int num;
   char s1[1000];
   char s2[1000];

   scanf("hello %d %s %s", &num, s1, s2);
}
```

Looks for the word "hello" followed by a number and two words (all separated by whitespace). scanf() uses the pointers `&num`, `s1`, and `s2` to store what it finds into the local variables.

ctype.h

ctype.h includes macros for doing simple tests and operations on characters

`isalpha(ch)` // ch is an upper or lower case letter

`islower(ch), isupper(ch)` // same as above, but upper/lower specific

`isspace(ch)` // ch is a whitepace character such as tab, space, newline, etc.

`isdigit(ch)` // digit such as '0'..'9'

`toupper(ch), tolower(ch)` // Return the lower or upper case version of a alphabetic character, otherwise pass it through unchanged.

string.h
None of these string routines allocate memory or check that the passed in memory is the right size. The caller is responsible for making sure there is "enough" memory for the operation. The type `size_t` is an unsigned integer wide enough for the computer's address space -- most likely an `unsigned long`.

```
size_t strlen(const char* string);
```
Return the number of chars in a C string. EG strlen("abc")==3

```
char* strcpy(char* dest, const char* source);
```
Copy the characters from the source string to the destination string.

```
size_t strlcpy(char* dest, const char* source,
                 size_t dest_size);
```
Like strcpy(), but knows the size of the dest. Truncates if necessary. Use this to avoid memory errors and buffer-overflow security problems. This function is not as standard as strcpy(), but most sytems have it. Do not use the old strncpy() function -- it is difficult to use correctly.

```
char *strcat(char* dest, const char* source);
```
Append the characters from the source string to the end of destination string. (There is a non-standard strlcat() variant that takes the size of the dest as third argument.)

```
int strcmp(const char* a, const char* b);
```
Compare two strings and return an int which encodes their ordering. zero:a==b, negative:a<b, positive:a>b. It is a common error to think of the result of strcmp() as being boolean true if the strings are equal which is, unfortunately, exactly backwards.

```
char* strchr(const char* searchIn, char ch);
```
Search the given string for the first occurence of the given character. Returns a pointer to the character, or NULL if none is found.

```
char* strstr(const char* searchIn, const char* searchFor);
```
Similar to strchr(), but searches for an entire string instead of a single character. The search is case sensitive.

```
void* memcpy(void* dest, const void* source, size_t n);
```
Copy the given number of bytes from the source to the destination. The source and destination must not overlap. This may be implemented in a specialized but highly optimized way for a particular computer.

```
void* memmove(void* dest, const void* source, size_t n);
```
Similar to memcpy() but allows the areas to overlap. This probably runs slightly slower than memcpy().

stdlib.h

`int rand();`

Returns a pseudo random integer in the range 0..RAND_MAX (limits.h) which is at least 32767.

`void srand(unsigned int seed);`

The sequence of random numbers returned by rand() is initially controlled by a global "seed" variable. srand() sets this seed which, by default, starts with the value 1. Pass the expression `time(NULL)` (time.h) to set the seed to a value based on the current time to ensure that the random sequence is different from one run to the next.

`void* malloc(size_t size);`

Allocate a heap block of the given size in bytes. Returns a pointer to the block or NULL on failure. A cast may be required to store the void* pointer into a regular typed pointer. [ed: see the Heap Allocation section above for the longer discussion of malloc(), free(), and realloc()]

`void free(void* block);`

Opposite of malloc(). Returns a previous malloc block to the system for reuse

`void* realloc(void* block, size_t size);`

Resize an existing heap block to the new size. Takes care of copying bytes from the old block to the new. Returns the new base address of the heap block. It is a common error to forget to catch the return value from realloc(). Returns NULL if the resize operation was not possible.

`void exit(int status);`

Halt and exit the program and pass a condition int back to the operating sytem. Pass 0 to signal normal program termination, non-zero otherwise.

```
void* bsearch(const void* key, const void* base, size_t len,
     size_t elem_size, <compare_function>);
```

Do a binary search in an array of elements. The last argument is a function which takes pointers to the two elements to compare. Its prototype should be:
int compare(const void* a, const void* b);, and it should return 0, -1, or 1 as strcmp() does. Returns a pointer to a found element, or NULL otherwise. Note that strcmp() itself cannot be used directly as a compare function for bsearch() on an array of char* strings because strcmp() takes char* arguments and bsearch() will need a comparator that takes pointers to the array elements -- char**.

```
void qsort(void* base, size_t len, size_t elem_size,
     <compare_function>);
```

Sort an array of elements. Takes a function pointer just like besearch().

Revision History

11/1998 -- original major version. Based on my old C handout for CS107. Thanks to Jon Becker for proofreading and Mike Cleron for the original inspiration.

Revised 4/2003 with many helpful typo and other suggestions from Negar Shamma and A. P. Garcia

First-Order Homogeneous Equations

A function *f*(*x,y*) is said to be **homogeneous of degree *n*** if the equation

$$f(zx, zy) = z^n f(x, y)$$

holds for all *x,y*, and *z* (for which both sides are defined).

Example 1: The function *f*(*x,y*) = $x^2 + y^2$ is homogeneous of degree 2, since

$$f(zx, zy) = (zx)^2 + (zy)^2 = z^2(x^2 + y^2) = z^2 f(x, y)$$

Example 2: The function $f(x, y) = \sqrt{x^8 - 3x^2y^6}$ is homogeneous of degree 4, since

$$\begin{aligned} f(zx, zy) &= \sqrt{(zx)^8 - 3(zx)^2(zy)^6} = \sqrt{z^8(x^8 - 3x^2y^6)} \\ &= \sqrt{z^8}\,\sqrt{x^8 - 3x^2y^6} = z^4 f(x, y) \end{aligned}$$

Example 3: The function *f*(*x,y*) = 2 *x* + *y* is homogeneous of degree 1, since

$$f(zx, zy) = 2(zx) + (zy) = z(2x + y) = z^1 f(x, y)$$

Example 4: The function *f*(*x,y*) = $x^3 - y^2$ is not homogeneous, since

$$f(zx, zy) = (zx)^3 - (zy)^2 = z^3x^3 - z^2y^2$$

which does not equal $z^n f(x,y)$ for any n.

Example 5: The function $f(x,y) = x^3 \sin(y/x)$ is homogeneous of degree 3, since

$$f(zx, zy) = (zx)^3 \sin \frac{zy}{zx} = z^3\left(x^3 \sin \frac{y}{x}\right) = z^3 f(x, y)$$

A first-order differential equation $M(x,y)\,dx + N(x,y)\,dy = 0$ is said to be **homogeneous** if $M(x,y)$ and $N(x,y)$ are both homogeneous functions of the same degree.

Example 6: The differential equation

$$(x^2 - y^2)\,dx + xy\,dy = 0$$

is homogeneous because both $M(x,y) = x^2 - y^2$ and $N(x,y) = xy$ are homogeneous functions of the same degree (namely, 2).

The method for solving homogeneous equations follows from this fact:

The substitution $y = xu$ (and therefore $dy = xdu + udx$) transforms a homogeneous equation into a separable one.

Example 7: Solve the equation $(x^2 - y^2)\,dx + xy\,dy = 0$.

This equation is homogeneous, as observed in Example 6. Thus to solve it, make the substitutions $y = xu$ and $dy = x\,dy + u\,dx$:

$$\begin{aligned}
[x^2 - (xv)^2]\,dx + [x(xv)](x\,dv + v\,dx) &= 0 \\
(x^2 - x^2v^2)\,dx + x^3v\,dv + x^2v^2\,dx &= 0 \\
x^2dx + x^3v\,dv &= 0 \\
dx + xv\,dv &= 0
\end{aligned}$$

This final equation is now separable (which was the intention). Proceeding with the solution,

$$v\,dv = -\frac{dx}{x}$$

$$\int v\,dv = \int -\frac{dx}{x}$$

$$\tfrac{1}{2}v^2 = -\ln|x| + c'$$

Therefore, the solution of the separable equation involving *x* and *v* can be written

$$\tfrac{1}{2}v^2 = \ln\left|\frac{c}{x}\right|$$

To give the solution of the original differential equation (which involved the variables *x* and *y*), simply note that

$$y = xv \Rightarrow v = \frac{y}{x}$$

Replacing *v* by *y*/ *x* in the preceding solution gives the final result:

$$\frac{1}{2}\left(\frac{y}{x}\right)^2 = \ln\left|\frac{c}{x}\right| \Rightarrow y^2 = 2x^2\ln\left|\frac{c}{x}\right|$$

This is the general solution of the original differential equation.

Example 8: Solve the IVP

$$\begin{aligned} 2(x+2y)\,dx + (y-x)\,dy &= 0 \\ y(1) &= 0 \end{aligned}$$

Since the functions

$$M(x,y) = 2(x+2y) \quad \text{and} \quad N(x,y) = y - x$$

are both homogeneous of degree 1, the differential equation is homogeneous. The substitutions *y* = *xv* and *dy* = *x dv* + *v dx* transform the equation into

$$2(x+2xv)\,dx + (xv - x)(x\,dv + v\,dx) = 0$$

which simplifies as follows:

$$\begin{aligned} 2x\,dx + 4xv\,dx + x^2v\,dv - x^2\,dv + xv^2\,dx - xv\,dx &= 0 \\ (2x + 3xv + xv^2)\,dx + (x^2v - x^2)\,dv &= 0 \\ x(2 + 3v + v^2)\,dx + x^2(v-1)\,dv &= 0 \\ (2 + 3v + v^2)\,dx + x(v-1)\,dv &= 0 \end{aligned}$$

The equation is now separable. Separating the variables and integrating gives

$$\begin{aligned} x(v-1)\,dv &= -(2 + 3v + v^2)\,dx \\ \frac{v-1}{v^2+3v+2}\,dv &= -\frac{dx}{x} \qquad (\dagger) \end{aligned}$$

The integral of the left-hand side is evaluated after performing a partial fraction decomposition:

$$\frac{v-1}{v^2+3v+2} = \frac{v-1}{(v+1)(v+2)} = \frac{-2}{v+1} + \frac{3}{v+2}$$

Therefore,

$$\begin{aligned}\int \frac{v-1}{v^2+3v+2}\,dv &= \int \left(\frac{-2}{v+1} + \frac{3}{v+2}\right) dv \\ &= -2\ln|v+1| + 3\ln|v+2| \\ &= \ln|(v+1)^{-2}(v+2)^3|\end{aligned}$$

The right-hand side of (†) immediately integrates to

$$\int -\frac{dx}{x} = -\ln|x| + c' = \ln|cx^{-1}|$$

Therefore, the solution to the separable differential equation (†) is

$$(v+1)^{-2}(v+2)^3 = cx^{-1}$$

Now, replacing *v* by *y*/ *x* gives

$$\left(\frac{y}{x}+1\right)^{-2}\left(\frac{y}{x}+2\right)^{3}=cx^{-1}$$

as the general solution of the given differential equation. Applying the initial condition *y*(1) = 0 determines the value of the constant *c*:

$$\left(\frac{0}{1}+1\right)^{-2}\left(\frac{0}{1}+2\right)^{3}=c\cdot 1^{-1}\Rightarrow 8=c$$

Thus, the particular solution of the IVP is

$$\left(\frac{y}{x}+1\right)^{-2}\left(\frac{y}{x}+2\right)^{3}=8x^{-1}$$

which can be simplified to

$$(2x+y)^{3}=8(x+y)^{2}$$

as you can check.

Technical note: In the separation step (†), both sides were divided by (*v* + 1)(*v* + 2), and *v* = –1 and *v* = –2 were lost as solutions. These need not be considered, however, because even though the equivalent functions *y* = – *x* and *y* = –2 *x* do indeed satisfy the given differential equation, they are inconsistent with the initial condition.

Formation of a Differential Equation

In this page we are going to discuss about **formation of a differential equation** concept. Below you will get explanation about formation of a differential equation.

Consider the family of lines represented by y = mx ….(1)

This equation represents infinite number of lines passing through the origin.

Differentiating (1), we get

$$\frac{dy}{dx} = m$$

Substituting this value of m, we get the differential equation.

$$y = \frac{dy}{dx}x \qquad \ldots.(2)$$

Consider the family of lines represented by

y = mx + c ….(3)

where m and c are arbitrary constants. Any line on the co-ordinate plane can be represented by (3)

Let us form a differential equation for equation (3)

Differentiating equation (3) , we have

$$\frac{dy}{dx} = m$$

Differentiating again, we have

$$\frac{d^2y}{dx^2} = 0 \qquad \ldots.(4)$$

This is the differential equation which represents the family of straight lines y = mx +c.

The equation y = mx has one arbitrary constant and its differential equation is of order 1. The equation y = mx + c has two arbitrary constants and its differential equation is of order 2.

In general, if an equation contains n arbitrary constants, then we obtain its differential equation which is of order n, after eliminate all the n constants.

Equation(1) is called the primitive of Differentiating equation(2).

Equation (3) is called the primitive of Differentiating equation (4).

The formation of a differential equation may be done by differentiating and eliminating arbitrary constants from the given equation.

The given equation is differentiated as many times as there are arbitrary constants.

Family of Curves

In this section we explained with examples how to form differential equation that represent family of curves .

Suppose a family of curves depending on one constant is given by

F_1 : f (x, y, a)= 0 ….(1)

where a $\in$ R is the parameter

Differentiating (1) with respect to x, we have

g (x, y, y', a) = 0 ….(2)

Now eliminating 'a' from equation (1) and equation (2), we get the required differential equation.

f (x, y, y') = 0 ….(3)

This equation represents the family of curves F_1. Equation (1) is called the primitive of the differential equation (3).

In this section, we discuss in general how to form Differential Equation which represent a family of curves.

Let

F_2: f (x, y, a, b) = 0 ….(4)

where a, b$\in$ R

represents a family of curves which depend on two constants (parameters) a, b.

Differentiating (4) with respect to x, we have

g (x, y, y', a, b) = 0 ….(5)

We can not eliminate a and b from equation (4) and equation (5). Therefore we need another equation which can be obtained by differentiation equation (5).

Differentiating equation (5) with respect to x, we have

h(x, y, y ',y'', a, b) = 0 ….(6)

Now the arbitrary constants can be eliminated from equation (4), (5) and (6), to obtain the differential equation of the family of curves.

As discussed earlier, the family of curves containing one parameter, is represented by a differential equation of order 1.

The family of curves which depend on two parameter is represented by differential equation of order 2.

In general, the family of curves which depend on n parameter is represented by differential equation of order 3.

Formation of a Differential Equation Examples

Below are the examples on formation of a differential equation:

Example 1:

Form a differential equations by eliminating 'a' from the family of curves $y^2=4ax$.

Solution:

$y^2 = 4ax$ …(1)

Differentiating with respect to x

$$2y\frac{dy}{dx} = 4a$$

Substitute for 4a in (1), we get

$$y^2 = 2y.\frac{dy}{dx}.x$$

$$y = 2x\frac{dy}{dx}$$

$y - 2xy' = 0$

Note that the given equation is differentiated only once to obtain the differential equation since it has only one constant.

Example 2:

Form a differential equation by eliminating the parameter A and B from the family of curves given by $y = Ae^{2x}+Be^{-2x}$.

Solution:

The given equation has two arbitrary constants.

To obtain the differential equation, we differentiate the given equation twice.

Differentiate with respect to x.

$$y_1 = 2Ae^{2x} - 2Be^{-2x}$$

$$y_2 = 4Ae^{2x} + 4Be^{-2x}$$

$$= 4\,[Ae^{2x}+Be^{-2x}]$$

$$\left[\begin{array}{l} \text{Notation:} \\ y_1 = \text{First derivative of } y = \dfrac{dy}{dx} \\ y_2 = \text{Second derivative of } y = \dfrac{d^2y}{dx^2} \end{array}\right]$$

$y_2 = 4y$

$y_2 - 4y = 0$ which is the required differential equation.

Formation of a differential equation whose general solution is given

We know that the equation

$$x^2 + y^2 + 2x - 4y + 4 = 0 \qquad \dots (1)$$

represents a circle having centre at (– 1, 2) and radius 1 unit.

Differentiating equation (1) with respect to x, we get

$$\frac{dy}{dx} = \frac{x+1}{2-y} \quad (y \neq 2) \qquad \dots (2)$$

which is a differential equation. You will find later on that this equation represents the family of circles and one member of the family is the circle given in equation (1).

Let us consider the equation

$$x^2 + y^2 = r^2 \qquad \dots (3)$$

By giving different values to r, we get different members of the family e.g. x2 + y2 = 1, x2 + y2 = 4, x2 + y2 = 9 etc. (see Fig 9.1).

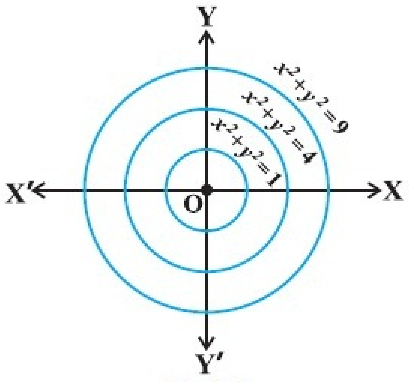


Fig 9.1

Thus, equation (3) represents a family of concentric circles centred at the origin and having different radii.

We are interested in finding a differential equation that is satisfied by each member of the family. The differential equation must be free from r because r is different for different members of the family. This equation is obtained by differentiating equation (3) with respect to x, i.e.,

$$2x + 2y\frac{dy}{dx} = 0 \quad \text{or} \quad x + y\frac{dy}{dx} = 0 \qquad \dots (4)$$

which represents the family of concentric circles given by equation (3).

Again, let us consider the equation

$$y = mx + c \qquad \dots (5)$$

By giving different values to the parameters m and c, we get different members of the family, e.g.,

$$y = x \qquad (m = 1, \quad c = 0)$$

$$y = \sqrt{3}\,x \qquad (m = \sqrt{3}, \ c = 0)$$

$$y = x + 1 \qquad (m = 1, \ c = 1)$$

$$y = -x \qquad (m = -1, \ c = 0)$$

$$y = -x - 1 \qquad (m = -1, \ c = -1) \text{ etc.}$$

(see Fig 9.2). Thus, equation (5) represents the family of straight lines, where m, c are parameters.

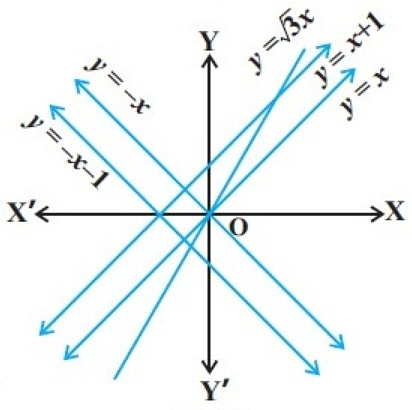


Fig 9.2

We are now interested in finding a differential equation that is satisfied by each member of the family. Further, the equation must be free from m and c because m and c are different for different members of the family. This is obtained by differentiating equation (5) with respect to x, successively we get

$$\frac{dy}{dx} = m \text{, and } \frac{d^2y}{dx^2} = 0 \qquad \dots (6)$$

The equation (6) represents the family of straight lines given by equation (5).
Note that equations (3) and (5) are the general solutions of equations (4) and (6) respectively.

Procedure to form a differential equation that will represent a given family of curves

(a) If the given family F1 of curves depends on only one parameter then it is represented by an equation of the form

$$F_1(x, y, a) = 0 \qquad \dots (1)$$

For example, the family of parabolas y2 = ax can be represented by an equation of the form f (x, y, a) : y2 = ax.
Differentiating equation (1) with respect to x, we get an equation involving y′, y, x, and a, i.e.,

$$g(x, y, y', a) = 0 \qquad \dots (2)$$

The required differential equation is then obtained by eliminating a from equations (1) and (2) as

$$F(x, y, y') = 0 \qquad \dots (3)$$

(b) If the given family F2 of curves depends on the parameters a, b (say) then it is represented by an equation of the from

$$F_2(x, y, a, b) = 0 \qquad \dots (4)$$

Differentiating equation (4) with respect to x, we get an equation involving y′, x, y, a, b, i.e.,

$$g(x, y, y', a, b) = 0 \qquad \dots (5)$$

But it is not possible to eliminate two parameters a and b from the two equations and so, we need a third equation. This equation is obtained by differentiating equation (5), with respect to x, to obtain a relation of the form

$$h(x, y, y', y'', a, b) = 0 \qquad \dots (6)$$

The required differential equation is then obtained by eliminating a and b from equations (4), (5) and (6) as

$$F(x, y, y', y'') = 0 \qquad \dots (7)$$

NOTE: The order of a differential equation representing a family of curves is same as the number of arbitrary constants present in the equation corresponding to the family of curves.

Example Form the differential equation representing the family of curves y = mx, where, m is arbitrary constant.
Solution We have

$$y = mx \qquad \dots (1)$$

Differentiating both sides of equation (1) with respect to x, we get

$$\frac{dy}{dx} = m$$

Substituting the value of m in

$$y = \frac{dy}{dx} \cdot x$$

equation (1) we get

or

$$x \frac{dy}{dx} - y = 0$$

which is free from the parameter m and hence this is the required differential equation.

Example Form the differential equation representing the family of curves y = a sin (x + b), where a, b are arbitrary constants.
Solution We have

$$y = a \sin(x + b) \qquad \ldots (1)$$

Differentiating both sides of equation (1) with respect to x, successively we get

$$\frac{dy}{dx} = a \cos(x + b) \qquad \ldots (2)$$

$$\frac{d^2y}{dx^2} = -a \sin(x + b) \qquad \ldots (3)$$

Eliminating a and b from equations (1), (2) and (3), we get

$$\frac{d^2y}{dx^2} + y = 0 \qquad \ldots (4)$$

which is free from the arbitrary constants a and b and hence this the required differential equation.

Graph Theory and Applications

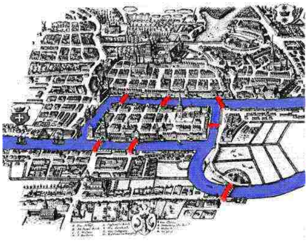
Paul Van Dooren
Université catholique de Louvain
Louvain-la-Neuve, Belgium

Large Graphs and Networks
UCL Université catholique de Louvain

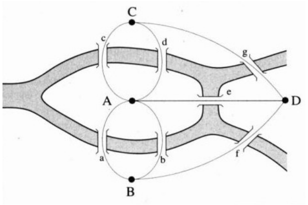
Dublin, August 2009

Inspired from the course notes of V. Blondel and L. Wolsey (UCL)

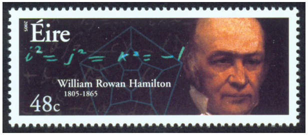
Graph theory started with Euler who was asked to find a nice path across the seven Köningsberg bridges



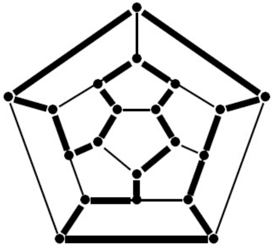
The (Eulerian) path should cross over each of the seven bridges exactly once



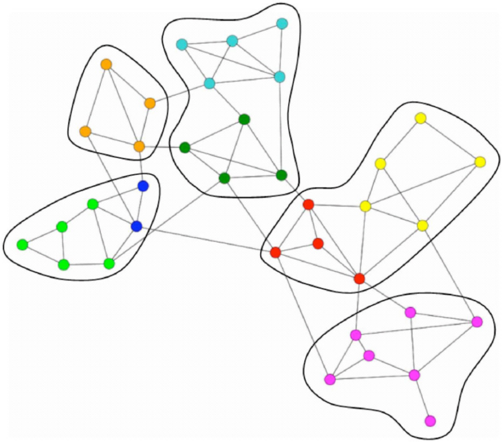
Another early bird was Sir William Rowan Hamilton (1805-1865)



In 1859 he developed a toy based on finding a path visiting all cities in a graph exactly once and sold it to a toy maker in Dublin. It never was a big success.

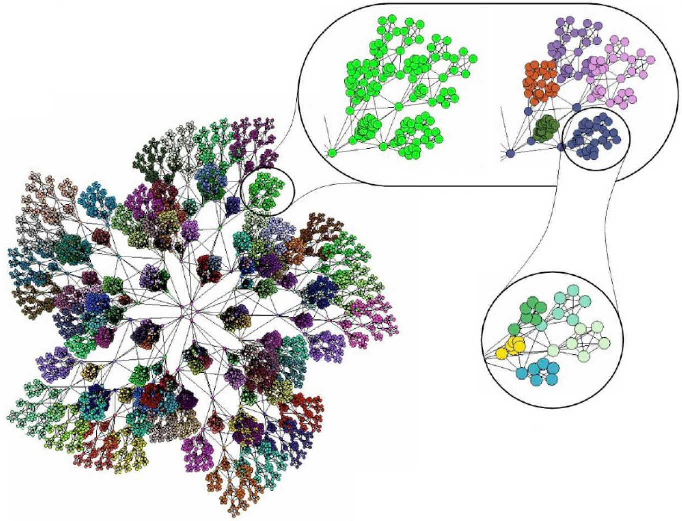


But now graph theory is used for finding communities in networks

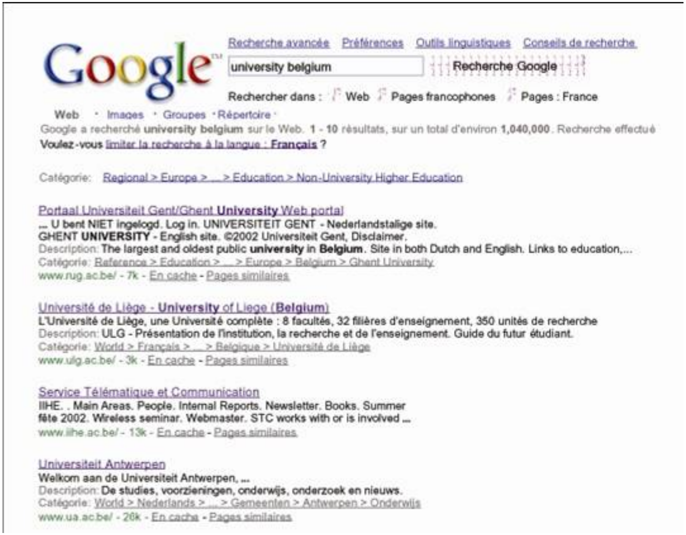


where we want to detect hierarchies of substructures

and their sizes can become quite big ...



It is also used for ranking (ordering) hyperlinks



Recherche avancée Préférences Outils linguistiques Conseils de recherche
university belgium
Recherche Google
Rechercher dans : Web Pages francophones Pages : France
Web · Images · Groupes · Répertoire ·

Google a recherché **university belgium** sur le Web. 1 - 10 résultats, sur un total d'environ 1,040,000. Recherche effectué

Voulez-vous limiter la recherche à la langue : **Français** ?

Catégorie: Regional > Europe > ... > Education > Non-University Higher Education

Portaal Universiteit Gent/Ghent **University** Web portal
... U bent NIET ingelogd. Log in. UNIVERSITEIT GENT - Nederlandstalige site.
GHENT **UNIVERSITY** - English site. ©2002 Universiteit Gent, Disclaimer.
Description: The largest and oldest public **university** in **Belgium**. Site in both Dutch and English. Links to education,...
Catégorie: Reference > Education > ... > Europe > Belgium > Ghent University
www.rug.ac.be/ - 7k - En cache - Pages similaires

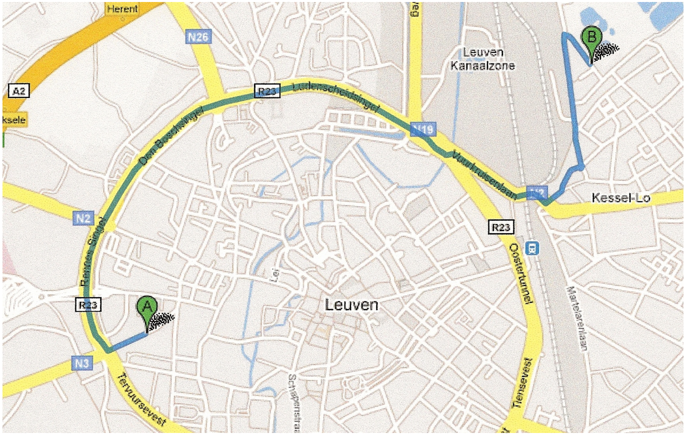
Université de Liège - **University** of Liege (**Belgium**)
L'Université de Liège, une Université complète : 8 facultés, 32 filières d'enseignement, 350 unités de recherche
Description: ULG - Présentation de l'institution, la recherche et de l'enseignement. Guide du futur étudiant.
Catégorie: World > Français > ... > Belgique > Université de Liège
www.ulg.ac.be/ - 3k - En cache - Pages similaires

Service Télématique et Communication
IIHE. . Main Areas. People. Internal Reports. Newsletter. Books. Summer
fête 2002. Wireless seminar. Webmaster. STC works with or is involved ...
www.iihe.ac.be/ - 13k - En cache - Pages similaires

Universiteit Antwerpen
Welkom aan de Universiteit Antwerpen, ...
Description: De studies, voorzieningen, onderwijs, onderzoek en nieuws.
Catégorie: World > Nederlands > ... > Gemeenten > Antwerpen > Onderwijs
www.ua.ac.be/ - 26k - En cache - Pages similaires

or by your GPS to find the shortest path home ...

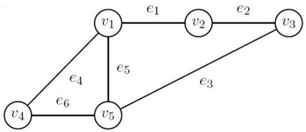
or by your GPS to find the shortest path home ...



What we will cover in this course

- Basic theory about graphs
  - Connectivity
  - Paths
  - Trees
  - Networks and flows
  - Eulerian and Hamiltonian graphs
  - Coloring problems
  - Complexity issues
- A number of applications (in large graphs)
  - Large scale problems in graphs
  - Similarity of nodes in large graphs
  - Telephony problems and graphs
  - Ranking in large graphs
  - Clustering of large graphs

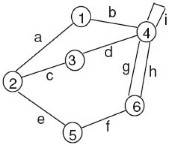
A graph $G = (V, E)$ is a pair of vertices (or nodes) V and a set of edges E, assumed finite i.e. $|V| = n$ and $|E| = m$.

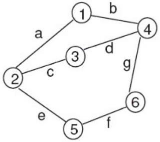


Here $V(G) = \{v_1, v_2, \ldots, v_5\}$ and $E(G) = \{e_1, e_2, \ldots, e_6\}$.

An edge $e_k = (v_i, v_j)$ is incident with the vertices v_i and v_j.

A simple graph has no self-loops or multiple edges like below





Some properties

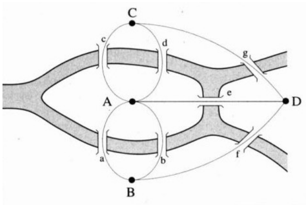
The degree $d(v)$ of a vertex V is its number of incident edges

A self-loop counts for 2 in the degree function.

An isolated vertex has degree 0.

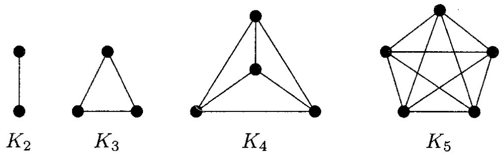
Proposition The sum of the degrees of a graph $G = (V, E)$ equals $2|E| = 2m$ (trivial)

Corollary The number of vertices of odd degree is even (trivial)

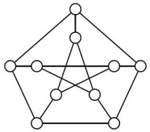


Special graphs

A complete graph K_n is a simple graph with all $B(n,2) := \frac{n(n-1)}{2}$ possible edges, like the matrices below for $n = 2, 3, 4, 5$.

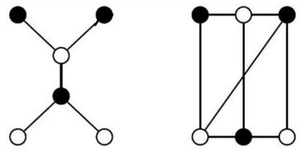


A *k*-regular graph is a simple graph with vertices of equal degree k

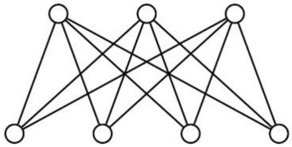


Corollary The complete graph K_n is $(n-1)$-regular

A bipartite graph is one where $V = V_1 \cup V_2$ such that there are no edges between V_1 and V_2 (the black and white nodes below)



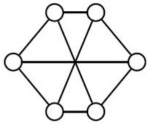
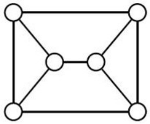
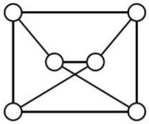
A complete bipartite graph is one where all edges between V_1 and V_2 are present (i.e. $|E| = |V_1|.|V_2|$). It is noted as K_{n_1,n_2}.



When is complete bipartite graph regular ?

When is G bipartite ?

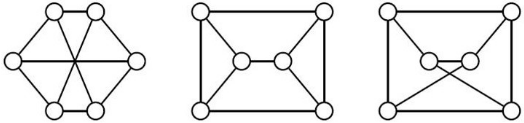
Which graph is bipartite ?

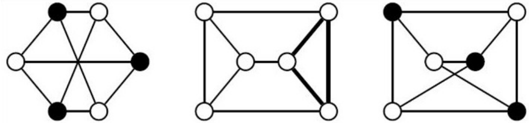
It suffices to find 2 colors that separate the edges as below

When is G bipartite ?

Which graph is bipartite ?



It suffices to find 2 colors that separate the edges as below



The second example is not bipartite because it has a triangle

(to be continued)

Walking in a graph

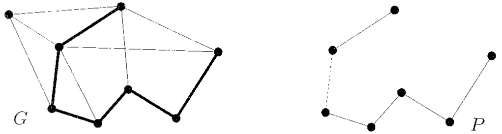
A walk of length k from node v_0 to node v_k is a non-empty graph $P = (V, E)$ of the form

$$V = \{v_0, v_1, \dots, v_k\} \quad E = \{(v_0, v_1), \dots, (v_{k-1}, v_k)\}$$

where edge j connects nodes $j - 1$ and j (i.e. $|V| = |E| + 1$).

A trail is a walk with all different edges.

A path is a walk with all different nodes (and hence edges).



A walk or trail is closed when $v_0 = v_k$.

A cycle is a walk with different nodes except for $v_0 = v_k$.

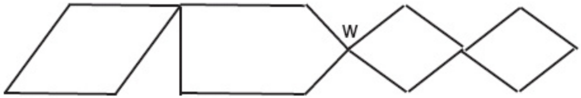
Try to prove the following wo (useful) lemmas

Proposition A walk from u to $v \neq u$ contains a path from u to v

Hint : eliminate subcycles

Proposition A closed walk of odd length contains a cycle of odd length

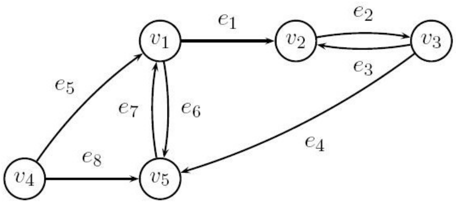
Hint : decompose recursively into distinct subgraphs and use induction



Question Is this only for simple graphs ?

Directed graphs

In a directed graph or digraph, each edge has a direction.



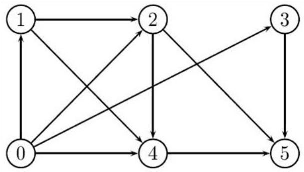
For $e = (v_s, v_t)$, v_s is the source node and v_t is the terminal node.

Each node v has an in-degree $d_{in}(v)$ and an out-degree $d_{out}(v)$.

A graph is balanced if $d_{in}(v) = d_{out}(v)$ for all nodes.

Topological order

Let us now try to order the nodes in a digraph.



Define a bijection $f_{ord} : V \to \{1, 2, \ldots, n\}$, then $f_{ord}(\cdot)$ is a topological order for the graph $G = (V, E)$ iff

$$f_{ord}(i) < f_{ord}(j), \quad \forall (i, j) \in E$$

This is apparently possible for the above graph.
It is easy to see that such a graph should have no cycles.

But is this also sufficient ?

An acyclic graph is a graph without cycles.

Proposition
Every acyclic graph contains at least one node with zero in-degree

Proof By contradiction.

Assume $d_{in}(v) > 0$ for all nodes, then each node i has a predecessor $p(i)$ such that $(v_{p(i)}, v_i) \in E$.

Start from an arbitrary v_0 to form a list of predecessors as below

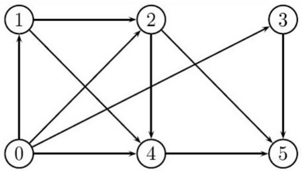
$$\begin{tikzcd} v_0 \arrow[r, bend left] & v_1 = p(v_0) \arrow[r, bend left] & v_2 = p(v_1) \end{tikzcd}$$

Since $|V|$ is bounded, one must eventually return to a node that was already visited; hence there is a cycle.

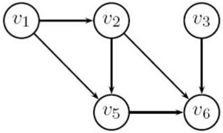
Let us use this to find a topological order

```
**Algorithm** FindTopOrd(G)
$t := 0; G^0 := G;$
**while** $\exists v \in G^t : d_{in}(v) = 0$ **do**
    $G^{t+1} := G^t / \{v\}; order(v) := t + 1; t := t + 1;$
**end while**
**if** $t = n$ **then** $G$ is acyclic;
    **else if** $t < n$ **then** $G$ has a cycle; **end if**
**end if**
```

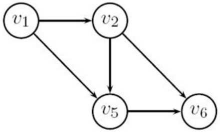
Let us verify this algorithm on the above example.



The only node of in-degree 0 is v_4. So for $t = 1$ we have



After removing v_4 there are two nodes of in-degree 0, v_1 and v_3. If we pick v_3 then we have for $t = 2$

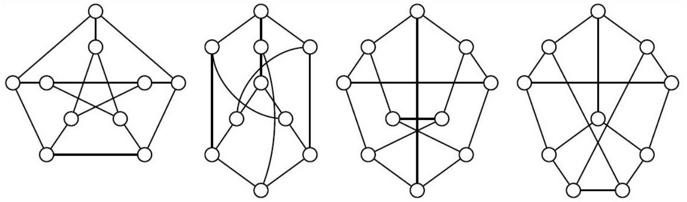


Further reductions yield the final order $\{v_4, v_3, v_1, v_2, v_5, v_6\}$.

What is the complexity of this algorithm ?

Isomorphic graphs

Two graphs G_1 and G_2 are isomorphic iff there is a bijection between their respective nodes which make each edge of G_1 correspond to exactly one edge of G_2, and vice versa.



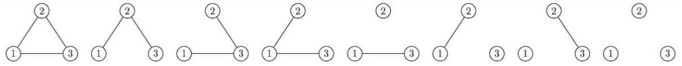
One must find a label numbering that makes the graphs identical

This problem is still believed to be NP hard

Counting graphs

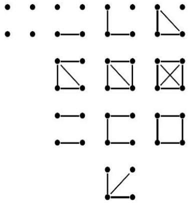
How many different simple graphs are there with n nodes ?

A graph with n nodes can have $B(n, 2) := n(n-1)/2$ different edges and each of them can be present or not.



Hence there can be at most $2^{n(n-1)/2}$ graphs with n nodes.
For $n = 3$ only 4 of the graphs are different
(omitting the isomorphic ones)

With $n = 4$ one finds eventually 11 different graphs after collapsing the isomorphic ones



Let there be T_n non-isomorphic (simple) graphs with n nodes. Then

$$L_n := \frac{2^{n(n-1)/2}}{n!} \leq T_n \leq 2^{n(n-1)/2}$$

Exercise Explain the lower bound

Taking logarithms and using $n! < n^n$ yields the bounds

$$B(n,2) - n \log n \leq \log T_n \leq B(n,2)$$

which gives an idea of the growth of T_n

n	2	3	4	5	6	7	8
T_n	2	4	11	34	156	1044	12346
$\lceil L_n \rceil$	2	2	3	9	46	417	6658

Bipartite revisited

Let us look again at bipartite graphs

Proposition A graph is bipartite iff it has no cycles of odd length

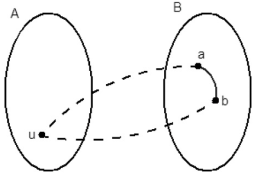
Necessity Trivial : color the nodes of the cycle black and white.

Sufficiency Pick $u \in V$ and let $f(v)$ be the length of a shortest path from u to v (∞ if there is no such path)

$$A = \{v \in V | f(v) = odd\} \quad B = \{v \in V | f(v) = even\}$$

Then A and B form a partition of the nodes of V connected to u.

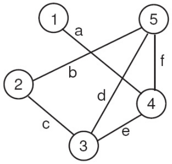
One then needs to show that there can be no links between any two nodes of A or any two nodes of B. If this would be the case, one could construct a cycle of odd length. Repeat on each subgraph.

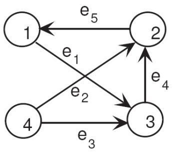


Representing graphs

A graph $G = (V, E)$ is often represented by its adjacency matrix.

It is an $n \times n$ matrix A with $A(i, j) = 1$ iff $(i, j) \in E$. For the graphs



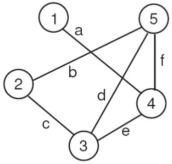


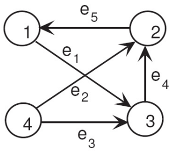
the adjacency matrices are

$$A_1 = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix} \quad A_2 = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

A graph can also be represented by its $n \times m$ incidence matrix T.

For an undirected graph $T(i,k) = T(j,k) = 1$ iff $e_k = (v_i, v_j)$.
For a directed graph $T(i,k) = -1$; $T(j,k) = 1$ iff $e_k = (v_i, v_j)$.
For the graphs

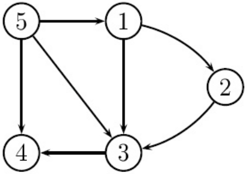




the incidence matrices are

$$T_1 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} \quad T_2 = \begin{bmatrix} -1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & -1 \\ 1 & 0 & 1 & -1 & 0 \\ 0 & -1 & -1 & 0 & 0 \end{bmatrix}$$

One can also use a sparse matrix representation of A and T. This is in fact nothing but a list of edges, organized e.g. by nodes.



$$\begin{aligned} V(1) &= \{2, 3\} \\ V(2) &= \{3\} \\ V(3) &= \{4\} \\ V(4) &= \emptyset \\ V(5) &= \{1, 3, 4\} \end{aligned}$$

Notice that the size of the representation of a graph is thus linear in the number of edges in the graph (i.e. in $m = |E|$).

To be more precise, one should count the number of bits needed to represent all entries :

$$L = (n + m) \log n$$

since one needs $\log n$ bits to represent the vertex pointers.

Counting degrees

Let $\mathbf{1}$ be the vector of all ones, then $d_{in} = A^T\mathbf{1}$ and $d_{out} = A\mathbf{1}$ are the vectors of in-degrees and out-degrees of the nodes of A and $d_{out} = d_{in} = d$ for undirected graphs.

How should we then take self-loops into account ?
In an adjacency matrix of an undirected graph $A(i,i) = 2$
In an adjacency matrix of a directed graph $A(i,i) = 1$

For an undirected graph, we have $d = T\mathbf{1}$.
For a directed graph one can define T_t and T_s as the matrices containing the terminal and source nodes : $T = T_t - T_s$ with

$$T_t := \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}, T_s := \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix}$$

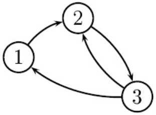
Then also we have $d_{in} = T_t\mathbf{1}$ and $d_{out} = T_s\mathbf{1}$.

Powers of A

Proposition $(A^k)_{ij}$ is the number of walks of length k from i to j

Proof Trivial for k=1; by induction for larger k.
The element (i,j) of $A^{k+1} = A^k \cdot A$ is the sum of the walks of length k to nodes that are linked to node j via the adjacency matrix A.

One verifies this in the following little example



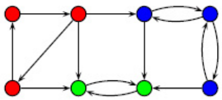
$$A = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}, \quad A^2 = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}$$

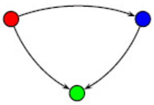
Corollary In a simple undirected graph one has the identities
$tr(A) = 0, tr(A^2)/2 = |E|$ and
$tr(A^3)/6$ equals the number of triangles in G.

Connected components

In a directed graph $G = (V, E)$, u and v are strongly connected if there exists a walk from u to v and from v to u.

This is an equivalence relation and hence leads to equivalence classes, which are called th connected components of the graph G.





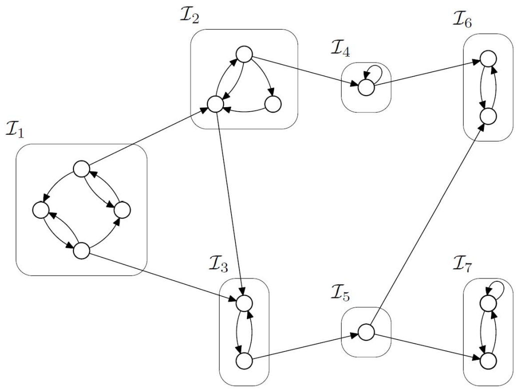
The graph reduced to its connected components is acyclic (why ?)

This shows up in many applications, e.g. in the dictionary graph.

The connected components are the groups of words that use each other in their definition (see later).

After the reduction one has an acyclic graph, which can be ordered topologically.

What do you obtain then ? Class orderings

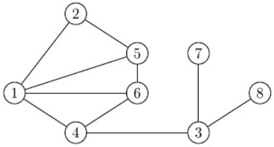


An initial class has $d_{in}(c) = 0$. A final class has $d_{out}(c) = 0$. The other ones are intermediate.

Verify (strong) connectivity of a graph based on its adjacency list

Idea : start from node s, explore the graph, mark what you visit

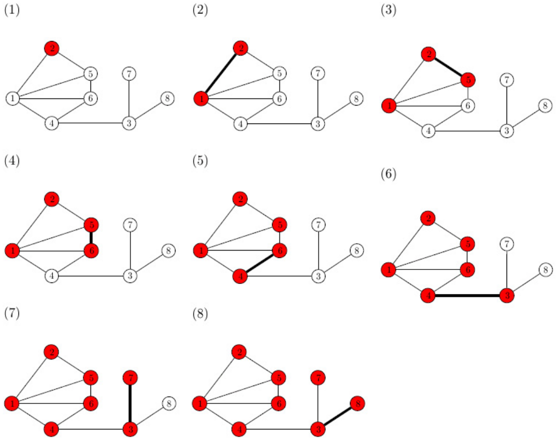
$V(1) = \{2, 4, 5, 6\}$
$V(2) = \{1, 5\}$
$V(3) = \{4, 7, 8\}$
$V(4) = \{1, 3, 6\}$
$V(5) = \{1, 2, 6\}$
$V(6) = \{1, 4, 5\}$
$V(7) = \{3\}$
$V(8) = \{3\}$



```
**Algorithm** GenericSearch(G,s)
mark($s$); $L := \{s\}$
**while** $L \neq \emptyset$ **do**
    choose $u \in L$;
    **if** $\exists(u, v)$ such that $v$ is unmarked **then**
        mark($v$); $L := L \cup \{v\}$;
    **else**
        $L := L \backslash \{u\}$;
    **end if**
**end while**
```

Below we marked the chosen nodes and the discovered nodes

L	*mark*
{2}	2
{2, 1}	1
{2, 1, 5}	5
{2, 1, 5, 6}	6
{1, 5, 6}	
{1, 5, 6, 4}	4
{5, 6, 4}	
{5, 4}	
{5, 4, 3}	3
{5, 3}	
{5, 3, 7}	7
{5, 3}	
{3}	
{3, 8}	8
{3}	
{}	



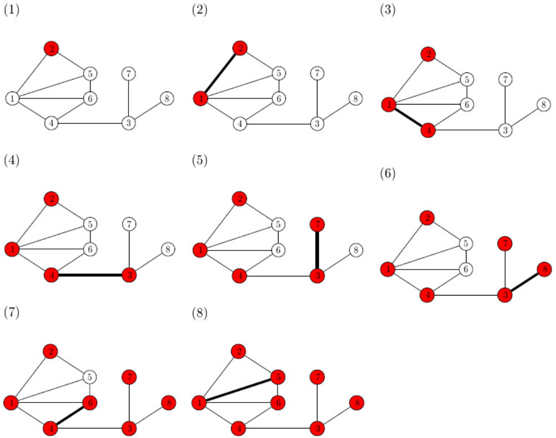
This algorithm has $2n$ steps : each node is added once and removed once. Its complexity is therefore linear in n.

Because of the choices, this algorithm allows for different versions
Let us use a LIFO list for L (Last In First Out) and choose for u the last element added to L. This is a depth first search (DFS).

```
**Algorithm** DeptFirstSearch(G,s)
mark($s$); $L := \{s\}$;
**while** $L \neq \emptyset$ **do**
    $u := last(L)$
    **if** $\exists(u, v)$ such that $v$ is unmarked **then**
        choose $(u, v)$ with $v$ of smallest index;
        mark($v$); $L := L \cup \{v\}$;
    **else**
        $L := L \backslash \{u\}$
    **end if**
**end while**
```

Below we marked the chosen nodes and the discovered nodes

L	*mark*
$\{2\}$	2
$\{2, 1\}$	1
$\{2, 1, 4\}$	4
$\{2, 1, 4, 3\}$	3
$\{2, 1, 4, 3, 7\}$	7
$\{2, 1, 4, 3\}$	
$\{2, 1, 4, 3, 8\}$	8
$\{2, 1, 4, 3\}$	
$\{2, 1, 4\}$	
$\{2, 1, 4, 6\}$	6
$\{2, 1, 4, 6, 5\}$	5
$\{2, 1, 4, 6\}$	
$\{2, 1, 4\}$	
$\{2, 1\}$	
$\{2\}$	
$\{\}$	



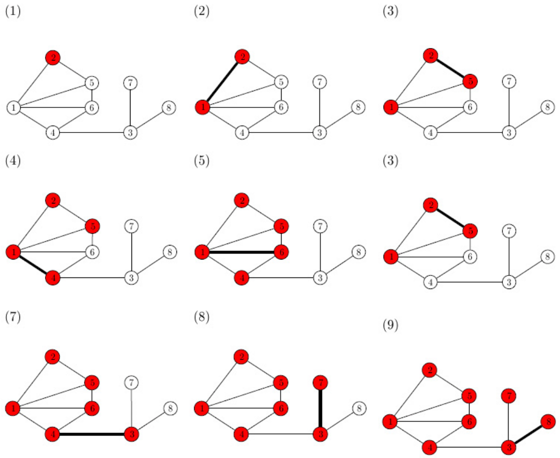
This algorithm builds longer paths than the generic one (depth first).

We now use a FIFO list for L (First In First Out) and choose for u the first element added to L. This is a breadth first search (BFS).

```
**Algorithm** BreadthFirstSearch(G,s)
mark($s$); $L := \{s\}$;
**while** $L \neq \emptyset$ **do**
    $u := first(L)$
    **if** $\exists(u, v)$ such that $v$ is unmarked **then**
        choose $(u, v)$ with $v$ of smallest index;
        mark($v$); $L := L \cup \{v\}$;
    **else**
        $L := L \setminus \{u\}$
    **end if**
**end while**
```

Below we marked the chosen nodes and the discovered nodes

L	*mark*
{2}	2
{2, 1}	1
{2, 1, 5}	5
{1, 5}	
{1, 5, 4}	4
{1, 5, 4, 6}	6
{5, 4, 6}	
{4, 6}	
{4, 6, 3}	3
{6, 3}	
{3}	
{3, 7}	
{3, 7, 8}	8
{7, 8}	
{8}	
{}	

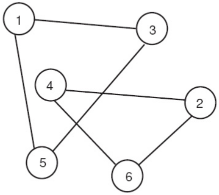


This algorithm builds a wider tree (breadth first).

Testing connectivity

The exploration algorithm finds the set of all nodes that can be reached by a path from a given node $u \in V$.

If the graph is undirected, each node in that set can follow a path back to u. They thus form the connected component $C(u)$ of u.



To find all connected components, repeat this exploration on a node of $V \setminus C(u)$, etc.

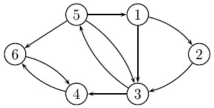
Testing strong connectivity

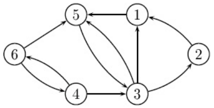
Proposition Let $G = (V, E)$ be a digraph and let $u \in V$.
If $\forall v \in V$ there exists a path from u to v and a path from v to u, then G is strongly connected.

The exploration algorithm finds the set of all nodes that can be reached by a path from a given node $u \in V$.

How can one find the nodes from which u can be reached ?

Construct for that the inverse graph by reversing all arrows

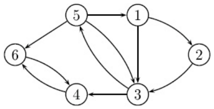


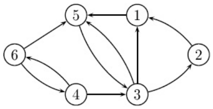


Show that the adjacency matrix of this graph is just A^T.

Proposition Let $G = (V, E)$ be a digraph and let $u \in V$.
Let $R_+(u)$ be the nodes that can be reached from u
and let $R_-(u)$ be the nodes that can reach u,
then the strongly connected component of u is
$C(u) = R_+(u) \cap R_-(u)$

The exploration algorithm applied to the inverse graph, starting from u finds the set $R_-(u)$





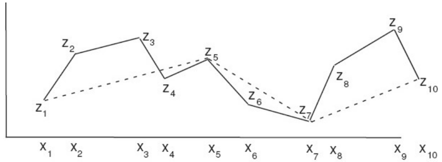
Here $R_+(v_6) = \{4, 6\}$ while $R_-(v_6) = V$ hence $C(v_6) = \{4, 6\}$

Find the other connected components.

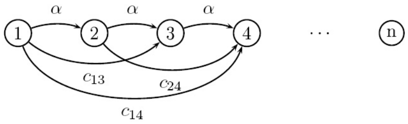
Shortest path problems

Find the shortest total length of a path between two nodes of a directed graph with lengths associated with each edge.

E.g. Find the best piecewise linear approximation of a function

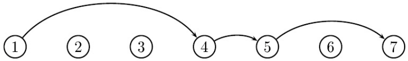


A cost $c_{ij} = \alpha + \beta \sum_{k=i}^{j} (f(x_k) - g(x_k))^2$ is associated with each linear section. This amounts to finding the shortest path in



Other example : Find the best production policy for a plant with a monthly demand d_i, a launching cost f_i, a storage cost h_i and a unit price p_i, for each period $i = 1, \ldots, n$.

In the path below, we are e.g. producing in stages 1, 4 and 5.

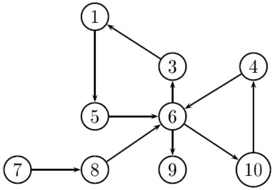


A cost is associated with each section. For the path (1,4) it is e.g. $c_{14} = f_1 + p_1(d_1 + d_2 + d_3) + h_1(d_2 + d_3) + h_2(d_3)$ which is the fixed cost + the production cost in periods 1, 2 and 3 + storage costs at the end of periods 1 and 2.

The minimization of the total cost amounts to a shortest path problem in a graph combining paths as above.

Proposition If there is a shortest walk from *s* to *t*, there is also a shortest path from *s* to *t*

Proof
Assume the walk is not a path; hence there is a recurring node. Eliminate the cycle between the first and last occurrence of this node. Repeat this procedure.



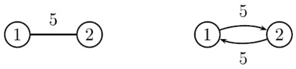
In the above graph the path $(7, 8, 6, 3, 1, 5, 6, 10, 4, 6, 9)$ has a cycle $(6, 3, 1, 5, 6, 10, 4, 6)$. After its elimination we have a path $(7, 8, 6, 9)$.

Corollary If *G* does not contain cycles of negative length, the resulting path is one of lower cost.

Proof Trivial

Dijkstra's algorithm

This method is for a digraph G that has positive edge lengths.
For undirected graphs one can duplicate each edge as follows



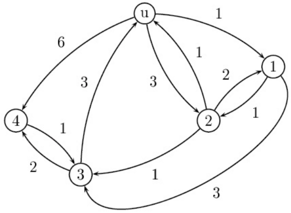
Below, $V^+(u)$ denotes the set of children of u.

```
**Algorithm** Dijkstra(G,u)
$S := \{u\}; d(u) := 0; d(v) := c(u,v) \;\; \forall v \neq u;$
while $S \neq V$ do
    choose $v' \notin S : d(v') \leq d(v) \;\; \forall v \notin S;$
    $S := S \cup \{v'\};$
    for each $v \in V^+(v')$ do
        $d(v) = \min\{d(v), d(v') + c(v', v)\}$
    end for
end while
```

Idea : Update a set S for which we know all shortest paths from u

Let us see the behavior of this algorithm on an example.

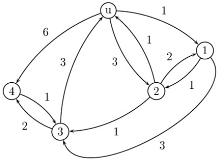
The table below indicates the steps and the distances computed for each node



Iter	S	$d(u)$	$d(1)$	$d(2)$	$d(3)$	$d(4)$
0	$\{u\}$	0	1	3	∞	6
1	$\{u,1\}$	0	1	2	4	6
2	$\{u,1,2\}$	0	1	2	3	6
3	$\{u,1,2,3\}$	0	1	2	3	5
4	$\{u,1,2,3,4\}$	0	1	2	3	5

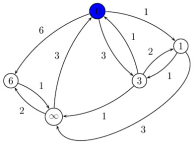
We indicate in more detail the exploration of the graph

S	$d(u)$	$d(1)$	$d(2)$	$d(3)$	$d(4)$
$\{u\}$	0	1	3	∞	6
$\{u,1\}$	0	1	2	4	6
$\{u,1,2\}$	0	1	2	3	6
$\{u,1,2,3\}$	0	1	2	3	5
$\{u,1,2,3,4\}$	0	1	2	3	5

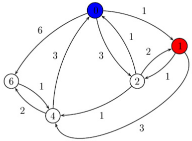


Below, the node *u* is blue and the explored nodes are red

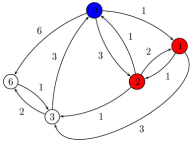
(1)



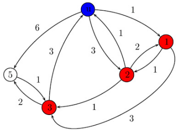
(2)



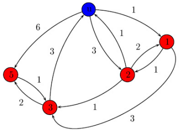
(3)



(4)

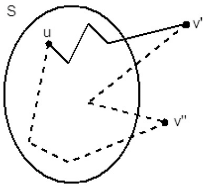


(5)



Proposition Dijkstra's algorithm finds in $O(n^2)$ time the shortest path from u to all other nodes of V.

Proof By induction on the size of S, we show that
1. $\forall v \in S, d(v)$ is the length of the shortest path from u to v
2. $\forall v \in S^+$ (children of nodes of S), $d(v)$ is the length of the shortest path from u to v not passing exclusively via nodes of S



Trivial for $S = \{u\}, d(u) = 0, d(v) = c(u, v)$.

Let $v' \notin S : d(v') = \min_{v \notin S} d(v)$ then the shortest path to v' must lie completely in S. If not, $\exists v''$ outside S at a shorter distance.

We can update $S := S \cup \{v'\}$ and compute the shortest path from u to children of v' as $d(v) = \min\{d(v), d(v') + c(v', v))\}$. This gives the length of the shortest path to all $v \in S^+$.

The other distances are unknown as yet and hence set to ∞.

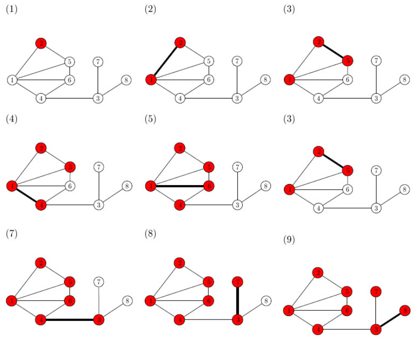
Variants

For a graph with edge lengths 1 it suffices to do a BFSearch and to keep track of the path lengths by incrementing them with 1 during the exploration phase. This is thus an $\mathcal{O}(m)$ time algorithm.

```
**Algorithm** ShortestPathBFS(G,v)
mark($v$); $S := \{v\}$; $d(v) = 0$;
while not $S = \emptyset$ do
    $v := first(S)$
    if $\exists(v, v')$ such that $v'$ is unmarked then
       choose $(v, v')$ with $v'$ of smallest index;
       mark($v'$); $S := S \cup \{v'\}$; $d(v') = d(v) + 1$;
    else
       $S := S \backslash \{v\}$
    end if
end while
```

Proposition All nodes at distance exactly k are correctly identified before proceeding further.

Proof For $k = 0$ this is trivial (S is the original node u).
Induction step : suppose the statement is correct up to k.
After all nodes at distance k have been found, one finds nodes that are at a distance larger than k but since they are all neighboring nodes, they must be at distance exactly $k + 1$.



For an acyclic graph, one can just compute the topological order in $O(m)$ time (see earlier).

To solve the shortest path problem one then uses the algorithm

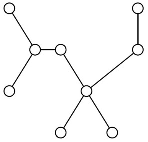
```
**Algorithm** ShortestPathAcyclic(G,v)
$d(1) = 0; d(i) := \infty$ for $i = 2, \dots, n$;
for $i = 1 : n - 1$ do
    for $j \in V^+(i)$ do
        $d(j) := min_j\{d(j), d(i) + c(i,j)\}$;
    end for
end for
```

What is the complexity of this second step ?

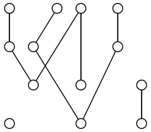
One can also see the shortest path problem as a flow problem or as a linear programming problem.
This leads to other algorithms like the Bellman-Ford Algorithm.

Trees and forests

A tree is an acyclic and connected graph



A forest is an acyclic graph (and hence a union of trees)



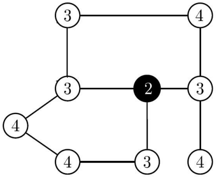
Proposition For a graph $G = (V, E)$ of order $n = |V|$, the following are equivalent

1. G is connected and has $n - 1$ edges
2. G is acyclic and has $n - 1$ edges
3. G is connected and acyclic
4. $\forall u, v \in V$ there is one and only one path from u to v
5. G is acyclic and adding an edge creates one and only one cycle
6. G is connected and removing an arbitrary edge disconnects it

Proofs ?

The following definitions are especially relevant for trees.

The eccentricity $\varepsilon(u) = \max_{v \in V} d(u, v)$ of a node is the maximum distance to any node $v \in V$. The eccentricity of each node is indicated in the graph below



The radius $rad(G) = \min_{u \in V} \varepsilon(u)$ of a graph G is the minimal eccentricity of all nodes in V

The diameter $diam(G) = \max_{u \in V} \varepsilon(u)$ of a graph G is the maximal eccentricity of all nodes in V. It is also the maximal distance between any two nodes in V

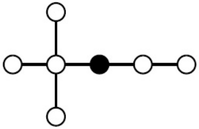
The center of a graph G is the set of nodes in V of minimal eccentricity (the black node)

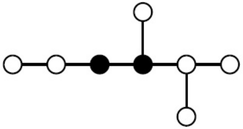
A leaf of a tree T is a node of degree 1

Proposition Let T be a tree and let T' be the tree obtained by removing all its leafs, then $\varepsilon(T') = \varepsilon(T) - 1$ for all nodes of T'.

Proof ?

Proposition The center of a tree is a single node or a pair of adjacent nodes.

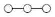
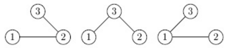
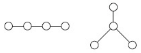
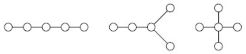




Proof By induction using the previous proposition.
Show that the center does not change.

Counting trees

How many different (labeled) trees are there with *n* nodes ?
The following table gives the count for small *n*

1			$\to 1$
2			$\to 1$
3			$\to 3$
4			$\to 16$
5			$\to 125$

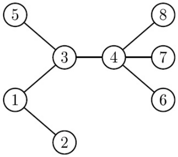
The following theorem of Cayley gives the exact formula.

Proposition
The number of distinct labeled trees of order *n* equals n^{n-2}

We construct a bijection of T_n with a sequence via the algorithm

```
**Algorithm** PrüferSequence(T)
$s := (); t := ();$
**while** $|E| > 1$ **do**
    choose the leaf of smallest index $i$;
    $T := T \setminus \{i\}; s := (s, i); t := (t, neighbour(i));$
**end while**
```

On the graph below, it yields the table next to it



i	s_i	t_i
1	2	1
2	1	3
3	5	3
4	3	4
5	6	4
6	7	4

One shows that the graph can be reconstructed from the sequence t_i which are $n-2$ numbers from $\{1, \ldots, n\}$ and there are exactly n^{n-2} such sequences.

Spanning tree

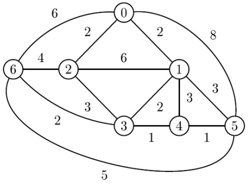
Remove from a connected graph as many edges as possible while remainig connected; this should yield a tree with $n - 1$ edges.

This is the minimal spanning tree problem solved by the following algorithm, of time complexity $\mathcal{O}(m \log m)$

```
Algorithm KruskalMST(G)
$E_{ord} := sort(E)$; $E' := \emptyset$; $E_{rest} := E_{ord}$;
while $|E'| > n - 1$ do
   $\alpha := first(E_{rest})$; $E_{rest} := E_{rest} \setminus \{\alpha\}$;
    if $(V, E' \cup \{\alpha\})$ is acyclic then
      $E' := E' \cup \{\alpha\}$;
    end if
end while
```

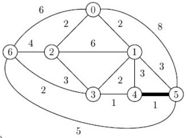
The sorting is done efficiently in $\mathcal{O}(m \log m)$ time as well.

Let us look at an example

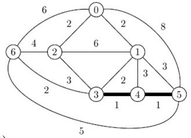


The different steps of the algorithm are

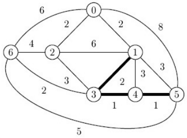
(1)



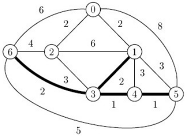
(2)



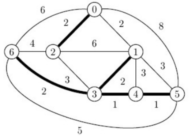
(3)



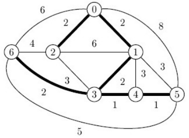
(4)



(5)



(6)



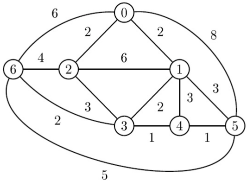
This constructs a tree which is a subgraph with $n - 1$ edges.

Now we look at an alternative algorithm of time complexity $O((m+n)\log n)$

The idea is to pick a random node and then grow a minimal tree from there

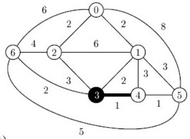
```
**Algorithm** PrimMST(G)
Choose $u \in V$; $V' := \{u\}$; $E' := \emptyset$;
**for** $i = 1 : n-1$ **do**
    $E''$ := edges linking $V$ to $V'$;
    choose $e = (u, v) \in E''$ of minimal weight and such that
    $(V' \cup \{v\}, E'\{e\}$ is acyclic;
    $V' := V' \cup \{v\}$; $E' := E' \cup \{e\}$;
**end for**
```

Let us look at the same example

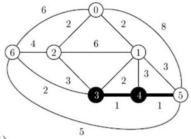


The different steps of the algorithm are

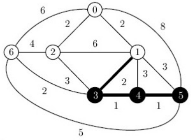
(1)



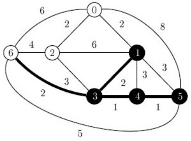
(2)



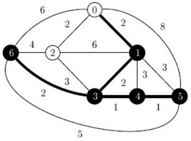
(3)



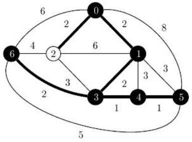
(4)



(5)



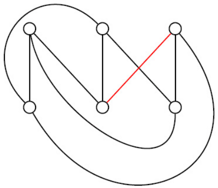
(6)

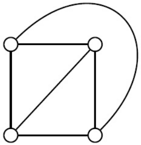


The graph (V, E') is a minimal spanning tree with $n - 1$ edges

Planar graphs

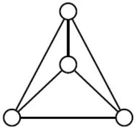
When drawing connected graphs one is naturally lead to the question of crossing edges. One says that a graph is planar if it can be drawn (or represented) without crossing edges



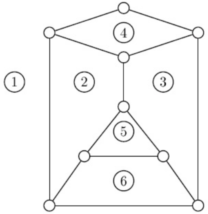


The above graphs represent $K_{3,3}$ (not planar) and K_4 (planar)

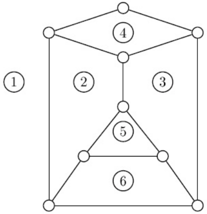
Proposition (Fary, 1948)
Every planar graph can be represented in the plane using straight edges only

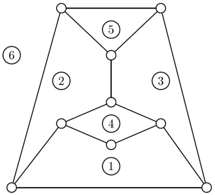


For such graphs, one can now define faces. These are the regions encircled by edges that form a cycle. One has to identify also an exterior face as shown in this figure with 6 faces



Proposition A planar representation of a graph can be transformed to another one where any face becomes the exterior face (a proof comes later)

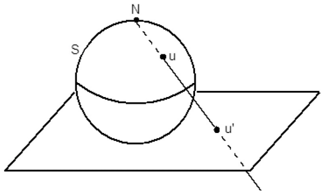




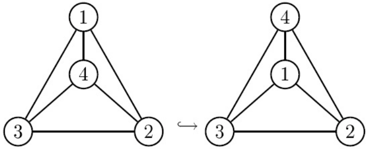
Proposition A graph can be represented in a plane if and only if it can be represented on a sphere (immersion)

Proof

Use a stereographic projection



Every face of the plane is mapped to a sector on the sphere. No point on the sphere can therefore belong to two different sectors. The external face is mapped to a sector containing the north pole.

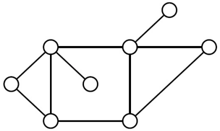


For the external face result, notice that by rotating the sphere, one can move any point (and hence sector) to the north pole

Characterisation

Proposition (Euler formula) Let G be planar, and let $n(G)$ be its number of vertices, $e(G)$ its number of edges, and $f(G)$ its number of faces. Then $f = e - n + 2$.

In the example shown here
$n = 8$; $e = 10$; $f = 4$



Proof Use induction on the number of faces f.
For $f = 1$ there are no cycles and hence the connected graph is a three, for which we know $e = n - 1$ and hence $f = e - n + 2$.
For $f \geq 2$, remove an edge (u, v) between two faces to construct $G' := G \backslash (u, v)$. Then $f(G') = f(G) - 1$; $e(G') = e(G) - 1$ and $n(G') = n(G)$. Use the result for smaller f to prove it for f.

Some exercices

Proposition Let G be planar with $f > 1$, then $3f \leq 2e$

Proposition Let G be planar with $f > 1$ and G have no triangles, then $2f \leq e$

Proposition Let G be a planar (connected) graph.
If $n \geq 3$ then $e \leq 2n - 6$

Proposition Let G be a planar (connected) graph.
If G has no triangles or is bipartite, then $e \leq 2n - 4$

These help to prove the following lemma

Proposition K_5 and $K_{3,3}$ are not planar.

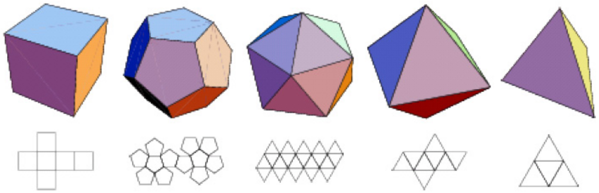
Corollary The average degree of the vertices of a planar connected graph G is smaller than $6 - \frac{12}{n}$

Corollary In a planar (connected) graph there always exists a vertex such that $d(v) \leq 5$

Corollary A planar graph can be colored with 6 colors (see later)

Proposition (Platonic solid) There are only 5 regular polyhedra

These so-called Platonic solids are shown below

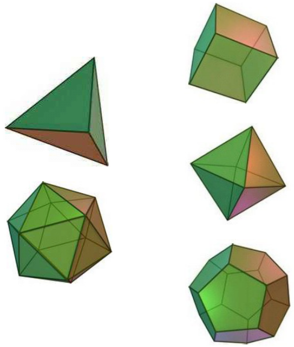


The Platonic solids are characterized by three equations $nk = 2e$, $fl = 2e$ for k, l integers, and $n + f = e + 2$

Explain why

It then follows that $2e/k + 2e/l - e = 2$ hence $2/k + 2/l > 1$ or $(k-2)(l-2) < 4$. The integer solutions are given by

Name	k	l	e	n	f
Tetraeder	3	3	6	4	4
Cube	3	4	12	8	6
Dodecaeder	3	5	30	20	12
Octaeder	4	3	12	6	8
Icosaeder	5	3	30	12	20

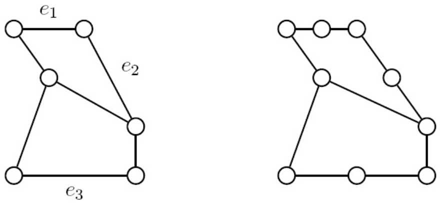


Test for planar graphs

We first need to introduce subdivisions and subgraphs.

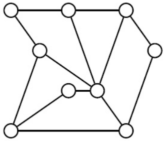
Let us expand a graph $G = (V, E)$ by a subdivision of one of its edges $e = (u, v) \in E$. We put a new node w on e and replace it by two new edges $e_1 = (u, w)$ and $e_2 = (w, v)$. The new graph is thus given by $G' = (V \cup \{w\}, E \cup \{e_1, e_2\} \backslash \{e\})$.

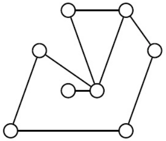
Two graphs are said to be homeomorphic to each other iff one can be derived from the other via a sequence of subdivisions.



Corollary Homeomorphism is an equivalence relation.

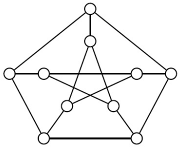
A graph $G' = (V', E')$ is a subgraph of a graph $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$ (edges must disappear along with nodes)

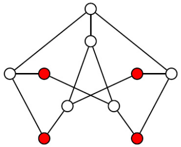


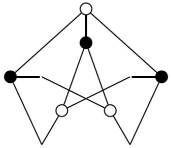


Proposition (Kuratowsky, 1930) A graph is planar iff it does not contain a subgraph homeomorphic to $K_{3,3}$ or K_5.

Example : the Petersen graph (subgraph + homeomorphism)

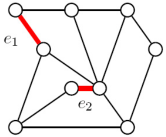


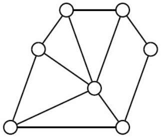




Minors

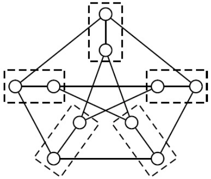
Let $e = (u, v)$ be an edge of a graph $G = (V, E)$. A contraction of the edge e consists of eliminating e and merging the nodes u and v into a new node w. The new graph G' is thus $G' = (V \backslash \{u, v\} \cup \{w\}, E \backslash \{e\})$

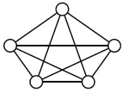




Proposition (Wagner, 1937) A graph is planar iff it does not have $K_{3,3}$ or K_5 as a minor.

Example :
the Petersen
graph again

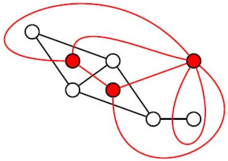




Proposition (Robertson-Seymour)
For a graph G, determining if a given graph H is a minor of H, can be solved in polynomial time (with respect to $n(G)$ and $m(G)$).

A dual graph G^* of a planar graph is obtained as follows
1. G^* has a vertex in each face of G
2. G^* has an edge between two vertices if G has an edge between the corresponding faces

This is again a planar graph but it might be a multigraph (with more than one edge betwee two vertices)

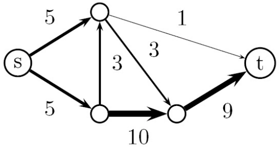


Exercise Show that Euler's formula is preserved

Exercise Show that $G = (G^*)^*$

Networks and flows

A network is a directed graph $N = (V, E)$ with a source node s (with $d_{out}(s) > 0$) and a terminal node t (with $d_{in}(t) > 0$). Moreover each edge has a strictly positive capacity $c(e) > 0$.

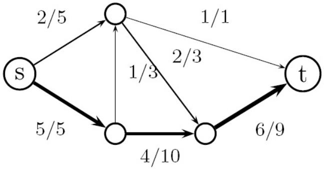


A flow $f : V^2 \to \mathbb{R}^+$ is associated with each edge $e = (u, v)$ s.t.
1. for each edge $e \in E$ we have $0 \leq f(e) \leq c(e)$
2. for each intermediate node $v \in V \backslash \{s, t\}$ the in- and out-flow at that node $\sum_{u \in V^-(v)} f(u, v) = \sum_{u \in V^+(v)} f(v, u)$ match

The total flow F of the network is then what leaves s or reaches t

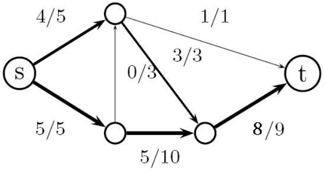
$$F(N) := \sum_{u \in V} f(s, u) - \sum_{u \in V} f(u, s) = \sum_{u \in V} f(u, t) - \sum_{u \in V} f(t, u)$$

Here is an example of a flow



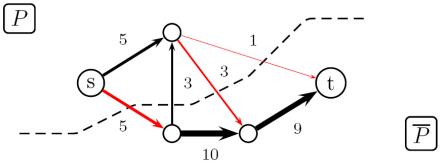
It has a value of $F(N) = 7$ and the conservation law is verified inside.

But the flow is not maximal, while the next one is ($F(N) = 9$) as we will show later. Notice that one edge is not being used ($f = 0$)



Cut of a network

A cut of a network is a partition of the vertex set $V = P \cup \overline{P}$ into two disjoint sets P (containing s) and $\overline{P}$ (containing t)



The capacity of a cut is the sum of the capacities of the edges (u, v) between P and $\overline{P}$

$$\kappa(P, \overline{P}) = \sum_{u \in P; v \in \overline{P}} c(u, v)$$

which in the above example equals $5 + 3 + 3 + 1 = 9$.

We now derive important properties of this capacity.

Proposition Let $(P, \overline{P})$ be any cut of a network $N = (V, E)$ then the associated flow is given by

$$F(N) = \sum_{u \in P; v \in \overline{P}} f(u, v) - \sum_{u \in P; v \in \overline{P}} f(v, u)$$

Proof First show that $F(N) = \sum_{u \in P} \left(\sum_v f(u, v) - \sum_v f(v, u) \right)$ by summing all contributions in P and using conservation.
For all $v \in P$ the term between brackets is zero (conservation).
Hence we only need to keep the edges across the partition.

Corollary A flow is bounded by the capacity of any cut
$F(N) \leq \kappa(P, \overline{P})$

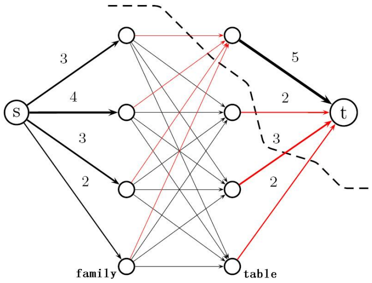
A minimal cut (with minimal capacity) also bounds $F(N)$

(we will construct one and will see it is in fact equal to $F(N)$)

Applications

The dining problem

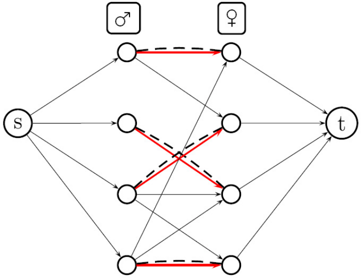
Can we seat 4 families with number of members (3,4,3,2) at 4 tables with number of seats (5,2,3,2) so that no two members of a same family sit at the same table ?



The central edges are the table assignments (a capacity of 1).
The cut shown has a capacity 11 which upper bounds $F(N)$.
We can therefore not seat all 12 members of the four families.

The marriage problem

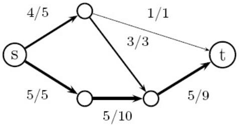
One wants to find a maximimum number of couplings between men and women where each couple has expressed whether or not this coupling was acceptable (central edges that exist or not)

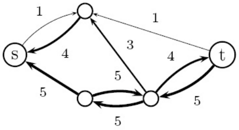


One wants to find a maximum number of disjoint paths in this directed graph. All the capacities of the existing edges are 1.

Given a network $N(V,E)$ and a flow f then its residual network N_f is a network with the same nodes V but with new capacities

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \textit{if } (u,v) \in E; \\ f(v,u) & \textit{if } (v,u) \in E \\ 0 & \textit{otherwise}. \end{cases}$$





An augmenting path is a directed path $v_0, \ldots, v_k$ from $S = v_0$ to $t = v_k$ for which
$\Delta_i = c(v_i, v_{i+1}) - f(v_i, v_{i+1}) > 0 \; \forall (v_i, v_{i+1}) \in E$ or
$\Delta_i = c(v_i, v_{i+1}) - f(v_{i+1}, v_i) > 0 \; \forall (v_{i+1}, v_i) \in E$
This path is not optimal since the original flow can be increased.

Max-flow Min-cut

Proposition
The flow is optimal if there exists no augmentation path from s to t

Proof Construct a cut $(P, \overline{P})$ where $u \in P$ if there is an augmentation path from s to u and $u \in \overline{P}$ otherwise.
Show that $(P, \overline{P})$ is a valid cut for which $F(N) = \kappa(P, \overline{P})$.

Proposition In a network N the following are equivalent
1. A flow is optimal
2. The residual graph does not contain an augmenting path
3. $F(N) = \kappa(P, \overline{P})$ for some cut $(P, \overline{P})$

The value of the optimal flow thus equals $F(N) = \min \kappa(P, \overline{P})$

Proof Left to the reader (combine earlier results)

This becomes an LP problem in the flows x_{ij} on the edges (i, j)
max $\left(\sum_{i:(s,i)} x_{si} = \sum_{i:(i,s)} x_{is}\right)$ subject to $\sum_i x_{ij} = \sum_i x_{ji}$ and $0 \leq x_{ij} \leq c_{ij}$

The Ford-Fulkerson algorithm (1956) calculates this optimal flow using augmentation paths.

```
**Algorithm** MaxFlowFF(N,s,t)
$f(u,v) := 0 \ \forall (u,v) \in E$;
**while** $N_f$ contains a path from $s$ to $t$ **do**
    choose an augmentation path $A_p$ from $s$ to $t$
    $\Delta := \min_{(u,v) \in A_p} \Delta_i$
    Augment the flow by $\Delta$ along $A_p$
    Update $N_f$
**end while**
```

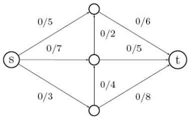
Finding a path in the residual graph can be implemented with a BFS or DFS exploration as shown below

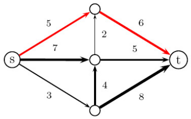
At each step we show the graph (left) and the residual graph (right)

Augmentation paths are in red. In 5 steps we find $F(N) = 14$

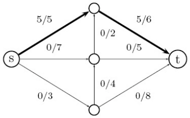
Flows

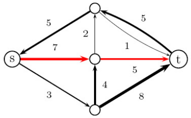
(1)


0/5
0/6
0/2
0/7
0/5
s
t
0/4
0/3
0/8

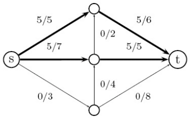

5
6
2
7
5
s
t
4
3
8

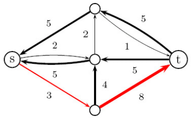
(2)


5/5
5/6
0/2
0/7
0/5
s
t
0/4
0/3
0/8

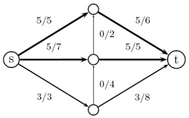

5
5
2
7
1
s
t
5
4
3
8

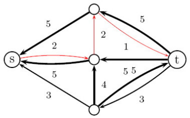
(3)


5/5
5/6
0/2
5/7
5/5
s
t
0/4
0/3
0/8

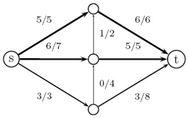

5
5
2
2
1
s
t
5
5
4
3
8

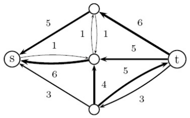
(4)


5/5
5/6
0/2
5/7
5/5
s
t
0/4
3/3
3/8


5
5
2
2
1
s
t
5
5 5
4
3
3

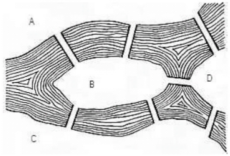
(5)

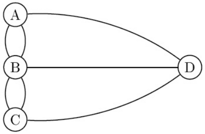

5/5
6/6
1/2
6/7
5/5
s
t
0/4
3/3
3/8


5
6
1
1
1
5
s
t
6
5
4
3
3

Eulerian tour (1756)

An Eulerian cycle (path) is a subgraph $G_e = (V, E_e)$ of $G = (V, E)$ which passes exactly once through each edge of G.
G must thus be connected and all vertices V are visited (perhaps more than once). One then says that G is Eulerian

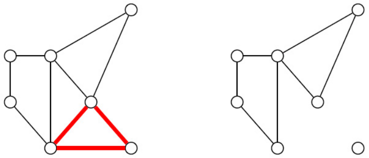




Proposition A graph G has an Eulerian cycle iff it is connected and has no vertices of odd degree
A graph G has an Eulerian path (i.e. not closed) iff it is connected and has 2 or no vertices of odd degree

This would prove that the above graph is not Eulerian.

Proof (of the first part regarding cycles)
Necessity Since G is Eulerian there is a cycle visiting all nodes. Each time we visit $v \in V$, we leave it again, hence $d(v)$ is even.
Sufficiency For a single isolated node, it is trivial. For $|V| > 1$ there must be a cycle ϕ in the graph. Consider the subgraph H with the same nodes but with the edges of ϕ removed.

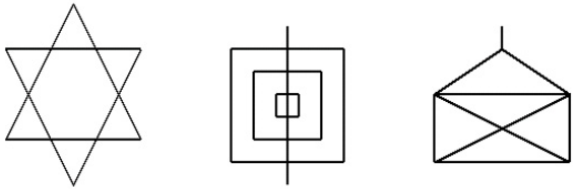


Each of its components H_i satisfy the even degree condition and again have an Eulerian cycle ϕ_i. By recurrence we then reduce G to its isolated vertices.

To reconstruct the Eulerian cycle, start from a basic cycle ϕ. Each time a node of another cycle ϕ_i is encountered, substitute that cycle to the node (and do this recursively).

Proof (of the second part regarding paths) Left as an exercise

The path problem says if you can draw a graph without lifting your pen. Apply this to the following examples.



Proposition A directed graph $G = V, E_d$ has an Eulerian tour G_e iff it is connected and balanced, i.e. all its nodes have $d_{in}(v) = d_{out}(v)$.

Proof Left as an exercise

The following algorithm of Fleury (1883) reconstructs a cycle C if it exists. E' is the set of edges already visited by the algorithm.

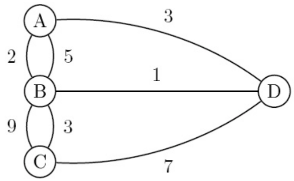
```
**Algorithm** FindEulerianCycle(G)
Choose $v_0 \in V$; $E' := \emptyset$; $C := \langle\rangle$;
**for** $i = 1 : m$ **do**
    choose $e = (v_{i-1}, v_i)$ s.t. $G' = (V, E \backslash E')$ has 1 conn. comp.;
    $E' := E' \cup \{e\}$; $C := \langle C, e\rangle$; $v_i := v_{i-1}$;
**end for**
```

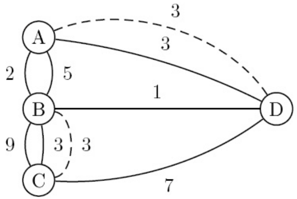
Exercise
Propose a modification addressing the Eulerian Path Problem

But what if the graph is not Eulerian ? Can we find a mininimum cost modification of the problem ?

Chinese postman (1962)

We consider a minimum cost modification of the Eulerian cycle problem. A chinese postman needs to find a tour passing along all edges of a graph and minimize the length of the path.



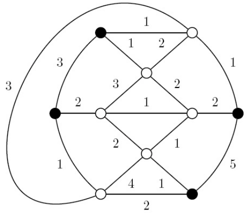


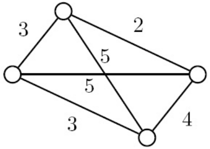
The edges have a cost and we need to make the graph Eulerian

Exercise

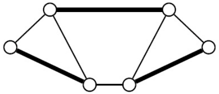
1. Give a simple lower bound. 2. When can this bound be met ?
3. Is there another solution (or a better one) ?

Solution : find all odd degree vertices and find the shortest paths between them





Now find a perfect matching of the nodes in this graph.
A perfect matching in a graph is a set of disjoint edges of a graph to which all vertices are incident.

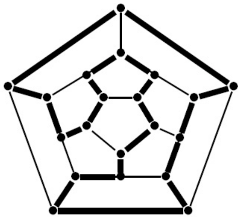


This can be solved in $0(n^3)$ time with the Hungarian algorithm.

Hamiltonian cycle (1859)

Was a game sold by Hamilton in 1859 to a toy maker in Dublin.

A Hamiltonian cycle is a cyclic subgraph $G_h = (V, E_h)$ of $G = (V, E)$ which passes exactly once through all nodes



It is a so-called hard problem and there is no general condition for its existence (in contrast with the Eulerian path problem).

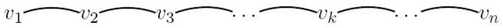
It exists for Platonic solids and complete graphs, but not for the Petersen graph

Proposition (Dirac, 1951) A graph G with $n \geq 3$ nodes and $d(v) \geq n/2,\ \forall v \in V$, is Hamiltonian

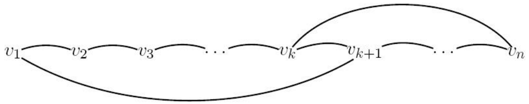
Proof

G is connected, otherwise its smallest component would have all edges with $d(v) < n/2$

Then consider a longest path $v_1 v_2 ... v_n$ (with maybe $n < |V|$)



Because $d(v_1), d(v_n) \geq n/2$, it must also be covered by a cycle (because all the neigbors of v_1 and v_n are on that path)

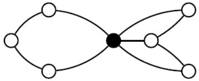


Because of connectedness $n = |V|$ and it is a Hamiltonian cycle.

Exercise Construct a graph with $d(v) < n/2$ and yet has a Hamiltonian cycle

Proposition
If $G = (V, E)$ has a Hamiltonian cycle, then $G - V'$ has at most $|V'|$ connected components for any subset of vertices $V' \subset V$.
Proof Let H be a Hamiltonian subgraph of G, then $H - V'$ has less than $|V'|$ connected components. But $G - V'$ has the same vertices as $H - V'$ and it has additional edges.

Exercise Does this graph have a Hamiltonian cycle ?



Exercise Prove that a complete bipartite graph $K_{m,n}$ is Hamiltonian iff $m = n$

Traveling Salemen Problem

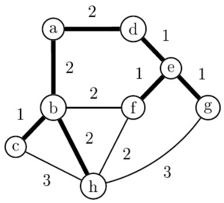
A traveling salesman is supposed to visit a number of cities (nodes in a graph) and minimize the travel time (or total length)

This is NP-hard but can often be solved approximately in reasonable time. Consider a distance graphs with triangle inequality $d(u, v) + d(v, w) \geq d(u, w)\ \forall u, v, w \in V$

Construct a minimal weight spanning tree T and visit the nodes using BFS.

For this example we would have a cycle (a,b,c,b,h,b,a,d,e,f,e,g,e,a)

Notice that all edges are visited twice.



The optimal path P^* satisfies the inequalities $cost(T) < cost(P^*) \leq 2.cost(T)$

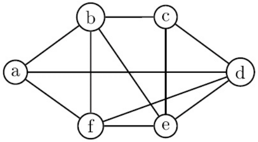
Exercise Explain why

Test for planar graph

There is a simple way to test if a Hamiltonian graph is planar

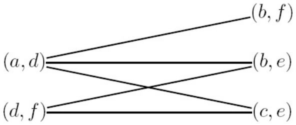
1. Draw G with the Hamiltonian graph H at the outside

The following graph is already drawn with $H = (a, b, c, d, e, f, a)$ outside



2. Define K as the graph whose nodes are the edges $e_1, \ldots, e_r$ not in H and with an edge between e_i and e_j if they cross in G.

The following graph has the vertices $(a, d), (b, f), b, e), c, e), (d, f)$ and five edges, corresponding to the crossings in G

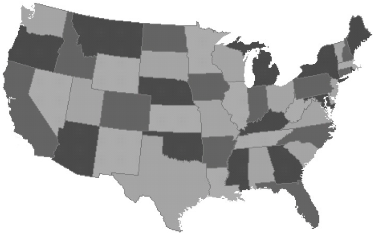


Then G is planar iff K is bipartite

Exercise Explain why

Four color problem

In 1852 it was conjectured that a country map (like the USA map) could always be colored with only four colors. There is an underlying assumption for point borders.



This was proven in 1976 by K. Appel and W. Haken but their proof used a computer search over 1200 so-called critical cases.

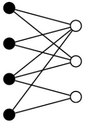
Exercise What property does the underlying graph have ?

Coloring nodes

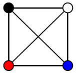
A *k*-coloring of a graph $G = (V, E)$ is a mapping $f : V \rightarrow 1, \ldots, k$ such that $f(v_i) \neq f(v_j)$ if $(v_i, v_j) \in E$.
The chromatic number of a graph is the smallest number k for which there exists a k-coloring.

Some examples of known chromatic numbers are :

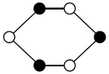
Bipartite graph
$\chi(G) = 2$



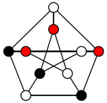
Clique
$\chi(K_n) = n$



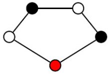
Even cycle
$\chi(G) = 2$



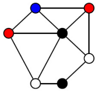
Petersen Graph
$\chi(G) = 3$



Odd cycle
$\chi(G) = 3$

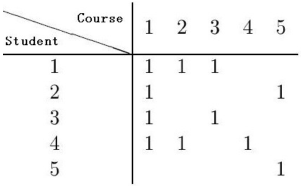


Planar graph
$\chi(G) = 4$

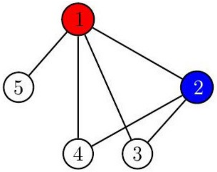


The coloring problem for general graphs is NP-complete but such problems often lead to more interesting applications

Exam scheduling problem



Student \ Course	1	2	3	4	5
1	1	1	1		
2	1				1
3	1		1		
4	1	1		1	
5					1



The table on the left gives the exams each student takes
The chromatic number $\chi(G)$ of the corresponding graph gives the minimum numbers of time slots for the exams

Exercise Can you formulate such a slot problem with students choosing out of k pre-set programs ?

Bounds

Proposition

Let G be connected and $m = |E|$, then $\chi(G) \leq \frac{1}{2} + \sqrt{2m + \frac{1}{4}}$

Proof Let $C = \{C_1, \ldots, C_k\}$ be the partition of V according to colors. There is at least one edge between two colors, which implies $m > B(k, 2)$ and hence $k^2 - k - 2m \leq 0$.

Proposition

Let $\Delta(G) = \max\{d(v) | v \in V\}$, then $\chi(G) \leq \Delta(G) + 1$ (trivial)

Proposition (Brooks, 1941)

$\chi(G) \leq \Delta(G)$ for any graph different from K_n or an odd cycle

Proposition

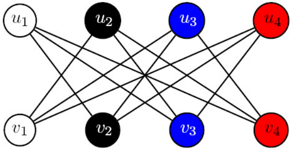
$\chi(G) \leq 1 + \max_i\{\min(d_i, i - 1)\}$ when ordering $d_1 \geq \ldots \geq d_n$.

Proof Order the nodes like the d_i's and use the greedy algorithm

Greedy algorithm

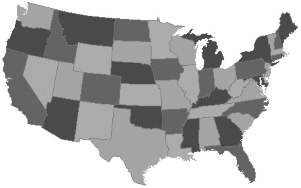
```
Algorithm GreedyColor(G)
$L := sort(V)$; $c := sort(colors)$
for $v \in V$ do
    choose smallest $c_i$ not used by colored neigbors
end for
```

On a bipartite graph this greedy algorithm is optimal when numbering the nodes per part but it can be bad for other numberings, such as $\{u_1, v_1, u_2, v_2, u_3, v_3, u_4, v_4\}$



Exercise
Does each graph have a good numbering for the greedy algorithm

Let us come back to the map coloring problem



and try to prove the following (simpler) result

Exercise Every planar graph can be colored with 6 colors

Show that $e \leq 3n - 6$

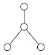
Show then that for planar graphs $\mathit{average}(d(v)) \leq 6 - 12/n$

Finally prove that there exists a v such that $d(v) \leq 5$

Now use induction to prove the proposition (remove nodes)

Chromatic polynomial (Birkhoff-Lewis 1918)

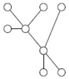
The chromatic polynomial of a graph $p_G(k)$ indicates how many different ways a graph can be colored with k colors. E.g.



k	1	2	3	k
$p_G(k)$	0	2	24	$k(k-1)^3$

k	1	2	3	k
$p_G(k)$	0	0	6	$k(k-1)(k-2)$

k	1	2	k
$p_G(k)$	1	16	k^4



k	1	2	k
$p_G(k)$	0	2	$k(k-1)^{n-1}$

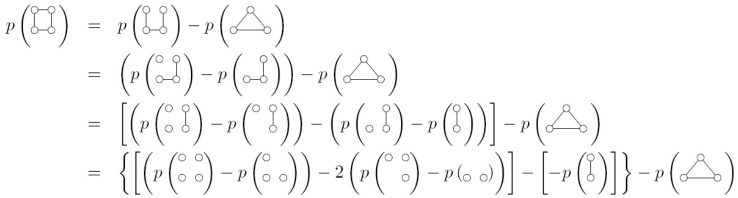
Exercise Prove the above formulas

Notice that $\chi(G) = \min\{p(G)(k) > 0\}$. Does this help ?

There is a powerful induction theorem using the simpler graphs $G-(u,v)$ (remove an edge) and $G \circ (u,v)$ (contract an edge)

Proposition If $(u,v) \in E$ then $p_G(k) = p_{G-(u,v)}(k) - p_{G\circ(u,v)}(k)$
Proof u and v have different colors in G and the same in $G \circ (u,v)$

This can be used to compute the chromatic polynomial of more complex networks



$$
\begin{aligned}
&= (k^4 - k^3) - 2(k^3 - k^2) + k(k-1) - k(k-1)(k-2) \\
&= k^4 - 4k^3 + 6k^2 - 3k \\
&= k(k-1)(k^2 - 3k + 3)
\end{aligned}
$$

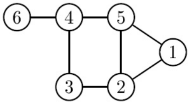
but the problem remains combinatorial and thus hard

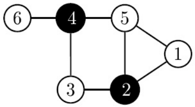
Exercise Derive this quicker using the result for a tree

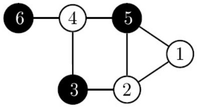
Stable sets

An independent or stable set S in a graph $G = (V, E)$ is a subgraph of G without any edges, i.e. $\forall u, v \in S : (u, v) \in E$

The two sets of black nodes are stable sets of the left graph







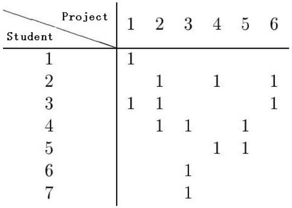
Such sets can clearly be colored with only one color, which proves

Proposition If a graph is k-colorable then V can be partitioned as k stable sets

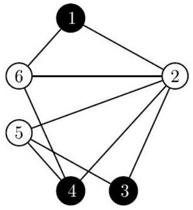
The independence number $\alpha(G)$ is the size of the largest possible stable set.

Proposition One has $\chi(G) \cdot \alpha(G) \geq n$ (trivial)

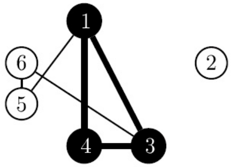
The following example requires finding a maximal stable set. Find the maximum number of projects one can realize when the table indicates which students are needed for each project.



Student \ Project	1	2	3	4	5	6
1	1					
2		1		1		1
3	1	1				1
4		1	1		1	
5				1	1	
6			1			
7			1			



Notice that it is equivalent to finding a maximal clique (or complete subgraph) in the complementary graph $G_c = (V, E_c)$, where E_c is the complement of E



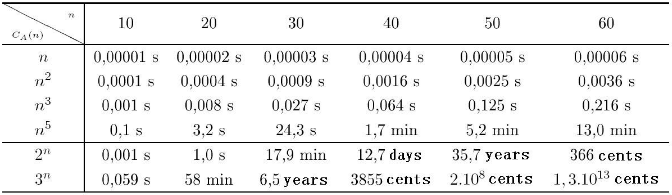
Algorithm complexity

We distinguish problems from algorithms used to solve them.
There is also the issue of time complexity and space complexity.

The function $C_A(s)$ of an algorithm is the number of time steps needed to solve a problem of size s with that algorithm.
A problem is called polynomial if there exists an algorithm with $C_A(s) = \mathcal{O}(p(n))$ for some polynomial $p(\cdot)$, meaning

$$\exists n_0 : C_A(s) \leq p(n) \quad \forall n \geq n_0.$$

The relative times needed to solve problems of different complexity



$C_A(n)$ \ n	10	20	30	40	50	60
n	0,00001 s	0,00002 s	0,00003 s	0,00004 s	0,00005 s	0,00006 s
n^2	0,0001 s	0,0004 s	0,0009 s	0,0016 s	0,0025 s	0,0036 s
n^3	0,001 s	0,008 s	0,027 s	0,064 s	0,125 s	0,216 s
n^5	0,1 s	3,2 s	24,3 s	1,7 min	5,2 min	13,0 min
2^n	0,001 s	1,0 s	17,9 min	12,7 days	35,7 years	366 cents
3^n	0,059 s	58 min	6,5 years	3855 cents	2.10^8 cents	$1,3.10^{13}$ cents

This shows the importance of having a polynomial problem

Better is to look at the size of the problems one can solve when the machines speed up 100 or 1000 times

$C_A(n)$	size	100 times	1000 times
n	N_1	$100N_1$	$1000N_1$
n^2	N_2	$10N_2$	$31,6N_2$
n^3	N_3	$4,64N_3$	$10N_3$
n^5	N_4	$2,5N_4$	$3,98N_4$
2^n	N_5	$N_5+6,64$	$N_5+9,97$
3^n	N_6	$N_6+4,19$	$N_6+6,29$

Here are a number of polynomial time problems

Finding the shortest path between 2 vertices
Testing if a graph is planar
Testing if a graph is Eulerian
Finding a spanning tree
Solving the perfect marriage problem

Here are a number of problems that are not polynomial

Finding the chromatic number of a graph
Finding a Hamiltonian cycle in a graph
Finding the largest stable set in a graph
Solving the travelling salesman problem
Testing if two graphs are isomorphic (not known)

Comparing problems

A problem Y is reducible (in polynomial time) to a problem X if X is at least as difficult to solve as Y, denoted as $X \geq_p Y$. Then

$X \geq_p Y$ and $X \in \mathcal{P}$ implies $Y \in \mathcal{P}$
$X \geq_p Y$ and $Y \notin \mathcal{P}$ implies $X \notin \mathcal{P}$

Define the problem $[\mathit{LongestPath}(u, v, w, N)]$ of finding a path of length $\geq$ any N from u to v in a graph with integer weights w

Proposition $[\mathit{HamiltonianCycle}] \leq_p [\mathit{LongestPath}(u, v, w, N)]$

Proof Choose unit weights w. Pick an edge $e = (u, v)$.
If there is a longest path of length $N = n - 1$ in $G' = G \backslash e$, then G is Hamiltonian. Try out all $m < n^2/2$ edges.

Since we know that the Hamiltonian cycle problem in not in $\mathcal{P}$ the longest path problem is also not in $\mathcal{P}$.

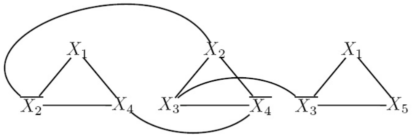
A Boolean clause is a disjunction of Boolean terms $X_i \in \{0,1\}$ and their negation $\overline{X}_i \in \{0,1\}$, e.g. $X_1 \vee \overline{X}_2 \vee X_4 \vee \overline{X}_7$ is a 4-term.

Define the problem [SAT] as checking if a set of Boolean clauses can be simultaneously satisfied ([3SAT] involves only 3-terms). E.g. $\{\overline{X}_2 \vee X_2, \overline{X}_2 \vee X_3 \vee \overline{X}_4, \overline{X}_1 \vee X_4\}$ can be satisfied by choosing $X_1 = 1, X_2 = 0, X_3 = 1, X_4 = 0$.

Proposition $[SAT] \leq_p [3SAT]$ and $[3SAT] \leq_p [StableSet]$

Proof We do not prove the first part involving only 3-terms.

Construct a triangle for each 3-term and then connect the negations across triangles



For a stable set, I can choose only one node in each triangle. Then there is a stable set of size $n/3$ iff [3SAT] is satisfiable.

$\mathcal{NP}$ and $\mathcal{NP}$-complete

A problem is Non-deterministic Polynomial ($\mathcal{NP}$) if the validity of a solution can be checked in polynomial time.

Checking if a given cycle is Hamiltonian can be solved in polynomial time, but finding it is difficult.

In $\mathcal{P}$ the problem can be solved in polynomial time,
in $\mathcal{NP}$ a solution can be checked in polynomial time.

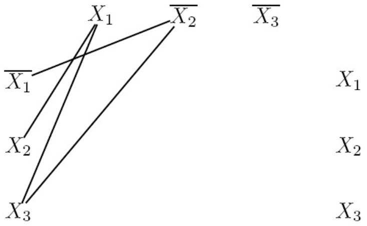
It is still an open question of $\mathcal{P} = \mathcal{NP}$ (Cray prize = 1 million dollar)

A problem X is $\mathcal{NP}$-complete if $X \in \mathcal{NP}$ and $\forall Y \in \mathcal{NP}$, $Y \leq_p X$.

Corollary If one $\mathcal{NP}$-complete problem is in $\mathcal{P}$ then $\mathcal{P} = \mathcal{NP}$

Corollary If one $\mathcal{NP}$-complete problem is not in $\mathcal{P}$ then $\mathcal{P} \neq \mathcal{NP}$

[3SAT] is known to be $\mathcal{NP}$-complete.
We now prove that also the [CLIQUE] problem is $\mathcal{NP}$-complete
The [CLIQUE] problem is checking if there exists a clique (complete subgraph) of size k in a graph $G = (V, E)$
Proof Consider $\{X_1 \vee \overline{X}_2 \vee \overline{X}_3, \overline{X}_1 \vee X_2 \vee X_3, X_1 \vee X_2 \vee X_3\}$.
Construct a graph with the terms of each clause as nodes.
Then connect all pairs of variable except their negation (partially done below)



If this graph contains a clique of size 3, the clause is satisfiable.

Some useful literature

J.A. Bondy and U.S.R. Murty, *Graph Theory with Applications*, (2nd Edition), North Holland, 1976.

Reinhard Diestel, *Graph Theory*, Graduate Texts in Mathematics, Vol. 173, Springer Verlag, Berlin, 1991.

Douglas West, *Introduction to Graph Theory*, (2nd Edition), Prentice Hall, 2000.

B. Bollobas, *Modern Graph Theory*, Springer-Verlag.

Fan Cheung and Linyuan Lu, *Complex Graphs and Networks*, Regional Conference Series in Mathematics, Vol. 107, AMS, 2004

Dieter Jungnickel, *Graphs, Networks and Algorithms*, Algorithms and Computation in Mathematics, Vol. 5, Springer Verlag, Berlin, 2005.

MATHEMATICS-I

DIFFERENTIAL EQUATIONS-II

I YEAR B.TECH

By

Y. Prabhaker Reddy
Asst. Professor of Mathematics
Guru Nanak Engineering College
Ibrahimpatnam, Hyderabad.

SYLLABUS OF MATHEMATICS-I (AS PER JNTU HYD)

Name of the Unit	Name of the Topic
Unit-I Sequences and Series	1.1 Basic definition of sequences and series 1.2 Convergence and divergence. 1.3 Ratio test 1.4 Comparison test 1.5 Integral test 1.6 Cauchy's root test 1.7 Raabe's test 1.8 Absolute and conditional convergence
Unit-II Functions of single variable	2.1 Rolle's theorem 2.2 Lagrange's Mean value theorem 2.3 Cauchy's Mean value theorem 2.4 Generalized mean value theorems 2.5 Functions of several variables 2.6 Functional dependence, Jacobian 2.7 Maxima and minima of function of two variables
Unit-III Application of single variables	3.1 Radius , centre and Circle of curvature 3.2 Evolutes and Envelopes 3.3 Curve Tracing-Cartesian Co-ordinates 3.4 Curve Tracing-Polar Co-ordinates 3.5 Curve Tracing-Parametric Curves
Unit-IV Integration and its applications	4.1 Riemann Sum 4.3 Integral representation for lengths 4.4 Integral representation for Areas 4.5 Integral representation for Volumes 4.6 Surface areas in Cartesian and Polar co-ordinates 4.7 Multiple integrals-double and triple 4.8 Change of order of integration 4.9 Change of variable
Unit-V Differential equations of first order and their applications	5.1 Overview of differential equations 5.2 Exact and non exact differential equations 5.3 Linear differential equations 5.4 Bernoulli D.E 5.5 Newton's Law of cooling 5.6 Law of Natural growth and decay 5.7 Orthogonal trajectories and applications
Unit-VI Higher order Linear D.E and their applications	6.1 Linear D.E of second and higher order with constant coefficients 6.2 R.H.S term of the form *exp(ax)* 6.3 R.H.S term of the form *sin ax and cos ax* 6.4 R.H.S term of the form exp(ax) v(x) 6.5 R.H.S term of the form exp(ax) v(x) 6.6 Method of variation of parameters 6.7 Applications on bending of beams, Electrical circuits and simple harmonic motion
Unit-VII Laplace Transformations	7.1 LT of standard functions 7.2 Inverse LT –first shifting property 7.3 Transformations of derivatives and integrals 7.4 Unit step function, Second shifting theorem 7.5 Convolution theorem-periodic function 7.6 Differentiation and integration of transforms 7.7 Application of laplace transforms to ODE
Unit-VIII Vector Calculus	8.1 Gradient, Divergence, curl 8.2 Laplacian and second order operators 8.3 Line, surface , volume integrals 8.4 Green's Theorem and applications 8.5 Gauss Divergence Theorem and applications 8.6 Stoke's Theorem and applications

CONTENTS

UNIT-6

Differential Equations-II

LINEAR DIFFERENTIAL EQUATIONS OF SECOND AND HIGHER ORDER

A D.E of the form $\frac{d^n y}{dx^n} + P_1 \frac{d^{n-1}y}{dx^{n-1}} + \ldots + P_{n-1}\frac{dy}{dx} + P_n = Q(x)$ is called as a Linear Differential Equation of order n with constant coefficients, where $P_1, P_2, \ldots, P_n$ are Real constants.

Let us denote $\frac{d}{dx} \equiv D, \frac{d^2}{dx^2} \equiv D^2, \frac{d^3}{dx^3} \equiv D^3$ etc, then above equation becomes

$(D^n + P_1 D^{n-1} + \ldots + P_{n-1}D + P_n)y = Q(x)$ which is in the form of $f(D)y = Q(x)$, where

$f(D) = (D^n + P_1 D^{n-1} + \ldots + P_{n-1}D + P_n)$.

The **General Solution** of the above equation is $y = C.F + P.I$ C.F= Complementary Function

(or) $\boldsymbol{y = y_c + y_p}$ P.I= Particular Function

Now, to find Complementary Function y_c , we have to find Auxillary Equation

Auxillary Equation: An equation of the form $f(m) = 0$ is called as an Auxillary Equation.

Since $f(m) = 0$ is a polynomial equation, by solving this we get roots. Depending upon these roots we will solve further.

Complimentary Function: The General Solution of $f(D)y = 0$ is called as Complimentary Function and it is denoted by y_c

Depending upon the Nature of roots of an Auxillary equation we can define y_c

Case I: If the Roots of the A.E are real and distinct, then proceed as follows

If α_1, α_2 are two roots which are real and distinct (different) then complementary function is given by $y_c = c_1 e^{\alpha_1 x} + c_2 e^{\alpha_2 x}$

Generalized condition: If $\alpha_1, \alpha_2, \alpha_3, \ldots, \alpha_n$ are real and distinct roots of an A.E then

$$y_c = c_1 e^{\alpha_1 x} + c_2 e^{\alpha_2 x} + c_3 e^{\alpha_3 x} + \ldots + c_n e^{\alpha_n x}$$

Case II: If the roots of A.E are real and equal then proceed as follows

If $\alpha_1 = \alpha_2 = \alpha$ then $y_c = e^{\alpha x}(c_1 + c_2 x)$

Generalized condition: If $\alpha_1 = \alpha_2, = \alpha_3, = \ldots, = \alpha_n = \alpha$ then

$$y_c = e^{\alpha x}(c_1 + c_2 x + c_3 x^2 + \cdots + c_n x^{n-1})$$

Case III: If roots of A.E are Complex conjugate i.e. $m = \alpha \pm i\beta$ then

$$y_c = e^{\alpha x}(c_1 \cos \beta x + c_2 \sin \beta x)$$

$$\text{(Or)} \quad y_c = c_1 e^{\alpha x} \cos(\beta x + c_2)$$

$$\text{(Or)} \quad y_c = c_1 e^{\alpha x} \sin(\beta x + c_2)$$

Note: For repeated Complex roots say, $m = \alpha \pm i\beta,\ \alpha \pm i\beta$

$$y_c = e^{\alpha x}[(c_1 + c_2 x)\cos\beta x + (c_3 + c_4 x)\sin\beta x]$$

Case IV: If roots of A.E are in the form of Surds i.e. $m = \alpha \pm \sqrt{\beta}$, where β is not a perfect square then,

$$y_c = e^{\alpha x}\left(c_1 \cosh\sqrt{\beta}x + c_2 \sin\sqrt{\beta}x\right)$$

$$\text{(Or)}\quad y_c = c_1 e^{\alpha x}\cos\left(\sqrt{\beta}x + c_2\right)$$

$$\text{(Or)}\quad y_c = c_1 e^{\alpha x}\sin\left(\sqrt{\beta}x + c_2\right)$$

Note: For repeated roots of surds say, $m = \alpha \pm \sqrt{\beta},\ \alpha \pm \sqrt{\beta}$

$$y_c = e^{\alpha x}\left[(c_1 + c_2 x)\cosh\sqrt{\beta}x + (c_3 + c_4 x)\sinh\sqrt{\beta}x\right]$$

Particular Integral

The evaluation of $\frac{1}{f(D)}Q(x)$ is called as Particular Integral and it is denoted by y_p

i.e. $y_p = \frac{1}{f(D)}Q(x)$

Note: The General Solution of $f(D)y = Q(x)$ is called as Particular Integral and it is denoted by y_p

Methods to find Particular Integral

Method 1: Method to find P.I of $f(D)y = Q(x)$ where $Q(x) = e^{ax}$, where a is a constant.

We know that $y_p = \frac{1}{f(D)}Q(x)$

$$= \frac{1}{f(D)}e^{ax}$$

$\therefore\ y_p = \frac{1}{f(a)}e^{ax}$ if $f(a) \neq 0$ Directly substitute a in place of D

$= e^{ax}\frac{1}{f(D+a)}$ if $f(a) = 0$ Taking e^{ax} outside the operator by replacing D with $D + a$

Depending upon the nature of $f(D + a)$ we can proceed further.

Note: while solving the problems of the type $\frac{1}{f(D)}Q(x)$, where Denominator =0, Rewrite the Denominator quantity as product of factors, and then keep aside the factor which troubles us. I.e the term which makes the denominator quantity zero, and then solve the remaining quantity. finally substitute $D + a$ in place of D.

Method 2: Method to find P.I of $f(D)y = Q(x)$ where $Q(x) = \sin ax\ (or) \cos ax$, a is constant

We know that $y_p = \frac{1}{f(D)} Q(x)$

$= \frac{1}{f(D)} \sin ax\ (or)\ \frac{1}{f(D)} \cos ax$

Let us consider $f(D) = \emptyset(D^2)$, then the above equation becomes

$\therefore\ y_p = \frac{1}{\emptyset(D^2)} \sin ax\ (or)\ \frac{1}{\emptyset(D^2)} \cos ax$

Now Substitute $D^2 = -a^2$ if $\emptyset(D^2) \neq 0$

If $\emptyset(D^2) = 0$ then i.e. $y_p = \frac{1}{D^2+a^2} \sin ax\ (or)\ \frac{1}{D^2+a^2} \cos ax$

Then $y_p = \frac{x}{2} \int \sin ax\ dx\ \ (or)\ \frac{x}{2} \int \cos ax\ dx$ respectively.

Method 3: Method to find P.I of $f(D)y = Q(x)$ where $Q(x) = x^k, k \in \mathbb{Z}^+$

We know that $y_p = \frac{1}{f(D)} Q(x)$

$= \frac{1}{f(D)} x^k$

Now taking Lowest degree term as common in $f(D)$, above relation becomes $y_p = \frac{1}{[1+\emptyset(D)]} x^k$

$\Rightarrow\ y_p = [1 + \emptyset(D)]^{-1} x^k$

Expanding this relation upto k^{th} derivative by using Binomial expansion and hence get y_p

Important Formulae:

1) $(1 - D)^{-1} = 1 + D + D^2 + \ \dots$
2) $(1 + D)^{-1} = 1 - D + D^2 - \ \dots$
3) $(1 - D)^{-2} = 1 + 2D + 3D^2 + \ \dots$
4) $(1 + D)^{-2} = 1 - 2D + 3D^2 - \ \dots$
5) $(1 - D)^{-3} = 1 + 3D + 6D^2 + \ \dots$
6) $(1 + D)^{-3} = 1 - 3D + 6D^2 - \ \dots$

Method 4: Method to find P.I of $f(D)y = Q(x)$ where $Q(x) = e^{ax} V$, where V is a function of x and a is constant

We know that $y_p = \frac{1}{f(D)} Q(x)$

$= \frac{1}{f(D)} e^{ax} V$

In such cases, first take e^{ax} term outside the operator, by substituting $D + a$ in place of D.

$$\Rightarrow\ y_p = e^{ax} \frac{1}{f(D+a)} V$$

Depending upon the nature of V we will solve further.

Method 5: Method to find P.I of $f(D)y = Q(x)$ where $Q(x) = x^k.v$, where $k \epsilon \mathbb{Z}^+$, v is any function of x (i.e. $v = sin\ ax$ $(or) cos\ ax$)

$$e^{i\theta} = \underset{R.P}{\cos\theta} + i\underset{I.P}{\sin\theta}$$

$$\cos\theta = R.P\left(e^{i\theta}\right)$$

$$\sin\theta = I.P\left(e^{i\theta}\right)$$

We know that $y_p = \frac{1}{f(D)}Q(x)$

$$= \frac{1}{f(D)}\ x^k.v$$

Case I: Let $k = 1$, then $y_p = \left[x - \frac{f'(D)}{f(D)}\right]\frac{1}{f(D)}v$

Case II: Let $k \neq 1$ and $v = \sin ax$

$$y_p = \frac{1}{f(D)}x^k \sin ax$$

We know that $e^{i\theta} = \cos\theta + i\sin\theta$

$$y_p = \frac{1}{f(D)}\ x^k\ I.P(e^{iax})$$

$$= I.P\ \frac{1}{f(D)}\ x^k e^{iax}$$

$$= I.P\ e^{iax}\ \frac{1}{f(D+ia)}\ x^k$$

By using previous methods we will solve further

Finally substitute $e^{iax} = \cos ax + i\sin ax$

Let $k \neq 1$ and $v = \sin ax$

$$y_p = \frac{1}{f(D)}x^k \cos ax$$

We know that $e^{i\theta} = \cos\theta + i\sin\theta$

$$y_p = \frac{1}{f(D)}\ x^k\ R.P(e^{iax})$$

$$= R.P\ \frac{1}{f(D)}\ x^k e^{iax}$$

$$= R.P\ e^{iax}\ \frac{1}{f(D+ia)}\ x^k$$

By using previous methods we will solve further

Finally substitute $e^{iax} = \cos ax + i\sin ax$

General Method

To find P.I of $f(D)y = Q(x)$ where $Q(x)$ is a function of x

We know that $y_p = \frac{1}{f(D)}Q(x)$

Let $f(D) = (D - \alpha)$ then $y_p = \frac{1}{(D-\alpha)}Q(x)$

$$= e^{\alpha x}\int e^{-\alpha x}\,Q(x)\,dx$$

Similarly, $f(D) = (D + \alpha)$ then $y_p = \frac{1}{(D+\alpha)}Q(x)$

$$= e^{-\alpha x}\int e^{\alpha x}\,Q(x)\,dx$$

Note: The above method is used for the problems of the following type

- $(D^2 - 3D + 2)y = \sin(e^{-x})$
- $(D^2 + a^2)y = \sec ax$
- $(D^2 + a^2)y = \tan ax$
- $(D^2 + a^2)y = \text{cosec}\, ax$

Cauchy's Linear Equations (or) Homogeneous Linear Equations

A Differential Equation of the form $[x^n D_n + A_1 x^{n-1} D_{n-1} + \ \dots + A_{n-1} x D + A_n] y = Q(x)$ where $D \equiv \frac{d}{dx}$ is called as n^{th} order Cauchy's Linear Equation in terms of dependent variable y and independent variable x, where $A_1, A_2, A_3, \dots, A_n$ are Real constants and $D \equiv \frac{d}{dx}$.

Substitute $x = e^z \implies \log x = z$ and

$xD = \theta, x^2D^2 = \theta(\theta - 1), x^3D^3 = \theta(\theta - 1)(\theta - 2), \dots \quad \theta \equiv \frac{d}{dz}$

Then above relation becomes $(\theta)y = Q(z)$, which is a Linear D.E with constant coefficients. By using previous methods, we can find Complementary Function and Particular Integral of it, and hence by replacing z with $\log x$ we get the required General Solution of Cauchy's Linear Equation.

Legendre's Linear Equation

An D.E of the form $[(ax + b)^n D_n + A_1(ax + b)^{n-1} D_{n-1} + \ \dots + A_{n-1}(ax + b)D + A_n] y = Q(x)$ is called as Legendre's Linear Equation of order , where $a, b, A_1, A_2, A_3, \dots, A_n$ are Real constants.

Now substituting, $(ax + b) = e^z \implies z = \log(ax + b)$

$(ax + b)D = a\theta, (ax + b)^2D^2 = a^2\theta(\theta - 1), (ax + b)^3D^3 = a^3\theta(\theta - 1)(\theta - 2), \dots \quad \theta \equiv \frac{d}{dz}$

Then, above relation becomes $f(\theta)y = Q(z)$ which is a Linear D.E with constant coefficients. By using previous methods we can find general solution of it and hence substituting $z = \log(ax + b)$ we get the general solution of Legendre's Linear Equation.

Method of Variation of Parameters

To find the general solution of $\frac{d^2y}{dx^2} + P\frac{dy}{dx} + Qy = R(x)$

Let us consider given D.E $\frac{d^2y}{dx^2} + P\frac{dy}{dx} + Qy = R(x) \longrightarrow$ **(I)**

Let the Complementary Function of above equation is $y_c = c_1 u + c_2 v$

Let the Particular Integral of it is given by $y_p = Au + Bv$, where

$$A = \int \frac{-vR}{uv' - vu'} dx \qquad B = \int \frac{uR}{uv' - vu'}$$

* * *

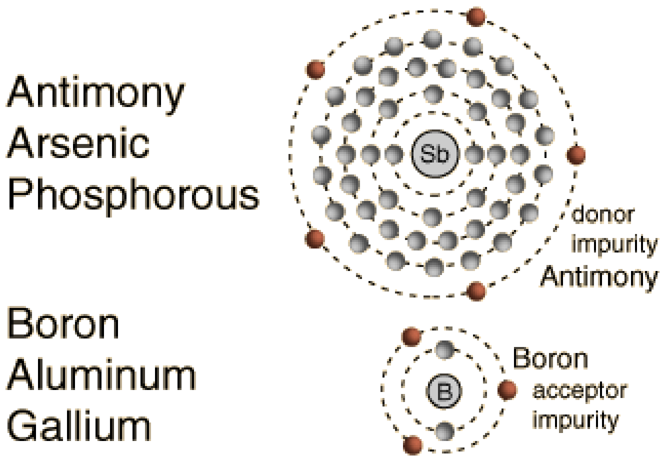
Doped Semiconductors

The Doping of Semiconductors

The addition of a small percentage of foreign atoms in the regular crystal lattice of silicon or germanium produces dramatic changes in their electrical properties, producing n-type and p-type semiconductors.

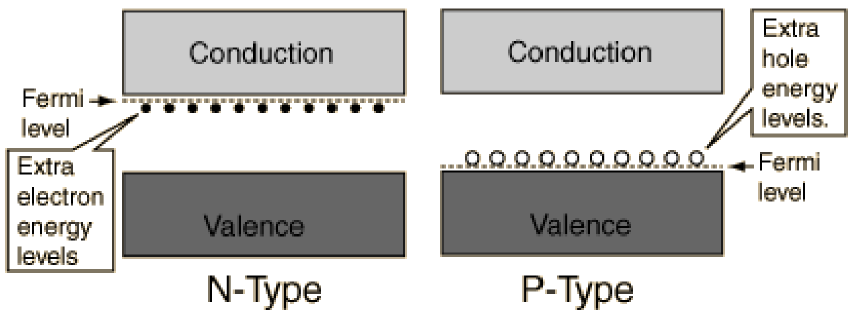
Pentavalent impurities
Impurity atoms with 5 valence electrons produce n-type semiconductors by contributing extra electrons.

Trivalent impurities
Impurity atoms with 3 valence electrons produce p-type semiconductors by producing a "hole" or electron deficiency.



Bands for Doped Semiconductors

The application of band theory to n-type and p-type semiconductors shows that extra levels have been added by the impurities. In n-type material there are electron energy levels near the top of the band gap so that they can be easily excited into the conduction band. In p-type material, extra holes in the band gap allow excitation of valence band electrons, leaving mobile holes in the valence band.



Lecture 1

First Steps in Graph Theory

This lecture introduces Graph Theory, the main subject of the course, and includes some basic definitions as well as a number of standard examples.

Reading: Some of the material in today's lecture comes from the beginning of Chapter 1 in

> Dieter Jungnickel (2008), *Graphs, Networks and Algorithms*, 3rd edition, which is available online via SpringerLink.

If you are at the university, either physically or via the VPN, you can download the chapters of this book as PDFs.

1.1 The Königsberg Bridge Problem

Graph theory is usually said to have been invented in 1736 by the great Leonhard Euler, who used it to solve the Königsberg Bridge Problem. I used to find this hard to believe—the graph-theoretic graph is such a natural and useful abstraction that it's difficult to imagine that no one hit on it earlier—but Euler's paper about graphs[1] is generally acknowledged[2] as the first one and it certainly provides a satisfying solution to the bridge problem. The sketch in the left panel of Figure 1.1 comes from Euler's original paper and shows the main features of the problem. As one can see by comparing Figures 1.1 and 1.2, even this sketch is already a bit of an abstraction.

The question is, can one make a walking tour of the city that (a) starts and finishes in the same place and (b) crosses every bridge exactly once. The short answer to this question is "No" and the key idea behind proving this is illustrated in the right panel of Figure 1.1. It doesn't matter what route one takes while walking around on, say, the smaller island: all that really matters are the ways in which the bridges connect the four land masses. Thus we can shrink the small island to a

[1]L. Euler (1736), Solutio problematis ad geometriam situs pertinentis, *Commentarii Academiae Scientiarum Imperialis Petropolitanae* **8**, pp. 128–140.

[2] See, for example, Robin Wilson and John J. Watkins (2013), *Combinatorics: Ancient & Modern*, OUP. ISBN 978-0-19-965659-2.

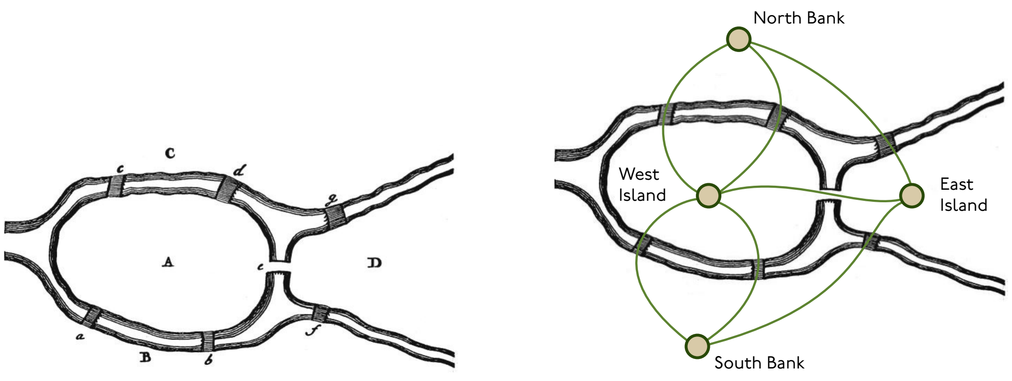


Figure 1.1: *The panel at left shows the seven bridges and four land masses that provide the setting for the Königsberg bridge problem, which asks whether it is possible to make a circular walking tour of the city that crosses every bridge exactly once. The panel at right includes a graph-theoretic abstraction that helps one prove that no such tour exists.*

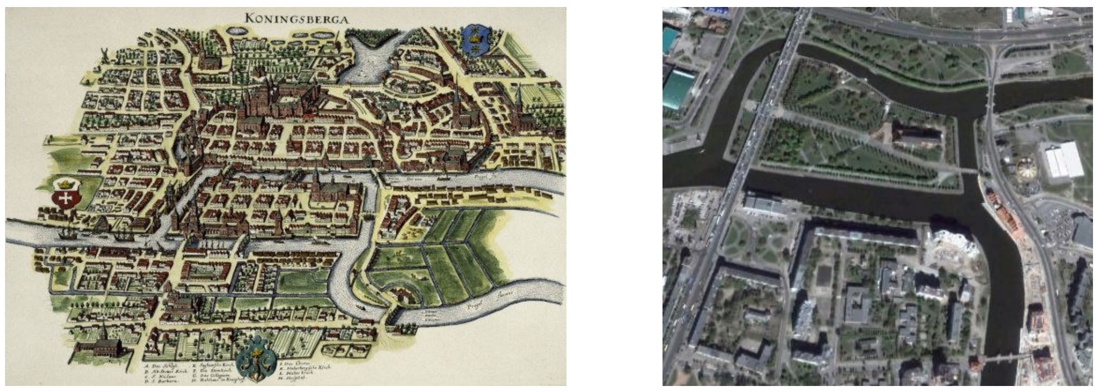


Figure 1.2: *Königsberg is a real place—a port on the Baltic—and during Euler's lifetime it was part of the Kingdom of Prussia. The panel at left is a bird's-eye view of the city that shows the celebrated seven bridges. It was made by Matthäus Merian and published in 1652. The city is now called Kaliningrad and is part of the Russian Federation. It was bombed heavily during the Second World War: the panel at right shows a recent satellite photograph and one can still recognize the two islands and modern versions of some of the bridges, but very little else appears to remain.*

point—and do the same with the other island, as well as with the north and south banks of the river—and then connect them with arcs that represent the bridges. The problem then reduces to the question whether it is possible to draw a path that starts and finishes at the same dot, but traces each of over the seven arcs exactly once.

One can prove that such a tour is impossible by contradiction. Suppose that one exists: it must then visit the easternmost island (see Figure 1.3) and we are free to imagine that the tour actually starts there. To continue we must leave the island, crossing one of its three bridges. Then, later, because we are required to

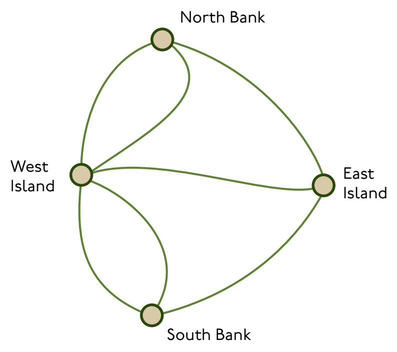


Figure 1.3: *The Königsberg Bridge graph on its own: it is not possible to trace a path that starts and ends on the eastern island without crossing some bridge at least twice.*

cross each bridge exactly once, we will have to return to the eastern island via a different bridge from the one we used when setting out. Finally, having returned to the eastern island once, we will need to leave again in order to cross the island's third bridge. But then we will be unable to return without recrossing one of the three bridges. And this provides a contradiction: the walk is supposed to start and finish in the same place and cross each bridge exactly once.

1.2 Definitions: graphs, vertices and edges

The abstraction behind Figure 1.3 turns out to be very powerful: one can draw similar diagrams to represent "connections" between "things" in a very general way. Examples include: representations of social networks in which the points are people and the arcs represent acquaintance; genetic regulatory networks in which the points are genes and the arcs represent activation or repression of one gene by another and scheduling problems in which the points are tasks that contribute to some large project and the arcs represent interdependence among the tasks. To help us make more rigorous statements, we'll use the following definition:

Definition. *A **graph** is a finite, nonempty set V, the **vertex set**, along with a set E, the **edge set**, whose elements $e \in E$ are pairs $e = (a, b)$ with $a, b \in V$.*

We will often write $G(V, E)$ to mean the graph G with vertex set V and edge set E. An element $v \in V$ is called a *vertex* (plural *vertices*) while an element $e \in E$ is called an *edge*.

The definition above is deliberately vague about whether the pairs that make up the edge set E are ordered pairs—in which case (a, b) and (b, a) with $a \neq b$ are distinct edges—or unordered pairs. In the unordered case (a, b) and (b, a) are just two equivalent ways of representing the same pair.

Definition. *An **undirected graph** is a graph in which the edge set consists of unordered pairs.*

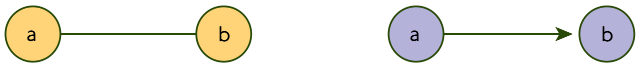


Figure 1.4: *Diagrams representing graphs with vertex set $V = \{a, b\}$ and edge set $E = \{(a, b)\}$. The diagram at left is for an undirected graph, while the one at right shows a directed graph. Thus the arrow on the right represents the ordered pair (a, b).*

Definition. *A **directed graph** is a graph in which the edge set consists of ordered pairs. The term "directed graph" is often abbreviated as **digraph**.*

Although graphs are defined abstractly as above, it's very common to draw *diagrams* to represent them. These are drawings in which the vertices are shown as points or disks and the edges as line segments or arcs. Figure 1.4 illustrates the graphical convention used to mark the distinction between directed and undirected edges: the former are drawn as line segments or arcs, while the latter are shown as arrows. A directed edge $e = (A, B)$ appears as an arrow that points from A to B.

Sometimes one sees graphs with more than one edge[3] connecting the same two vertices; the Königsberg Bridge graph is an example. Such edges are called *multiple* or *parallel* edges. Additionally, one sometimes sees graphs with edges of the form $e = (v, v)$. These edges, which connect a vertex to itself, are called *loops* or *self loops.* All these terms are illustrated in Figure 1.5.

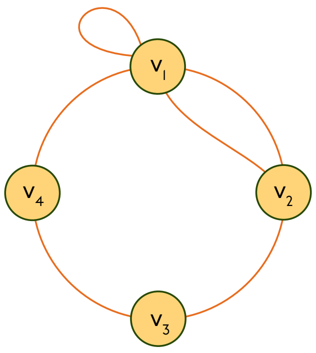


Figure 1.5: *A graph whose edge set includes the self loop (v_1, v_1) and two parallel copies of the edge (v_1, v_2).*

It is important to bear in mind that diagrams such as those in Figures 1.3–1.5 are only illustrations of the edges and vertices. In particular, the arcs representing edges may cross, but this does not necessarily imply anything: see Figure 1.6.

Remark. In this course when we say "graph" we will normally mean an undirected graph that contains no loops or parallel edges: if you look in other books you may

[3]In this case it is a slight abuse of terminology to talk about the edge "set" of the graph, as sets contain only a single copy of each of their elements. Very scrupulous books (and students) might prefer to use the term *edge list* in this context, but I will not insist on this nicety.

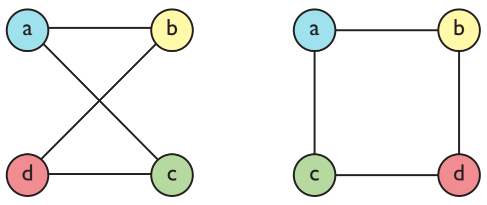


Figure 1.6: *Two diagrams for the same graph: the crossed edges in the leftmost version do not signify anything.*

see such objects referred to as *simple graphs*. By contrast, we will refer to a graph that contains parallel edges as a *multigraph*.

Definition. *Two vertices $a \neq b$ in an undirected graph $G(V, E)$ are said to be* ***adjacent*** *or to be* ***neighbours*** *if $(a, b) \in E$. In this case we also say that the edge $e = (a, b)$ is* ***incident on*** *the vertices a and b.*

Definition. *If the directed edge $e = (u, v)$ is present in a directed graph $H(V', E')$ we will say that u is a* ***predecessor*** *of v and that v is a* ***successor*** *of u. We will also say that u is the* ***tail*** *or* ***tail vertex*** *of the edge (u, v), while v is the* ***tip*** *or* ***tip vertex****.*

1.3 Standard examples

In this section I'll introduce a few families of graphs that we will refer to throughout the rest of the term.

The complete graphs K_n

The *complete graph* K_n is the undirected graph on n vertices whose edge set includes every possible edge. If one numbers the vertices consecutively the edge and vertex set are

$$\begin{aligned} V &= \{v_1, v_2, \ldots, v_n\} \\ E &= \{(v_j, v_k) \mid 1 \leq j \leq (n-1), (j+1) \leq k \leq n\}. \end{aligned}$$

There are thus

$$|E| = \binom{n}{2} = \frac{n(n-1)}{2}$$

edges in total: see Figure 1.7 for the first few examples.

The path graphs P_n

These graphs are formed by stringing n vertices together in a path. The word "path" actually has a technical meaning in graph theory, but you needn't worry about that

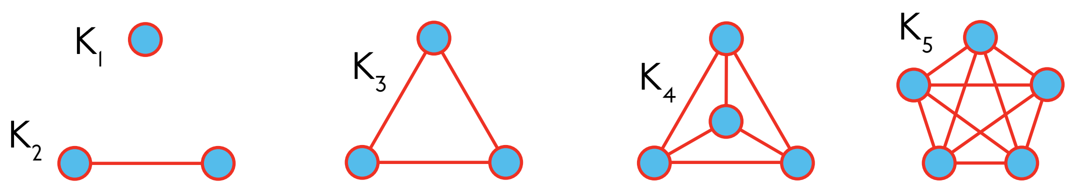


Figure 1.7: *The first five members of the family K_n of complete graphs.*

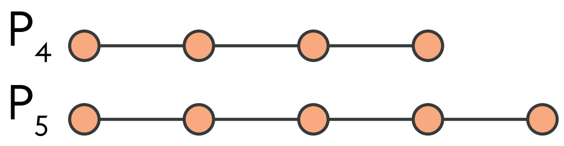


Figure 1.8: *Diagrams for the path graphs P_4 and P_5.*

today. P_n has vertex and edge sets as listed below,

$$\begin{aligned} V &= \{v_1, v_2, \ldots, v_n\} \\ E &= \{(v_j, v_{j+1}) \,|\, 1 \le j < n\}, \end{aligned}$$

and Figure 1.8 shows two examples.

The cycle graphs C_n

The *cycle graph* C_n, sometimes also called the *circuit graph*, is a graph in which $n \ge 3$ vertices are arranged in a ring. If one numbers the vertices consecutively the edge and vertex set are

$$\begin{aligned} V &= \{v_1, v_2, \ldots, v_n\} \\ E &= \{(v_1, v_2), (v_2, v_3), \ldots, (v_j, v_{j+1}), \ldots, (v_{n-1}, v_n), (v_n, v_1)\}. \end{aligned}$$

C_n has n edges that are often written (v_j, v_{j+1}), where the subscripts are taken to be defined periodically so that, for example, $v_{n+1} \equiv v_1$. See Figure 1.9 for examples.

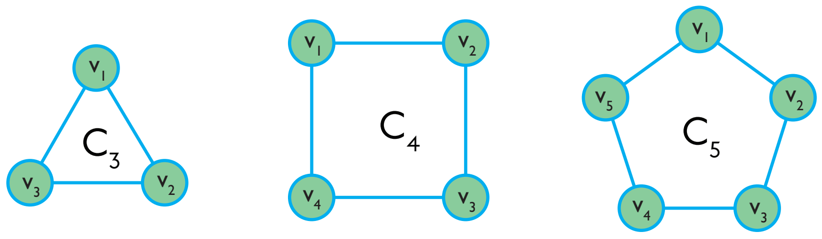


Figure 1.9: *The first three members of the family C_n of cycle graphs.*

The complete bipartite graphs $K_{m,n}$

The *complete bipartite graph* $K_{m,n}$ is a graph whose vertex set is the union of a set V_1 of m vertices with second set V_2 of n different vertices and whose edge set includes every possible edge running between these two subsets:

$$\begin{aligned} V &= V_1 \cup V_2 \\ &= \{u_1, \ldots, u_m\} \cup \{v_1, \ldots, v_n\} \\ E &= \{(u,v) \,|\, u \in V_1,\, v \in V_2\}. \end{aligned}$$

$K_{m,n}$ thus has $|E| = mn$ edges: see Figure 1.10 for examples.

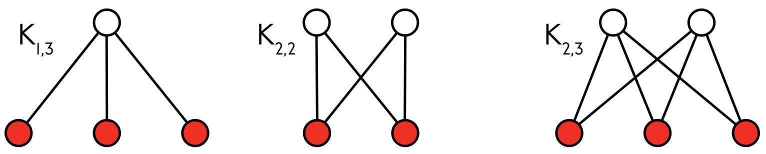


Figure 1.10: *A few members of the family $K_{m,n}$ of complete bipartite graphs. Here the two subsets of the vertex set are illustrated with colour: the white vertices constitute V_1, while the red ones form V_2.*

There are other sorts of bipartite graphs too:

Definition 1.1. *A graph $G(V, E)$ is said to be a* ***bipartite graph*** *if*

- *it has a nonempty edge set: $E \neq \emptyset$ and*
- *its vertex set V can be decomposed into two nonempty, disjoint subsets*

$$V = V_1 \cup V_2 \;\; \textit{with} \;\; V_1 \cap V_2 = \emptyset \;\; \textit{and} \;\; V_1 \neq \emptyset \;\; \textit{and} \;\; V_2 \neq \emptyset$$

in such a way that all the edges in E connect a member of V_1 with a member of V_2. That is, we need

$$(u,v) \in E \Rightarrow \begin{cases} u \in V_1 \textit{ and } v \in V_2 \\ \textit{or } u \in V_2 \textit{ and } v \in V_1. \end{cases}$$

The cube graphs I_d

These graphs are specified in a way that's closer to the purely combinatorial, set-theoretic definition of a graph given above. I_d, the *d-dimensional cube graph*, has vertices that are strings of d zeroes or ones, and all possible labels occur. Edges connect those vertices whose labels differ in exactly one position. Thus, for example, I_2 has vertex and edge sets

$$V = \{00,\, 01,\, 10,\, 11\} \qquad \text{and} \qquad E = \{(00,01),\, (00,10),\, (01,11),\, (10,11)\}.$$

Figure 1.11 shows diagrams for the first few cube graphs and these go a long way toward explaining the name. More generally, I_d has vertex and edge sets given by

$$\begin{aligned} V &= \{w \,|\, w \in \{0,1\}^d\} \\ E &= \{(w, w') \,|\, w \text{ and } w' \text{ differ in a single position}\}. \end{aligned}$$

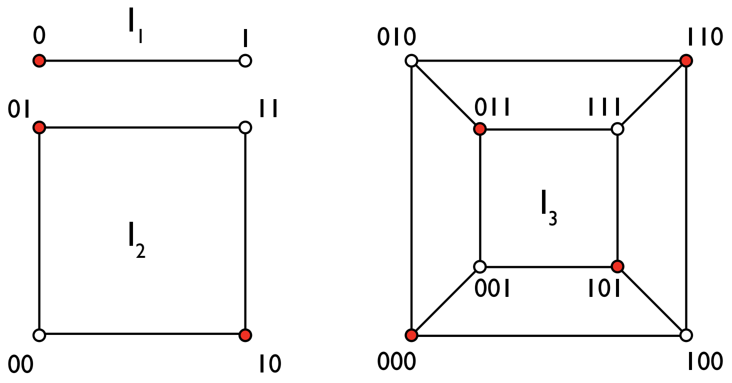


Figure 1.11: *The first three members of the family I_d of cube graphs. Notice that all the cube graphs are bipartite (the red and white vertices are the two disjoint subsets from Definition 1.1), but that, for example, I_3 is* not *a complete bipartite graph.*

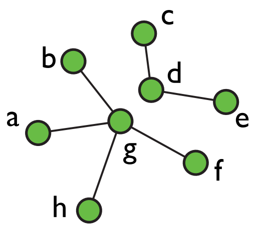
This means that I_d has $|V| = 2^d$ vertices, but it's a bit harder to count the edges. In the last part of today's lecture we'll prove a theorem that enables one to show that I_d has $|E| = d\,2^{d-1}$ edges.

1.4 A first theorem about graphs

I find it wearisome to give, or learn, one damn definition after another and so I'd like to conclude the lecture with a small, but useful theorem. To do this we need one more definition:

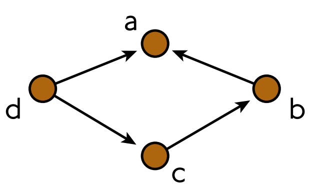
Definition. *In an undirected graph $G(V, E)$ the* ***degree*** *of a vertex $v \in V$ is the number of edges that include the vertex. One writes* $\deg(v)$ *for "the degree of v".*

So, for example, every vertex in the complete graph K_n has degree $n - 1$, while every vertex in a cycle graph C_n has degree 2; Figure 1.12 provides more examples. The generalization of degree to directed graphs is slightly more involved. A vertex v in a digraph has two degrees: an *in-degree* that counts the number of edges having v at their tip and an *out-degree* that counts number of edges having v at their tail. See Figure 1.13 for an example.



v	a	b	c	d	e	f	g	h
$\deg(v)$	1	1	1	2	1	1	4	1

Figure 1.12: *The degrees of the vertices in a small graph. Note that the graph consists of two "pieces".*



v	$\deg_{in}(v)$	$\deg_{out}(v)$
a	2	0
b	1	1
c	1	1
d	0	2

Figure 1.13: *The degrees of the vertices in a small digraph.*

Once we have the notion of degree, we can formulate our first theorem:

Theorem 1.2 (Handshaking Lemma, Euler 1736)**.** *If $G(V, E)$ is an undirected graph then*

$$\sum_{v \in V} \deg(v) = 2|E|. \tag{1.1}$$

Proof. Each edge contributes twice to the sum of degrees, once for each of the two vertices on which it is incident. □

The following two results are immediate consequences:

Corollary 1.3. *In an undirected graph there must be an even number of vertices that have odd degree.*

Corollary 1.4. *The cube graph I_d has $|E| = d\,2^{d-1}$.*

The first is fairly obvious: the right hand side of (1.1) is clearly an even number, so the sum of degrees appearing on the left must be even as well. To get the formula for the number of edges in I_d, note that it has 2^d vertices, each of degree d, so the Handshaking Lemma tells us that

$$2|E| \;=\; \sum_{v \in V} \deg(v) \;=\; 2^d \times d$$

and thus $|E| = (d \times 2^d)/2 = d\,2^{d-1}$.

Lesson 31

Linear Differential Equation of Higher Order

In connection to the last lesson, we discuss solution methodologies of getting particular integral of the linear differential equations of higher order. In particular, in this lesson we present operator method which is somewhat easier than other methods for finding particular integrals.

31.1 Determination of Particular Integral (P.I.)

As we have seen in the earlier lesson that a general nonhomogeneous linear differential equations with constant coefficients can be written in operator form as $f(D)y = F(x)$. The operator, $1/f(D)$ is called inverse operator which gives a particular integral when operated on both the sides of the given differential equation. Hence, a particular integral of the given differential equation is given as $\frac{1}{f(D)}F(x)$. First we give a rather general idea of getting a particular integral with this method and then state some other useful direct results. Note that the operator $f(D)$ can be expressed as $(D-\alpha_1)(D-\alpha_2)\dots(D-\alpha_n)$ and thus a particular integral is given as

$$\frac{1}{f(D)}F(x) = \frac{1}{D-\alpha_1}\frac{1}{D-\alpha_2}\cdots\frac{1}{D-\alpha_n}F(x) \tag{31.1}$$

We give a general idea of evaluating an expression of the type $\dfrac{1}{D-\alpha}F(x)$. This procedure can be repeatedly applied to find a particular integral (31.1). However, applicability of this method depends upon the form of $F(x)$.

We give a general theorem that can be applied to any problem for finding particular integral of a differential equation.

31.1.1 Theorem 1

If $F(x)$ is function of x and α is a constant, then

$$\frac{1}{D-\alpha}F(x) = e^{\alpha x}\int F(x)e^{-\alpha x}dx.$$

Proof: Let us assume that

$$y = \frac{1}{D-\alpha}F(x)$$

On operating $(D-\alpha)$ both sides, we get

$$(D-\alpha)y = F(x) \quad \Rightarrow \quad \frac{dy}{dx} - \alpha y = F(x)$$

The above equation is a linear differential equation of first order whose integrating factor is $e^{-\int \alpha dx} = e^{-\alpha x}$. Hence, the solution is given by

$$ye^{-\alpha x} = \int F(x)e^{-\alpha x}dx \quad \Rightarrow \quad y = e^{\alpha x}\int F(x)e^{-\alpha x}dx$$

Since our interest is finding a particular integrals, the constant of integration is dropped. Thus,

$$\frac{1}{D-\alpha}F(x) = e^{\alpha x}\int F(x)e^{-\alpha x}dx. \qquad \blacksquare$$

Now we state some useful result those will be used to find P.I. of certain special forms of $F(x)$.

31.1.2 Theorem 2

If α is a constant, then $f(D)e^{\alpha x} = f(\alpha)e^{\alpha x}$

Proof: We know that $De^{\alpha x} = \alpha e^{\alpha x}$ and similarly $D^2e^{\alpha x} = \alpha^2e^{\alpha x}$. With induction we can prove that $D^ne^{\alpha x} = \alpha^ne^{\alpha x}$ for any natural number n. This proves the result $f(D)e^{\alpha x} = f(\alpha)e^{\alpha x}$. $\blacksquare$

31.1.3 Theorem 3

If α is a constant and $g(x)$ is any function, then $f(D)\left(e^{\alpha x}g(x)\right) = e^{\alpha x}f(D+\alpha)g(x)$

Proof: We know that $D\left(e^{\alpha x}g(x)\right) = \alpha e^{\alpha x}g(x) + e^{\alpha x}Dg(x) = e^{\alpha x}(\alpha + D)g(x)$. Similar to the proof of previous theorem we can prove with induction that $D^ne^{\alpha x}g(x) = e^{\alpha x}(\alpha + D)^ng(x)$ for any natural number n. This proves the result $f(D)\left(e^{\alpha x}g(x)\right) = e^{\alpha x}f(D+\alpha)g(x)$. This result is known as shifting property of operator $f(D)$. $\blacksquare$

31.1.4 Theorem 4

If α and β are arbitrary constants, then

$$f(D^2)\sin(\alpha x+\beta) = f(-\alpha^2)\sin(\alpha x+\beta) \quad \textit{and} \quad f(D^2)\cos(\alpha x+\beta) = f(-\alpha^2)\cos(\alpha x+\beta)$$

Proof: It can easily be verified that $D^2\sin(\alpha x+\beta) = -\alpha^2\sin(\alpha x+\beta)$ and $D^2\cos(\alpha x+\beta) = -\alpha^2\cos(\alpha x+\beta)$. In other words, we can replace D^2 by $-\alpha^2$ and this proves the desired result. ■

Now we describe the method for some special form of $F(x)$.

31.2 Rule I: $F(x)$ is of the form e^{ax}

We know from Theorem 31.1.2 that $f(D)e^{\alpha x} = f(\alpha)e^{\alpha x}$. Operating on both sides by $1/f(D)$ we get

$$e^{\alpha x} = \frac{1}{f(D)}f(\alpha)e^{\alpha x} \quad \Rightarrow \quad e^{\alpha x} = f(\alpha)\frac{1}{f(D)}e^{\alpha x}$$

This implies that

$$\frac{1}{f(D)}e^{\alpha x} = \frac{1}{f(\alpha)}e^{\alpha x}, \quad \text{provided} \quad f(\alpha)\neq 0$$

If $f(\alpha)=0$, then $(D-\alpha)$ is a factor of $f(D)$, say $f(D)=(D-\alpha)g(D)$. Then

$$\frac{1}{f(D)}e^{\alpha x} = \frac{1}{(D-\alpha)}\frac{1}{g(D)}e^{\alpha x} = \frac{1}{(D-\alpha)}\frac{1}{g(\alpha)}e^{\alpha x} \quad \text{provided} \quad g(\alpha)\neq 0$$

Now using Theorem 31.1.1, we get

$$\frac{1}{f(D)}e^{\alpha x} = \frac{1}{g(\alpha)}\frac{1}{(D-\alpha)}e^{\alpha x} = \frac{1}{g(\alpha)}e^{\alpha x}x$$

In case $g(\alpha)=0$ then , say $f(D)=(D-\alpha)^2h(D)$. In this case we get

$$\frac{1}{f(D)}e^{\alpha x} = \frac{1}{h(\alpha)}\frac{1}{(D-\alpha)^2}e^{\alpha x} = \frac{1}{g(\alpha)}\frac{x^2}{2!}e^{\alpha x} \quad \text{provided} \quad h(\alpha)\neq 0$$

Again, if $h(\alpha)=0$, the same procedure can be repeated. To conclude, we have the following results:

(i) $\frac{1}{f(D)}e^{\alpha x} = \frac{1}{f(\alpha)}e^{\alpha x}$, where $f(\alpha)\neq 0$

(ii) If $f(\alpha) = 0$, then $f(D)$ must posses a factor of the type $(D-\alpha)^r$, say $f(D) = (D-\alpha)^r g(D)$ where $g(\alpha) \neq 0$. Then the following formula is applicable

$$\frac{1}{(D-\alpha)^r} e^{\alpha x} = \frac{x^r}{r!} e^{\alpha x}$$

.

31.3 Example Problems

31.3.1 Problem 1

Find the general solution of the differential equation $(D^2 - 3D + 2)y = e^{3x}$.

Solution: The auxiliary equation is

$$(m^2 - 3m + 2) = 0 \quad \Rightarrow \quad (m-1)(m-2) = 0 \quad \Rightarrow \quad m = 1, 2.$$

The complimentary function is given as

$$\text{C.F.} = c_1 e^x + c_2 e^{2x}$$

The particular integral is

$$\text{P.I.} = \frac{1}{D^2 - 3D + 2} e^{3x} = \frac{1}{3^2 - 3.3 + 2} e^{3x} = \frac{1}{2} e^{3x}.$$

The general solution is: $y = c_1 e^x + c_2 e^{2x} + \dfrac{1}{2} e^{3x}$.

31.3.2 Problem 2

Solve $(4D^2 - 12D + 9)y = 144e^{3x/2}$

Solution: The auxiliary equation is

$$(4m^2 - 12m + 9) = 0 \quad \Rightarrow \quad m = 3/2, 3/2.$$

The complimentary function is

$$\text{C.F.} = (c_1 + c_2 x) e^{3x/2}$$

The particular integral is

$$\text{P.I.} = \frac{144}{(2D-3)^2} e^{3x/2} = \frac{144}{4} \frac{1}{(D-3/2)^2} e^{3x/2} = 36 \frac{x^2}{2!} e^{3x/2}$$

The required solution is: $y = (c_1 + c_2 x) e^{3x/2} + 36\frac{x^2}{2!} e^{3x/2}$.

31.4 Rule II: $F(x)$ is of the form $\cos ax$ or $\sin ax$

We express $f(D)$ as a function of D^2, say $f(D) = \phi(D^2)$. From Theorem 31.1.4 we know that $\phi(D^2)\sin(\alpha x+\beta) = \phi(-\alpha^2)\sin(\alpha x+\beta)$. Applying $[\phi(D^2)]^{-1}$ both sides we obtain

$$\sin(\alpha x+\beta) = \frac{1}{\phi(D^2)}\phi(-\alpha^2)\sin(\alpha x+\beta)$$

If $\phi(-\alpha^2)\neq 0$, we can divide the above equation by $\phi(-\alpha^2)$ to get

$$\frac{1}{\phi(D^2)}\sin(\alpha x+\beta) = \frac{1}{\phi(-\alpha^2)}\sin(\alpha x+\beta)$$

Similarly,

$$\frac{1}{\phi(D^2)}\cos(\alpha x+\beta) = \frac{1}{\phi(-\alpha^2)}\cos(\alpha x+\beta), \quad \text{provided} \quad \phi(-\alpha^2)\neq 0$$

In case, $\phi(-\alpha^2) = 0$, we can rewrite $\sin(\alpha x+\beta) = \mathrm{Im}(e^{i(\alpha x+\beta)})$ and $\cos(\alpha x+\beta) = \mathrm{Re}(e^{i(\alpha x+\beta)})$. Now case I can be applied as

$$\frac{1}{f(D)}\sin(\alpha x+\beta) = \mathrm{Im}\left(\frac{1}{f(D)}e^{i(\alpha x+\beta)}\right) = \mathrm{Im}\left(\frac{1}{f(i\alpha)}e^{i(\alpha x+\beta)}\right) \quad \text{provided} \quad f(i\alpha)\neq 0$$

Similarly,

$$\frac{1}{f(D)}\cos(\alpha x+\beta) = \mathrm{Re}\left(\frac{1}{f(i\alpha)}e^{i(\alpha x+\beta)}\right) \quad \text{provided} \quad f(i\alpha)\neq 0$$

31.5 Example Problems

31.5.1 Problem 1

Solve the differential equation $(D^2+1)y = \cos 2x$.

Solution: The characteristic equation of the corresponding homogeneous equation is

$$(m^2+1) = 0 \quad \Rightarrow \quad m = \pm i$$

Hence, C.F. $= (c_1\cos x + c_2\sin x)$. The particular integral is given by

$$\text{P.I.} = \frac{1}{D^2+1}\cos 2x = \frac{1}{(-2^2+1)}\cos 2x = \frac{1}{-3}\cos 2x.$$

The required solution is: $y = (c_1\cos x + c_2\sin x) - \dfrac{1}{3}\cos 2x$.

31.5.2 Problem 2

Solve the differential equation $(D^2 - 4D + 3)y = \sin x.$

Solution: The roots of the characteristic equations are 1 and 3. The complementary function is $C.F. = c_1 e^x + c_2 e^{3x}$. The particular integral is

$$\text{P.I.} = \frac{1}{D^2 - 4D + 3} \sin x$$

Replacing D^2 by -1, we get

$$\text{P.I.} = \frac{1}{2 - 4D} \sin x = \frac{1}{2} \frac{1}{1 - 2D} \sin x = \frac{1}{2} \frac{1 + 2D}{1 - 4D^2} \sin x$$

Again, replacing D^2 by -1, we obtain

$$\text{P.I.} = \frac{1}{10}(1 + 2D) \sin x = \frac{1}{10}(\sin x + 2 \cos x)$$

Hence the complete solution is

$$y = c_1 e^x + c_2 e^{3x} + \frac{1}{10}(\sin x + 2 \cos x),$$

where c_1 and c_2 are arbitrary constants.

Suggested Readings

McQuarrie, D.A. (2009). Mathematical Methods for Scientist and Engineers. First Indian Edition. Viva Books Pvt. Ltd. New Delhi.

Raisinghania, M.D. (2005). Ordinary & Partial Differential Equation. Eighth Edition. S. Chand & Company Ltd., New Delhi.

Kreyszig, E. (1993). Advanced Engineering Mathematics. Seventh Edition, John Willey & Sons, Inc., New York.

Arfken, G.B. (2001). Mathematical Methods for Physicists. Fifth Edition, Harcourt Academic Press, San Diego.

Grewal, B.S. (2007). Higher Engineering Mathematics. Fourteenth Edition. Khanna Publishilers, New Delhi.

Edwards, C.H., Penney, D.E. (2007). Elementary Differential Equations with Boundary Value Problems. Sixth Edition. Pearson Higher Ed, USA.

Loop Analysis of Electric Circuits

In this method, we set up and solve a system of equations in which the unknowns are **loop currents**. The currents in the various branches of the circuit are then easily determined from the loop currents. (Click here for a tutorial on loop currents vs. branch currents.)

The steps in the loop current method are:

- Count the number of loop currents required. Call this number *m*.
- Choose *m* independent loop currents, call them $I_1, I_2, \ldots, I_m$ and draw them on the circuit diagram.
- Write down Kirchhoff's Voltage Law for each loop. The result, after simplification, is a system of *n* linear equations in the *n* unknown loop currents in this form:

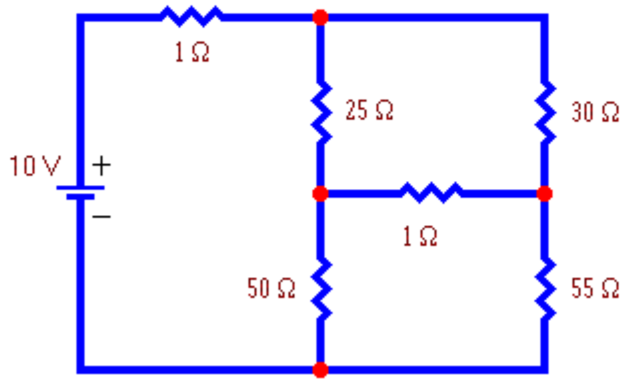
$$\begin{cases} R_{11}\cdot I_1 + R_{12}\cdot I_2 + \cdots + R_{1m}\cdot I_m = V_1 \\ R_{21}\cdot I_1 + R_{22}\cdot I_2 + \cdots + R_{2m}\cdot I_m = V_2 \\ \vdots \qquad \vdots \qquad \qquad \vdots \qquad \vdots \\ R_{m1}\cdot I_1 + R_{m2}\cdot I_2 + \cdots + R_{mm}\cdot I_m = V_m \end{cases}$$

  where $R_{11}, R_{12}, \ldots, R_{mm}$ and $V_1, V_2, \ldots, V_m$ are constants.

  Alternatively, the system of equations can be gotten (already in simplified form) by using the inspection method.

- Solve the system of equations for the *m* loop currents $I_1, I_2, \ldots, I_m$ using Gaussian elimination or some other method.
- Reconstruct the branch currents from the loop currents.

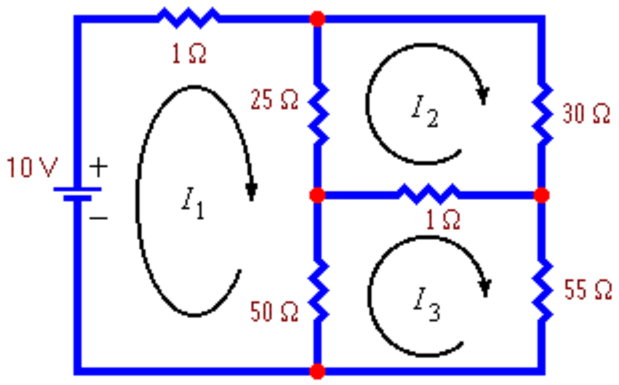
Example 1: Find the current flowing in each branch of this circuit.



Solution:

- The number of loop currents required is 3.

- We will choose the loop currents shown to the right. In fact these loop currents are **mesh currents**.



- Write down Kirchoff's Voltage Law for each loop. The result is the following system of equations:

$$\begin{cases} 1i_1 + 25(i_1 - i_2) + 50(i_1 - i_3) = 10 \\ 25(i_2 - i_1) + 30i_2 + 1(i_2 - i_3) = 0 \\ 50(i_3 - i_1) + 1(i_3 - i_2) + 55i_3 = 0 \end{cases}$$

Collecting terms this becomes:

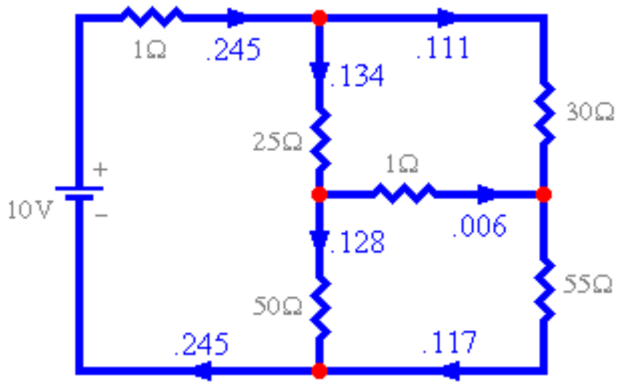
$$\begin{cases} 76i_1 - 25i_2 - 50i_3 = 10 \\ -25i_1 + 56i_2 - 1\,i_3 = 0 \\ -50i_1 - 1\,i_2 + 106i_3 = 0 \end{cases}$$

This form for the system of equations could have been gotten immediately by using the inspection method.

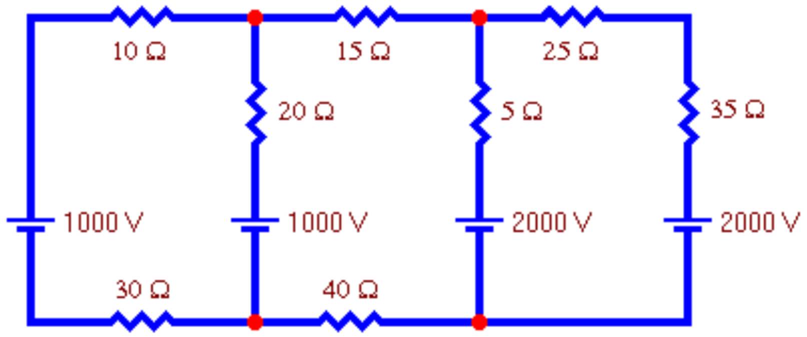
- Solving the system of equations using Gaussian elimination or some other method gives the following currents, all measured in amperes:

I_1=0.245, I_2=0.111 and I_3=0.117

- Reconstructing the branch currents from the loop currents gives the results shown in the picture to the right.

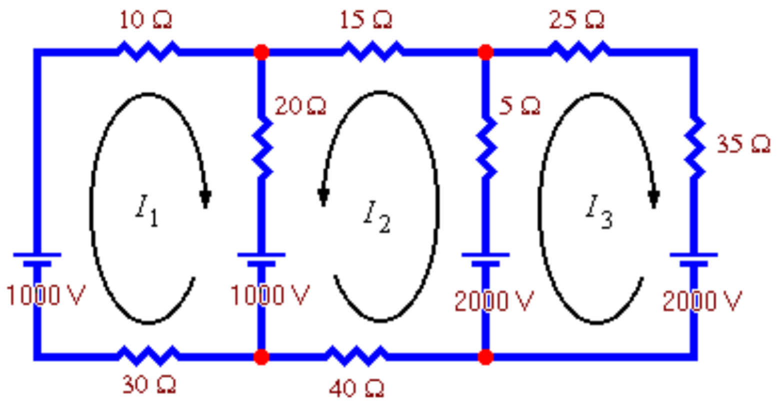


Example 2: Find the current flowing in each branch of this circuit.



Solution:

- The number of loop currents required is 3.
- This time we will choose the loop currents shown to the right.



- Write down Kirchoff's Voltage Law for each loop. The result is the following system of equations:

$$\begin{cases} 10i_1 + 20(i_1 + i_2) + 30i_1 = 1000 - 1000 \\ 15i_2 + 20(i_2 + i_1) + 40i_2 + 5(i_2 + i_3) = 2000 - 1000 \\ 25i_3 + 35i_3 + 5(i_3 + i_2) = 2000 - 2000 \end{cases}$$

Collecting terms this becomes:

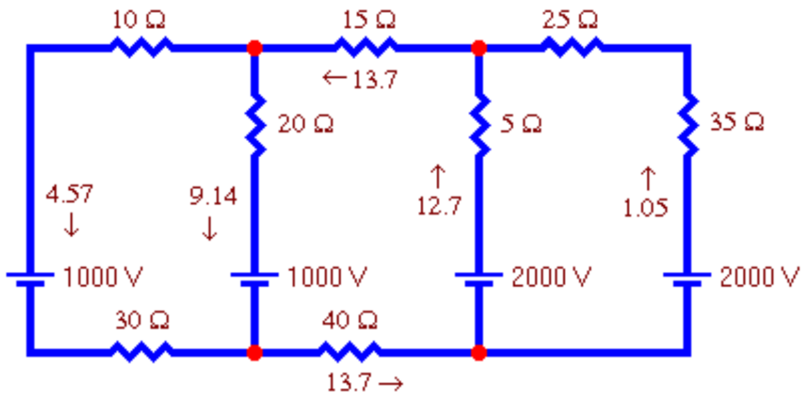
$$\begin{cases} 60i_1 + 20i_2 \ + 0i_3 = 0 \\ 20i_1 + 80i_2 \ + 5\ i_3 = 1000 \\ 0i_1 + \ 5\ i_2 + 65i_3 = 0 \end{cases}$$

This form for the system of equations could have been gotten immediately by using the inspection method.

- Solving the system of equations using Gaussian elimination or some other method gives the following currents, all measured in amperes:

I_1 = - 4.57, I_2 = 13.7 and I_3 = - 1.05

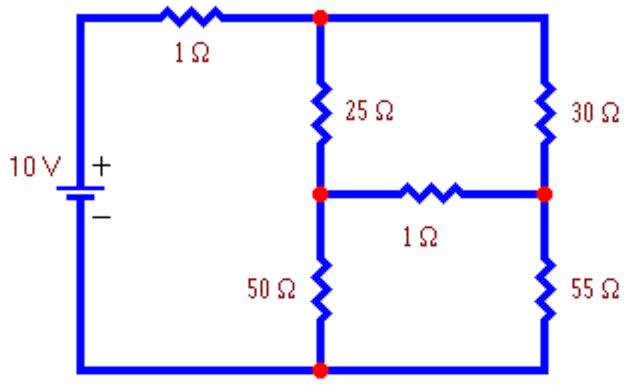
- Reconstructing the branch currents from the loop currents gives the results shown in the picture to the right.



Return to Linear Algebra and Electricity

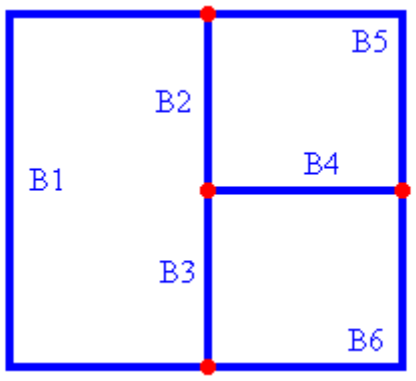
Loop Currents and Branch Currents

An **electric network** is a set of voltage sources and resistances connected in some way.

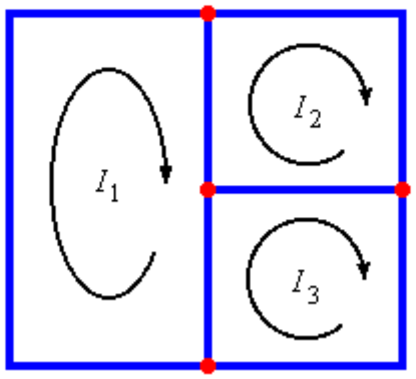


In this diagram we have removed all the resistors and voltage sources so that we can focus attention on the **topology** of the network (ie. the structure of the circuit) and count its nodes and branches.

"*To solve a network*" means to find the current flowing in each branch of the network. Since this circuit has 6 branches, this means calculating 6 **branch currents**.

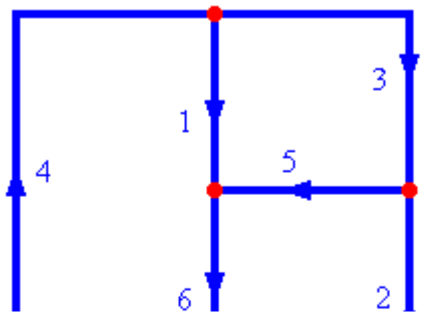


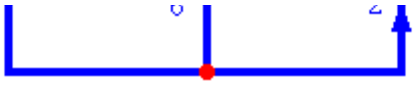
Loop currents offer a more economical way to describe the current flow in a network. The currents in all 6 branches can be described in terms of just 3 loop currents as shown to the right. A **loop current** is defined as a constant current that flows around a closed path or loop. (A **closed path** is a path through the network that ends where it starts.)



Each branch current is given by the algebraic sum of all the loop currents present in that branch. (By algebraic sum we mean that the sign and direction of loop currents must be taken into account in the sum.)

For example, if the 3 loop currents in the picture above have these values:

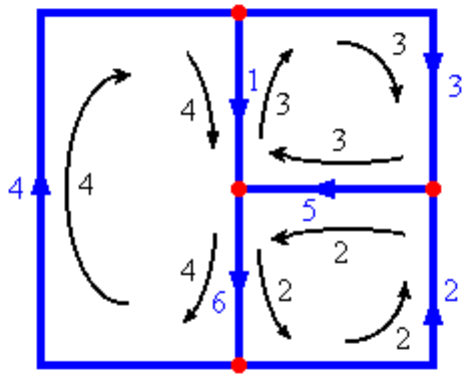




$$\begin{cases} I_1 = 4 \\ I_2 = 3 \\ I_3 = -2 \end{cases}$$

then the 6 branch currents in the picture to the right have the magnitudes and directions shown.

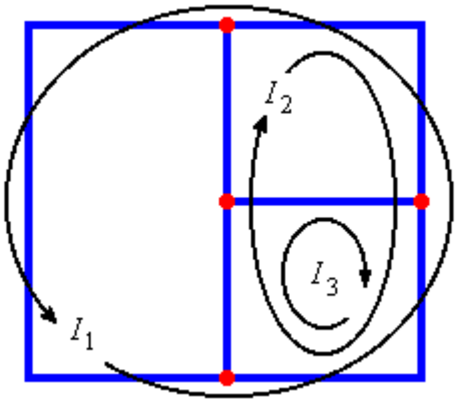
This drawing shows in detail how the various loop currents contribute to the various branch currents.



Here is a different set of 3 loop currents. If the values of the loop currents are:

$$\begin{cases} I_1 = -4 \\ I_2 = -1 \\ I_3 = -5 \end{cases}$$

then the 6 branch currents have exactly the same values as before.



Return to Loop Analysis of Electric Circuits

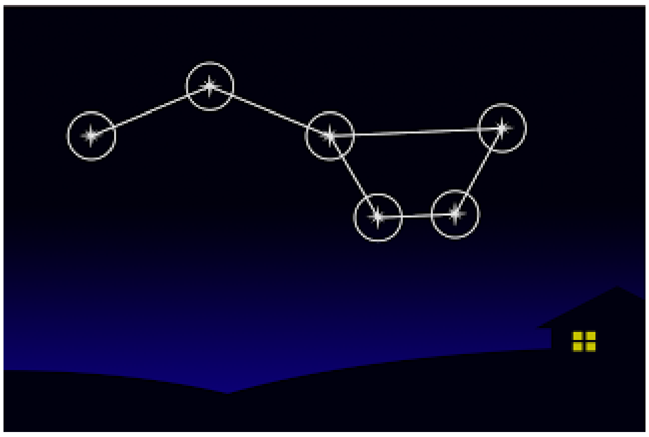
Constellations: Preliminary Design

Because I am way too excited to begin work on this, I thought I'd go ahead and be the first to submit my idea for the game jam. For those of you who don't know me, please see my previous introduction for Crossword-Z here!

Title: Constellations
Catchphrase: "Graphs in the Sky"
Type: Puzzle Game!
Inspiration: Flow (Android/Apple top-selling game)

Idea Behind the Game:
Recently I've become very interested in Graph Theory, precisely because it lends itself to so many different puzzles and interesting questions. I think that there is so much that can be visualized in Graph Theory that it is a shame that there isn't already an app that allows you to learn by seeing, touching, and experimenting with graphs. Thus Constellations was born as a solution to this problem.

Theme: The idea for the theme is that the player is looking out her window and tracing graphs through the stars. I believe that there is so much beauty in graph theory in that it is a powerful tool for visualizing the relationships of our world. I want the art and style of my game to to capture this beauty, and there is no more beautiful a graph in nature than the constellations. I want the player to become immersed in a starry night sky and feel time pass slowly by as she relaxes and observes the constellations she creates. To make this atmosphere I will use a dark palette with quiet colors and storybook art and bright white stars in the sky (See the screenshot below for an example). I will also include some relaxing, non-vocal music playing in the background to fully immerse the player in the experience. I'm thinking of sampling some tracks by the Japanese band Mono, who have a very quiet, fantasy sound.


The puzzle screen, with an example graph drawn in the sky.

Mechanics: My idea for this game jam is to create a number of interesting levels based around two different game modes: Create and Traverse.

- In the Create puzzles, the player will be given certain restraints (such as a degree sequence), and then they are told to draw a graph that meets all of the requirements. I want as many of these puzzles as possible to have multiple solutions, or even some solutions that better meet the requirement (such as, "a connected graph that has the fewest possible edges") to allow the player to experiment and improve upon her solutions.
- In the Traverse puzzles, the player will be given a Constellation, and told to analyze it in some way (such as finding the shortest path). Again, for most of these puzzles I want the player to have the option to experiment and improve upon their solutions. The game will only tell the player if their solution does not meet the criteria, but it will not tell her if it is sub-optimal.

Goals: I want to make this game educational in that playing will serve as an introduction to the topics and ideas of Graph Theory. However, I don't want the game to feel like a textbook. Instead, I want the game to be an interactive experiment, where the player can learn herself by pushing for optimal solutions in each puzzle. To this end, I have a few ideas for how to incorporate an educational component:

1. Make the many theorems of Graph Theory unlockable as "achievements" when a player creates a graph that demonstrates knowledge of the theorem.
2. Include narration that comments on each puzzles after a solution is reached. This would just be enough to get the player curious, but not require a great deal of advanced mathematics to understand.
3. Include a "Storybook" section that gradually fills with information as levels are cleared. I would model this book as a student's journal that she gradually fills up with thoughts about the constellations shes sees. I could even include hints about to how to improve solutions that are less than optimal.

With all of these elements, I want the game itself to be a guide, not a lecturer. I want all insights to come from the player, so that she can learn on her own without being taught. I think that interaction and discovery are the best ways for anybody to learn a subject, and that's what I want this game to be about.

UNIT - V

SECOND ORDER DIFFERENTIAL EQUATIONS

5.1. Solution of second order differential equations with constant coefficients in the form $a\frac{d^2y}{dx^2}+b\frac{dy}{dx}+cy=0.$ Simple Problems

5.2 Solution of second order differential equations in the form $a\ \frac{d^2y}{dx^2}+b\frac{dy}{dx}+cy=f(x)$. Where a,b and c are constants and f(x) = e^{mx}. Simple problems.

5.3. Solution of second order differential equations in the form $a\frac{d^2y}{dx^2}+b\frac{dy}{dx}+cy=f(x).$ Where a,b and c are constants and f(x) = sinmx or cosmx. Simple problems

5.1 SECOND ORDER DIFFERENTIAL EQUATIONS

Introduction:

In the last unit, we learnt first order differential equation. In this unit, we will learn second order differential equation.

The second order differential equation is of the form

$$a\frac{d^2y}{dx^2}+b\frac{dy}{dx}+cy=f(x). \qquad (1)$$

Where a,b and c are real numbers and f(x) is a function of x.

We use differential operators Dy, D^2y in (1), we get

$$(aD^2+bD+c)y=f(x) \text{ where } D=\frac{d}{dx} \qquad (2)$$

Now, we put f(x) = 0 in (1), we get

$$a\frac{d^2y}{dx^2}+b\frac{dy}{dx}+cy=0 \qquad (3)$$

The solution of (3) is called complementary function (CF) of (1).

To solve (3), we assume a trial solution $y = e^{px}$ for some value of p. Then $\frac{dy}{dx} = pe^{px}$ and $\frac{d^2y}{dx^2} = p^2e^{px}$.

Substituting these values in (3), we get

$$ap^2e^{px} + bpe^{px} + ce^{px} = 0$$

$$\Rightarrow e^{px}[ap^2 + bp + c] = 0 \qquad (4)$$

$$\Rightarrow ap^2 + bp + c = 0$$

This equation in p is called the Auxillary Equation (AE)

Solving (4), we get two roots say p_1 and p_2. Then the following three cases arise.

Case (i)

If the roots p_1 and p_2 are real and distinct, then the solution of (3) is

$$y = Ae^{p_1x} + Be^{p_2x}$$

Case (ii)

If the roots p_1 and p_2 are real and equal, then the solution of (3) is

$$y = e^{p_1x}(Ax + B)$$

Case (iii)

If the roots p_1 and p_2 are complex say $p_1 = \alpha + i\beta$ and $p_2 = \alpha - i\beta$, then the solution of (3) is

$$y = e^{\alpha x}[A\cos\beta x + B\sin\beta x]$$

In all cases, A and B are arbitrary constants.

5.1 WORKED EXAMPLES
PART – A

1. If roots of the auxillary equation are $\frac{1}{2} \pm i\frac{\sqrt{3}}{2}$, what is the solution of the differential equation?

Solution:

Here, the roots are complex and $\alpha = \frac{1}{2}$, $\beta = \frac{\sqrt{3}}{2}$,

$\therefore$ The solution of differential equation is

$$y = e^{\frac{1}{2}x}[A\cos\frac{\sqrt{3}}{2}x + B\sin\frac{\sqrt{3}}{2}x]$$

2. Find the solution of $(D^2 – 81)\ y = 0$

Solution:

The auxillary equation is $p^2 – 81 = 0$

$\Rightarrow (p+9)\ (p-9) = 0$

$\Rightarrow p_1 = -9,\ p_2 = 9$

Here, the roots are real and distinct

$\therefore$ The solution of differential equation is

$$y = Ae^{-9x} + Be^{9x}$$

3. Solve $\frac{d^2y}{dx^2} + 64y = 0$

Solution:

Given $\frac{d^2y}{dx^2} + 64y = 0 \Rightarrow (D^2 + 64)y = 0$

The auxillary equation is $p^2+64=0$

$\Rightarrow p = \pm 8i$

Here, the roots are complex, $\alpha=0$ and $\beta = 8$

$\therefore$ The solution is $y = A\cos 8x + B\sin 8x$

4. Solve $(D^2-2D-3)y=0$

Solution:

The auxillary equation is $p^2-2p-3=0$

$\Rightarrow (p+1)(p-3) = 0$

$\Rightarrow p_1 = -1, p_2 = 3$

Here, the roots are real and distinct

$\therefore$ The solution is $y = Ae^{-x} + Be^{3x}$

5. Solve $(D^2-4D-1)y=0$

Solution:

The auxillary equation is $p^2-4p-1 = 0$

Here a = 1, b = -4, c = -1

$$P = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$$

$$= \frac{4 \pm \sqrt{16-4(1)(-1)}}{2(1)}$$

$$= \frac{4 \pm \sqrt{20}}{2}$$

$$= \frac{4 \pm 2\sqrt{5}}{2} = 2 \pm \sqrt{5}$$

So, $p_1 = 2 + \sqrt{5}$ and $p_2 = 2 - \sqrt{5}$

Here, the roots are real and distinct

$\therefore$ The solution is

$y = Ae^{(2+\sqrt{5})x} + Be^{(2-\sqrt{5})x}$

6. Solve $\frac{d^2y}{dx^2}-6\frac{dy}{dx}+9y=0$

Solution:

Given: $\frac{d^2y}{dx^2}-6\frac{dy}{dx}+9y=0 \Rightarrow (D^2-6D+9)\,y=0$

The auxillary equation is $p^2 - 6p+9 = 0$

$\Rightarrow (p-3)(p-3)=0$

$\Rightarrow p_1=3,\ p_2 = 3$

Here, the roots are real and equal.

.. The solution is $y = e^{3x}[Ax+B]$

7. Solve $(D^2+D+2)y=0$

Solution:

The auxillary equation is $p^2+p+2 = 0$

Here a = 1, b = 1, c = 2

$$P = \frac{-b\pm\sqrt{b^2-4ac}}{2a}$$

$$= \frac{-1\pm\sqrt{1-4(1)(2)}}{2(1)}$$

$$= \frac{-1\pm\sqrt{-7}}{2}$$

$$= \frac{-1\pm i\sqrt{7}}{2}$$

$$= \frac{-1}{2}\pm i\frac{\sqrt{7}}{2}$$

Here, the roots are complex, $\alpha=-\frac{1}{2},\beta=\frac{\sqrt{7}}{2}$

$\therefore$ The solution is y = $e^{\frac{-1}{2}x}[A\cos\frac{\sqrt{7}}{2}x+B\sin\frac{\sqrt{7}}{2}x$

PART – B

1. Solve $(D^2+1)\, y = 0$ when $x = 0$, $y = 2$ and $x = \frac{\pi}{2}$, $y=-2$.

Solution:

The auxillary equation is $p^2 + 1 = 0$

$$\Rightarrow p = \pm i$$

Here, the roots are complex, $\beta = 1$

$\therefore$ The solution is

$y = A \cos x + B \sin x$ …1

When x=0, y=2, the equation (1) becomes

$A \cos 0 + B \sin 0 = 2$

$A + 0 = 2$

$A = 2$

When $x = \frac{\pi}{2}$, $y=-2$, the equation (1) becomes

$A \cos \frac{\pi}{2} + B \sin \frac{\pi}{2} = -2$

$0 + B = -2$

$B = -2$

$\therefore$ The required solution is

$y = 2 \cos x – 2 \sin x$

2. Show that the solution of the equation $(D^2 + 3D + 2)\, y = 0$ if $y(0) = 1$ and $y^1(0) = 0$ is $y = 2e^{-x} – e^{-2x}$

Solution:

The auxillary equation is $p^2+3p+2=0$

$$\Rightarrow (p+1)(p+2) = 0$$

$$\Rightarrow p_1 = -1,\ p_2 = -2$$

Here, the roots are real and distinct

$\therefore$ The solution is $y = Ae^{-x} + Be^{-2x}$ …1

Now, $y' = -Ae^{-x} - 2Be^{-2x}$ …2

If $y(0) = 1$, the equation (1) becomes

$A + B = 1$ …3

If $y'(0) = 0$, the equation (2) becomes

$A+2B=0$ …4

Solving (3) and (4) we get $A=2$, $B=-1$

$\therefore$The required solution is

$y=2e^{-x}-e^{-2x}$

5.2. SOLUTION OF SECOND ORDER EQUATIONS IN THE FORM $a\frac{d^2y}{dx^2}+b\frac{dy}{dx}+cy=f(x)$ WHERE A,B AND C ARE CONSTANTS AND $f(x)=e^{mx}$.

Introduction:

In previous section, we find the complementary function . In this section, we have to find the particular integral (PI) and the general solution of a second order differential equation.

The Solution of Differential equation with Constant Coefficients is y=CF+PI

Method of finding particular integral

Consider $(aD^2+bD+c)y = e^{mx}$ where m is a constant.

Let $f(D) = aD^2+bD+c$

Then PI is given by $\frac{1}{f(D)}e^{mx} = \frac{e^{mx}}{f(m)}$

Three cases arise in PI

Case (i)

If $f(m) \neq 0$ then $PI = \frac{1}{f(D)}e^{mx} = \frac{e^{mx}}{f(m)}$

Case (ii)

If $f(m)=0$ and $f'(m)\neq 0$ then $\quad PI=\dfrac{x\,e^{mx}}{f'(m)}$

Case (iii)

If $f(m)=0$ and $f'(m)=0$ and $f''(m)\neq 0$ then $\quad PI=\dfrac{x^2e^{mx}}{f''(m)}$

5.2 WORKED EXAMPLE

PART – A

1. Find the complementary function of $(D^2+16)y= e^x$

Solution:

The auxiliary equation is $p^2+16=0 \Rightarrow p=\pm 4i$

Here, the roots are complex , $\beta=4$

$\therefore CF = A \cos 4x + B \sin 4x$

2. Find the complementary function of $(D^2-60D+800)y=e^{40x}$

Solution:

The auxiliary equation is $p^2-60p+800=0$

$\Rightarrow (p-40)\ (P-20) =0$

$\Rightarrow P_1=40,\ P_2=20$

Here the roots are real and distinct

$\therefore CF = Ae^{40x} + Be^{20x}$

3. Find the particular integral of $(D^2+1)\ y =1$

Solution:

$$PI=\frac{1}{D^2+1}=\frac{1}{D^2+1}\,e^0$$

$$=\frac{1}{0+1}=\frac{1}{1}=1$$

4. Find the particular integral of $(D^2+7D+14) = 8e^{-x}$

Solution:

$$PI = \frac{1}{D^2+7D+14}\, 8e^{-x}$$

$$= \frac{8e^{-x}}{(-1)^2+7(-1)+14} \quad = \frac{8e^{-x}}{8} = e^{-x}$$

5. Find the particular integral of $(D^2-2D-3)y = e^{-x}$

Solution:

$$PI = \frac{1}{D^2-2D-3}\, e^{-x}$$

$$= \frac{x\,e^{-x}}{2D-2} \qquad \text{Since } f(-1)=0$$

$$= \frac{x\,e^{-x}}{2(-1)-2} \quad = -\frac{x\,e^{-x}}{4}$$

PART - B

1. Solve $(D^2+5D+6)y=30$

Solution:

The auxiliary equation is $p^2+5p+6=0$

$\Rightarrow (p+2)\ (P+3) = 0$

$\Rightarrow P_1=-2,\ P_2=-3$

Here, the roots are real and distinct

$\therefore CF = Ae^{-2x}+Be^{-3x}$

$$\text{Now}\, PI = \frac{1}{D^2+5D+6}\, 30$$

$$= \frac{30e^{\circ}}{D^2+5D+6}$$

$$= \frac{30e^{\circ}}{0^2+5(0)+6}$$

$$= \frac{30}{6}$$

$$PI = 5$$

$\therefore$ The Required solution is

$Y=CF+PI = Ae^{-2x}+Be^{-3x}+5$

2. Solve $(D^2+6D+5)\ y = 2e^x$

Solution:

The auxiliary equation is $p^2+6p+5=0$

$\Rightarrow (p+1)\ (P+5) = 0$

$\Rightarrow P_1=-1,\ P_2=-5$

Here the roots are real and distinct

$\therefore CF = Ae^{-x} + Be^{-5x}$

$$\text{Now PI} = \frac{1}{D^2+6D+5}\ 2e^x$$

$$= \frac{2e^x}{1^2+6(1)+5}$$

$$= \frac{2e^x}{12}$$

$$PI = \frac{e^x}{6}$$

$\therefore$ The required solution is

Y=CF+PI

$$Ae^{-x} + Be^{-5x} + \frac{e^x}{6}$$

3. Solve $(D^2+D)y = e^{\frac{x}{2}}$

Solution:

The auxiliary equation is $p^2+p=0$

$\Rightarrow p\ (p+1)=0$

$\Rightarrow P_1=0,\ P_2=-1$

Here the roots are real and distinct

$\therefore CF = Ae^0 + Be^{-x} = A+Be^{-x}$

$$\text{Now PI} = \frac{1}{D^2+D}\ e^{\frac{x}{2}}$$

$$= \frac{e^{\frac{x}{2}}}{\left(\frac{1}{2}\right)^2 + \frac{1}{2}}$$

$$= \frac{e^{\frac{x}{2}}}{3/4}$$

$$PI = \frac{4}{3} e^{\frac{x}{2}}$$

$\therefore$ The required solution is
y=CF+PI

$$= A + Be^{-x} + \frac{3}{4} e^{\frac{x}{2}}$$

4. Solve $(D^2 - D - 12)y = e^{4x}$

Solution:

The auxiliary equation is p^2-p-12=0

$\Rightarrow$(p-4) (p+3)=0

$\Rightarrow p_1=4,\ p_2=-3$

Here the roots are real and distinct

$\therefore CF = Ae^{4x} + Be^{-3x}$

$$\text{Now } PI = \frac{1}{D^2 - D - 12} e^{4x}$$

$$= \frac{x e^{4x}}{2D - 1} \qquad \text{Since } f(4) = 0$$

$$= \frac{x e^{4x}}{2(4) - 1}$$

$$PI = \frac{x e^{4x}}{7}$$

$\therefore$ The required solution is
y=CF + PI

$$= Ae^{4x} + Be^{-3x} + \frac{x e^{4x}}{7}$$

5. Solve $(D^2-2D+1)\ y = e^x$

Solution:

The auxiliary equation is $p^2-2p+1=0$

$\Rightarrow (p-1)\ (p-1) = 0$

$\Rightarrow p_1=1,\ p_2=1$

Here the roots are real and equal

$\therefore CF = e^x\ (Ax+B)$

$$\text{Now } PI = \frac{1}{D^2-2D+1}\ e^x$$

$$PI = \frac{x^2}{2}\ e^x \qquad \text{Since } f(1)=0, f'(1)=0$$

$\therefore$ The required solution is

Y=CF+PI

$$= e^x\ (Ax+B) + \frac{x^2}{2} e^x$$

6 Solve $\dfrac{d^2y}{dx^2} - 13\dfrac{dy}{dx} + 12y = 2e^{-2x} + 5e^x$

Solution:

Given $\dfrac{d^2y}{dx^2} - 13\dfrac{dy}{dx} + 12y = 2e^{-2x} + 5e^x$

$\Rightarrow (D^2 - 13D + 12)\ y = 2e^{-2x} + 5e^x$

The auxilary equation is $p^2-13p+12=0$

$\Rightarrow (p-1)\ (p-12) = 0$

$\Rightarrow p_1=1,\ p_2=12$

Here the roots are real and distinct

$\therefore CF = Ae^x+Be^{12x}$

$$\text{Now } PI_1 = \frac{1}{D^2 - 13D + 12}\, 2e^{-2x}$$

$$= \frac{2e^{-2x}}{(-2)^2 - 13(-2) + 12}$$

$$= \frac{2e^{-2x}}{4 + 26 + 12}$$

$$= \frac{e^{-2x}}{21}$$

$$\text{Now } PI_2 = \frac{1}{D^2 - 13D + 12}\, 5e^{x}$$

$$= \frac{5xe^{x}}{2D - 13} \qquad \text{Since } f(1) = 0$$

$$= \frac{5xe^{x}}{2(1) - 13}$$

$$= -\frac{5xe^{x}}{11}$$

$\therefore$ The required solution is

Y =CF+PI$_1$+PI$_2$

$$= Ae^{x} + Be^{12x} + \frac{e^{-2x}}{21} - \frac{5xe^{x}}{11}$$

5.3 SOLUTION OF SECOND ORDER DIFFERENTIALEQUATIONS IN THE FORM $a\frac{d^2y}{dx^2} - b\frac{dy}{dx} + cy = f(x)$ WHERE a,b AND c ARE CONSTANTS AND f(x) = sin mx or cos mx where m is a constant $\neq 0$

INTRODUCTION

In this section, we have to find the particular integral when f(x) =sin mx or cos m x where m is a constant

Methods of finding PI

Consider f(x) =sin m x

Case (i)

Express f(D) as function of D^2,say ϕ (D^2) and then replace $D^2$with $-m^2$

If $\phi(-m^2)\neq 0$,then

$$PI = \frac{1}{f(D)} \sin mx$$

$$= \frac{1}{\phi(D^2)} \sin mx$$

$$PI = \frac{1}{\phi(-m^2)} \sin mx$$

Case (ii)

Sometimes we cannot form ϕ (D^2) Then we shall try to get $\phi(D,D^2)$ that is a function of D and D^2. In such cases we proceed as follows.

For Example

$$\text{Now PI} = \frac{1}{D^2 + 2D + 3} \sin 2x$$

$$= \frac{1}{-2^2 + 2D + 3} \sin 2x \text{ Replace } D^2 \text{ by } -2^2$$

$$= \frac{1}{2D - 1} \text{Sin } 2x$$

$$= \frac{2D + 1}{4D^2 - 1} \sin 2x \text{ multiply and divide by } 2D + 1$$

$$= \frac{2D(\sin 2x) + \sin 2x}{4(-2^2) - 1}$$

$$= \frac{4\cos 2x + \sin 2x}{-17}$$

$$= \frac{1}{-17}\left[4\cos 2x + \sin 2x\right]$$

Now consider f(x) = cos m x

Case (i): $PI = \frac{1}{\phi(-m^2)} \cos mx$

Case(ii): Same as sin m x method

General Solution:

The general solution is y= CF+PI

5.3 WORKED EXAMPLE

PART - A

1. Find the complementary function of (D^2+49) y= cos 4x

Solution:

The auxiliary equation is $p^2+49=0$

$\Rightarrow p=\pm 7i$

Here, the roots are complex ,β =7

$\therefore$ CF = A cos 7x+B sin 7x

2. Find the particular integral of (D^2+14) y = sin 3x

Solution:

$$PI = \frac{1}{D^2+14}\sin 3x$$

$$= \frac{1}{-3^2+14}\sin 3x$$

$$= \frac{\sin 3x}{5}$$

3. Find the particular integral of (D^2+a^2) y = Cos b x

Solution:

$$PI = \frac{1}{D^2+a^2}\cos bx$$

$$= \frac{1}{-b^2+a^2}\cos bx$$

$$= \frac{\cos bx}{a^2-b^2}$$

PART - B

1.) Solve $(D^2-4)y = \sin 2x$

Solution:

The auxiliary equation is $p^2-4=0$

$$\Rightarrow p^2 = 4$$

$$\Rightarrow p = \underline{+}2$$

$$\Rightarrow p_1 = 2, p_2 = -2$$

Here, the roots are real and distinct

$\therefore\ CF = Ae^{2x} + Be^{-2x}$

Now $PI = \frac{1}{D^2 - 4}(\sin 2x)$

$= \frac{1}{-2^2 - 4}\sin 2x$

$= -\frac{\sin 2x}{8}$

$\therefore$ The Required solution is

$y = CF + PI$

$= Ae^{2x} + Be^{-2x} - \frac{\sin 2X}{8}$

2.) Solve $D^2y = -16\sin 4x$

Solution:

The auxiliary equation is $p^2 = 0$

$\Rightarrow p_, = 0, p_2 = 0$

Here, the roots are real and equal

$\therefore\ CF = e^0(Ax + B) = Ax + B$

Now $PI = \frac{1}{D^2} - 16\sin 4x$

$= \frac{1}{-4^2} - 16\sin 4x$

PI= Sin4x

$\therefore$ The Required solution is

$y = CF + PI$

$= Ax + B + \text{Sin}4x$

3.) Solve $\frac{d^2y}{dx^2} + 16y = \cos^2 x$

Solution:

Given $\frac{d^2y}{dx^2} + 16y = \cos^2 x$

$$\Rightarrow (D^2 + 16)y = \cos^2 x$$

$$\Rightarrow (D^2 + 16)y = \frac{1}{2} + \frac{\cos 2x}{2}$$

$$= \frac{1}{2}e^0 + \frac{1}{2}\cos 2x$$

The auxiliary equation is $p^2 + 16 = 0$

$$\Rightarrow p = \underline{+}4i$$

Here, the roots are complex, $\beta = 4$

$$\therefore CF = A\cos 4x + B\text{Sin}4x$$

$$PI_1 = \frac{\frac{1}{2}e^0}{D^2 + 16}$$

$$= \frac{1}{2}\,\frac{e^0}{0 + 16}$$

$$= \frac{1}{32}$$

$$PI_2 = \frac{1}{2} \cdot \frac{\cos 2x}{D^2 + 16}$$

$$= \frac{1}{2} \cdot \frac{\cos 2x}{-2^2 + 16}$$

$$= \frac{\cos 2x}{24}$$

$\therefore$ The Required solution is

$y = CF + PI$

$$= A\cos 4x + B\text{Sin}4x + \frac{1}{32} + \frac{\cos 2x}{24}$$

4.) Solve $\left(D^2+3D+2\right)y=\sin 2x$

Solution:

The auxiliary equation is $p^2+3p+2=0$

$$\Rightarrow (p+2)(p+1)=0$$

$$\Rightarrow p_1=-2,\ p_2=-1$$

Here, the roots are real and distinct

$\therefore CF = Ae^{-2x}+Be^{-x}$

Now, $PI=\dfrac{1}{D^2+3D+2}.\text{Sin}2x$

$$=\frac{1}{-2^2+3D+2}.\text{Sin}2x$$

$$=\frac{1}{3D-2}.\text{Sin}2x$$

$$=\frac{3D+2}{9D^2-4}.\text{Sin}2x$$

$$=\frac{3D+2}{-36-4}.\text{Sin}2x$$

$$=\frac{3D(\sin 2x)+2\sin 2x}{-40}$$

$$=\frac{6\cos 2x+2\sin 2x}{-40}$$

$$=\frac{-1}{20}\left[3\cos 2x+\sin 2x\right]$$

$\therefore$ The Required solution is

$y=CF+PI$

$$=Ae^{-2x}+Be^{-x}-\frac{1}{20}\left[3\cos 2x+\sin 2x\right]$$

5.) Solve $(D^2 - 2D - 8)y = 4\cos 3x$

Solution:

Solution: The auxiliary equation is $p^2 - 2p - 8 = 0$

$$\Rightarrow (p-4)(p+2) = 0$$
$$\Rightarrow p_1 = 4,\ p_2 = -2$$

Here, the roots are real and distinct

$\therefore CF = Ae^{4x} + Be^{-2x}$

Now, $PI = \dfrac{1}{D^2 - 2D - 8} 4\cos 3x$

$$= \frac{1}{-3^2 - 2D - 8} 4\cos 3x$$
$$= \frac{1}{-2D - 17} 4\cos 3x$$
$$= -4\left[\frac{1}{2D+17} 4\cos 3x\right]$$
$$= -4\left[\frac{2D-17}{4D^2 - 289} \cos 3x\right]$$
$$= -4\left[\frac{2D(\cos 3x) - 17\cos 3x}{-325}\right]$$
$$= -4\left[\frac{-6\sin 3x - 17\cos 3x}{-325}\right]$$
$$= \frac{-4}{325}[6\sin 3x + 17\cos 3x]$$

$\therefore$ The Required solution is

$y = CF + PI$

$$= Ae^{4x} + Be^{-2x} - \frac{4}{325}[6\sin 3x + 17\cos 3x\ ]$$

EXERCISE

PART - A

1.) If roots of the auxilary equation are 2,7 what is the solution of the differential equation?

2.) If roots of the auxilary equation are 0,1 what is the solution of the differential equation?

3.) If roots of the auxilary equation are -2,$\pm$ i, what is the solution of the differential equation?

4.) Find the solution of $\left(D^2 - 1\right)y = 0$

5.) Find the solution of $\frac{d^2y}{dx^2} - 16y = 0$

6.) Solve $\left(D^2 + 9\right)y = 0$

7.) Find the solution of $\left(D^2 + 100\right)y = 0$

8.) Solve $\left(D^2 + 4D - 1020\right)y = 0$

9.) Solve $\left(3D^2 - 5D + 2\right)y = 0$

10.) Solve $\left(3D^2 - 7D - 6\right)y = 0$

11.) Solve $\frac{d^2y}{dx^2} + \frac{dy}{dx} = 0$

12.) Solve $\left(D^2 - D - 1\right)y = 0$

13.) Solve $\left(D^2 + 4D + 4\right)y = 0$

14.) Solve $\frac{d^2y}{dx^2} - 12\frac{dy}{dx} + 36y = 0$

15.) Solve $\left(D^2 + D + 1\right)y = 0$

16.) Solve $\left(3D^2 - D + 1\right)y = 0$

17.) Find the Complementary function of $\left(D^2 + 13D - 90\right)y = e^x$

18.) Find the Particular integral of $(D^2 - 3D + 2)y = e^{-x}$

19.) Find the Particular integral of $(D^2 + D + 4)y = 10e^{2x}$

20.) Find the Particular integral of $(D^2 - 8D + 15)y = e^{3x}$

21.) Find the Particular integral of $(D^2 + 10D + 25)y = e^{-5x}$

22.) Find the Complementary integral of $(D^2 + 25)y = \cos ax$

23.) Find the Particular integral of $(D^2 + 25)y = \text{Sin}x$

24.) Find the Particular integral of $(D^2 + 10)y = \sin 3x$

25.) Find the Particular integral of $\frac{d^2y}{dx^2} - 4y = \cos 4x$

PART - B

1.) Solve $(D^2 + 36)y = 0$ when $y(0) = 2$ and $y^1(0) = 12$

2.) Solve $\frac{d^2y}{dx^2} + y = 0$ given that $\frac{dy}{dx} = 2$ and y=1 when x=0

3.) Solve $(D^2 - 2D - 15)y = 0$ given that $\frac{dy}{dx} = 0$ and $\frac{d^2y}{dx^2} = 2$ when x=0

4.) Solve $(D^2 - D - 20)y = 0$ given that y=5 and $\frac{dy}{dx} = -2$ when x=0

5.) Solve $(D^2 + 7D + 12)y = 3$

6.) Solve $(D^2 + 3D + 2)y = 2e^x$

7.) Solve $(D^2 + 12D + 36)y = e^x$

8.) Solve $(D^2 + D + 4)y = e^{x/2}$

9.) Solve $(D^2 - 3D + 2)y = e^{2x}$

10.) Solve $(D^2 + 6D + 8)y = e^{-4x}$

11.) Solve $\frac{d^2y}{dx^2} - 4\frac{dy}{dx} + 4y = e^{2x}$

12.) Solve $(D^2 + 2aD + a^2)y = e^{-ax}$

13.) Solve $(D^2 + 14D + 49)y = 4e^{-7x}$

14.) Solve $(D^2 - 2D + 4)y = 5 + 3e^{-x}$

15.) Solve $\frac{d^2y}{dx^2} + 8\frac{dy}{dx} + 15y = e^{-3x} + e^{3x}$

16.) Solve $(D^2 + 10D + 25)y = e^{5x} + e^{-5x}$

17.) Solve $(D^2 + 16)y = \sin 9x$

18.) Solve $(D^2 - 25)y = \sin 5x$

19.) Solve $(D^2 + 100)y = \cos 2x$

20.) Solve $\frac{d^2y}{dx^2} - 2y = \cos 3x$

21.) Solve $(D^2 + 2D - 3)y = \sin x$

22.) Solve $(D^2 + D - 2)y = \text{Sin}3x$

23.) Solve $(D^2 + 4D + 13)y = 4\cos 3x$

24.) Solve $(D^2 - 8D + 9)y = 8\cos 5x$

25.) Solve $(D^2 - 2D - 8)y = 4\cos 2x$

ANSWERS

PART - A

1.) $y = Ae^{2x} + Be^{7x}$

2.) $y = A + Be^{x}$

3.) $y = e^{-2x}[A\cos x + B\sin x]$

4.) $y = Ae^{x} + Be^{-x}$

5.) $y = Ae^{4x} + Be^{-4x}$

6.) $y = A\cos 3x + B\sin 3x$

7.) $y = A\cos 10x + B\sin 10x$

8.) $y = Ae^{30x} + Be^{-34x}$

9.) $y = Ae^{x} + Be^{\frac{2}{3}x}$

10.) $y = Ae^{3x} + Be^{-\frac{2}{3}x}$

11.) $y = A + Be^{-x}$

12.) $y = Ae^{\left(\frac{1+\sqrt{5}}{2}\right)x} + Be^{\left(\frac{1-\sqrt{5}}{2}\right)x}$

13.) $y = e^{-2x}(Ax + B)$

14.) $y = e^{6x}(Ax + B)$

15.) $y = e^{-x/2}\left(A\cos\frac{\sqrt{3}}{2}x + B\sin\frac{\sqrt{3}}{2}x\right)$

16.) $y = e^{x/6}\left(A\cos\frac{\sqrt{11}}{6}x + B\sin\frac{\sqrt{11}}{6}x\right)$

17.) $CF = Ae^{5x} + Be^{-18x}$

18.) $\frac{e^{-x}}{6}$

19.) e^{2x}

20.) $-\frac{xe^{3x}}{2}$

21.) $\frac{x^2}{2}e^{-5x}$

22.) CF=Acos5x+Bsin5x

23.) $\frac{\sin x}{24}$

24.) Sin3x

25.) $-\frac{\text{Cos}4x}{20}$

Part - B

1.) $y = 2\cos 6x + 2\sin 6x$

2.) $y = \cos x + 2\sin x$

3.) $y = \frac{1}{20}e^{5x} + \frac{1}{2}e^{-3x}$

4.) $y = 2e^{5x} + 3e^{-4x}$

5.) $y = Ae^{-4x} + Be^{-3x} + \frac{1}{4}$

6.) $y = Ae^{-x} + Be^{-2x} + \frac{e^{x}}{3}$

7.) $y = e^{-6x}(Ax + B) + \frac{e^{x}}{49}$

8.) $y = e^{-x/2}\left[A\cos\frac{\sqrt{15}}{2}x + B\sin\frac{\sqrt{15}}{2}x\right] + \frac{4}{19}e^{x/2}$ 9.)

$y = Ae^{x} + Be^{2x} + xe^{2x}$

10.) $y = Ae^{-4x} + Be^{-2x} - \frac{xe^{-4x}}{2}$

11.) $y = e^{2x}(Ax + B) + \frac{x^2}{2}e^{2x}$

12.) $y = e^{-ax}(Ax + B) + \frac{x^2}{2}e^{-ax}$

13.) $y = e^{-7x}(Ax + B) + 2x^2e^{-7x}$

14.) $y = e^{x}\left(A\cos\sqrt{3}x + B\sin\sqrt{3}x\right) + \frac{5}{4} + \frac{3}{7}e^{-x}$

15.) $y = Ae^{-3x} + Be^{-5x} + \frac{xe^{-3x}}{2} + \frac{e^{3x}}{48}$

16.) $y = e^{-5x}(Ax + B) + \frac{e^{5x}}{100} + \frac{x^2e^{-5x}}{2}$

17.) $y = A\cos 4x + B\sin 4x - \frac{\sin 9x}{65}$

18.) $y = Ae^{5x} + Be^{-5x} - \frac{\sin 5x}{50}$

19.) $y = A\cos 10x + B\sin 10x + \frac{\cos 2x}{96}$

20.) $y = Ae^{\sqrt{2}x} + Be^{-\sqrt{2}x} - \frac{\cos 3x}{11}$

21.) $y = Ae^{-3x} + Be^{x} - \frac{1}{10}(\cos x + 2\sin x)$

22.) $y = Ae^{x} + Be^{-2x} - \frac{1}{130}(3\cos 3x + 11\sin 3x)$

23.) $y = e^{-2x}(A\cos 3x + B\sin 3x) + \frac{1}{10}3\sin 3x + \cos 3x$

24.) $y = Ae^{(4+\sqrt{7})x} + Be^{(4-\sqrt{7})x} - \frac{1}{29}(5\sin 5x + 2\cos 5x)$

25.) $y = Ae^{4x} + Be^{-2x} - \frac{1}{10}(\sin 2x + 3\cos 2x)$

MAXIMUM POWER TRANSFER THEOREM

tina.com/course/11maxim/maxim

Sometimes in engineering we are asked to design a circuit that will transfer the maximum power to a load from a given source. According to the maximum power transfer theorem, a load will receive maximum power from a source when its resistance (R_L) is equal to the internal resistance (R_I) of the source. If the source circuit is already in the form of a Thevenin or Norton equivalent circuit (a voltage or current source with an internal resistance), then the solution is simple. If the circuit is not in the form of a Thevenin or Norton equivalent circuit, we must first use Thevenin's or Norton's theorem to obtain the equivalent circuit.

Here's how to arrange for the maximum power transfer.

1. Find the internal resistance, R_I. This is the resistance one finds by looking back into the two load terminals of the source *with no load connected*. As we have shown in the Thevenin's Theorem and Norton's Theorem chapters, the easiest method is to replace voltage sources by short circuits and current sources by open circuits, then find the total resistance between the two load terminals.

2. Find the open circuit voltage (U_T) or the short circuit current (I_N) of the source between the two load terminals, with no load connected.

Once we have found R_I, we know the optimal load resistance ($R_{Lopt} = R_I$). Finally, the maximum power can be found

$$P_{\max} = \frac{I_N^2 * R_I}{4} = \frac{U_T^2}{4R_I}$$

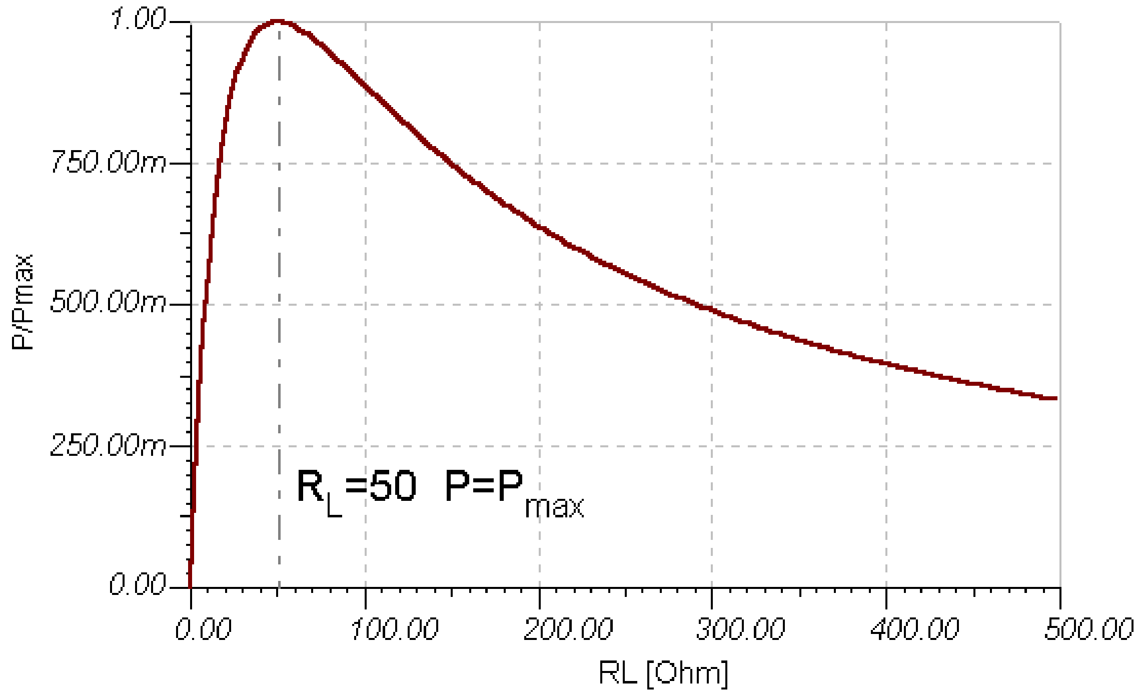
In addition to the maximum power, we might want to know another important quantity: the *efficiency*. Efficiency is defined by the ratio of the power received by the load to the total power supplied by the source. For the Thevenin equivalent:

$$\eta = \frac{P_L}{P_S} = \frac{I_L^2 R_L}{I_L^2 R_{tot}} = \frac{R_L}{R_{Th} + R_L}$$

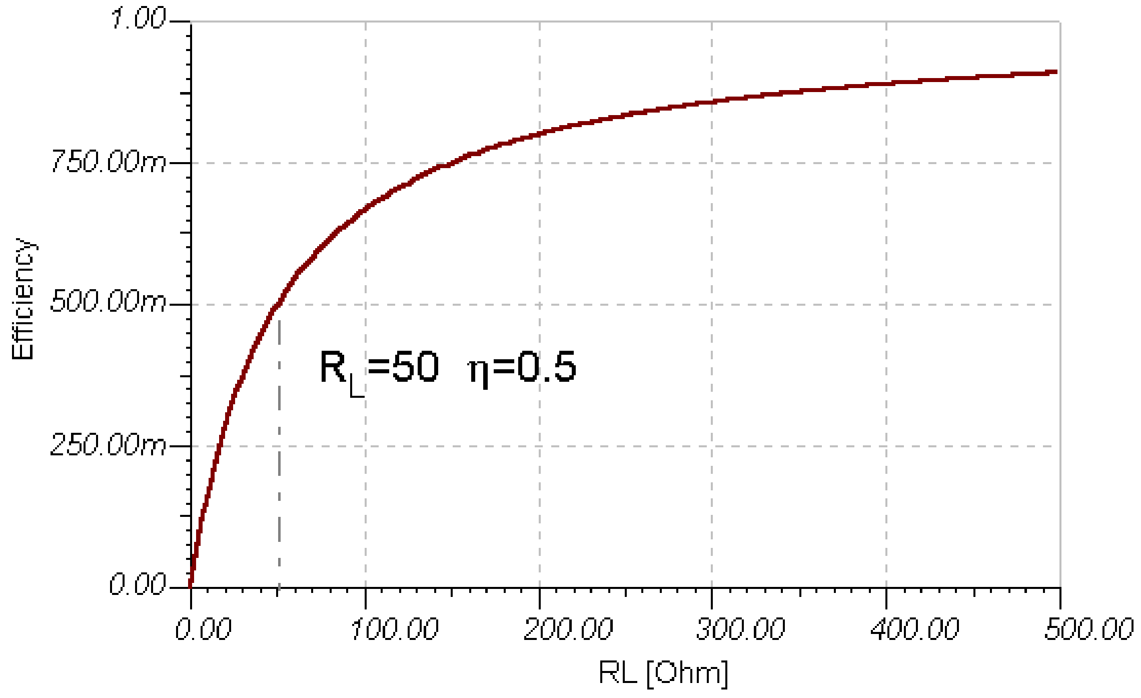
and for the Norton equivalent:

$$\eta = \frac{P_L}{P_S} = \frac{\dfrac{V_L^2}{R_L}}{\dfrac{V_L^2}{R_L} + \dfrac{V_L^2}{R_N}} = \frac{R_N}{R_N + R_L}$$

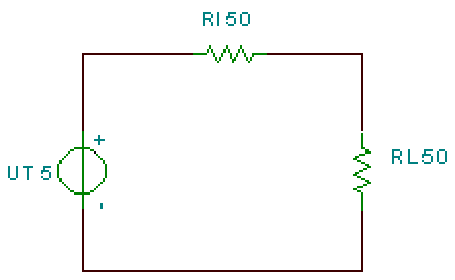
Using TINA's Interpreter, it is easy to draw *P, P/P_{max}*, and *h* as a function of R_L. The next graph shows *P/Pmax*, the power on R_L divided by the maximum power, P_{max}, as a function of R_L (for a circuit with internal resistance R_I=50).



Now let's see the efficiency *h* as a function of $R_{L.}$



The circuit and the TINA Interpreter program to draw the diagrams above are shown below. Note that we we also used the editing tools of TINA's Diagram window to add some text and the dotted line.



```
{ Double click here to invoke the interpreter
and then press the Run button to draw the functions }

Function efficiency(RL);
Begin
  efficiency:=RL/(RI+RL);
End;

Function power(RL);
Begin
  power:= sqr(UT/(RI+RL))*RL/Pmax;
End;

Pmax:=UT*UT/(4RI);
Draw(efficiency(RL), Eff)
Draw(power(RL), Power)
```

Now let's explore the efficiency (h) for the case of maximum power transfer, where R_L = R_{Th}.

The efficiency is:

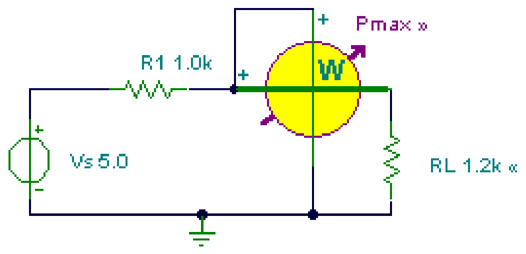
$$\eta = \frac{R_L}{R_{Th} + R_L} = 0.5$$

which when given as a percentage is only 50%. This is acceptable for some applications in electronics and telecommunication, such as amplifiers, radio receivers or transmitters However, 50% efficiency is not acceptable for batteries, power supplies, and certainly not for power plants.

Another undesirable consequence of arranging a load to achieve maximum power transfer is the 50% voltage drop on the internal resistance. A 50% drop in source voltage can be a real problem. What is needed, in fact, is a nearly constant load voltage. This calls for systems where the internal resistance of the source is much lower than the load resistance. Imagine a 10 GW power plant operating at or close to maximum power transfer. This would mean that half of the energy generated by the plant would be dissipated in the transmission lines and in the generators (which would probably burn out). It would also result in load voltages that would randomly fluctuate between 100% and 200% of the nominal value as consumer power usage varied.

To illustrate the application of the maximum power transfer theorem, let's find the optimum value of the resistor R_L to receive maximum power in the circuit below.

Click/tap the circuit above to analyze on-line or click this link to Save under Windows

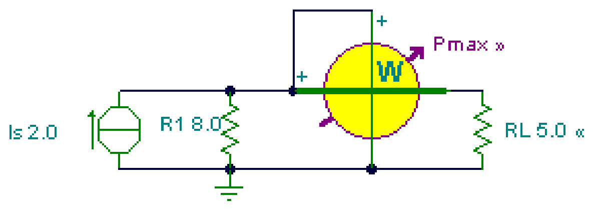


We get the maximum power if R_L= R_1, so R_L = 1 kohm. The maximum power:

A similar problem, but with a current source:

$$P_{max} = \frac{U_T^2}{4R_1} = \frac{5^2}{4*1} = 6.25\ mW$$

Click/tap the circuit above to analyze on-line or click this link to Save under Windows



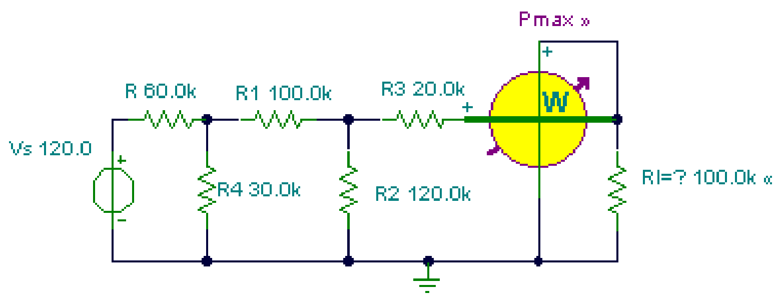
Find the maximum power of the resistor R_L .

We get the maximum power if R_L = R_1 = 8 ohm. The maximum power:

$$P_{max} = \frac{I_N^2 * R_I}{4} = \frac{2^2 * 8}{4} = 8W$$

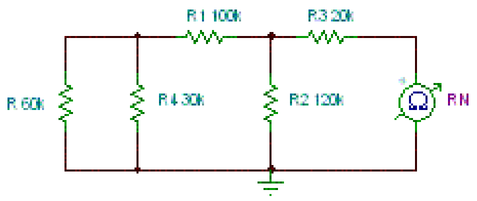
The following problem is more complex, so first we must reduce it to a simpler circuit.

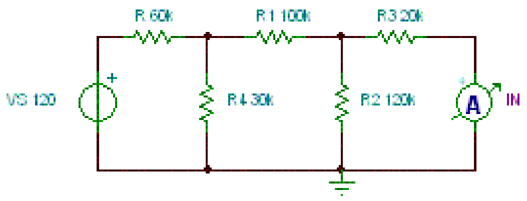
Find R_l to achieve maximum power transfer, and calculate this maximum power.

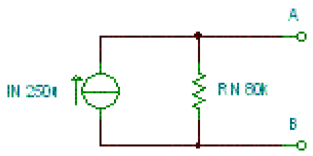


Click/tap the circuit above to analyze on-line or click this link to Save under Windows

First find the Norton equivalent using TINA.







Click/tap the circuit above to analyze on-line or click this link to Save under Windows

Finally the maximum power:

$$P_{\max} = \left(\frac{I_N}{2}\right)^2 * R_N = (0.125)^2 * 80 = 1.25\ mW$$

{Solution by TINA's Interpreter}
O1:=Replus(R4,(R1+Replus(R2,R3)))/(R+Replus(R4,
(R1+Replus(R2,R3))));
IN:=Vs*O1*Replus(R2,R3)/(R1+Replus(R2,R3))/R3;
RN:=R3+Replus(R2,(R1+Replus(R,R4)));
Pmax:=sqr(IN)/4*RN;
IN=[250u]
RN=[80k]
Pmax=[1.25m]

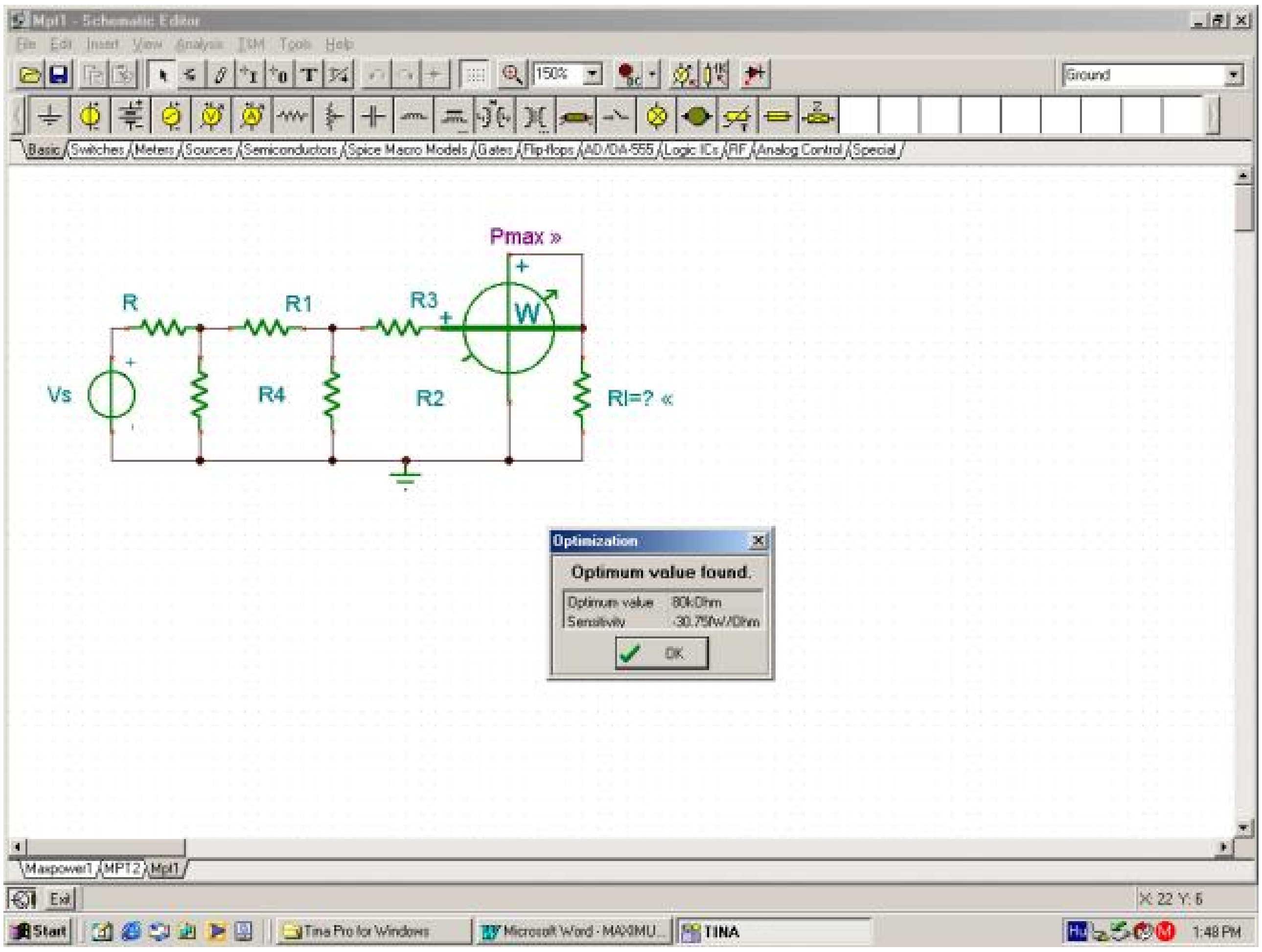
We can also solve this problem using one of TINA’s most interesting features, the *Optimization* analysis mode.

To set up for an Optimization, use the Analysis menu or the icons at the top right of the screen and select Optimization Target. Click on the Power meter to open its dialog box and select Maximum. Next, select Control Object, click on $R_{L,}$ and set the limits within which the optimum value should be searched.

To carry out the optimization in TINA v6 and above, simply use the Analysis/Optimization/DC Optimization command from the Analysis menu.

In older versions of TINA, you can set this mode from the menu, *Analysis/Mode/Optimization*, and then execute a DC Analysis.

After running Optimization for the problem above, the following screen appears:



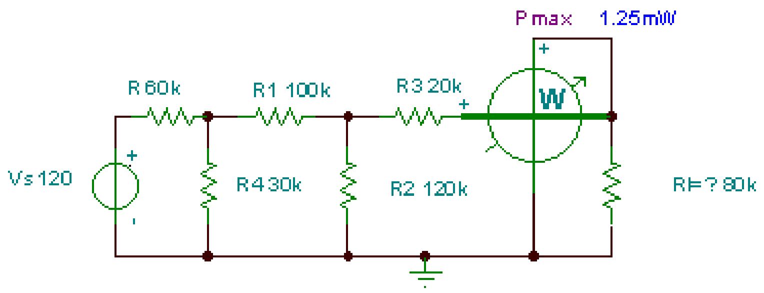
Optimization

Optimum value found.

Optimum value 80kOhm
Sensitivity -30.75fW/Ohm

OK

After Optimization, the value of Rl is automatically updated to the value found. If we next run an interactive DC analysis by pressing the DC button, the maximum power is displayed as shown in the following figure.



Magnetic Circuits

<u>Outline</u>

- Ampere’s Law Revisited
- Review of Last Time: Magnetic Materials
- Magnetic Circuits
- Examples

Electric Fields

Magnetic Fields

$$\oint_S \epsilon_o \vec{E} \cdot d\vec{A} = \int_V \rho dV$$

$$= Q_{enclosed}$$

GAUSS

$$\oint_S \vec{B} \cdot d\vec{A} = 0$$

GAUSS

FARADAY

$$\oint_C \vec{E} \cdot d\vec{l} = -\frac{d}{dt}\left(\int_S \vec{B} \cdot d\vec{A}\right)$$

$$emf = v = \frac{d\lambda}{dt}$$

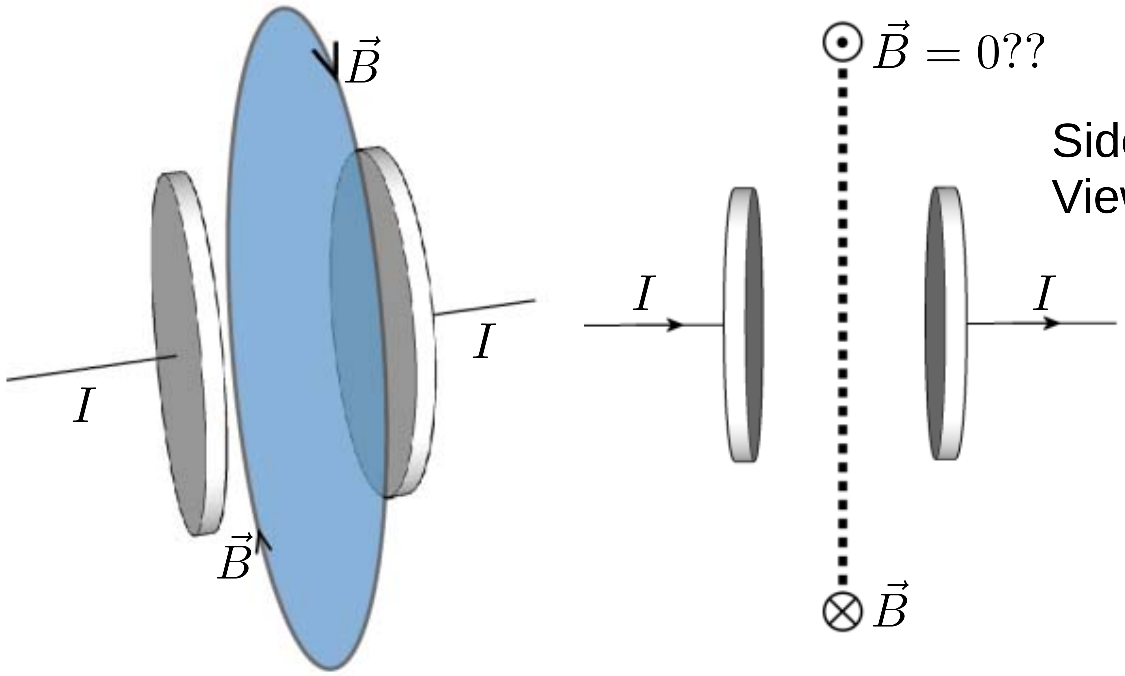
AMPERE

$$\oint_C \vec{H} \cdot d\vec{l}$$

$$= \int_S \vec{J} \cdot d\vec{A} + \frac{d}{dt}\int_S \epsilon \vec{E} \cdot d\vec{A}$$

Ampere's Law Revisited

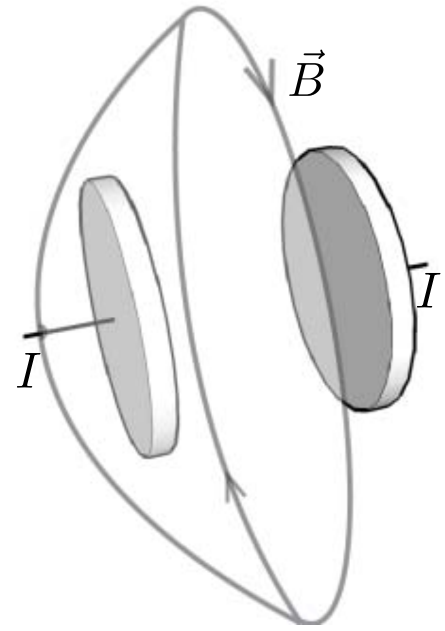
In the case of the magnetic field we can see that 'our old' Ampere's law can not be the whole story. Here is an example in which current does not gives rise to the magnetic field:

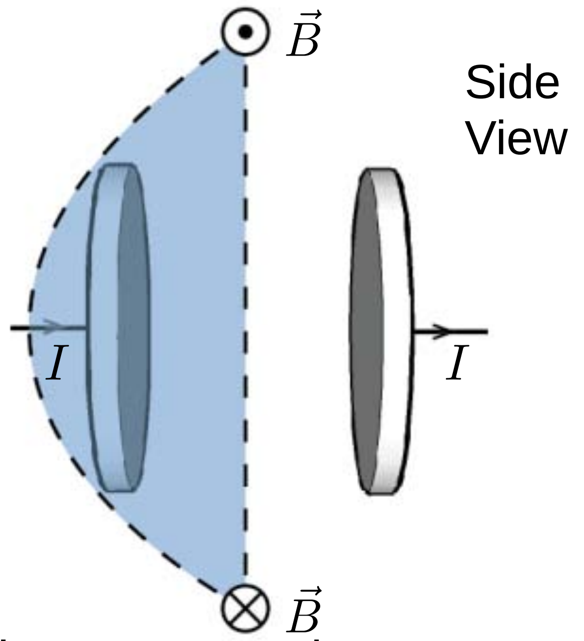


Consider the case of charging up a capacitor C which is connected to very long wires. The charging current is *I*. From the symmetry it is easy to see that an application of Ampere's law will produce *B* fields which go in circles around the wire and whose magnitude is $B(r) = \mu_o I/(2\pi r)$. But there is no charge flow in the gap across the capacitor plates and according to Ampere's law the *B* field in the plane parallel to the capacitor plates and going through the capacitor gap should be zero!
This seems unphysical.

Ampere's Law Revisited (cont.)

If instead we drew the Amperian surface as sketched below, we would have concluded that *B* in non-zero !





Maxwell resolved this problem by adding a term to the Ampere's Law. In equivalence to Faraday's Law,
the changing electric field can generate the magnetic field:

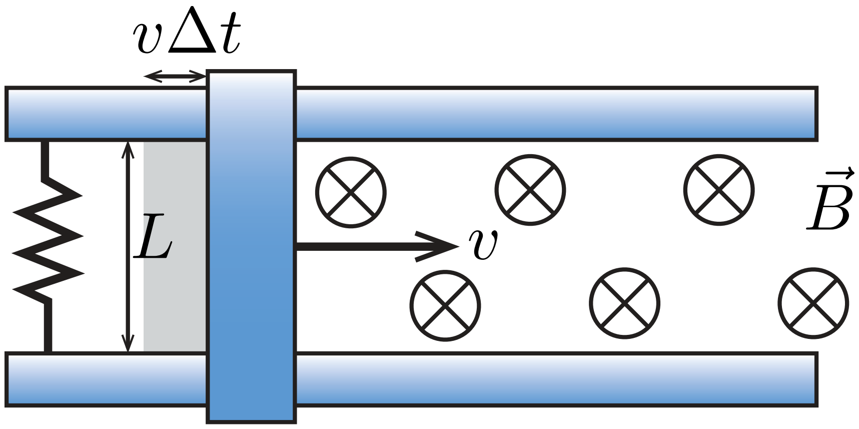
$$\oint_C \vec{H} \cdot d\vec{l} = \int_S \vec{J} \cdot d\vec{A} + \frac{d}{dt} \int_S \epsilon \vec{E} \cdot d\vec{A}$$

COMPLETE AMPERE'S LAW

Faraday's Law and Motional emf

What is the emf over the resistor ?

$$emf = -\frac{d\Phi_{mag}}{dt}$$



In a short time Δt the bar moves a distance Δx = v*Δt, and the flux increases by $\Delta\Phi_{mag}$ = B (L v*Δt)

$$emf = \frac{\Delta\Phi_{mag}}{\Delta t} = BLv$$

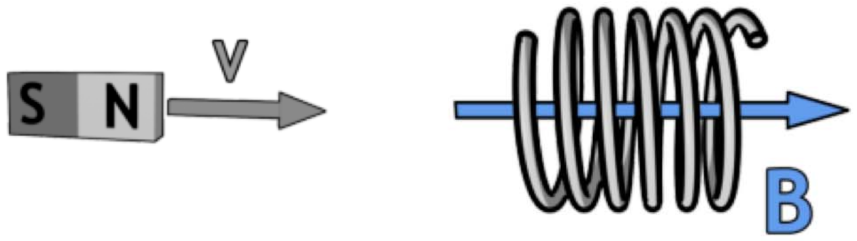
There is an increase in flux through the circuit as the bar of length *L* moves to the right (orthogonal to magnetic field H) at velocity, *v*.

from Chabay and Sherwood, Ch 22

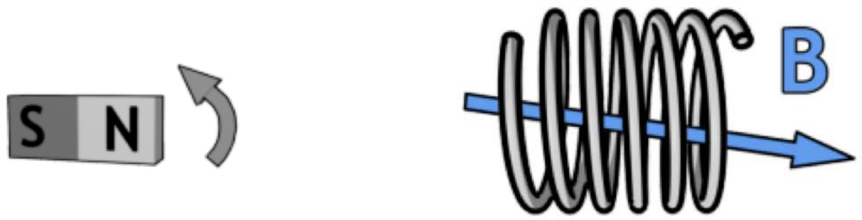
Faraday's Law for a Coil

The induced emf in a coil of N turns is equal to
N times the rate of change of the magnetic flux on one loop of the coil.

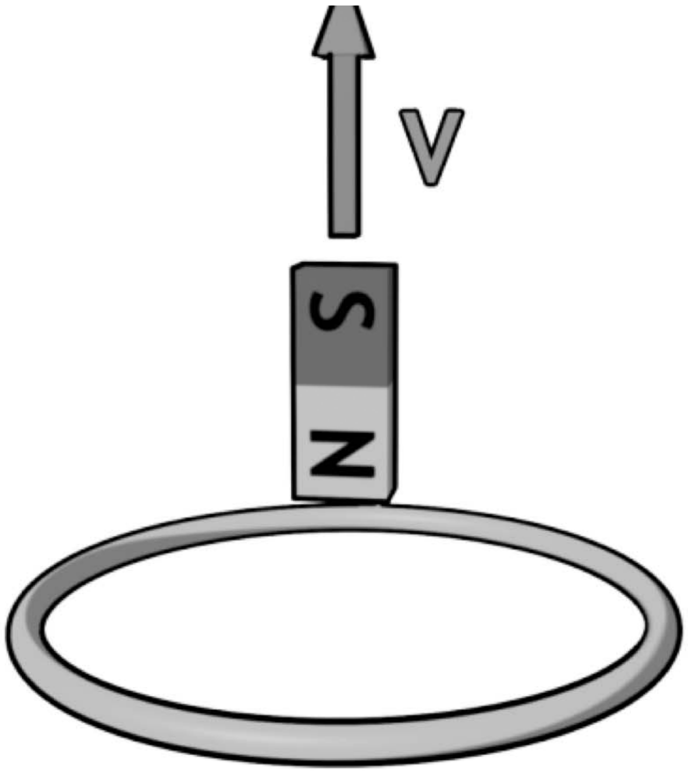
$$emf = -N\frac{d\Phi_{mag}}{dt}$$



Moving a magnet towards a coil produces a time-varying magnetic field inside the coil



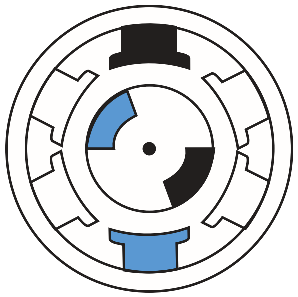
Rotating a bar of magnet (or the coil) produces a time-varying magnetic field inside the coil



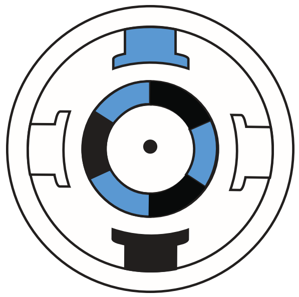
Will the current run CLOCKWISE or ANTICLOCKWISE ?

Complex Magnetic Systems

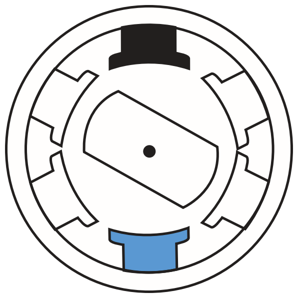
DC Brushless



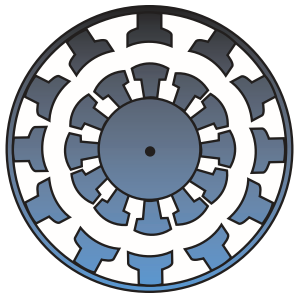
Stepper Motor



Reluctance Motor



Induction Motor



$$\int_C \vec{H} \cdot d\vec{l} = I_{enclosed} \qquad \int_S \vec{B} \cdot d\vec{A} = 0 \qquad \vec{f} = q\left(\vec{v} \times \vec{B}\right)$$

We need better (more powerful) tools...

Magnetic Circuits: Reduce Maxwell to (scalar) circuit problem

Energy Method: Look at change in stored energy to calculate force

Magnetic Flux	Φ	[Wb] (Webers)
Magnetic Flux Density	B	[Wb/m^2] = T (Teslas)
Magnetic Field Intensity	H	[Amp-turn/m]

due to macroscopic & microscopic

due to macroscopic currents

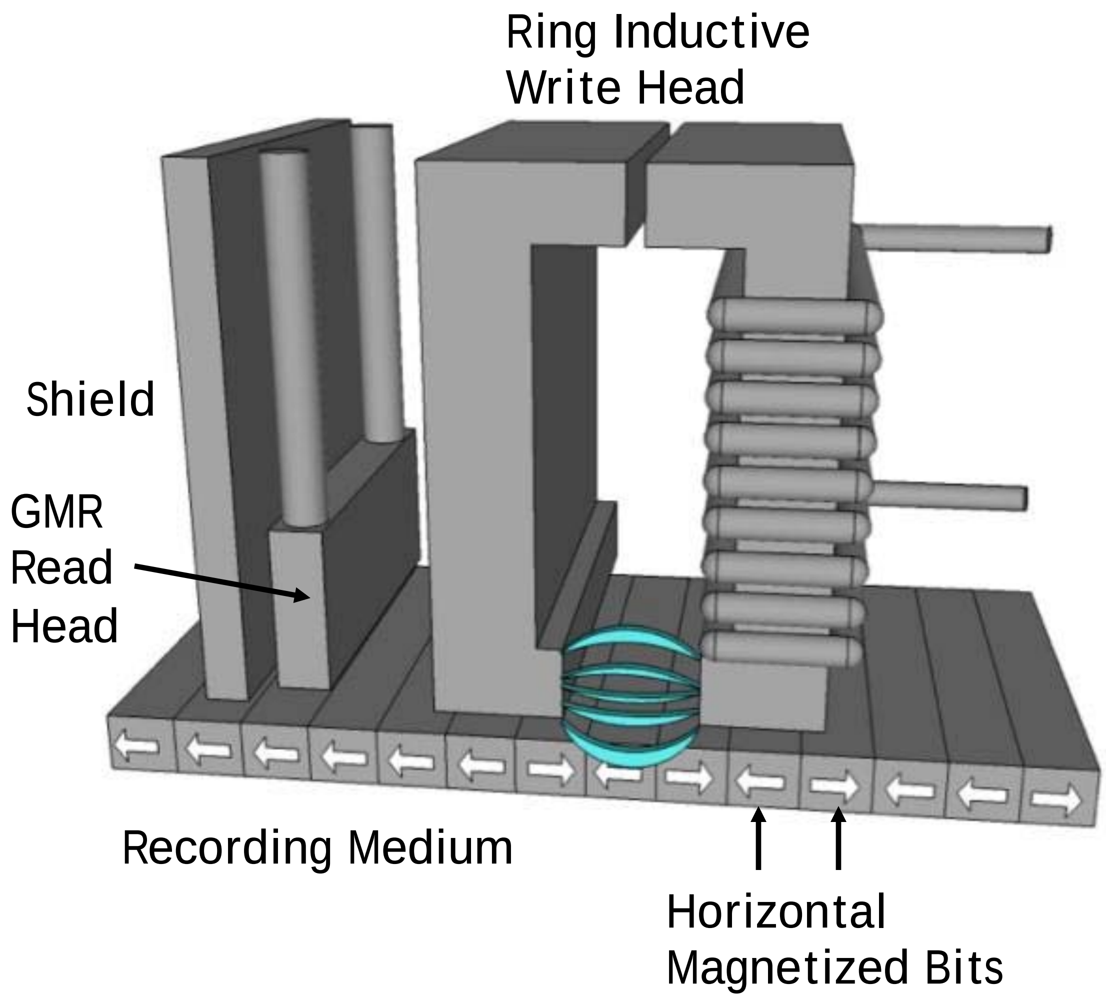
$$\vec{B} = \mu_o\left(\vec{H} + \vec{M}\right) = \mu_o\left(\vec{H} + \chi_m \vec{H}\right) = \mu_o \mu_r \vec{H}$$

Faraday's Law

$$emf = -\frac{d\Phi_{mag}}{dt}$$

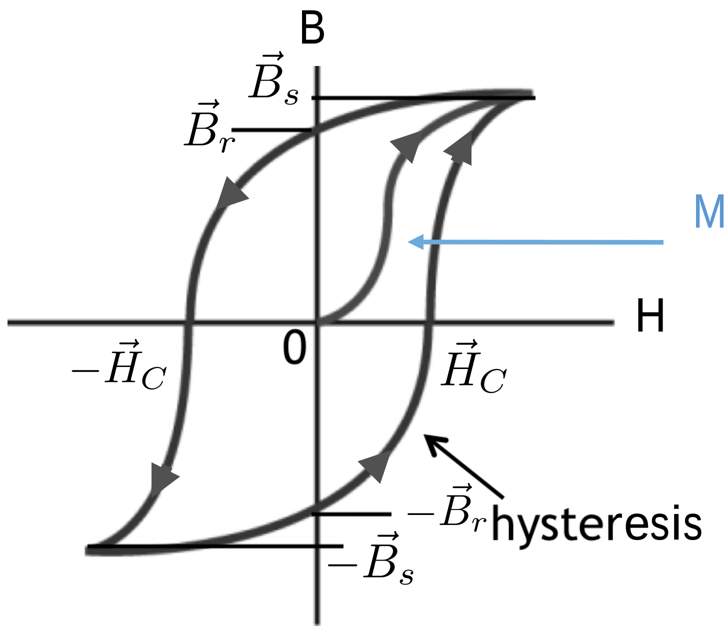
$$emf = \oint \vec{E}_{NC} \cdot d\vec{l} \quad \text{and} \quad \Phi_{mag} = \int \vec{B} \cdot \hat{n}\, d\vec{A}$$

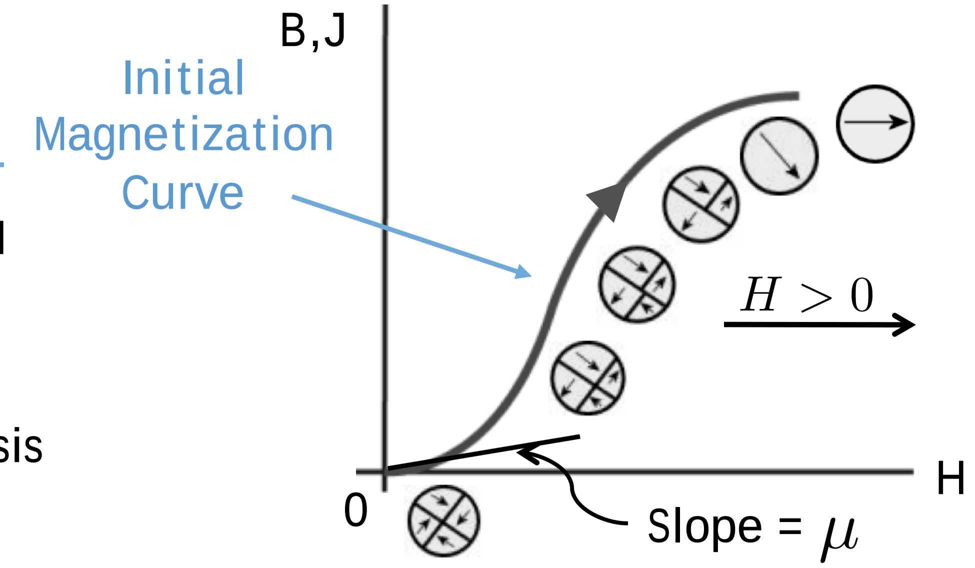
Example: Magnetic Write Head



Bit density is limited by how well the field can be localized in write head

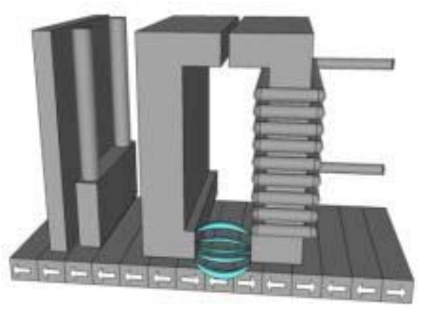
Review: Ferromagnetic Materials





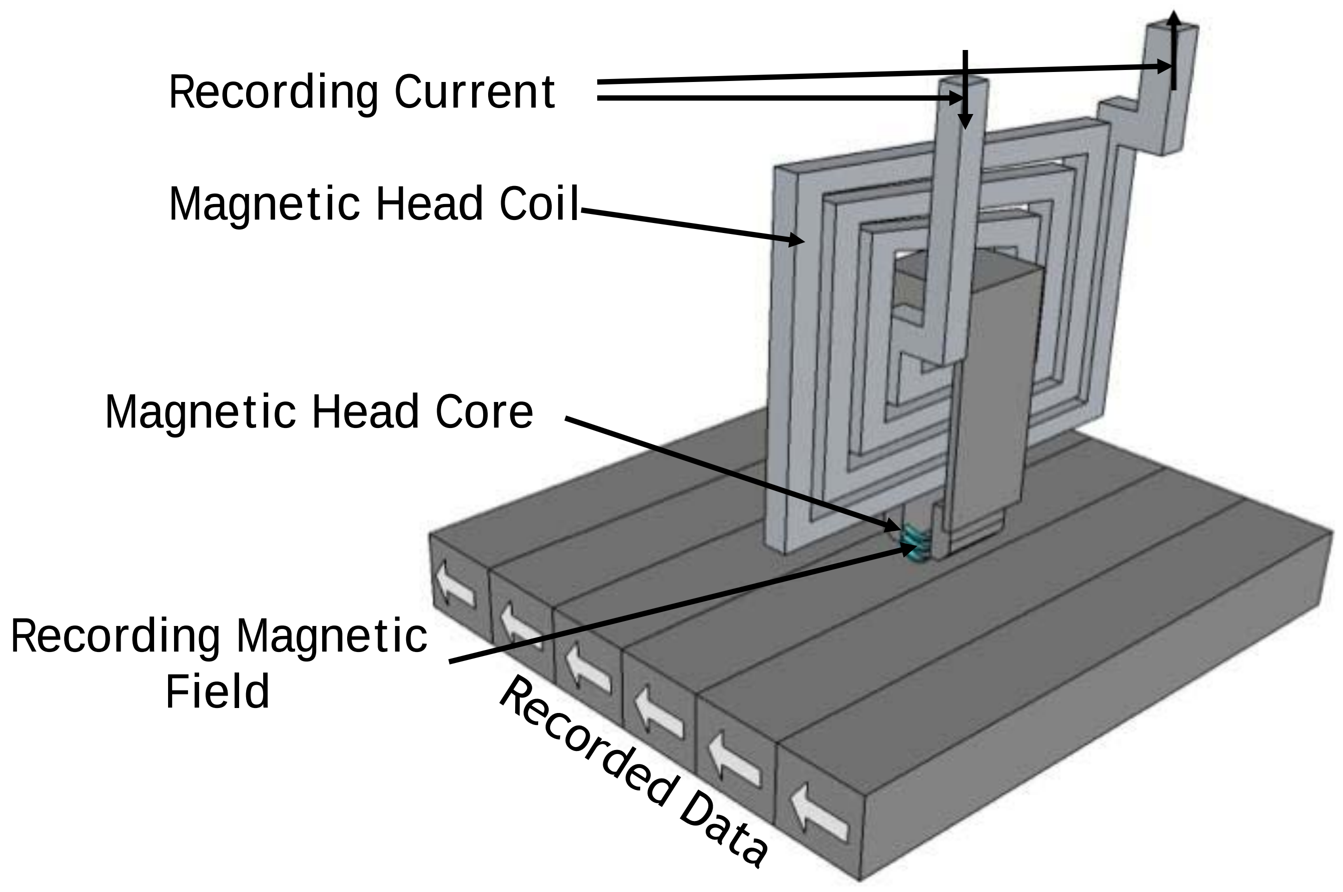
c
Hr : coercive magnetic field strength
Bs : remanence flux density
B : saturation flux density

Behavior of an initially unmagnetized material.
Domain configuration during several stages of magnetization.



$$H \sim i$$

Thin Film Write Head



How do we apply Ampere’s Law to this geometry (low symmetry) ?

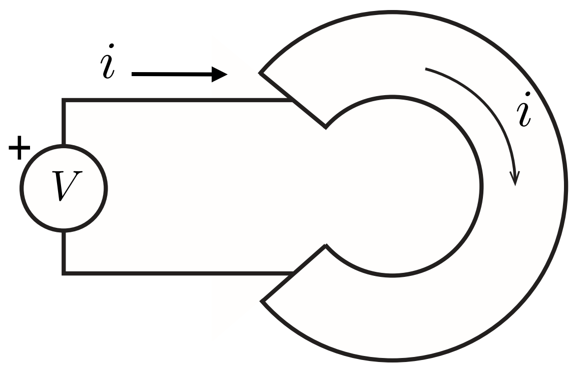
$$\int_C \vec{H} \cdot d\vec{l} = \int_S \vec{J} \cdot d\vec{A}$$

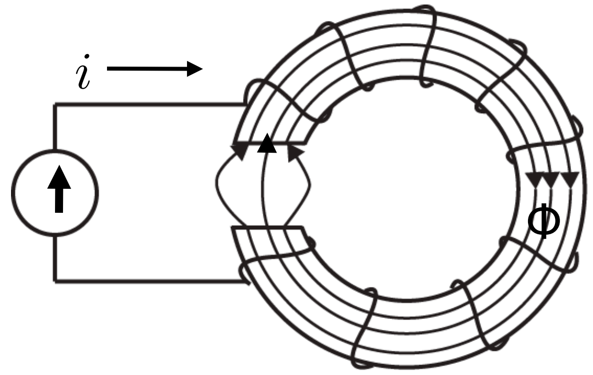
Electrical Circuit Analogy

Charge is conserved...

Flux is 'conserved'...

$$\int_S \vec{B} \cdot d\vec{A} = 0$$

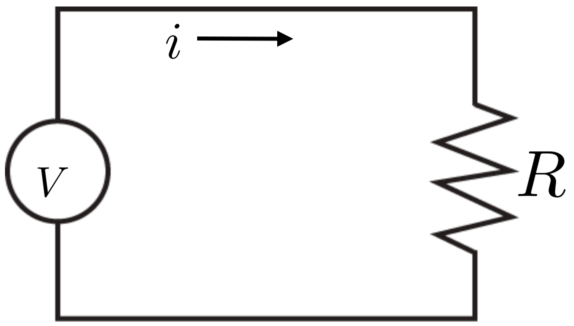


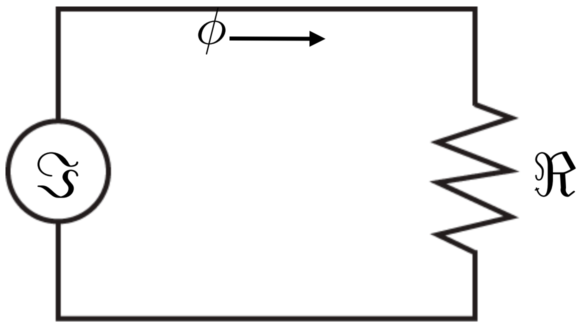


EQUIVALENT
CIRCUITS

Electrical

Magnetic





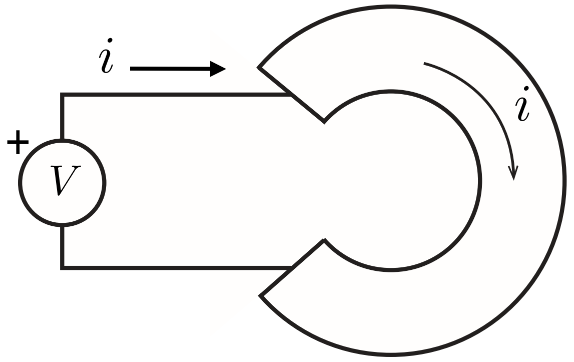
Electrical Circuit Analogy

Electromotive force (charge push)=

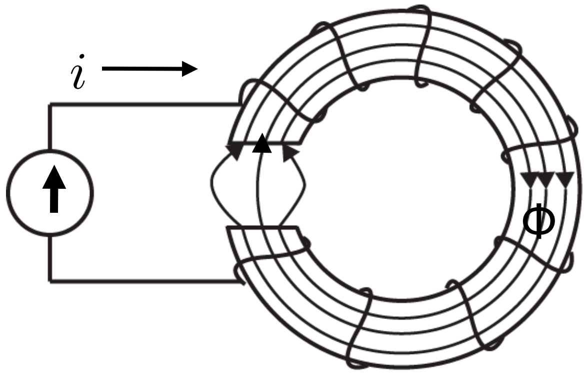
$$v = \int \vec{E} \cdot d\vec{l}$$

Magneto-motive force (flux push)=

$$\oint_C \vec{H} \cdot d\vec{l} = I_{enclosed}$$

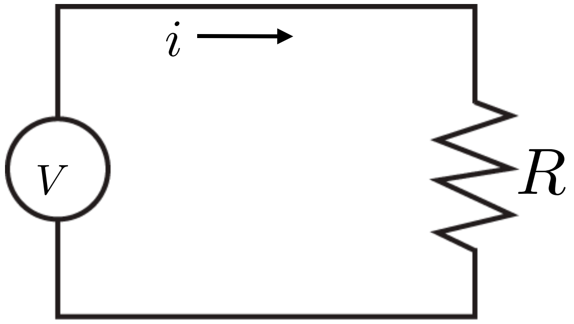


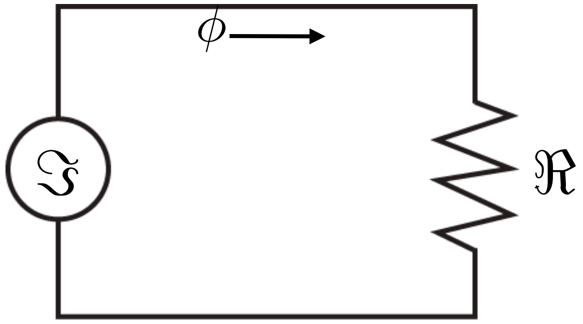
EQUIVALENT CIRCUITS



Electrical

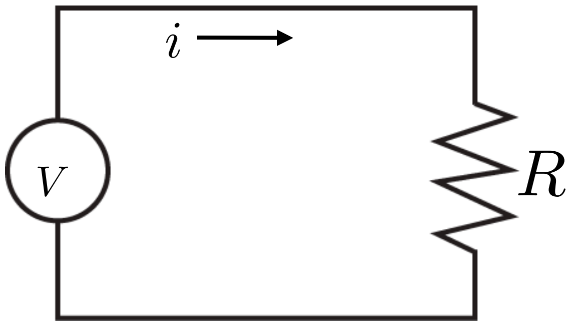
Magnetic

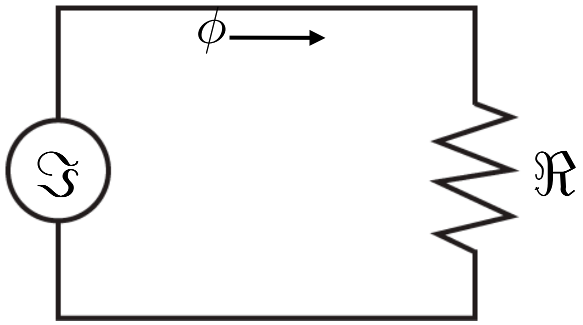




Electrical Circuit Analogy

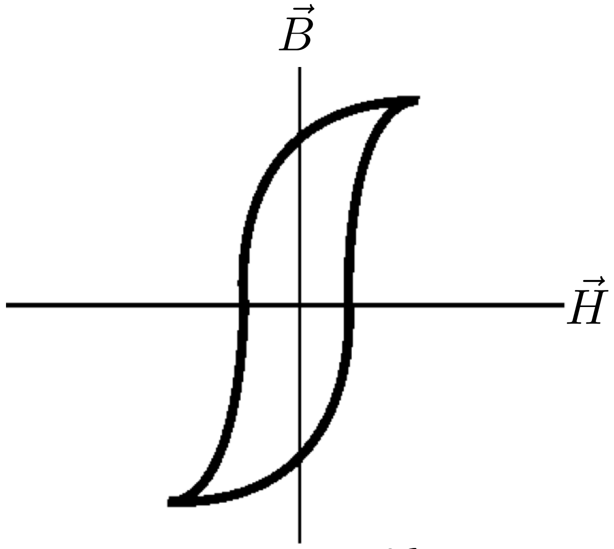
Material properties and geometry determine flow – push relationship





OHM's LAW $\vec{J} = \sigma \vec{E}_{DC}$

$$\vec{B} = \mu_o \mu_r \vec{H}$$



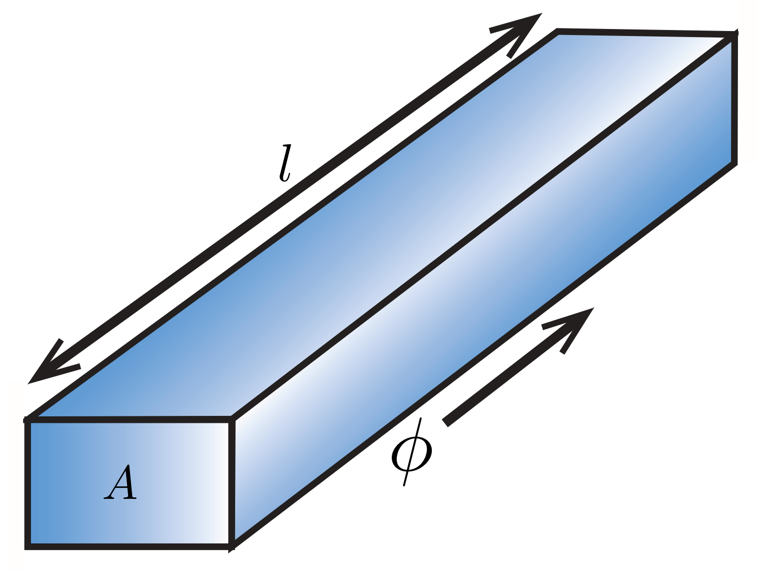
Recovering macroscopic variables:

$$I = \int \vec{J} \cdot d\vec{A} = \sigma \int \vec{E} \cdot d\vec{A} = \sigma \frac{V}{l} A$$

$$V = I \frac{l}{\sigma A} = I \frac{\rho l}{A} = IR$$

$$Ni = \Phi \Re$$

Reluctance of Magnetic Bar



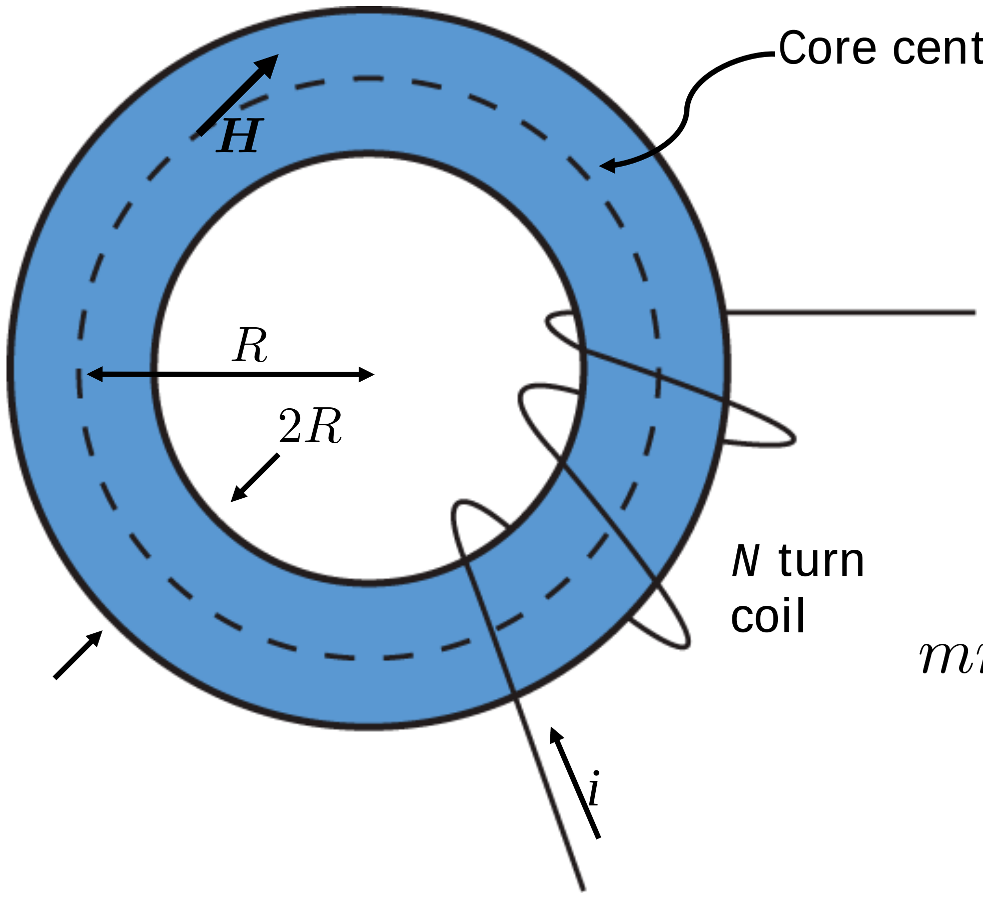
Magnetic "OHM's LAW"

$$Ni = \Phi \Re$$

$$\Re = \frac{l}{\mu A}$$

The reluctance $\mathcal{R}$ of a magnetic path depends on the mean length l, the area A, and the permeability μ of the material.

Flux Density in a Toroidal Core



$$B = \frac{\mu N i}{2\pi R}$$

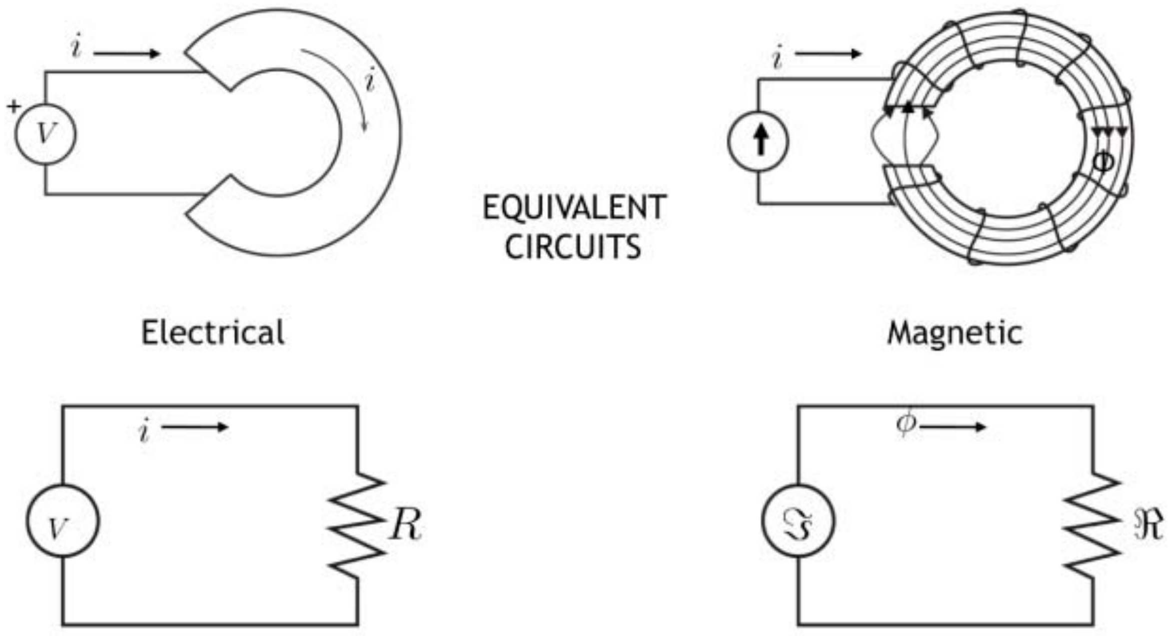
$$\mu N i = 2\pi R B = l B$$

$$mmf = Ni = \frac{lB}{\mu} = \boxed{\Phi \frac{l}{\mu A}}$$

(of an N-turn coil)

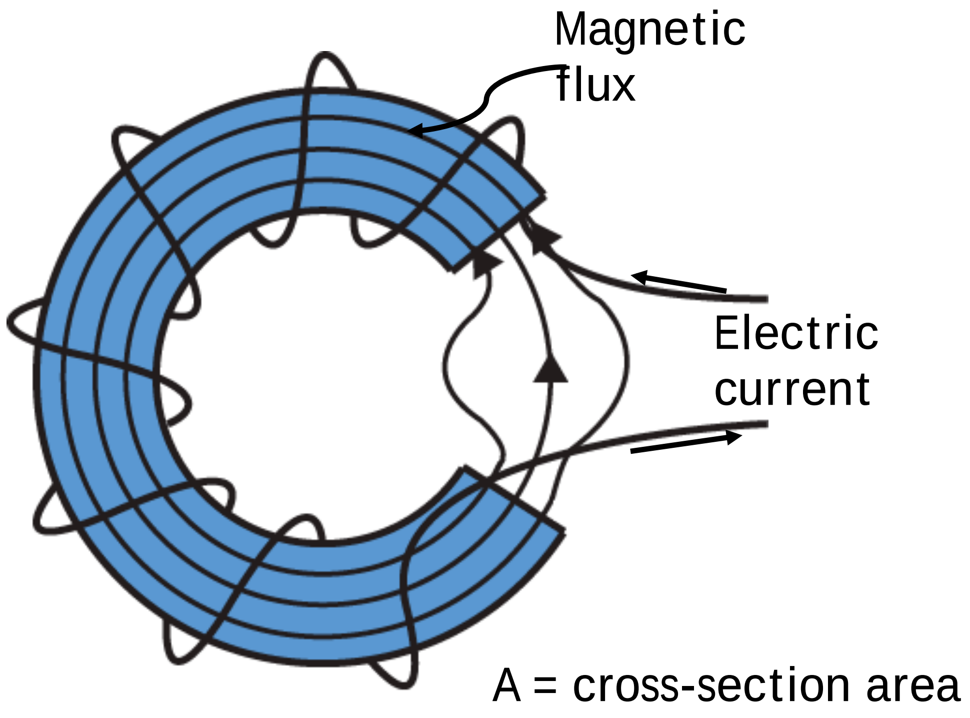
$$mmf = \Phi \Re$$

Electrical Circuit Analogy



Electrical	Magnetic
Voltage v	Magnetomotive Force $\Im = Ni$
Current i	Magnetic Flux ϕ
Resistance R	Reluctance $\Re$
Conductivity $1/\rho$	Permeability μ
Current Density J	Magnetic Flux Density B
Electric Field E	Magnetic Field Intensity H

Toroid with Air Gap

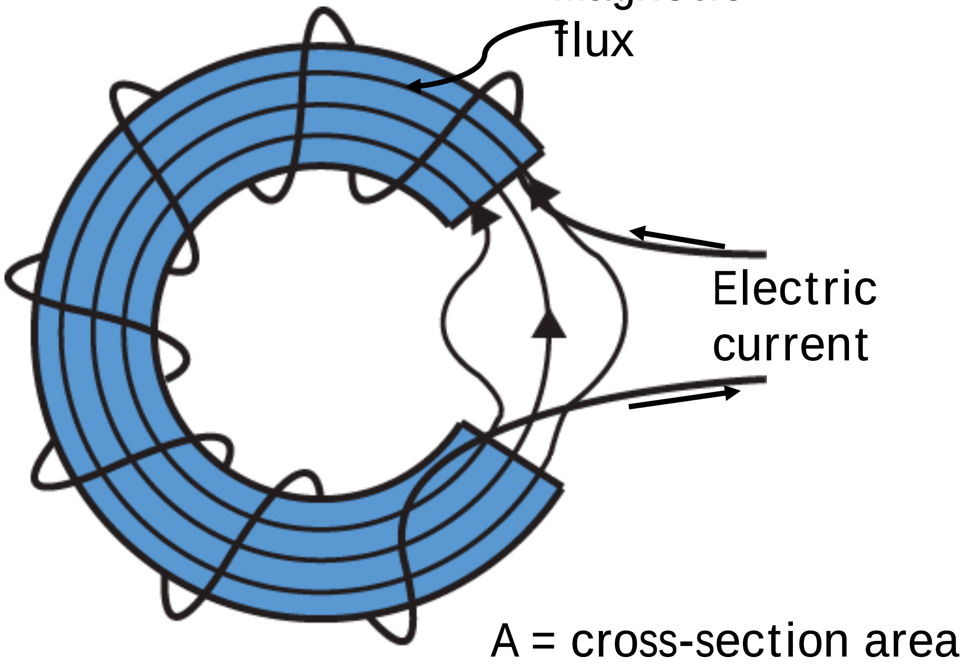


Why is the flux confined mainly to the core ?

Can the reluctance ever be infinite (magnetic insulator) ?

Why does the flux not leak out further in the gap ?

Fields from a Toroid



$$\int_C \vec{H} \cdot d\vec{l} = \int_S \vec{J} \cdot d\vec{A}$$

$$= I_{enclosed}$$

$$H = \frac{Ni}{2\pi R} \xrightarrow{\vec{B} = \mu_o \left(\vec{H} + \vec{M}\right)} B = \mu \frac{Ni}{2\pi R}$$

$$\Phi = BA = Ni \frac{\mu A}{2\pi R}$$

Scaling Magnetic Flux

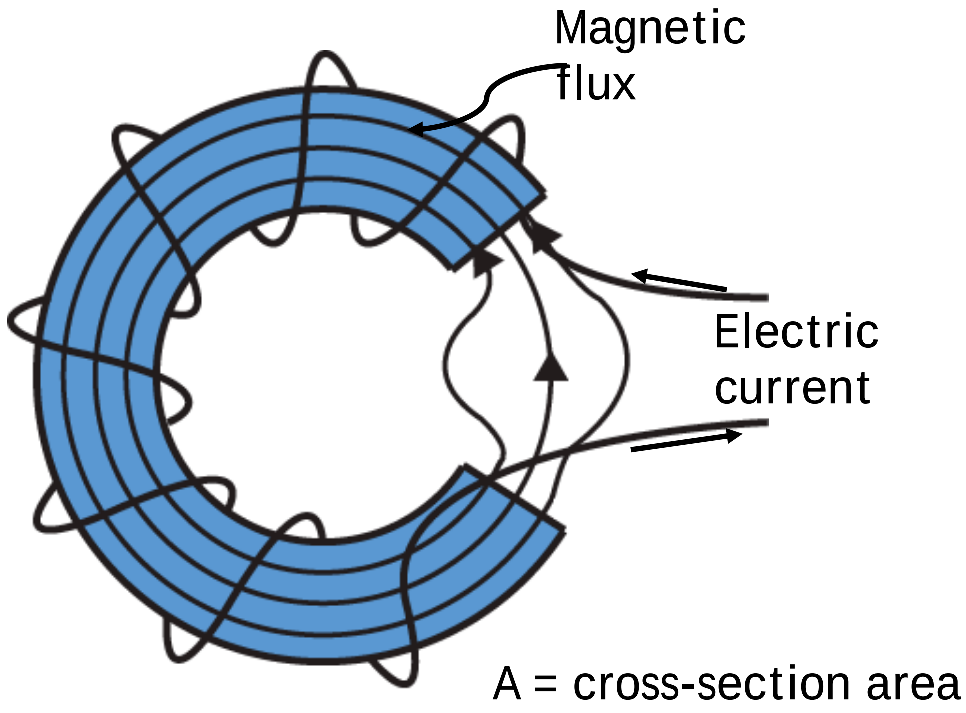
$$Ni = \Phi \Re$$

&

$$\Re = \frac{l}{\mu A} \Rightarrow \Re = \frac{2\pi R}{\mu A}$$

$$\Downarrow$$

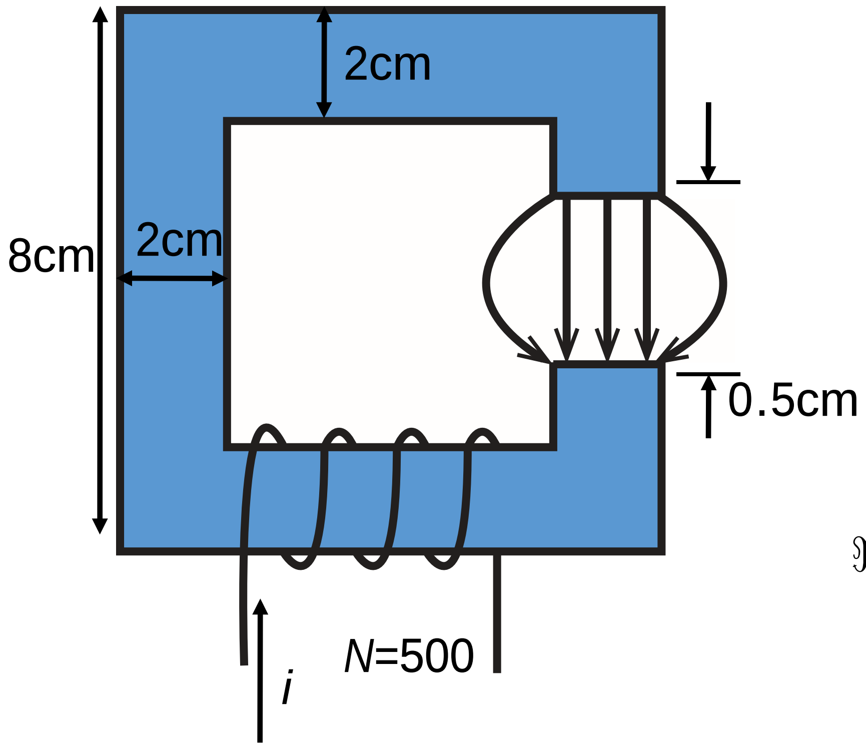
$$\Phi = BA = Ni\frac{\mu A}{2\pi R}$$



Same answer as Ampere’s Law (slide 9)

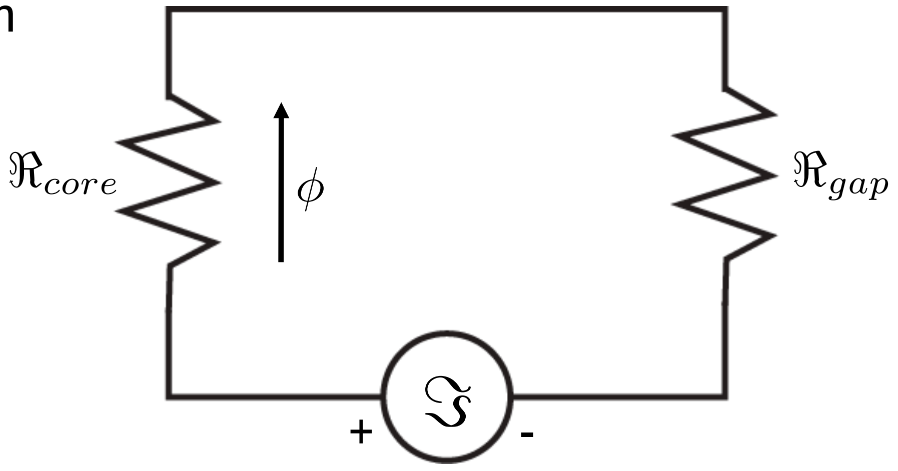
Magnetic Circuit for 'Write Head'

Core Thickness = 3cm



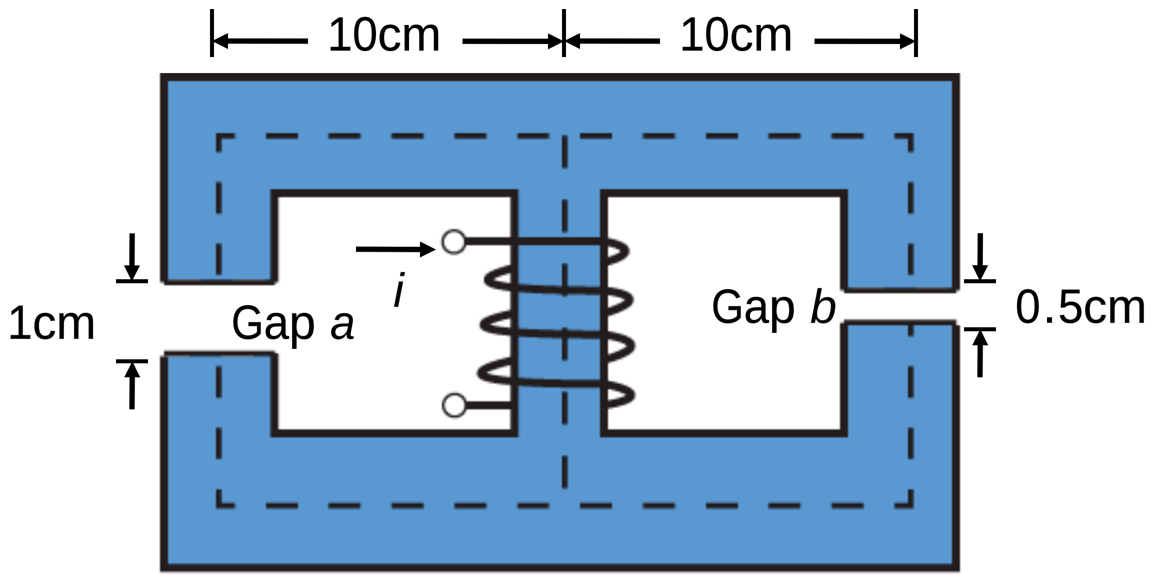
A = cross-section area

$$\Re = \frac{l}{\mu A}$$



$$\Phi \approx \boxed{}$$

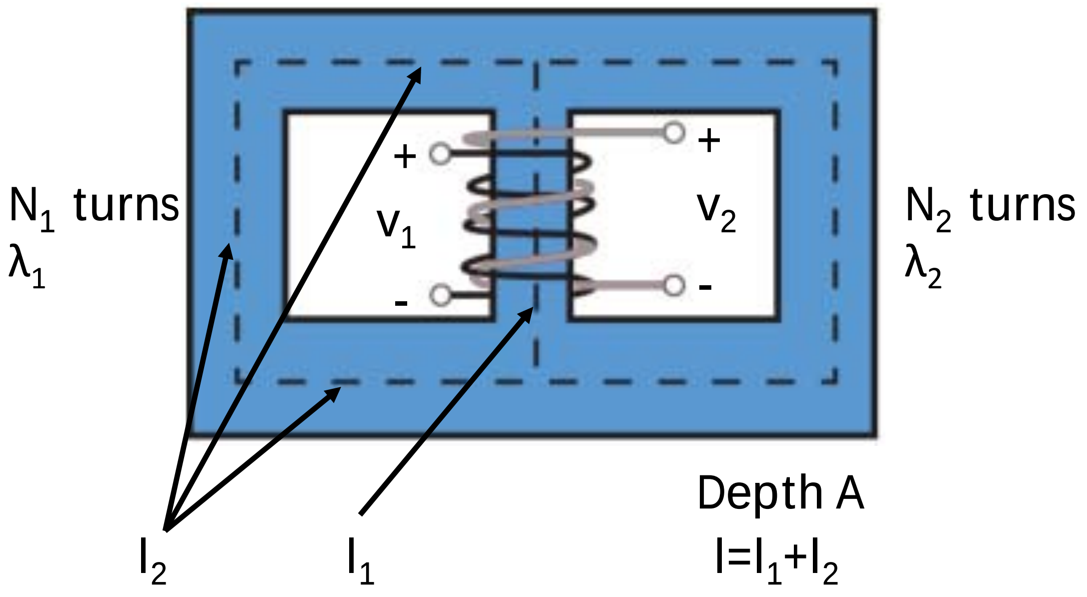
Parallel Magnetic Circuits



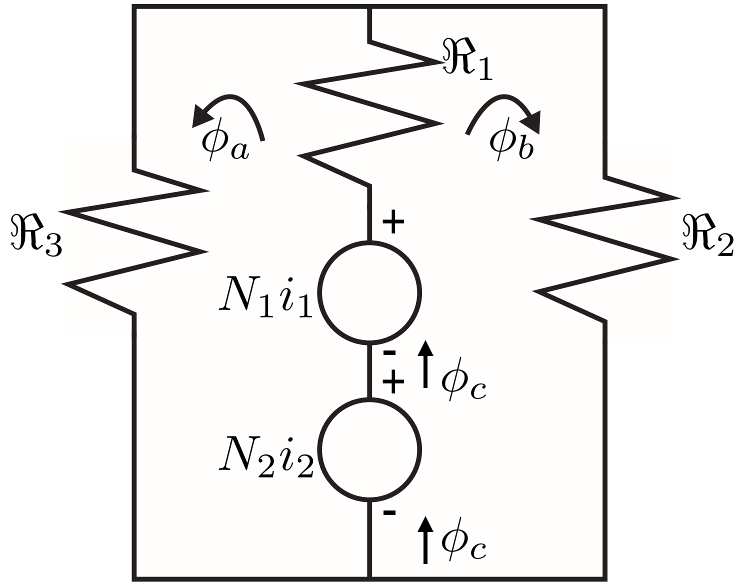
A = cross-section area

A Magnetic Circuit with Reluctances in Series and Parallel

“Shell Type” Transformer

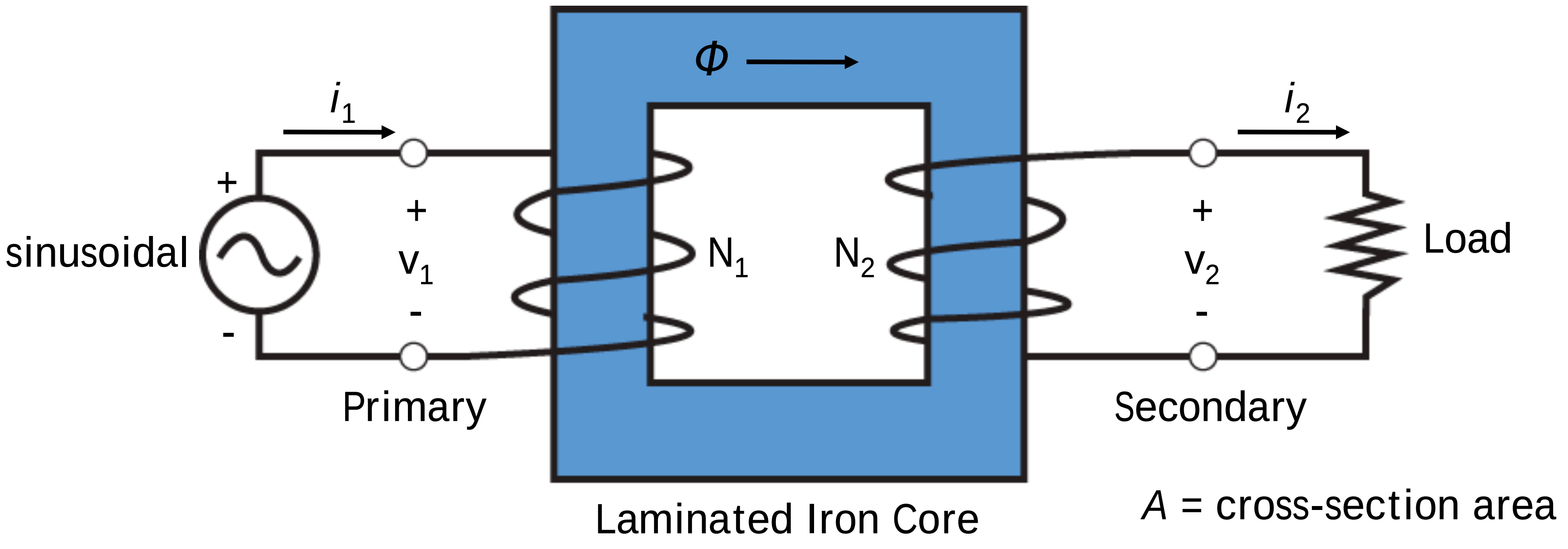


Magnetic Circuit



$$\Re_1 = \frac{l_1}{\mu A} \qquad \Re_2 = \frac{l_2}{\mu A}$$

Faraday Law and Magnetic Circuits



A = cross-section area

Flux linkage $\lambda = N\Phi$ $v = \dfrac{d\lambda}{dt}$

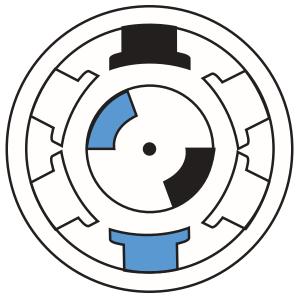
Step 1: Estimate voltage v_1 due to time-varying flux...

Step 2: Estimate voltage v_2 due to time-varying flux...

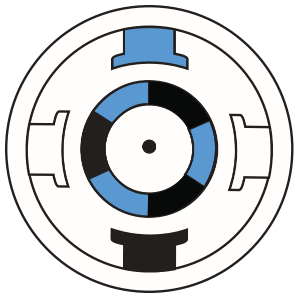
$$\frac{v_2}{v_1} = \boxed{}$$

Complex Magnetic Systems

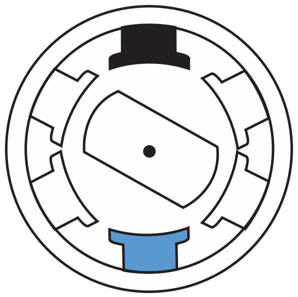
DC Brushless



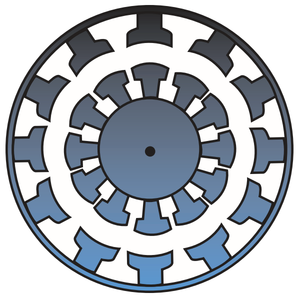
Stepper Motor



Reluctance Motor



Induction Motor



Powerful tools…

Magnetic Circuits: *Reduce Maxwell to (scalar) circuit problem*

Makes it easy to calculate B, H, λ

Energy Method: *Look at change in stored energy to calculate force*

Stored Energy in Inductors

In the absence of mechanical displacement…

$$W_S = \int P_{elec} dt = \int i v dt = \int i \frac{d\lambda}{dt} = \int i\left(\lambda\right) d\lambda$$

For a linear inductor:

$$i\left(\lambda\right) = \frac{\lambda}{L} \quad \Longrightarrow$$

Stored energy…

$$W_S = \int_0^{\lambda} \frac{\lambda'}{L} d\lambda' = \frac{\lambda^2}{2L}$$

Relating Stored Energy to Force

Lets use chain rule ...

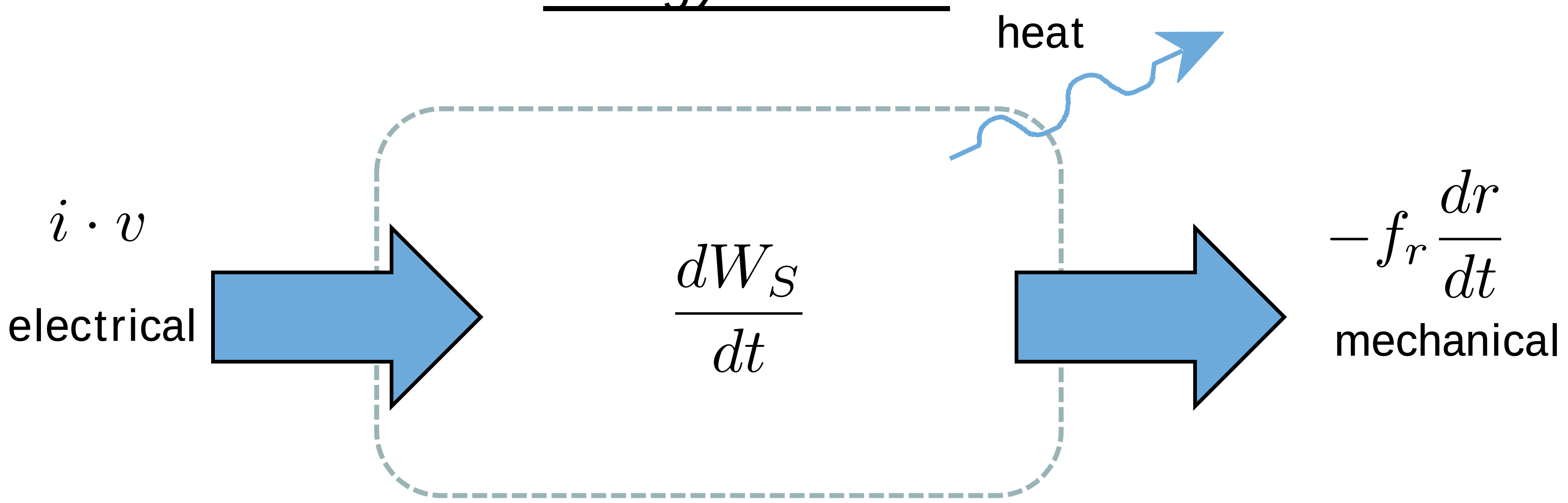
$$\frac{W_S\left(\Phi, r\right)}{dt} = \frac{\partial W_S}{\partial \Phi}\frac{d\Phi}{dt} + \frac{\partial W_S}{\partial r}\frac{dr}{dt}$$

This looks familiar ...

$$\begin{aligned}\frac{dW_S}{dt} &= i \cdot v - f_r\frac{dr}{dt}\\ &= iL\frac{di}{dt} - f_r\frac{dr}{dt}\end{aligned}$$

Comparing similar terms suggests ... $\boxed{f_r = -\frac{\partial W_S}{\partial r}}$

Energy Balance

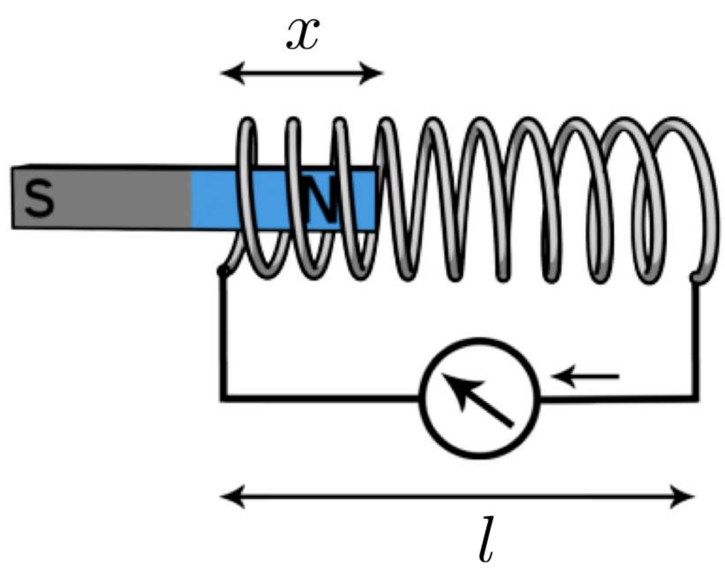


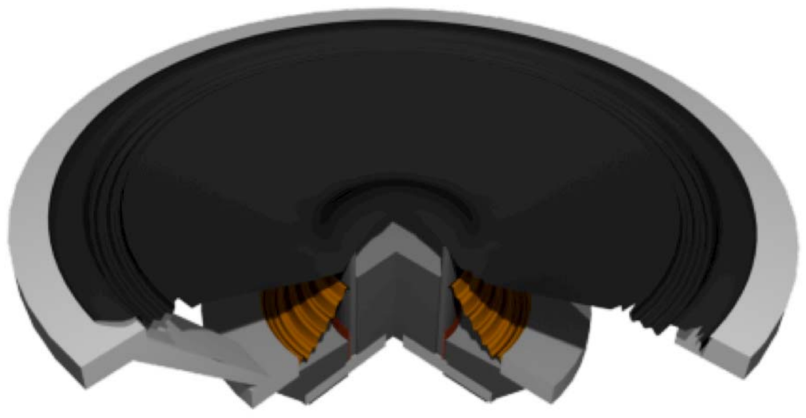
$$\frac{dW_S(\lambda, r)}{dt} = \frac{\partial W_S}{\partial \lambda}\frac{d\lambda}{dt} + \frac{\partial W_S}{\partial r}\frac{dr}{dt}$$ **neglect heat**

For magnetostatic system, dλ=0 **no electrical power** flow…

$$\frac{dW_S}{dt} = -f_r \frac{dr}{dt}$$

Linear Machines: Solenoid Actuator





Coil attached to cone

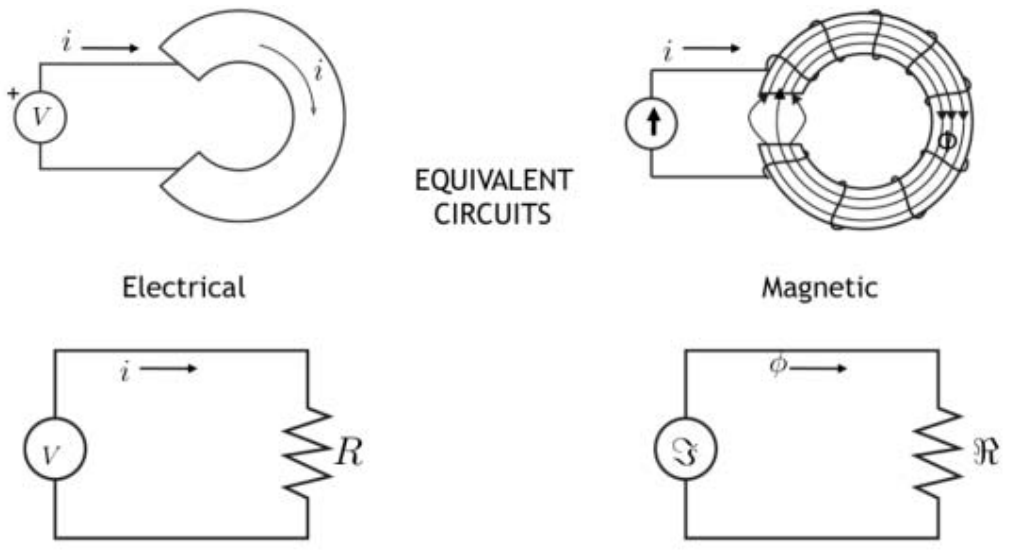
If we can find the stored energy, we can immediately compute the force...

...lets take all the things we know to put this together...

$$f_r = -\frac{\partial W_S}{\partial r}\Big|_\lambda \qquad W_S\left(\lambda, r\right) = \frac{1}{2}\frac{\lambda^2}{L}$$

KEY TAKEAWAYS

COMPLETE AMPERE'S LAW $$\int_C \vec{H}\cdot d\vec{l} = \int_S \vec{J}\cdot d\vec{A} + \frac{d}{dt}\int_S \epsilon\vec{E}\cdot d\vec{A}$$



Electrical	Magnetic
Voltage v	Magnetomotive Force $\Im = Ni$
Current i	Magnetic Flux ϕ
Resistance R	Reluctance $\Re$
Conductivity $1/\rho$	Permeability μ
Current Density J	Magnetic Flux Density B
Electric Field E	Magnetic Field Intensity H

RELUCTANCE

$$\Re = \frac{l}{\mu A}$$

NETWORK THEOREMS

- KIRCHHOFFS LAWS
- MESH ANALYSIS
- NODAL ANALYSIS
- NORTAN
- SUPERPOSITION
- THEVENIN
- MAXIMUM POWER TRANSFER

Kirchhoff's Laws

Kirchhoff's circuit laws are two equalities that deal with the conservation of charge and energy in electrical circuits.

There basically two Kirchhoff's law :-

1. **Kirchhoff's current law (KCL)** – Based on principle of conservation of electric charge.
2. **Kirchhoff's voltage law (KVL) -** Based on principle of conservation of energy.

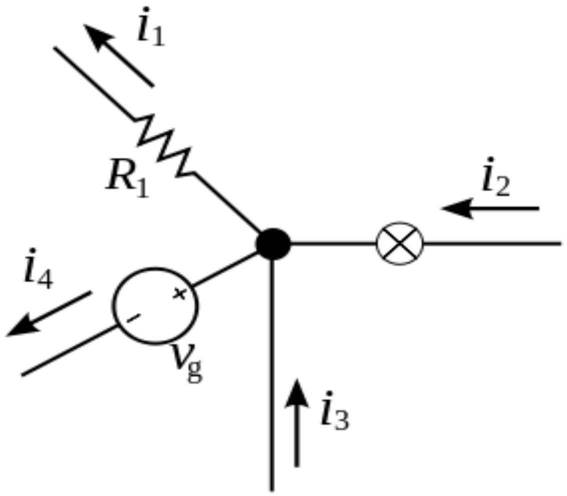
Kirchhoff's current law (KCL)

This law is also called **Kirchhoff's first law**, **Kirchhoff's point rule**, **Kirchhoff's junction rule** (or nodal rule), and **Kirchhoff's first rule**.

The principle of conservation of electric charge implies that:

At any node (junction) in an electrical circuit, the sum of currents flowing into that node is equal to the sum of currents flowing out of that node, or The algebraic sum of currents in a network of conductors meeting at a point is zero.

Strictly speaking KCL only applies to circuits with steady currents (DC).
However, for AC circuits having dimensions much smaller than a wavelength, KCL is also approximately applicable.



The current entering any junction is equal to the current leaving that junction. $i_1 + i_4 = i_2 + i_3$

Recalling that current is a signed (positive or negative) quantity reflecting direction towards or away from a node, this principle can be stated as:

$$\boxed{\sum I = 0}$$

Kirchhoff's voltage law (KVL)

This law is also called **Kirchhoff's second law**, **Kirchhoff's loop (or mesh) rule**, and **Kirchhoff's second rule**.

The principle of conservation of energy implies that

The directed sum of the electrical potential differences (voltage) around any closed circuit is zero, or

More simply, the sum of the emfs in any closed loop is equivalent to the sum of the potential drops in that loop

Strictly speaking KVL only applies to circuits with steady currents (DC).

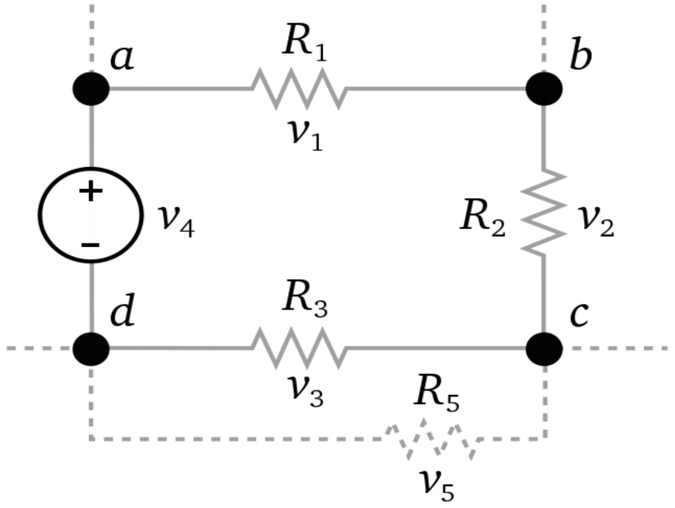
However, for AC circuits having dimensions much smaller than a wavelength, KVL is also approximately applicable.

The algebraic sum of the products of the resistances of the conductors and the currents in them in a closed loop is equal to the total emf available in that loop. Similarly to KCL, it can be stated as:

$$\boxed{\sum V_{emf} = I \sum R}$$

OR

KVL: $\boxed{\sum_{\text{loop}} V_n = 0}$



The sum of all the voltages around the loop is equal to zero. $v_1 + v_2 + v_3 - v_4 = 0$

Mesh Analysis

Mesh analysis (or the **mesh current method**) is a method that is used to solve planar circuits for the currents (and indirectly the voltages) at any place in the circuit. Planar circuits are circuits that can be drawn on a plane surface with no wires crossing each other.

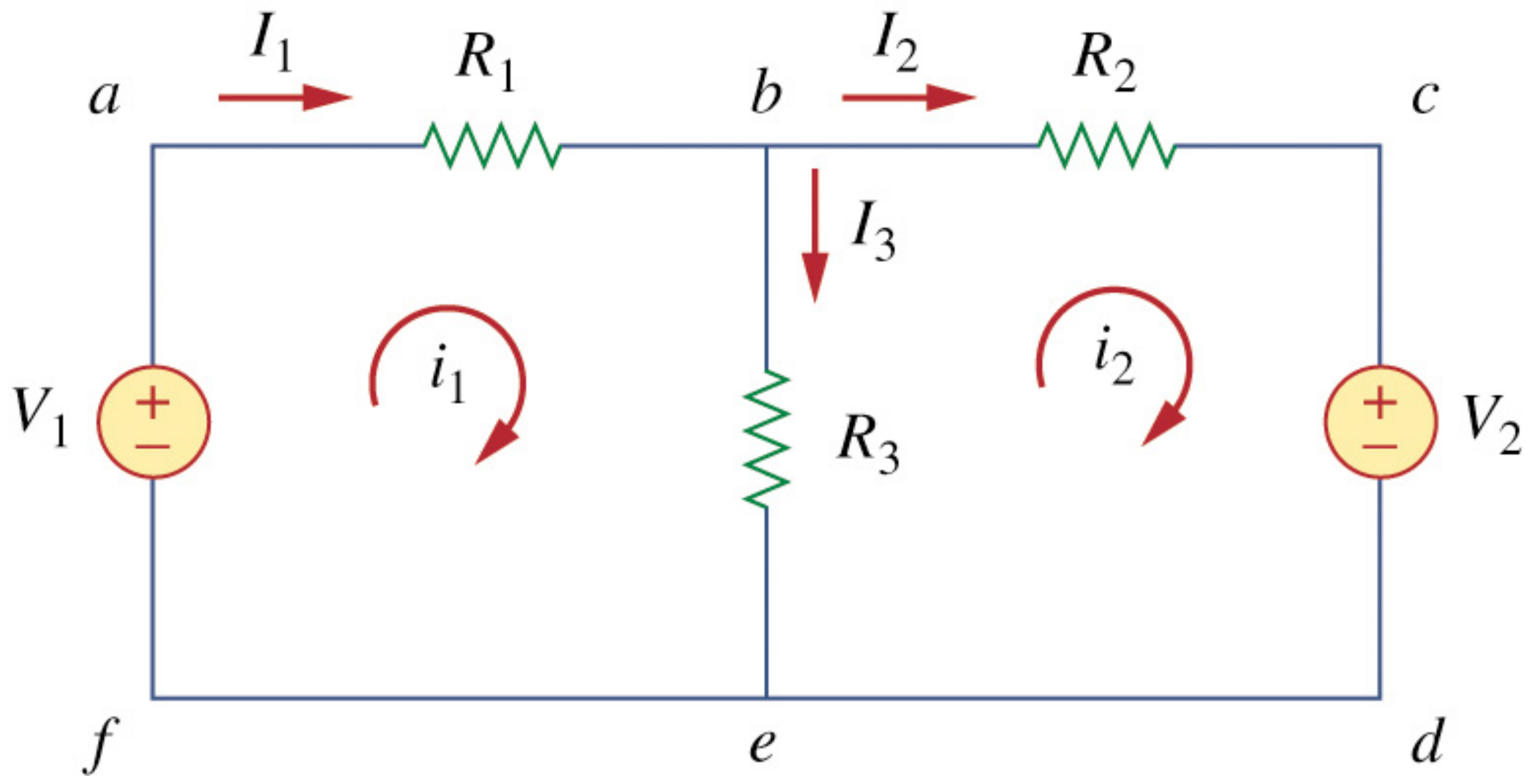
Mesh analysis works by arbitrarily assigning mesh currents in the essential meshes. An essential mesh is a loop in the circuit that does not contain any other loop.

Steps to Determine Mesh Currents:

1. Assign mesh currents i_1, i_2, .., i_n to the n meshes. Current direction need to be same in all meshes either clockwise or anticlockwise.
2. Apply KVL to each of the n meshes. Use Ohm's law to express the voltages in terms of the mesh currents.
3. Solve the resulting *n* simultaneous equations to get the mesh currents

Example

A circuit with two meshes.



Apply KVL to each mesh. For mesh 1,

$$-V_1 + R_1 i_1 + R_3(i_1 - i_2) = 0$$

$$(R_1 + R_3)i_1 - R_3 i_2 = V_1$$

For mesh 2,

$$R_2 i_2 + V_2 + R_3(i_2 - i_1) = 0$$

$$-R_3 i_1 + (R_2 + R_3)i_2 = -V_2$$

Solve for the mesh currents.

$$\begin{bmatrix} R_1 + R_3 & -R_3 \\ -R_3 & R_2 + R_3 \end{bmatrix} \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} = \begin{bmatrix} V_1 \\ -V_2 \end{bmatrix}$$

Use *i* for a mesh current and *I* for a branch current. It’s evident from Fig. 3.17 that

$$\boxed{I_1 = i_1, \quad I_2 = i_2, \quad I_3 = i_1 - i_2}$$

Nodal Analysis

In electric circuits analysis, **nodal analysis**, **node-voltage analysis**, or the **branch current method** is a method of determining the voltage (potential difference) between "nodes" (points where elements or branches connect) in an electrical circuit in terms of the branch currents.

Nodal analysis is possible when all the circuit elements branch constitutive relations have an admittance representation.

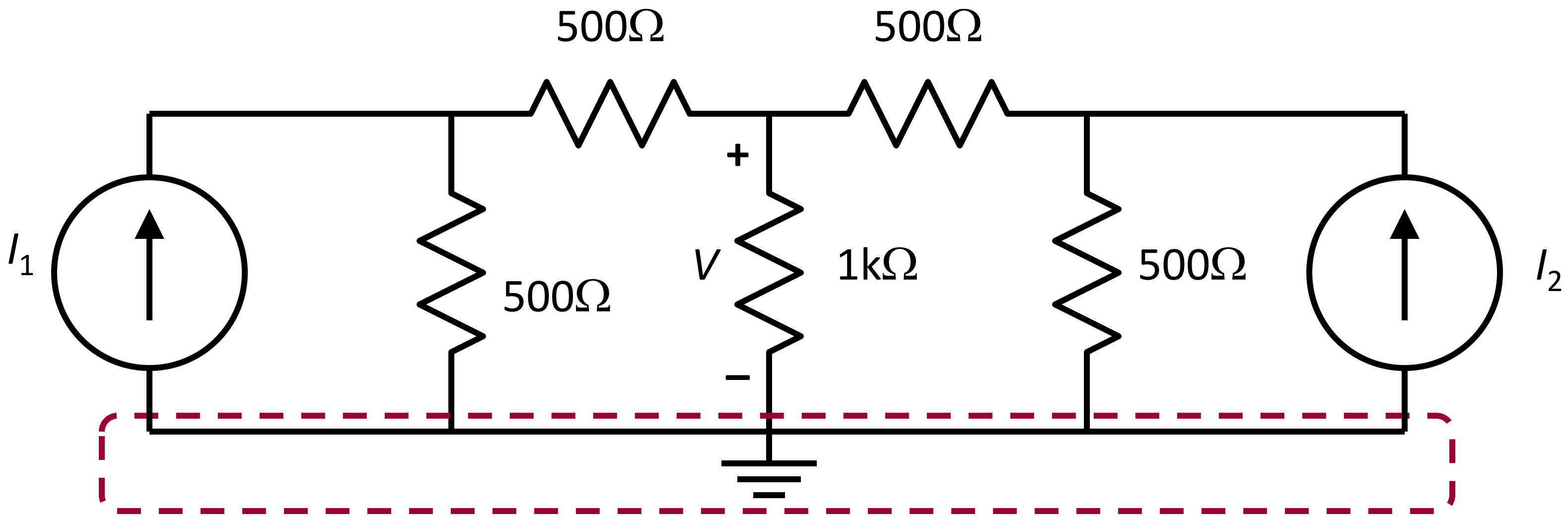
Kirchhoff’s current law is used to develop the method referred to as **nodal analysis**

STEPS FOR NODAL ANALYSIS:-

- Note all connected wire segments in the circuit. These are the *nodes* of nodal analysis.
- Select one node as the ground reference. The choice does not affect the result and is just a matter of convention. Choosing the node with most connections can simplify the analysis.
- Assign a variable for each node whose voltage is unknown. If the voltage is already known, it is not necessary to assign a variable.
- For each unknown voltage, form an equation based on Kirchhoff's current law. Basically, add together all currents leaving from the node and mark the sum equal to zero.

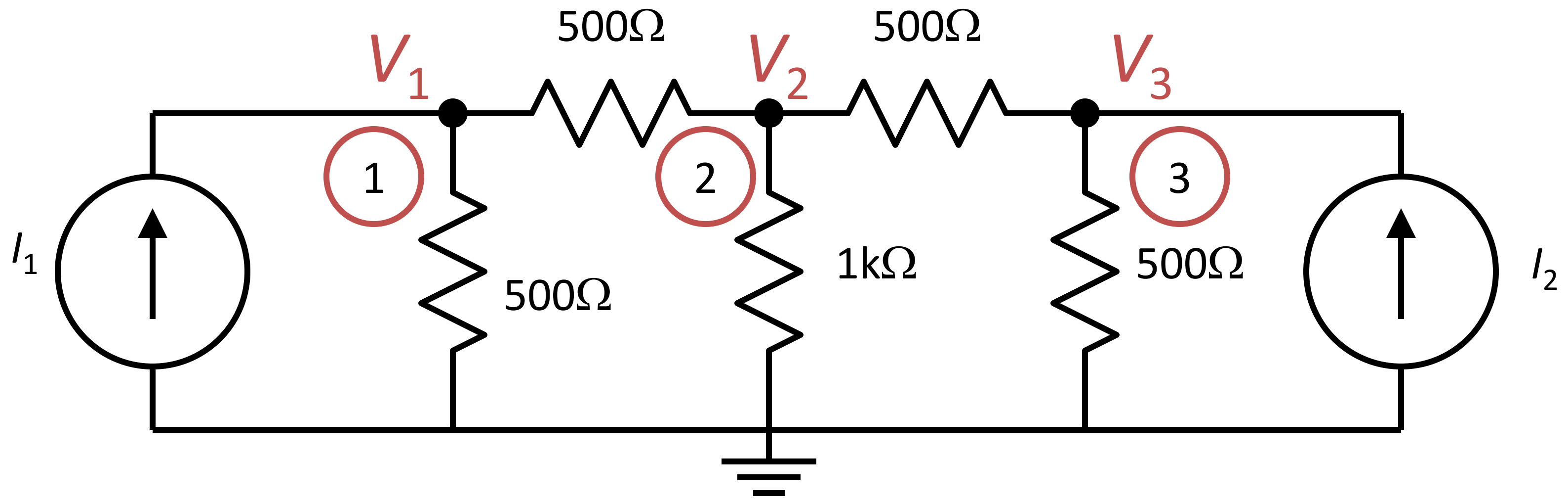
- If there are voltage sources between two unknown voltages, join the two nodes as a super node. The currents of the two nodes are combined in a single equation, and a new equation for the voltages is formed.
- Solve the system of simultaneous equations for each unknown voltage.

1. Reference Node



The reference node is called the *ground* node where $V = 0$

Example

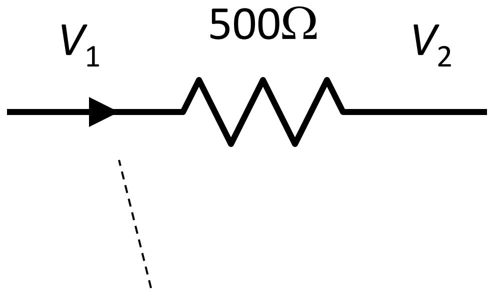


V_1, V_2, and V_3 are unknowns for which we solve using KCL

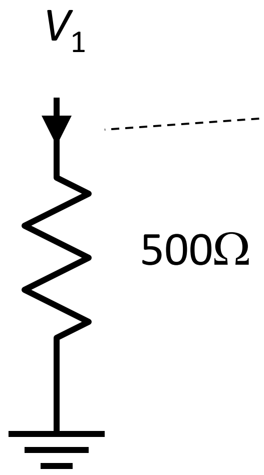
Steps of Nodal Analysis

1. Choose a reference (ground) node.
2. Assign node voltages to the other nodes.
3. Apply KCL to each node other than the reference node; express currents in terms of node voltages.
4. Solve the resulting system of linear equations for the nodal voltages.

Currents and Node Voltages

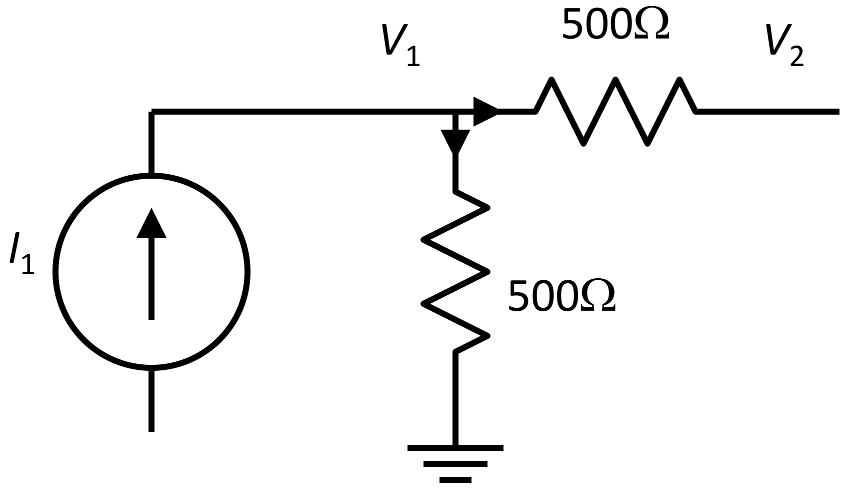


$$\frac{V_1 - V_2}{500\Omega}$$



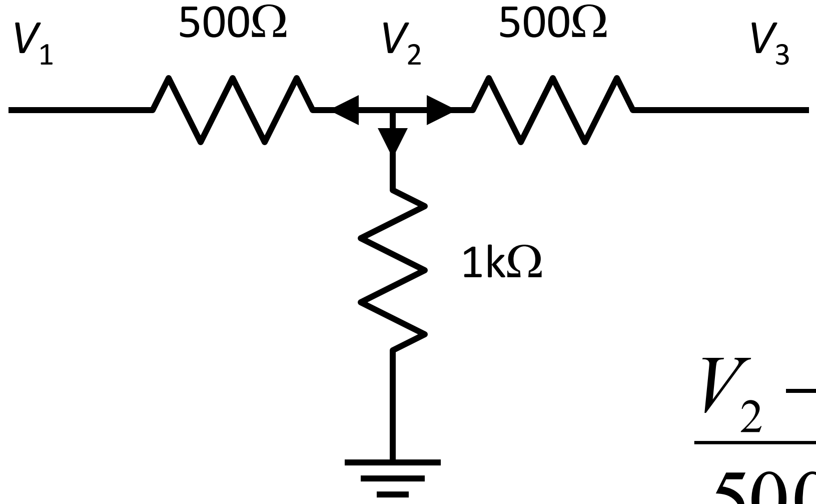
$$\frac{V_1}{500\Omega}$$

3. KCL at Node 1



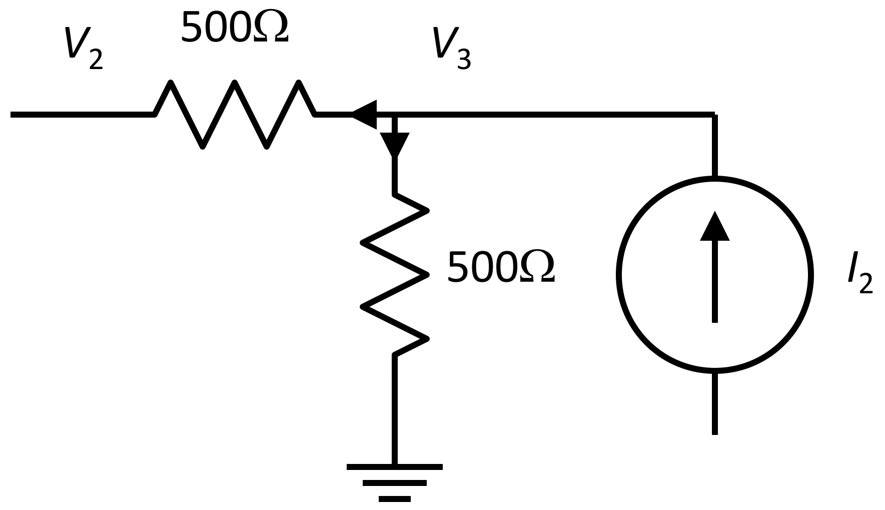
$$I_1 = \frac{V_1 - V_2}{500\Omega} + \frac{V_1}{500\Omega}$$

3. KCL at Node 2



$$\frac{V_2 - V_1}{500\Omega} + \frac{V_2}{1\text{k}\Omega} + \frac{V_2 - V_3}{500\Omega} = 0$$

3. KCL at Node 3



$$\frac{V_3 - V_2}{500\Omega} + \frac{V_3}{500\Omega} = I_2$$

Superposition Theorem

- It is used to find the solution to networks with two or more sources that are not in series or parallel
- The current through, or voltage across, an element in a linear bilateral network is equal to the algebraic sum of the currents or voltages produced independently by each source.
- For a two-source network, if the current produced by one source is in one direction, while that produced by the other is in the opposite direction through the same resistor, the resulting current is the difference of the two and has the direction of the larger
- If the individual currents are in the same direction, the resulting current is the sum of two in the direction of either current

Superposition Theorem

- The total power delivered to a resistive element must be determined using the total current through or the total voltage across the element and cannot be determined by a simple sum of the power levels established by each source

For applying Superposition theorem:-

- Replace all other independent voltage sources with a short circuit (thereby eliminating difference of potential. i.e. V=0, internal impedance of ideal voltage source is ZERO (short circuit)).
- Replace all other independent current sources with an open circuit (thereby eliminating current. i.e. I=0, internal impedance of ideal current source is infinite (open circuit).

Example:- Determine the branches current using Superposition theorem.

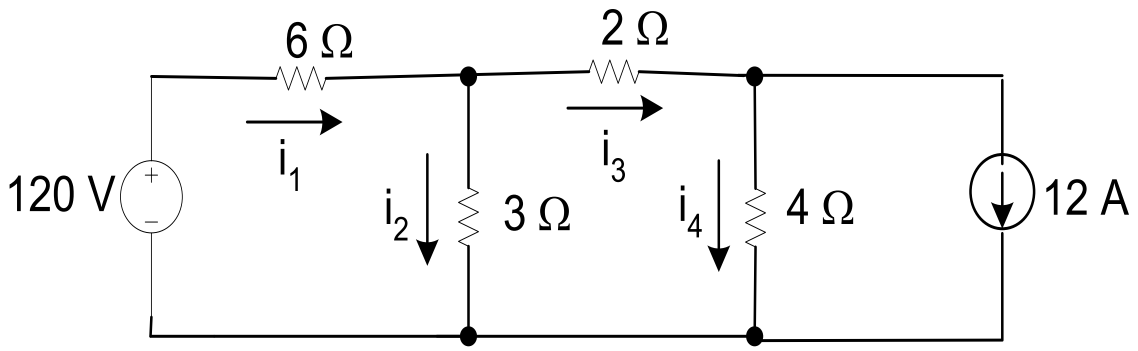


Figure 1

Solution:

- The application of the superposition theorem is shown in Figure 1, where it is used to calculate the branch current. We begin by calculating the branch current caused by the voltage source of 120 V. By substituting the ideal current with open circuit, we deactivate the current source, as shown in Figure 2.

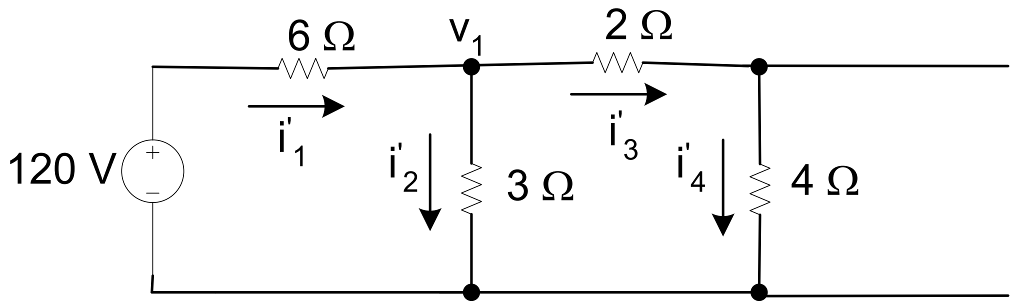


Figure 2

- To calculate the branch current, the node voltage across the 3Ω resistor must be known. Therefore

$$\frac{v_1 - 120}{6} + \frac{v_1}{3} + \frac{v_1}{2+4} = 0$$

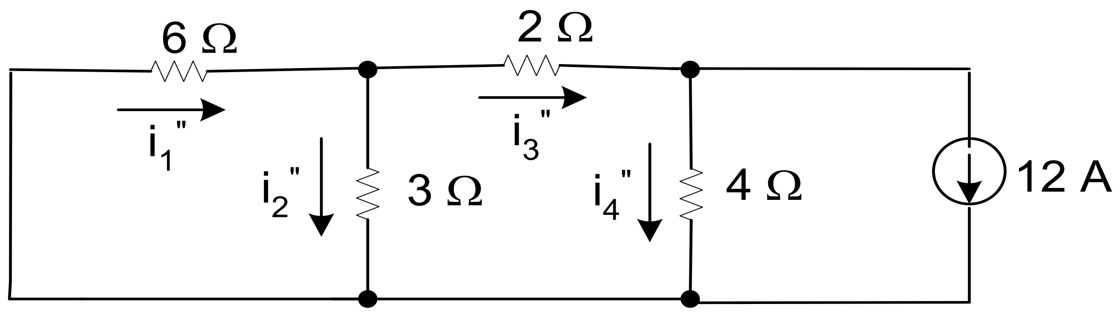
where $v_1 = 30$ V

The equations for the current in each branch,

$$i'_1 = \frac{120 - 30}{6} = 15\ \text{A}$$

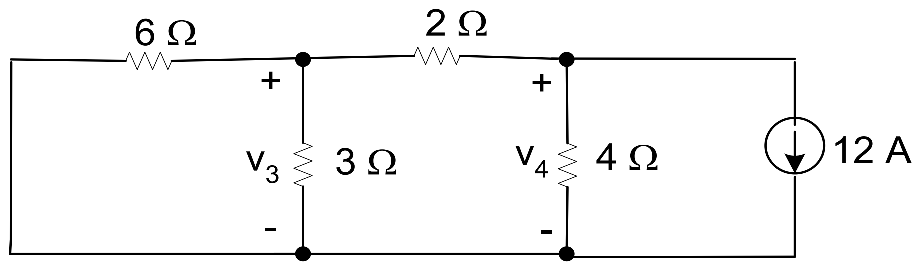
$$i'_2 = \frac{30}{3} = 10\ \text{A}$$

$$i'_3 = i'_4 = \frac{30}{6} = 5\ \text{A}$$



In order to calculate the current cause by the current source, we deactivate the ideal voltage source with a short circuit, as shown

To determine the branch current, solve the node voltages across the 3Ω dan 4Ω resistors as shown in Figure 4



$$\frac{v_3}{3}+\frac{v_3}{6}+\frac{v_3-v_4}{2} = 0$$

$$\frac{v_4-v_3}{2}+\frac{v_4}{4}+12 = 0$$

The two node voltages are

- By solving these equations, we obtain

v_3 = -12 V

v_4 = -24 V

Now we can find the branches current,

$$i_1'' = \frac{-v_3}{6} = \frac{12}{6} = 2A$$

$$i_2'' = \frac{v_3}{3} = \frac{-12}{3} = -4A$$

$$i_3'' = \frac{v_3 - v_4}{2} = \frac{-12+24}{2} = 6A$$

$$i_4'' = \frac{v_4}{4} = \frac{-24}{4} = -6A$$

To find the actual current of the circuit, add the currents due to both the current and voltage source,

$$i_1 = i'_1 + i''_1 = 15 + 2 = 17\,A$$

$$i_2 = i'_2 + i''_2 = 10 - 4 = 6\,A$$

$$i_3 = i'_3 + i''_3 = 5 + 6 = 11\,A$$

$$i_4 = i'_4 + i''_4 = 5 - 6 = -1\,A$$

Thevenin's theorem

Thevenin's theorem for linear electrical networks states that any combination of voltage sources, current sources, and resistors with two terminals is electrically equivalent to a single voltage source *V* and a single series resistor *R.*

Any two-terminal, linear bilateral dc network can be replaced by an equivalent circuit consisting of a voltage source and a series resistor

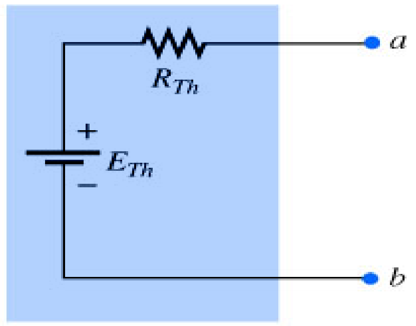


FIG. 9.24
Thévenin equivalent circuit.

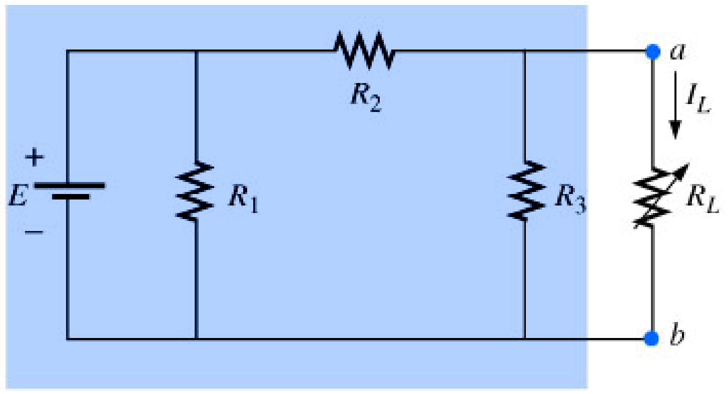
Thévenin’s Theorem

The Thévenin equivalent circuit provides an equivalence at the terminals only – the internal construction and characteristics of the original network and the Thévenin equivalent are usually quite different

- This theorem achieves two important objectives:
  - Provides a way to find any particular voltage or current in a linear network with one, two, or any other number of sources
  - We can concentration on a specific portion of a network by replacing the remaining network with an equivalent circuit

Calculating the Thévenin equivalent

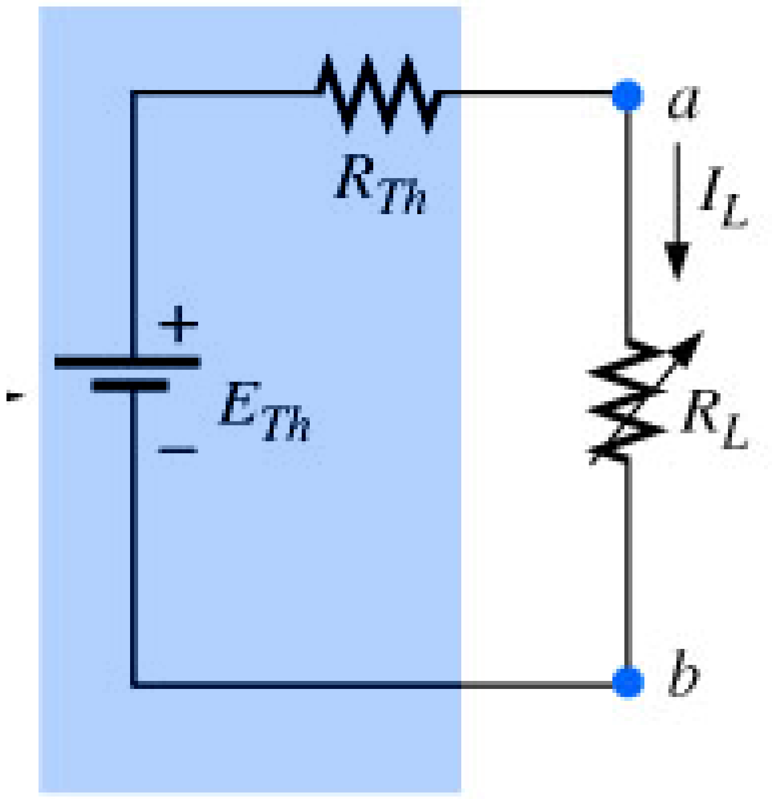
- Sequence to proper value of R_{Th} and E_{Th}
- Preliminary
  - 1. Remove that portion of the network across which the Thévenin equation circuit is to be found. In the figure below, this requires that the load resistor R_L be temporarily removed from the network.



- 2. Mark the terminals of the remaining two-terminal network. (The importance of this step will become obvious as we progress through some complex networks)
- R_{Th}:
- 3. Calculate R_{Th} by first setting all sources to zero (voltage sources are replaced by short circuits, and current sources by open circuits) and then finding the resultant resistance between the two marked terminals. (If the internal resistance of the voltage and/or current sources is included in the original network, it must remain when the sources are set to zero)

- E_{Th}:
  - 4. Calculate E_{Th} by first returning all sources to their original position and finding the open-circuit voltage between the marked terminals. (This step is invariably the one that will lead to the most confusion and errors. In all cases, keep in mind that it is the open-circuit potential between the two terminals marked in step 2)

- Conclusion:
  - 5. Draw the Thévenin equivalent circuit with the portion of the circuit previously removed replaced between the terminals of the equivalent circuit. This step is indicated by the placement of the resistor R_L between the terminals of the Thévenin equivalent circuit

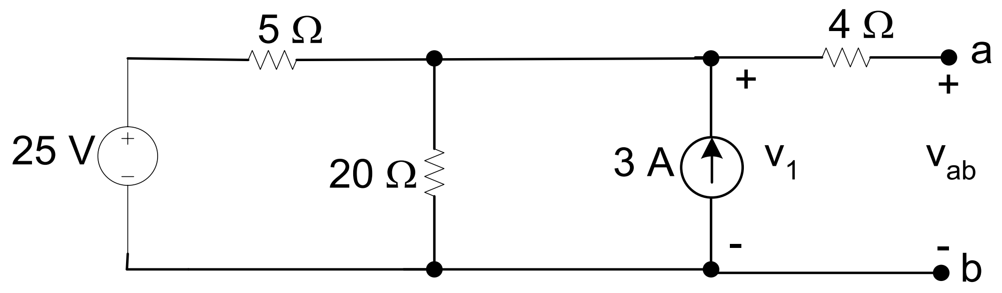


Another way of Calculating the Thévenin equivalent

- Measuring V_{OC} and I_{SC}
  - The Thévenin voltage is again determined by measuring the open-circuit voltage across the terminals of interest; that is, $E_{Th} = V_{OC}$. To determine R_{Th}, a short-circuit condition is established across the terminals of interest and the current through the short circuit I_{sc} is measured with an ammeter
  - Using Ohm's law:

$$R_{Th} = V_{oc} / I_{sc}$$

Example:- find the Thevenin equivalent circuit.



Solution

- In order to find the Thevenin equivalent circuit for the circuit shown in Figure1 , calculate the open circuit voltage, Vab. Note that when the a, b terminals are open, there is no current flow to 4Ω resistor. Therefore, the voltage vab is the same as the voltage across the 3A current source, labeled v_1.
- To find the voltage v_1, solve the equations for the singular node voltage. By choosing the bottom right node as the reference node,

$$\frac{v_1 - 25}{5} + \frac{v_1}{20} - 3 = 0$$

- By solving the equation, v1 = 32 V. Therefore, the Thevenin voltage Vth for the circuit is 32 V.
- The next step is to short circuit the terminals and find the short circuit current for the circuit shown in Figure 2. Note that the current is in the same direction as the falling voltage at the terminal.

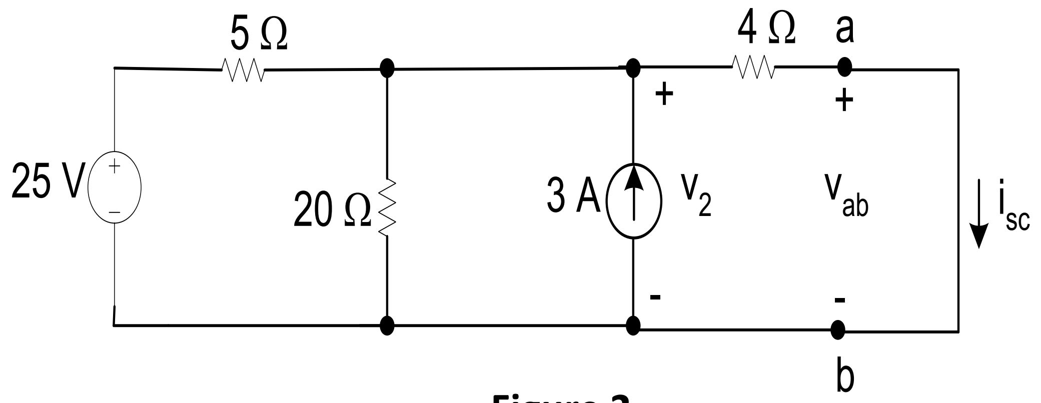


Figure 2

Current ***i***$_{sc}$ can be found if v_2 is known. By using the bottom right node as the reference node, the equationfor v_2 becomes

By solving the above equation, v_2 = 16 V. Therefore, the short circuit
current i_{sc} is

$$\frac{v_2 - 25}{5} + \frac{v_2}{20} - 3 + \frac{v_2}{4} = 0$$

$$i_{sc} = \frac{16}{4} = 4A$$

The Thevenin resistance R_{Th} is

$$R_{Th} = \frac{V_{Th}}{i_{sc}} = \frac{32}{4} = 8\Omega$$

Figure 3 shows the Thevenin equivalent circuit for the Figure 1.

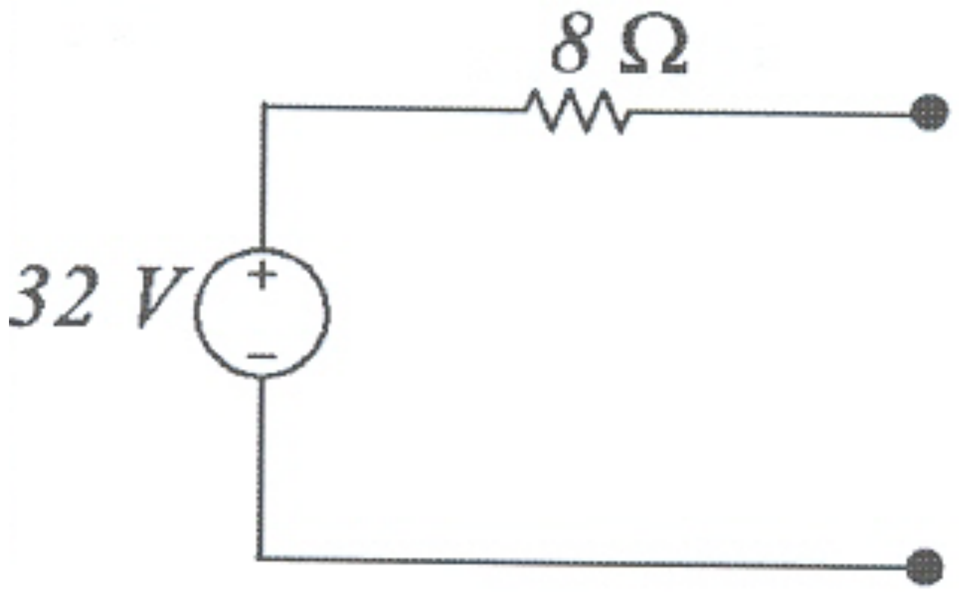
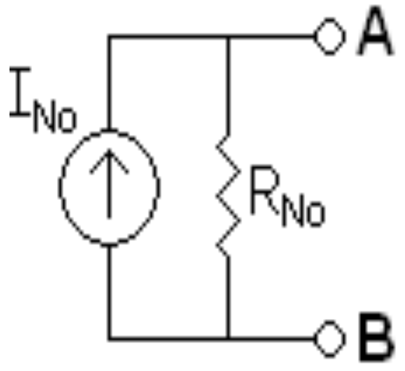


Figure 3

Norton theorem

Norton's theorem for linear electrical networks states that any collection of voltage sources, current sources, and resistors with two terminals is electrically equivalent to an ideal current source, *I*, in parallel with a single resistor.

Any two linear bilateral dc network can be replaced by an equivalent circuit consisting of a current and a parallel resistor.



Calculating the Norton equivalent

- The steps leading to the proper values of I_N and R_N
- Preliminary
  - 1. Remove that portion of the network across which the Norton equivalent circuit is found
  - 2. Mark the terminals of the remaining two-terminal network

- R_N:
  - 3. Calculate R_N by first setting all sources to zero (voltage sources are replaced with short circuits, and current sources with open circuits) and then finding the resultant resistance between the two marked terminals. (If the internal resistance of the voltage and/or current sources is included in the original network, it must remain when the sources are set to zero.) Since $R_N = R_{Th}$ the procedure and value obtained using the approach described for Thévenin's theorem will determine the proper value of R_N

Norton’s Theorem

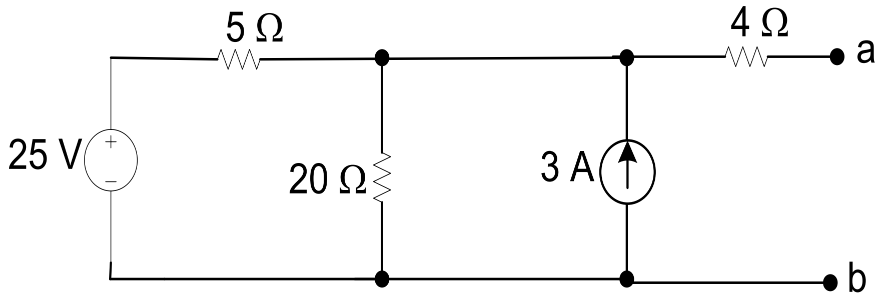
- I_N :
  - 4. Calculate I_N by first returning all the sources to their original position and then finding the short-circuit current between the marked terminals. It is the same current that would be measured by an ammeter placed between the marked terminals.
  - Conclusion:
  - 5. Draw the Norton equivalent circuit with the portion of the circuit previously removed replaced between the terminals of the equivalent circuit

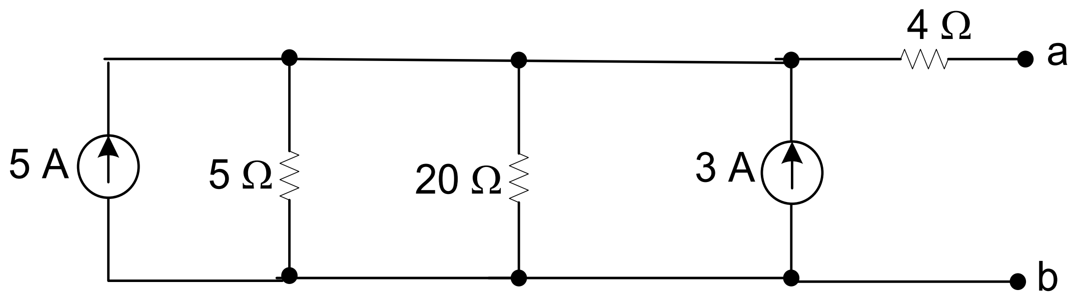
Example

Derive the Norton equivalent circuit

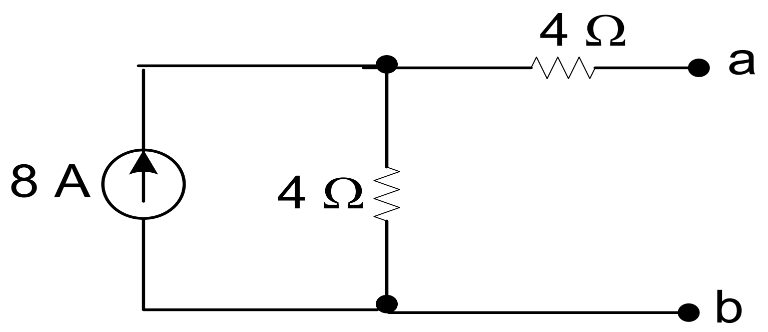
Solution

Step 1: Source transformation (The 25V voltage source is converted to a 5 A current source.)

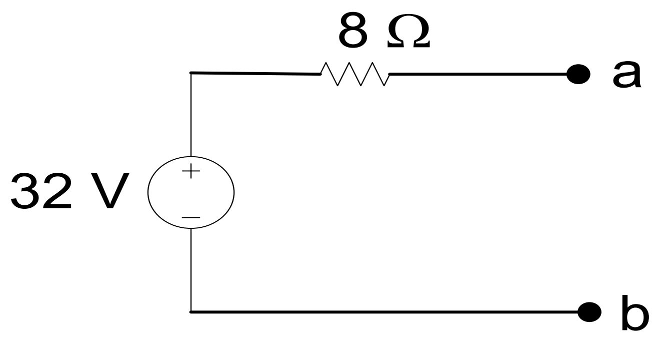




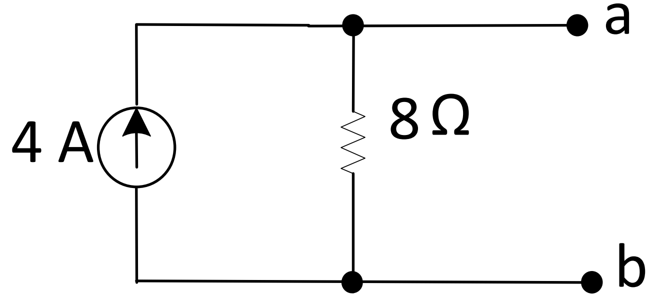
Step 2: Combination of parallel source and parallel resistance



Step 3: Source transformation (combined serial resistance to produce the Thevenin equivalent circuit.)



- Step 4: Source transformation (To produce the Norton equivalent circuit. The current source is 4A (I = V/R = 32 V/8 Ω))



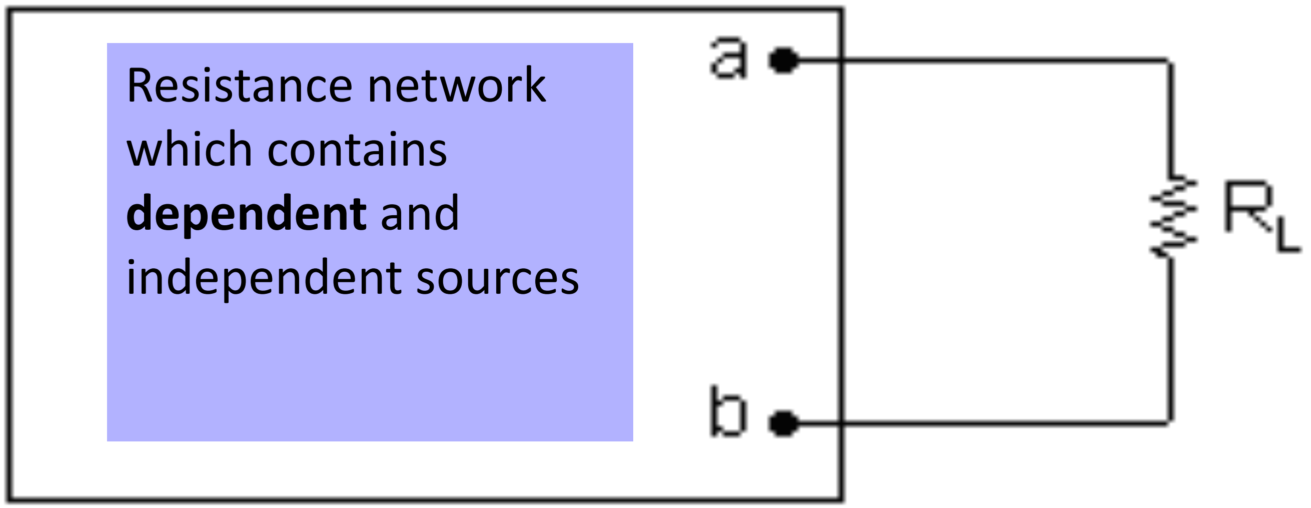
Norton equivalent circuit.

Maximum power transfer theorem

The **maximum power transfer theorem** states that, to obtain maximum external power from a source with a finite internal resistance, the resistance of the load must be equal to the resistance of the source as viewed from the output terminals.

A load will receive maximum power from a linear bilateral dc network when its total resistive value is exactly equal to the Thévenin resistance of the network as “seen” by the load

$$R_L = R_{Th}$$


Resistance network which contains **dependent** and independent sources
a
b
R_L

- Maximum power transfer happens when the load resistance R_L is equal to the Thevenin equivalent resistance, R_{Th}. To find the maximum power delivered to R_L,

$$p_{max} = \frac{V_{Th}{}^2 R_L}{(2R_L)^2} = \frac{V_{Th}{}^2}{4R_L}$$

Application of Network Theorems

- Network theorems are useful in simplifying analysis of some circuits. But the more useful aspect of network theorems is the insight it provides into the properties and behaviour of circuits
- Network theorem also help in visualizing the response of complex network.
- The Superposition Theorem finds use in the study of alternating current (AC) circuits, and semiconductor (amplifier) circuits, where sometimes AC is often mixed (superimposed) with DC

Second Order Linear Differential Equations

Second order linear equations with constant coefficients; Fundamental solutions; Wronskian; Existence and Uniqueness of solutions; the characteristic equation; solutions of homogeneous linear equations; reduction of order; Euler equations

In this chapter we will study ordinary differential equations of the standard form below, known as the *second order linear equations*:

$$\boxed{y'' + p(t)y' + q(t)y = g(t).}$$

Homogeneous Equations: If $g(t) = 0$, then the equation above becomes

$$y'' + p(t)y' + q(t)y = 0.$$

It is called a *homogeneous* equation. Otherwise, the equation is *nonhomogeneous* (or *inhomogeneous*).

Trivial Solution: For the homogeneous equation above, note that the function $y(t) = 0$ always satisfies the given equation, regardless what $p(t)$ and $q(t)$ are. This constant zero solution is called the *trivial solution* of such an equation.

Second Order Linear Homogeneous Differential Equations with Constant Coefficients

For the most part, we will only learn how to solve second order linear equation with constant coefficients (that is, when $p(t)$ and $q(t)$ are constants). Since a homogeneous equation is easier to solve compares to its nonhomogeneous counterpart, we start with second order linear homogeneous equations that contain constant coefficients only:

$$\boxed{a y'' + b y' + c y = 0.}$$

Where a, b, and c are constants, $a \neq 0$.

A very simple instance of such type of equations is

$$y'' - y = 0.$$

The equation's solution is any function satisfying the equality $y'' = y$. Obviously $y_1 = e^t$ is a solution, and so is any constant multiple of it, $C_1 e^t$. Not as obvious, but still easy to see, is that $y_2 = e^{-t}$ is another solution (and so is any function of the form $C_2 e^{-t}$).

It can be easily verified that any function of the form

$$y = C_1 e^t + C_2 e^{-t}$$

will satisfy the equation. In fact, this is the general solution of the above differential equation.

Comment: Unlike first order equations we have seen previously, the general solution of a second order equation has *two* arbitrary coefficients.

Principle of Superposition: If y_1 and y_2 are any two solutions of the homogeneous equation

$$y'' + p(t)y' + q(t)y = 0.$$

Then any function of the form $y = C_1 y_1 + C_2 y_2$ is also a solution of the equation, for any pair of constants C_1 and C_2.

That is, for a homogeneous linear equation, any multiple of a solution is again a solution; any sum/difference of two solutions is again a solution; and the sum / difference of the multiples of any two solutions is again a solution. (This principle holds true for a homogeneous linear equation of *any* order; it is not a property limited only to a second order equation. It, however, **does not** hold, in general, for solutions of a nonhomogeneous linear equation.)

Note: However, while the general solution of $y'' + p(t)y' + q(t)y = 0$ will always be in the form of $C_1 y_1 + C_2 y_2$, where y_1 and y_2 are some solutions of the equation, the converse is not always true. Not every pair of solutions y_1 and y_2 could be used to give a general solution in the form $y = C_1 y_1 + C_2 y_2$. We shall see shortly the exact condition that y_1 and y_2 must satisfy that would give us a general solution of this form.

Fact: The general solution of a second order equation contains two arbitrary constants / coefficients. To find a particular solution, therefore, requires two initial values. The initial conditions for a second order equation will appear in the form: $y(t_0) = y_0$, and $y'(t_0) = y'_0$.

Question: Just by inspection, can you think of two (or more) functions that satisfy the equation $y'' + 4y = 0$? (Hint: A solution of this equation is a function φ such that $\varphi'' = -4\varphi$.)

Example: Find the general solution of

$$y'' - 5y' = 0.$$

There is no need to "guess" an answer here. We actually know a way to solve the equation already. Observe that if we let $u = y'$, then $u' = y''$. Substitute them into the equation and we get a new equation:

$$u' - 5u = 0.$$

This is a first order linear equation with $p(t) = -5$ and $g(t) = 0$. (!)

The integrating factor is $\mu = e^{-5t}$.

$$u(t) = \frac{1}{\mu(t)}\left(\int \mu(t)g(t)\,dt\right) = e^{5t}\left(\int 0\,dt\right) = e^{5t}(C) = Ce^{5t}$$

The actual solution y is given by the relation $u = y'$, and can be found by integration:

$$y(t) = \int u(t)\,dt = \int Ce^{5t}\,dt = \frac{C}{5}e^{5t} + C_2 = C_1 e^{5t} + C_2.$$

The method used in the above example can be used to solve any second order linear equation of the form $y'' + p(t)\,y' = g(t)$, <u>regardless whether its coefficients are constant or nonconstant, or it is a homogeneous equation or nonhomogeneous.</u>

Equations of nonconstant coefficients with missing y-term

If the y-term (that is, the dependent variable term) is missing in a second order linear equation, then the equation can be readily converted into a first order linear equation and solved using the integrating factor method.

Example: $t y'' + 4 y' = t^2$

The standard form is $y'' + \frac{4}{t} y' = t$.

Substitute: $u' + \frac{4}{t} u = t \quad \rightarrow \quad p(t) = \frac{4}{t}, g(t) = t$

Integrating factor is $\mu = t^4$.

$$u(t) = \frac{1}{t^4}\left(\int t^5 \, dt\right) = t^{-4}\left(\frac{1}{6}t^6 + C\right) = \frac{1}{6}t^2 + C t^{-4}$$

Finally,

$$y(t) = \int u(t)\, dt = \frac{1}{18}t^3 - \frac{C}{3}t^{-3} + C_2 = \frac{1}{18}t^3 + C_1 t^{-3} + C_2$$

Comment: Notice the above solution is not in the form of $y = C_1 y_1 + C_2 y_2$. There is nothing wrong with this, because this equation is not homogeneous. The general solution of a nonhomogeneous linear equation has a slightly different form. We will learn about the solutions of nonhomogeneous linear equations a bit later.

In general, given a second order linear equation with the y-term missing

$$y'' + p(t)y' = g(t),$$

we can solve it by the substitutions $u = y'$ and $u' = y''$ to change the equation to a first order linear equation. Use the integrating factor method to solve for u, and then integrate u to find y. That is:

1. Substitute:	$u' + p(t)u = g(t)$
2. Integrating factor:	$\mu(t) = e^{\int p(t)dt}$
3. Solve for u:	$u(t) = \dfrac{\int \mu(t)g(t)\,dt \ \left(+C\right)}{\mu(t)}$
4. Integrate:	$y(t) = \int u(t)\,dt$

This method works regardless whether the coefficients are constant or nonconstant, or if the equation is nonhomogeneous.

The Characteristic Polynomial

Back to the subject of the second order linear homogeneous equations with constant coefficients (note that it is not in the standard form below):

$$ay'' + by' + cy = 0, \qquad a \neq 0. \qquad (*)$$

We have seen a few examples of such an equation. In all cases the solutions consist of exponential functions, or terms that could be rewritten into exponential functions[†]. With this fact in mind, let us derive a (very simple, as it turns out) method to solve equations of this type. We will start with the assumption that there are indeed some exponential functions of unknown exponents that would satisfy any equation of the above form. We will then devise a way to find the specific exponents that would give us the solution.

Let $y = e^{rt}$ be a solution of (*), for some as-yet-unknown constant r. Substitute y, $y' = re^{rt}$, and $y'' = r^2 e^{rt}$ into (*), we get

$$ar^2e^{rt} + bre^{rt} + ce^{rt} = 0, \qquad \text{or}$$

$$e^{rt}(ar^2 + br + c) = 0.$$

Since e^{rt} is never zero, the above equation is satisfied (and therefore $y = e^{rt}$ is a solution of (*)) if and only if $ar^2 + br + c = 0$. Notice that the expression $ar^2 + br + c$ is a quadratic polynomial with r as the unknown. It is always solvable, with roots given by the quadratic formula. Hence, we can always solve a second order linear homogeneous equation with constant coefficients (*).

[†] Sine and cosine are related to exponential functions by the identities
$$\sin\theta = \frac{e^{i\theta} - e^{-i\theta}}{2i} \quad \text{and} \quad \cos\theta = \frac{e^{i\theta} + e^{-i\theta}}{2}.$$

This polynomial, $ar^2 + br + c$, is called the *characteristic polynomial* of the differential equation (*). The equation

$$\boxed{ar^2 + br + c = 0}$$

is called the *characteristic equation* of (*). Each and every root, sometimes called a *characteristic root*, r, of the characteristic polynomial gives rise to a solution $y = e^{rt}$ of (*).

We will take a more detailed look of the 3 possible cases of the solutions thusly found:

1. (When $b^2 - 4ac > 0$) There are two distinct real roots r_1, r_2.
2. (When $b^2 - 4ac < 0$) There are two complex conjugate roots $r = \lambda \pm \mu i$.
3. (When $b^2 - 4ac = 0$) There is one repeated real root r.

Note: There is no need to put the equation in its standard form when solving it using the characteristic equation method. The roots of the characteristic equation remain the same regardless whether the leading coefficient is 1 or not.

Case 1 Two distinct real roots

When $b^2 - 4ac > 0$, the characteristic polynomial have two distinct real roots r_1, r_2. They give two distinct[‡] solutions $y_1 = e^{r_1 t}$ and $y_2 = e^{r_2 t}$. Therefore, a general solution of (*) is

$$\boxed{y = C_1 y_1 + C_2 y_2 = C_1 e^{r_1 t} + C_2 e^{r_2 t}.}$$

It is **that** easy.

Example: $y'' + 5y' + 4y = 0$

The characteristic equation is $r^2 + 5r + 4 = (r + 1)(r + 4) = 0$, the roots of the polynomial are $r = -1$ and -4. The general solution is then

$$y = C_1 e^{-t} + C_2 e^{-4t}.$$

Suppose there are initial conditions $y(0) = 1$, $y'(0) = -7$. A unique particular solution can be found by solving for C_1 and C_2 using the initial conditions. First we need to calculate $y' = -C_1 e^{-t} - 4C_2 e^{-4t}$, then apply the initial values:

$$1 = y(0) = C_1 e^0 + C_2 e^0 = C_1 + C_2$$

$$-7 = y'(0) = -C_1 e^0 - 4C_2 e^0 = -C_1 - 4C_2$$

The solution is $C_1 = -1$, and $C_2 = 2$ $\quad\rightarrow\quad y = -e^{-t} + 2e^{-4t}.$

[‡] We shall see the precise meaning of distinctness in the next section. For now just think that the two solutions are not constant multiples of each other.

Question: Suppose the initial conditions are instead $y(10000) = 1$, $y'(10000) = -7$. How would the new t_0 change the particular solution?

Apply the initial conditions as before, and we see there is a little complication. Namely, the simultaneous system of 2 equations that we have to solve in order to find C_1 and C_2 now comes with rather inconvenient irrational coefficients:

$$1 = y(10000) = C_1 e^{-10000} + C_2 e^{-40000}$$

$$-7 = y'(10000) = -C_1 e^{-10000} - 4C_2 e^{-40000}$$

With some good bookkeeping, systems like this can be solved the usual way. However, there is an easier method to simplify the inconvenient coefficients. The idea is *translation* (or *time-shift*). What we will do is to first construct a new coordinate axis, say $\check{T}$-axis. The two coordinate-axes are related by the equation $\check{T} = t - t_0$. (Therefore, when $t = t_0$, $\check{T} = 0$; that is, the initial t-value t_0 becomes the new origin.) In other words, we translate (or time-shift) t-axis by t_0 units to make it $\check{T}$-axis. In this example, we will accordingly set $\check{T} = t - 10000$. The immediate effect is that it makes the initial conditions to be back at 0: $y(0) = 1$, $y'(0) = -7$, with respect to the new $\check{T}$-coordinate. We then solve the translated system of 2 equations to find C_1 and C_2. What we get is the (simpler) system

$$1 = y(0) = C_1 e^0 + C_2 e^0 = C_1 + C_2$$

$$-7 = y'(0) = -C_1 e^0 - 4C_2 e^0 = -C_1 - 4C_2$$

As we have seen on the previous page, the solution is $C_1 = -1$, and $C_2 = 2$. Hence, the solution, in the new $\check{T}$-coordinate system, is $y(\check{T}) = -e^{-\check{T}} + 2e^{-4\check{T}}$.

Lastly, since this solution is in terms of $\check{T}$, but the original problem was in terms of t, we should convert it back to the original context. This conversion is easily achieved using the translation formula used earlier, $\check{T} = t - t_0 = t - 10000$. By replacing every occurrence of $\check{T}$ by $t - 1000$ in the solution, we obtain the solution, in its proper independent variable t.

$$y(t) = -e^{-(t-10000)} + 2e^{-4(t-10000)}.$$

Example: Consider the solution $y(t)$ of the initial value problem

$$y'' - 2y' - 8y = 0, \qquad y(0) = \alpha, \quad y'(0) = 2\pi.$$

Depending on the value of α, as $t \to \infty$, there are 3 possible behaviors of $y(t)$. Explicitly determine the possible behaviors and the respective initial value α associated with each behavior.

The characteristic equation is $r^2 - 2r - 8 = (r + 2)(r - 4) = 0$. Its roots are $r = -2$ and 4. The general solution is then

$$y = C_1 e^{-2t} + C_2 e^{4t}.$$

Notice that the long-term behavior of the solution is dependent on the coefficient C_2 only, since the $C_1 e^{-2t}$ term tends to 0 as $t \to \infty$, regardless of the value of C_1.

Solving for C_2 in terms of α, we get

$$\begin{aligned} y(0) = \alpha &= C_1 + C_2 \\ y'(0) = 2\pi &= -2C_1 + 4C_2 \end{aligned}$$

$$2\alpha + 2\pi = 6C_2 \qquad \to \qquad C_2 = \frac{\alpha + \pi}{3}.$$

Now, if $C_2 > 0$ then y tends to ∞ as $t \to \infty$. This would happen when $\alpha > -\pi$. If $C_2 = 0$ then y tends to 0 as $t \to \infty$. This would happen when $\alpha = -\pi$. Lastly, if $C_2 < 0$ then y tends to $-\infty$ as $t \to \infty$. This would happen when $\alpha < -\pi$. In summary:

When $\alpha > -\pi$,	$C_2 > 0$,	$\lim_{t \to \infty} y(t) = \infty$.
When $\alpha = -\pi$,	$C_2 = 0$,	$\lim_{t \to \infty} y(t) = 0$.
When $\alpha < -\pi$,	$C_2 < 0$,	$\lim_{t \to \infty} y(t) = -\infty$.

The Existence and Uniqueness (of the solution of a second order linear equation initial value problem)

A sibling theorem of the first order linear equation Existence and Uniqueness Theorem…

Theorem: Consider the initial value problem

$$y'' + p(t)\, y' + q(t)\, y = g(t), \qquad y(t_0) = y_0, \quad y'(t_0) = y'_0.$$

If the functions p, q, and g are continuous on the interval $\boldsymbol{I}$: $\alpha < t < \beta$ containing the point $t = t_0$. Then there exists a unique solution $y = \varphi(t)$ of the problem, and that this solution exists throughout the interval $\boldsymbol{I}$.

That is, the theorem guarantees that the given initial value problem will always have (existence of) exactly one (uniqueness) twice-differentiable solution, on any interval containing t_0 as long as all three functions $p(t)$, $q(t)$, and $g(t)$ are continuous on the same interval. Conversely, neither existence nor uniqueness of a solution is guaranteed at a discontinuity of $p(t)$, $q(t)$, or $g(t)$.

Examples: For each IVP below, find the largest interval on which a unique solution is guaranteed to exist.

(a) $(t + 2)y'' + ty' + \cot(t)y = t^2 + 1, \qquad y(2) = 11, \; y'(2) = -2.$

The standard form is $y'' + \dfrac{t}{t+2}y' + \dfrac{\cos(t)}{(t+2)\sin(t)}y = \dfrac{t^2+1}{t+2}$, and $t_0 = 2$. The discontinuities of p, q, and g are $t = -2, 0, \pm\pi, \pm2\pi, \pm3\pi\ldots$ The largest interval that contains $t_0 = 2$ but none of the discontinuities is, therefore, $(0, \pi)$.

(b) $\sqrt{16-t^2}\, y'' + \ln(t+1)y' + \cos(t)y = 0$, $y(0) = 2$, $y'(0) = 0$.

The standard form is $y'' + \dfrac{\ln(t+1)}{\sqrt{16-t^2}}y' + \dfrac{\cos(t)}{\sqrt{16-t^2}}y = 0$, $p(t)$ is only defined (and is continuous) on the interval $(-1, 4)$, and similarly $q(t)$ is only continuously defined on the interval $(-4, 4)$; $g(t)$ is continuous everywhere. Combining them we see that p, q, and g have discontinuities at any t such that $t \le -1$ or $t \ge 4$. That is, they are all continuous only on the interval $(-1, 4)$. Since that interval contains $t_0 = 0$, it must be the largest interval on which the solution is guaranteed to exist uniquely. Therefore, the answer is $(-1, 4)$

Similar to the previous instance (first order linear equation version) of the Existence and Uniqueness Theorem, the only time that a unique solution is not guaranteed to exist anywhere is whenever the initial time t_0 occurs at a discontinuity of either $p(t)$, $q(t)$, or $g(t)$.

Initial Value Problem vs. Boundary Value Problem

It might seem that there are more than one ways to present the initial conditions of a second order equation. Instead of locating both initial conditions $y(t_0) = y_0$ and $y'(t_0) = y'_0$ at the same point t_0, couldn't we take them at different points, for examples $y(t_0) = y_0$ and $y(t_1) = y_1$; or $y'(t_0) = y'_0$ and $y'(t_1) = y'_1$? The answer is NO. All the initial conditions in an initial value problem <u>must</u> be taken at the same point t_0. The sets of conditions above where the values are taken at different points are known as *boundary conditions*. A boundary value problem where a differential equation is bundled with (two or more) boundary conditions does not have the existence and uniqueness guarantee.

Example: Every function of the form $y = C\sin(t)$, where C is a real number satisfies the boundary value problem $y'' + y = 0$, $y(0) = 0$ and $y(\pi) = 0$. Therefore, the problem has infinitely many solutions, even though $p(t) = 0$, $q(t) = 1$, and $g(t) = 0$ are all continuous everywhere.

Exercises B.1-1:

1 – 4 Find the general solution of each equation.

1. $y'' + 10y' = t^2$

2. $y'' - 9y = 0$

3. $y'' + 4y' - 5y = 0$

4. $6y'' + y' - y = 0$

5 – 9 Solve each initial value problem. For each problem, state the largest interval in which the solution is guaranteed to uniquely exist.

5. $y'' + y' = 3e^{t/2}$, $y(0) = 4$, $y'(0) = 3$

6. $y'' + 2y' = te^{-t}$, $y(0) = 6$, $y'(0) = -1$

7. $ty'' - y' = t^2 + t$, $y(1) = 1$, $y'(1) = 5$

8. $y'' - y' - 2y = 0$, $y(0) = 2$, $y'(0) = 7$

9. $(t^2 + 9)y'' + 2ty' = 0$, $y(3) = 2\pi$, $y'(3) = 2/3$

10 – 15 Solve each initial value problem.

10. $y'' + y' - 12y = 0$, $y(0) = -2$, $y'(0) = -20$

11. $y'' + y' - 12y = 0$, $y(\pi) = -2$, $y'(\pi) = -20$

12. $y'' + 2y' - 3y = 0$, $y(0) = 1$, $y'(0) = 13$

13. $y'' + 2y' - 3y = 0$, $y(2\pi) = 1$, $y'(2\pi) = 13$

14. $y'' + 2y' - 4y = 0$, $y(0) = 6$, $y'(0) = -6$

15. $y'' + 2y' - 4y = 0$, $y(18) = 6$, $y'(18) = -6$

16. Without solving the given initial value problem, what is the largest interval in which a unique solution is guaranteed to exist?

$$(t+10)y'' - (5-t)y' + \ln|t|\,y = e^{2t}\cos t,$$

(a) $y(1) = -1, \quad y'(1) = 0$

(b) $y(-9) = 3, \quad y'(-9) = -2$

(c) $y(-12.5) = 1, \quad y'(-12.5) = 4$

17. Prove the Principle of Superposition: If y_1 and y_2 are any two solutions of the homogeneous equation

$$y'' + p(t)y' + q(t)y = 0.$$

Then any function of the form $y = C_1 y_1 + C_2 y_2$ is also a solution of the equation, for any pair of constants C_1 and C_2.

Answers B-1.1:

1. $y = \dfrac{t^3}{30} - \dfrac{t^2}{100} + \dfrac{t}{500} + C_1 e^{-10t} + C_2$
2. $y = C_1 e^{3t} + C_2 e^{-3t}$
3. $y = C_1 e^{t} + C_2 e^{-5t}$
4. $y = C_1 e^{t/3} + C_2 e^{-t/2}$
5. $y = -e^{-t} + 1 + 4e^{t/2}, \quad (-\infty, \infty)$
6. $y = -t e^{-t} + 6, \quad (-\infty, \infty)$
7. $y = \dfrac{t^3}{3} + \dfrac{7t^2}{4} + \dfrac{t^2}{2}\ln t - \dfrac{13}{12}, \qquad (0, \infty)$
8. $y = 3e^{2t} - e^{-t}, \quad (-\infty, \infty)$
9. $y = 4\tan^{-1}\left(\dfrac{t}{3}\right) + \pi, \qquad (-\infty, \infty)$
10. $y = -4e^{3t} + 2e^{-4t}$
11. $y = -4e^{3(t-\pi)} + 2e^{-4(t-\pi)}$
12. $y = 4e^{t} - 3e^{-3t}$
13. $y = 4e^{t-2\pi} - 3e^{-3(t-2\pi)}$
14. $y = 3e^{(-1+\sqrt{5})t} + 3e^{(-1-\sqrt{5})t}$
15. $y = 3e^{(-1+\sqrt{5})(t-18)} + 3e^{(-1-\sqrt{5})(t-18)}$
16. (a) $(0, \infty)$, (b) $(-10, 0)$, (c) $(-\infty, -10)$

Fundamental Solutions

We have seen that the general solution of a second order homogeneous linear equation is in the form of $y = C_1 y_1 + C_2 y_2$[§], where y_1 and y_2 are two "distinct" functions both satisfying the given equation (as a result, y_1 and y_2 are themselves particular solutions of the equation). Now we will examine the circumstance under which two arbitrary solutions y_1 and y_2 could give us a general solution.

Suppose y_1 and y_2 are two solutions of some second order homogeneous linear equation such that their linear combinations $y = C_1 y_1 + C_2 y_2$ give a general solution of the equation. Then, according to the Existence and Uniqueness Theorem, for any pair of initial conditions $y(t_0) = y_0$ and $y'(t_0) = y'_0$ there must exist uniquely a corresponding pair of coefficients C_1 and C_2 that satisfies the system of (algebraic) equations

$$\begin{aligned} y_0 &= C_1\, y_1(t_0) + C_2\, y_2(t_0) \\ y'_0 &= C_1\, y'_1(t_0) + C_2\, y'_2(t_0) \end{aligned}$$

From linear algebra, we know that for the above system to always have a unique solution (C_1, C_2) for any initial values y_0 and y'_0, the coefficient matrix of the system must be invertible, or, equivalently, <u>the determinant of the coefficient matrix must be nonzero[**]</u>. That is

$$\det\begin{pmatrix} y_1(t_0) & y_2(t_0) \\ y'_1(t_0) & y'_2(t_0) \end{pmatrix} = y_1(t_0)y'_2(t_0) - y'_1(t_0)y_2(t_0) \neq 0$$

This determinant above is called the *Wronskian* or the *Wronskian determinant*. It is a function of t as well, denoted $W(y_1, y_2)(t)$, and is given by the expression

$$\boxed{W(y_1, y_2)(t) = y_1\, y'_2 - y'_1\, y_2.}$$

[§] The expression $y = C_1 y_1 + C_2 y_2$ is called a *linear combination* of the functions y_1 and y_2.

[**] By nonzero it means that the Wronskian is not the constant zero function.

On the other hand, at each point t_0 where $W(y_1, y_2)(t_0) = 0$, a unique pair of coefficients C_1 and C_2 that satisfies the previous system of equations cannot always be found (see any linear algebra textbook for a proof of this). This could be due to one of two reasons. The first reason is that $y = C_1 y_1 + C_2 y_2$ is really not a general solution of our equation. Or, the second possibility is that t_0 is a discontinuity of either $p(t)$, $q(t)$, or $g(t)$. This second reason is, of course, a consequence of the Existence and Uniqueness theorem.

Assuming that not every point is a discontinuity of either $p(t)$, $q(t)$, or $g(t)$, then the fact that $W(y_1, y_2)(t)$ is constant zero implies that $y = C_1 y_1 + C_2 y_2$ is not a general solution of the given equation. Otherwise, if $W(y_1, y_2)(t)$ is nonzero at some points t_0 on the real line, then $y = C_1 y_1 + C_2 y_2$ will, together with different combinations of initial condition $y(t_0) = y_0$ and $y'(t_0) = y'_0$, give uniquely all the possible particular solutions, on some open intervals containing t_0. That is, $y = C_1 y_1 + C_2 y_2$ is a general solution of the given equation. Hence, our interest in knowing whether or not $W(y_1, y_2)(t)$ is the constant zero function.

Formally, if $W(y_1, y_2)(t) \neq 0$, then the functions y_1, y_2 are said to be *linearly independent*. Else they are called *linearly dependent* if $W(y_1, y_2)(t) = 0$.[††]

Note: In the simple instance of two functions, as is the case presently, their linear independence could equivalently be determined by the fact that two functions are linearly independent if and only if they are not constant multiples of each other.

Suppose y_1 and y_2 are two linearly independent solutions of a second order homogeneous linear equation

$$y'' + p(t)y' + q(t)y = 0.$$

That is, y_1 and y_2 both satisfy the equation, and $W(y_1, y_2)(t) \neq 0$. Then (and only then) their linear combination $y = C_1 y_1 + C_2 y_2$ forms a general solution of the differential equation. Therefore, a pair of such linearly independent solutions y_1 and y_2 is called a set of *fundamental solutions*, because they are

[††] Since $W(y_1, y_2)(t) = -W(y_2, y_1)(t)$, they are either both zero or both nonzero. Therefore, the order of the 2 functions y_1 and y_2 does not matter in the Wronskian calculation.

essentially the basic building blocks of all particular solutions of the equation.

To summarize, suppose y_1 and y_2 are two solutions of a second order homogeneous linear equation, then:

$$\boxed{\begin{array}{c} W(y_1, y_2)(t) \text{ is not the constant zero function} \\ \updownarrow \\ y_1, y_2 \text{ are linearly independent} \\ \updownarrow \\ y_1, y_2 \text{ are fundamental solutions} \\ \updownarrow \\ y = C_1 y_1 + C_2 y_2 \text{ is a general solution of the equation} \end{array}}$$

Example: Let $y_1 = e^{r_1 t}$ and $y_2 = e^{r_2 t}$, $r_1 \neq r_2$, be any two different exponential function. Then

$$W(y_1, y_2) = \det\begin{pmatrix} e^{r_1 t} & e^{r_2 t} \\ r_1 e^{r_1 t} & r_2 e^{r_2 t} \end{pmatrix} = r_2 e^{r_1 t} e^{r_2 t} - r_1 e^{r_1 t} e^{r_2 t}$$

$$= (r_2 - r_1) e^{r_1 t} e^{r_2 t} \neq 0, \qquad \text{for all } t.$$

Therefore, any two different exponential-function solutions of a second order homogeneous linear equation (as those found using its characteristic equation) are always linearly independent, thus they will always give a general solution. Better yet, in this case since the Wronskian is never zero for all real numbers, a unique solution can always be found.

Lastly, here is an interesting (and, as we shall see shortly, useful) relationship between the Wronskian of any two solutions of a second order linear equation with its coefficient function $p(t)$.

Abel's Theorem: If y_1 and y_2 are any two solutions of the equation

$$y'' + p(t)y' + q(t)y = 0,$$

where p and q are continuous on an open interval $\boldsymbol{I}$. Then the Wronskian $W(y_1, y_2)(t)$ is given by

$$W(y_1, y_2)(t) = C\, e^{-\int p(t)\,dt},$$

where C is a constant that depends on y_1 and y_2, but not on t. Further, $W(y_1, y_2)(t)$ is either zero for all t in $\boldsymbol{I}$ (if $C = 0$) or else is never zero in $\boldsymbol{I}$ (if $C \neq 0$).

Exercises B-1.2:

1. Previously, we have found that the equation $y'' - y = 0$ has a general solution $y = C_1 e^{t} + C_2 e^{-t}$. (a) Construct another general solution by first verifying that $y_1 = \cosh t = \dfrac{e^{t} + e^{-t}}{2}$ and $y_2 = \sinh t = \dfrac{e^{t} - e^{-t}}{2}$ also form a pair of fundamental solutions. Conclude that a general solution is not unique for this equation. (b) For each of the two general solutions, find the solution corresponding to the initial conditions $y(0) = 1$ and $y'(0) = 2$. Show that the two particular solutions are identical.

2. Suppose y_1 and y_2 are two solutions of the equation $t^2 y'' + 2t^3 y' - t^{-2} y = 0$. Find $\boldsymbol{W}(y_1, y_2)(t)$.

3. Suppose y_1 and y_2 are two solutions of the equation $t y'' - (t + 4) y' + e^{-3t} y = 0$, such that $\boldsymbol{W}(y_1, y_2)(1) = 10$. Find $\boldsymbol{W}(y_1, y_2)(t)$.

4. Suppose $y_1 = t$ and $y_2 = t e^{4t}$ are both solutions of a certain equation $y'' + p(t) y' + q(t) y = 0$. (a) Compute $\boldsymbol{W}(y_1, y_2)(t)$. (b) What is a general solution of this equation? (c) Does there exist a unique solution satisfying the initial conditions $y(0) = 0, y'(0) = 0$? (Use part b in your computation, is there a unique pair of coefficients C_1 and C_2?) (d) Find the solution satisfying the initial conditions $y(1) = 1, y'(1) = 5$. (e) What is the largest interval on which the solution from part d is guaranteed to exist uniquely?

5. Suppose $y_1 = 2 + 3 e^{-t}$ and $y_2 = 3 - 2 e^{-t}$ are both solutions of a certain equation $y'' + p(t) y' + q(t) y = 0$. (a) Compute $\boldsymbol{W}(y_1, y_2)(t)$. (b) What is a general solution of this equation? (c) Find the solution satisfying the initial conditions $y(0) = 2, y'(0) = 3$. (d) What is the largest interval on which the solution from part d is guaranteed to exist uniquely?

Answers B-1.2:
1. (a) Another general solution is $y = C_1 \cosh t + C_2 \sinh t$.
2. $\boldsymbol{W}(y_1, y_2)(t) = = Ce^{-t^2}$
3. $\boldsymbol{W}(y_1, y_2)(t) = 10t^4 e^{t-1}$
4. (a) $\boldsymbol{W}(y_1, y_2)(t) = 4t^2 e^{4t}$, (b) $y = C_1 t + C_2 t e^{4t}$, (c) Since $\boldsymbol{W}(y_1, y_2)(0) = 0$, there is no existence or uniqueness guarantee for a particular solution. As it turns out, there are infinitely many solutions satisfying the given initial conditions: any function of the form $y = C_1 t + C_2 t e^{4t}$, where $C_1 = -C_2$. (d) $y = e^{-4} t e^{4t}$, (e) $(0, \infty)$.
5. (a) $\boldsymbol{W}(y_1, y_2)(t) = 13e^{-t}$, (b) $y = C_1 (2 + 3e^{-t}) + C_2 (3 - 2e^{-t})$, which can be simplified to $y = K_1 + K_2 e^{-t}$, (c) $y = 5 - 3e^{-t}$, (d) $(-\infty, \infty)$.

Case 2 Two complex conjugate roots

When $b^2 - 4ac < 0$, the characteristic polynomial has two complex roots, which are conjugates, $r_1 = \lambda + \mu i$ and $r_2 = \lambda - \mu i$ (λ, μ are real numbers, $\mu > 0$). As before they give two linearly independent solutions $y_1 = e^{r_1 t}$ and $y_2 = e^{r_2 t}$. Consequently the linear combination $y = C_1 e^{r_1 t} + C_2 e^{r_2 t}$ will be a general solution. At this juncture you might have this question: "but aren't r_1 and r_2 complex numbers; what would become of the exponential function with a complex number exponent?" The answer to that question is given by the *Euler's formula*.

Euler's formula For any real number θ,

$$e^{\theta i} = \cos\theta + i\sin\theta .$$

Hence, when r is a complex number $\lambda + \mu i$, the exponential function e^{rt} becomes

$$e^{rt} = e^{(\lambda + \mu i)t} = e^{\lambda t} e^{\mu i t} = e^{\lambda t}(\cos \mu t + i \sin \mu t)$$

Similarly, when $r = \lambda - \mu i$, e^{rt} becomes

$$\begin{aligned} e^{(\lambda - \mu i)t} &= e^{\lambda t} e^{-\mu i t} = e^{\lambda t}(\cos(-\mu t) + i \sin(-\mu t)) \\ &= e^{\lambda t}(\cos \mu t - i \sin \mu t) \end{aligned}$$

Hence, the general solution found above is then

$$y = C_1\, e^{\lambda t}(\cos \mu t + i \sin \mu t) + C_2\, e^{\lambda t}(\cos \mu t - i \sin \mu t)$$

However, this general solution is a complex-valued function (meaning that, given a real number t, the value of the function $y(t)$ could be complex). It represents the general form of all particular solutions with either real or complex number coefficients. What we seek here, instead, is a real-valued expression that gives only the set of all particular solutions with real number coefficients only. In other words, we would like to "filter out" all functions containing coefficients with an imaginary part, that satisfy the given differential equation, keeping only those whose coefficients are real numbers.

Define
$$u(t) = e^{\lambda t}\cos \mu t$$
$$v(t) = e^{\lambda t}\sin \mu t$$

It is easy to verify that both u and v satisfy the differential equation (one way to see this is to observe that u can be obtain from the complex-valued general solution by setting $C_1 = C_2 = 1/2$; and v can be obtained similarly by setting $C_1 = 1/2i$ and $C_2 = -1/2i$). Their Wronskian is $W(u, v) = \mu e^{2\lambda t}$ is never zero. Therefore, the functions u and v are linearly independent solutions of the equation. They form a pair of real-valued fundamental solutions and the linear combination is a desired real-valued general solution:

$$\boxed{y = C_1 e^{\lambda t}\cos \mu t + C_2 e^{\lambda t}\sin \mu t.}$$

When $r = \lambda \pm \mu i$, $\mu > 0$, are two complex roots of the characteristic polynomial.

Example: $y'' + 4y = 0$

Answer: $y = C_1 \cos 2t + C_2 \sin 2t$

Example: $y'' + 2y' + 5y = 0, \qquad y(0) = 4, \quad y'(0) = 6$

The characteristic equation is $r^2 + 2r + 5 = 0$, which has solutions $r = -1 \pm 2i$. So $\lambda = -1$ and $\mu = 2$. Therefore, the general solution is

$$y = C_1 e^{-t} \cos 2t + C_2 e^{-t} \sin 2t$$

Apply the initial conditions to find that $C_1 = 4$ and $C_2 = 5$. Hence,

$$y = 4e^{-t} \cos 2t + 5e^{-t} \sin 2t.$$

Question: What would the solution be if the initial conditions are $y(25000) = 4$, and $y'(25000) = 6$ instead?

Answer: $y = 4e^{-(t-25000)} \cos 2(t - 25000) + 5e^{-(t-25000)} \sin 2(t - 25000)$

<u>Case 3</u> One repeated real root

When $b^2 - 4ac = 0$, the characteristic polynomial has a single repeated real root, $r = \dfrac{-b}{2a}$. This causes a problem, because unlike the previous two cases the roots of characteristic polynomial presently only give us one distinct solution $y_1 = e^{rt}$. It is not enough to give us a general solution. We would need to come up with a second solution, linearly independent with y_1, on our own. How do we find a second solution?

Take what we have: a solution $y_1 = e^{rt}$, where $r = \dfrac{-b}{2a}$. Let y_2 be another solution of the same equation $ay'' + by' + cy = 0$. The standard form of this equation is $y'' + \dfrac{b}{a}y' + \dfrac{c}{a}y = 0$, where $p(t) = \dfrac{b}{a}$. Compute the Wronskian two different ways:

$$W(y_1, y_2) = \det\begin{pmatrix} e^{rt} & y_2 \\ re^{rt} & y_2' \end{pmatrix} = e^{rt}y_2' - re^{rt}y_2, \quad r = \frac{-b}{2a}$$

and

$$W(y_1, y_2) = Ce^{-\int p(t)\,dt} = Ce^{-\int \frac{b}{a}dt} = Ce^{\frac{-b}{a}t}, \qquad C \neq 0.$$

By the Abel's Theorem, the fact $C \neq 0$ guarantees that y_1 and y_2 are going to be linearly independent. Now, we have two expressions for the Wronskian of the same pair of solutions. The two expressions must be equal:

$$e^{\frac{-b}{2a}t}\,y_2' + \frac{b}{2a}e^{\frac{-b}{2a}t}\,y_2 = Ce^{\frac{-b}{a}t}, \qquad C \neq 0.$$

This is a first order linear differential equation with y_2 as the unknown!

Put it into its standard form and solve by the integrating factor method.

$$y_2' + \frac{b}{2a} y_2 = C e^{\frac{-b}{2a}t}$$

The integrating factor is $\mu = e^{\int \frac{b}{2a} dt} = e^{\frac{b}{2a}t}$.

Hence,

$$y_2 = \frac{1}{e^{\frac{b}{2a}t}} \int e^{\frac{b}{2a}t} C e^{\frac{-b}{2a}t} dt = e^{\frac{-b}{2a}t} \int C \, dt = e^{\frac{-b}{2a}t} (Ct + C_1)$$

$$= Cte^{\frac{-b}{2a}t} + C_1 e^{\frac{-b}{2a}t} = Cte^{rt} + C_1 e^{rt}$$

Any such a function would be a second, linearly independent solution of the differential equation. We just need one instance of such a function. The only condition for the coefficients in the above expression is $C \neq 0$. Pick, say, $C = 1$, and $C_1 = 0$ would work nicely. Thus $y_2 = t e^{rt}$.

Therefore, the general solution in the case of a repeated real root r is

$$\boxed{y = C_1 e^{rt} + C_2 t e^{rt}.}$$

Example: $y'' - 4y' + 4y = 0, \qquad y(0) = 4, \quad y'(0) = 5$

The characteristic equation is $r^2 - 4r + 4 = (r - 2)^2 = 0$, which has solution $r = 2$ (repeated). Thus, the general solution is

$$y = C_1 e^{2t} + C_2 t e^{2t}.$$

Differentiate,

$$y' = 2C_1 e^{2t} + C_2 (2t e^{2t} + e^{2t}).$$

Apply the initial conditions to find that $C_1 = 4$ and $C_2 = -3$:

$$y = 4 e^{2t} - 3t e^{2t}.$$

Summary

Given a second order linear equation with constant coefficients

$$ay'' + by' + cy = 0, \qquad a \neq 0.$$

Solve its characteristic equation $ar^2 + br + c = 0$. The general solution depends on the type of roots obtained (use the quadratic formula to find the roots if you are unable to factor the polynomial!):

1. When $b^2 - 4ac > 0$, there are two distinct real roots r_1, r_2.

$$y = C_1 e^{r_1 t} + C_2 e^{r_2 t}.$$

2. When $b^2 - 4ac < 0$, there are two complex conjugate roots $r = \lambda \pm \mu i$. Then

$$y = C_1 e^{\lambda t} \cos \mu t + C_2 e^{\lambda t} \sin \mu t.$$

3. When $b^2 - 4ac = 0$, there is one repeated real root r. Then

$$y = C_1 e^{rt} + C_2 t e^{rt}.$$

Since $p(t) = b/a$ and $q(t) = c/a$, being constants, are continuous for every real number, therefore, according to the Existence and Uniqueness Theorem, in each case above there is always a unique solution valid on $(-\infty, \infty)$ for any pair of initial conditions $y(t_0) = y_0$ and $y'(t_0) = y'_0$.

Exercises B-1.3:

1. Verify that $y = t\,e^{rt}$, $r = \dfrac{-b}{2a}$, is a solution of the equation $a\,y'' + b\,y' + c\,y = 0$ if $b^2 - 4\,ac = 0$; and it is not a solution if $b^2 - 4\,ac \neq 0$.

2 − 10 For each of the following equations (a) find its general solution, (b) find the particular solution satisfying the initial conditions $y(0) = 2$, $y'(0) = -1$, and (c) find the limit, as $t \to \infty$, of the solution found in (b).

2. $y'' + 9y' + 8y = 0$

3. $y'' - 6y' + 25y = 0$

4. $y'' - 6y' + 8y = 0$

5. $2y'' + 5y' - 3y = 0$

6. $2y'' - 16y' + 32y = 0$

7. $y'' + 4y' + 13y = 0$

8. $2y'' - y' = 0$

9. $2y'' + 5y' + 2y = 0$

10. $16y'' - 8y' + y = 0$

11 – 15 Solve each initial value problem.

11. $y'' + 9y' + 14y = 0,\quad y(5\pi) = 4,\quad y'(5\pi) = 2$

12. $2y'' - 16y' + 32y = 0,\quad y(-2) = 2,\quad y'(-2) = -1$

13. $9y'' + y = 0,\quad y(0) = -2,\quad y'(0) = 2$

14. $y'' + 6y' + 34y = 0,\quad y(10) = 5,\quad y'(10) = -5$

15. $10y'' - 7y' + y = 0,\quad y(0) = -8,\quad y'(0) = -1$

16 – 21 Find a second order linear equation with constant coefficients that has the indicated solution. (The answer is not unique.)

16. The general solution is $y = C_1 e^{t} + C_2 t e^{t}$.

17. The general solution is $y = C_1 e^{5t} + C_2 e^{-2t}$.

18. The general solution is $y = C_1 \cos 10t + C_2 \sin 10t$.

19. A particular solution is $y = 7e^{3t} - e^{-2t}$.

20. A particular solution is $y = 12 e^{\pi - t} \sin 2t$.

21. A particular solution is $y = -2\pi t e^{-5t}$.

22. Consider all the nonzero solutions of the equation $y'' + 12y' + 36y = 0$, determine their behavior as $t \to \infty$.

23. Consider all the nonzero solutions of the equation $y'' - 2y' + 10y = 0$, determine their behavior as $t \to \infty$.

Answers B-1.3:

2. $y = C_1 e^{-t} + C_2 e^{-8t}$, $\quad y = \frac{15}{7}e^{-t} - \frac{1}{7}e^{-8t}$, 0

3. $y = C_1 e^{3t}\cos 4t + C_2 e^{3t}\sin 4t$, $\quad y = 2e^{3t}\cos 4t - \frac{7}{4}e^{3t}\sin 4t$, none

4. $y = C_1 e^{2t} + C_2 e^{4t}$, $\quad y = \frac{9}{2}e^{2t} - \frac{5}{2}e^{4t}$, $-\infty$

5. $y = C_1 e^{t/2} + C_2 e^{-3t}$, $\quad y = \frac{10}{7}e^{t/2} + \frac{4}{7}e^{-3t}$, ∞

6. $y = C_1 e^{4t} + C_2 te^{4t}$, $\quad y = 2e^{4t} - 9te^{4t}$, $-\infty$

7. $y = C_1 e^{-2t}\cos 3t + C_2 e^{-2t}\sin 3t$, $\quad y = 2e^{-2t}\cos 3t + e^{-2t}\sin 3t$, 0

8. $y = C_1 e^{t/2} + C_2$, $\quad y = -2e^{t/2} + 4$, $-\infty$

9. $y = C_1 e^{-t/2} + C_2 e^{-2t}$, $\quad y = 2e^{-t/2}$, 0

10. $y = C_1 e^{t/4} + C_2 te^{t/4}$, $\quad y = 2e^{t/4} - \frac{3}{2}te^{t/4}$, $-\infty$

11. $y = 6e^{-2(t-5\pi)} - 2e^{-7(t-5\pi)}$

12. $y = 2e^{4(t+2)} - 9(t+2)e^{4(t+2)} = -16e^{4t+8} - 9te^{4t+8}$

13. $y = -2\cos\frac{t}{3} + 6\sin\frac{t}{3}$

14. $y = 5e^{-3(t-10)}\cos 5(t-10) + 2e^{-3(t-10)}\sin 5(t-10)$

15. $y = 2e^{t/2} - 10e^{t/5}$

16. $y'' - 2y' + y = 0$

17. $y'' - 3y' - 10y = 0$

18. $y'' + 100y = 0$

19. $y'' - y' - 6y = 0$

20. $y'' + 2y' + 5y = 0$

21. $y'' + 10y' + 25y = 0$

22. The solutions are of the form $y = C_1 e^{-6t} + C_2 t e^{-6t}$, they all approach 0 as $t \to \infty$.

23. The solutions are of the form $y = C_1 e^{t}\cos 3t + C_2 e^{t}\sin 3t$. The zero solution (i.e. when $C_1 = C_2 = 0$) approaches 0, all the nonzero solutions oscillate with an increasing amplitude and do not reach a limit.

Reduction of Order

Problem: Given a second order, homogeneous, linear differential equation (with non-constant coefficients) and a known nonzero solution y_1, find the general solution of the given equation.

To start, assume that there exists a second solution in the form of $y_2 = y_1\, v(t)$, for some differentiable function $v(t)$.

First we want to make sure the equation is written in the standard form with leading coefficient 1:

$$y'' + p(t)y' + q(t)y = 0.$$

Next, we will compute the Wronskian $W(y_1, y_2)(t)$ two different ways, using the two methods that we know. By the definition of Wronskian:

$$W(y_1, y_2) = \det\begin{pmatrix} y_1 & y_1 v(t) \\ y_1' & y_1' v(t) + y_1 v'(t) \end{pmatrix} = y_1 y_1' v(t) + y_1^{\,2} v'(t) - y_1 y_1' v(t) = y_1^{\,2} v'(t)$$

By the Abel's Theorem:

$$W(y_1, y_2) = C\, e^{-\int p(t)\,dt}, \qquad \text{where } C \neq 0.$$

The fact that $C \neq 0$ is important, because it guarantees the linear independence of y_1 and y_2.

The two expressions computed above are the Wronskian of the same two functions, therefore, the two expressions must be the same. Equate them:

$$y_1^{\,2} v'(t) = C\, e^{-\int p(t)\,dt}.$$

Therefore,
$$v'(t) = C\,\frac{e^{-\int p(t)\,dt}}{y_1^{\,2}}, \qquad C \neq 0.$$

Integrate the right-hand side to find $v(t)$. Choose any convenient nonzero value for C. Letting $C = 1$ would work nicely, although it may not be the most convenient choice. Then find $y_2 = y_1\, v(t)$.

The general solution is still, of course, in the form $y = C_1 y_1 + C_2 y_2$. Therefore,

$$y = C_1 y_1 + C_2 y_2 = C_1 y_1 + C_2\, y_1\, v(t).$$

Note: It is actually not necessary to assume that $y_2 = y_1\, v(t)$. Although doing so makes the resulting first order differential equation easier to solve.

Example: If it is known that $y_1 = t$ is a solution of

$$t^2 y'' - t(t+2)y' + (t+2)y = 0, \qquad t > 0.$$

Find its general solution.

Rewrite the equation into the standard form

$$y'' - \frac{t+2}{t} y' + \frac{t+2}{t^2} y = 0$$

Identify $p(t) = -\dfrac{t+2}{t}$. Let $y_2 = y_1 v(t) = t\, v(t)$.

$$W(y_1, y_2) = \det\begin{pmatrix} t & tv(t) \\ 1 & v(t) + tv'(t) \end{pmatrix} = tv(t) + t^2 v'(t) - tv(t) = t^2 v'(t),$$

and,

$$W(y_1, y_2) = C\, e^{\int \frac{t+2}{t} dt} = C\, e^{\int \left(1+\frac{2}{t}\right) dt} = C\, e^{t + \ln(t^2)} = C\, t^2 e^t,$$

where $C \neq 0$.

Equating both parts: $\qquad t^2 v' = C t^2 e^t$

$$v' = C e^t \qquad \rightarrow \qquad v = C e^t + C_1$$

Choose $C = 1$ and $C_1 = 0 \;\rightarrow\; v = e^t$. Therefore, $y_2 = y_1 v(t) = t e^t$.

The general solution is

$$y = C_1 y_1 + C_2 y_2 = C_1 t + C_2 t e^t.$$

Example: Find the general solution of the equation below, given that $y_1 = t^2\cos(\ln t)$ is a known solution.

$$t^2 y'' - 3ty' + 5y = 0, \qquad t > 0.$$

Rewrite the equation into the standard form

$$y'' - \frac{3}{t}y' + \frac{5}{t^2}y = 0$$

Identify $p(t) = -\frac{3}{t}$. Let $y_2 = y_1\, v(t) = t^2\cos(\ln t)\, v(t)$.

$$W(y_1, y_2) = t^4\cos^2(\ln t)v'(t),$$

and,

$$W(y_1, y_2) = C\, e^{\int \frac{3}{t}dt} = C\, e^{3\ln(t)} = Ce^{\ln(t^3)} = C\,t^3, \quad C \neq 0.$$

Equating both parts: $\quad t^4\cos^2(\ln t)\, v' = C\,t^3$

$$v' = \frac{C}{t\cos^2(\ln t)} = \frac{C\sec^2(\ln t)}{t}$$

$$v = C\int \frac{\sec^2(\ln t)}{t}\,dt = C\tan(\ln t) + C_1 = C\frac{\sin(\ln t)}{\cos(\ln t)} + C_1$$

Choose $C = 1$ and $C_1 = 0 \rightarrow v = \tan(\ln t)$.

$$y_2 = y_1\, v(t) = t^2\cos(\ln t)\tan(\ln t) = t^2\sin(\ln t).$$

The general solution is

$$y = C_1\, y_1 + C_2\, y_2 = C_1\, t^2\cos(\ln t) + C_2\, t^2\sin(\ln t).$$

Exercises B-1.4:

1 – 7 For each equation below, a known solution is given. Find a second, linearly independent solution of the equation, and find the general solution.

1. $t^2y'' - 2ty' + 2y = 0, \qquad t > 0, \qquad y_1 = t^2.$

2. $t^2y'' - ty' - 3y = 0, \qquad t > 0, \qquad y_1 = t^{-1}.$

3. $ty'' + y' = 0, \qquad t > 0, \qquad y_1 = 1.$

4. $t^2y'' - 5ty' + 8y = 0, \qquad t > 0, \qquad y_1 = t^4.$

5. $t^2y'' - ty' + 10y = 0, \qquad t > 0, \qquad y_1 = t\sin(3\ln t).$

6. $(t-5)^2y'' - 2(t-5)y' + 2y = 0, \qquad t > 5, \qquad y_1 = (t-5)^2.$

7. $(t+2)^2y'' + 3(t+2)y' + y = 0, \qquad t > -2, \qquad y_1 = (t+2)^{-1}.$

8. Solve the initial value problem

$$t^2y'' - 3ty' + 4y = 0, \qquad t > 0, \qquad y(1) = -2, \quad y'(1) = 1.$$

Given that $y_1 = t^2 \ln t$ is a known solution.

9. (a) Find the general solution of $t^2y'' - 2y = 0$, $t > 0$, given $y_1 = t^2$.
(b) Find the particular solution satisfying $y(1) = 6$ and $y'(1) = 9$.
(c) Show that the initial value problem $t^2y'' - 2y = 0$, $y(0) = 0$ and $y'(0) = 0$, do not have a unique solution by verifying that any of the infinitely many functions of the form $y = Ct^2$ is a solution, regardless of the value of C. Does this fact violate the Existence and Uniqueness Theorem?

Answers B-1.4:

1. $y = C_1 t^2 + C_2 t$
2. $y = C_1 t^{-1} + C_2 t^3$
3. $y = C_1 + C_2 \ln t$
4. $y = C_1 t^4 + C_2 t^2$
5. $y = C_1 t \sin(3 \ln t) + C_2 t \cos(3 \ln t)$
6. $y = C_1(t-5)^2 + C_2 (t-5)$
7. $y = C_1 \dfrac{1}{t+2} + C_2 \dfrac{\ln(t+2)}{t+2}$
8. $y = 5t^2 \ln t - 2t^2$
9. (a) $y = C_1 t^2 + C_2 t^{-1}$, (b) $y = 5t^2 + t^{-1}$

(Optional topic) Euler Equations

A second order *Euler equation* (also known as an *Euler-Cauchy equation*) is a second order homogeneous linear equation of the form

$$t^2 y'' + \alpha t y' + \beta y = 0, \qquad (**)$$

or, in standard form

$$y'' + \frac{\alpha}{t} y' + \frac{\beta}{t^2} y = 0 .$$

In a course at this level, variations of the Euler equation most frequently appear as examples and exercises in lectures about reduction of order. In this context we have seen a few of them in the previous section. This type of equations, however, is very interesting in its own right. Despite the non-constant nature of their coefficients, Euler equations can be easily solved in a way that is analogous to the characteristic equation method of solving constant coefficient homogeneous linear equations. We shall develop this solution technique for Euler equations in this section.

By visual inspection (or by peeking back at the exercises previously encountered in the section about reduction of order technique) we might deduce that a function of the form $y = t^r$ could be a solution of (**), $t \neq 0$. Therefore, similar to how we have previously derived the characteristic equation method, we will assume that, for some power, r, yet to be determined, there exists a solution $y = t^r$. We then substitute it into (**) to get a better idea about what r should be.

For the time being, let us consider only the case of $t > 0$. Start with the trial solution $y = t^r$, then $y' = rt^{r-1}$ and $y'' = r(r-1)t^{r-2}$. Plug them into (**):

$$r(r-1)t^{r-2+2} + \alpha r t^{r-1+1} + \beta t^r = 0,$$

$$(r^2 - r + \alpha r + \beta)\, t^r = 0,$$

$$(r^2 + (\alpha - 1)r + \beta)\, t^r = 0.$$

Since $t^r \neq 0$, it follows that $r^2 + (\alpha - 1)r + \beta = 0$. This quadratic equation is the "characteristic equation" of (**). <u>Whatever value of r, real or complex, that satisfies the characteristic equation will yield a nontrivial solution of (**) in the form $y = t^r$.</u>

As you might have suspected, depending on the number and type of roots r of the characteristic equation, the equation (**) will have different forms of (real-valued) general solution. We will look at each case in turn.

Case I: There are two distinct real roots r_1 and r_2.

In this case $y_1 = t^{r_1}$ and $y_2 = t^{r_2}$ are two solutions linearly independent everywhere on the interval $(0, \infty)$. (Exercise: check that their Wronskian is nonzero for $t \neq 0$.) Therefore, a general solution of (**) is

$$\boxed{y = C_1 y_1 + C_2 y_2 = C_1 t^{r_1} + C_2 t^{r_2}.}$$

Case II: There are two complex conjugate roots $r = \lambda \pm \mu i, \quad \mu > 0$.

In this case $y_1 = t^{r_1}$ and $y_2 = t^{r_2}$ remain two solutions linearly independent everywhere on the interval $(0, \infty)$. They are complex-valued functions, however:

$$y_1 = t^{\lambda+\mu i} = t^{\lambda}\, t^{\mu i} = t^{\lambda}\, e^{\mu(\ln t)i} = t^{\lambda}\,(\cos(\mu \ln t) + i\sin(\mu \ln t))$$
$$y_2 = t^{\lambda-\mu i} = t^{\lambda}\, t^{-\mu i} = t^{\lambda}\, e^{-\mu(\ln t)i} = t^{\lambda}\,(\cos(\mu \ln t) - i\sin(\mu \ln t))$$

In the same fashion as we have done for the constant coefficients second order linear equation earlier, we can produce the following pair of real-valued, linearly independent solutions using linear combinations.

$$u = (y_1 + y_2)/2 = t^{\lambda}\cos(\mu \ln t)$$

$$v = (y_1 - y_2)/2i = t^{\lambda}\sin(\mu \ln t)$$

Therefore, a real-valued general solution is

$$\boxed{y = C_1 t^{\lambda}\cos(\mu \ln t) + C_2 t^{\lambda}\sin(\mu \ln t).}$$

Case III: There is a repeated real root *r*.

Initially, we have only $y_1 = t^r$ as a solution. The second solution can be readily found by the method of reduction of order to be $y_2 = t^r \ln t$.

To wit: If $r = k$ is a repeated root, then the characteristic equation have coefficients $\alpha - 1 = -2k$, i.e., $p(t) = \dfrac{\alpha}{t} = \dfrac{-2k+1}{t}$; and $\beta = k^2$. Now, let $y_1 = t^k$ and $y_2 = t^k v$.

It follows that $W(y_1, y_2) = y_1^{\,2} v'(t) = t^{2k} v'(t)$, and by Abel's theorem, it is also $W(y_1, y_2) = C\, e^{\int \frac{2k-1}{t} dt} = C\, e^{(2k-1)\ln(t)} = C\, t^{2k-1}$, $\qquad C \neq 0$.

Hence, $t^{2k} v'(t) = C\, t^{2k-1} \qquad \rightarrow \qquad v'(t) = \dfrac{C}{t}$, $\qquad C \neq 0$.

Integrate to obtain $v(t) = C \ln t + C_1$, then set $C = 1$ and $C_1 = 0$.

We have $v(t) = \ln t$.

Consequently, $y_1 = t^k = t^r$, and $y_2 = y_1 \ln t = t^r \ln t$, are the two required fundamental solutions.

Therefore, a general solution is

$$\boxed{y = C_1 t^r + C_2 t^r \ln t\,.}$$

For $t < 0$, the general solution will take the same forms described above, except each formula will be in terms of $|\, t\,|$.

Example: Find the general solution of

$$t^2 y'' - 3ty' + 20y = 0, \qquad t > 0.$$

The characteristic equation is $r^2 - 4r + 20 = 0$, which has roots $r = 2 \pm 4i$. Therefore, the general solution is

$$y = C_1\, t^2 \cos(4 \ln t\,) + C_2\, t^2 \sin(4 \ln t).$$

Example: Find the general solution of

$$t^2 y'' + 7ty' + 9y = 0, \qquad t < 0.$$

The characteristic equation is $r^2 + 6r + 9 = (r + 3)^2 = 0$, which has a repeated root $r = -3$. Therefore, with $t < 0$, the general solution is

$$y = C_1 |\, t\,|^{-3} + C_2 |\, t\,|^{-3} \ln |\, t\,|.$$

Solution by substitution

Alternatively, the Euler equation can also be solved by a simple substitution. This approach seeks to convert an Euler equation into one with constant coefficients, thus establish a direct relation between their characteristic equations discussed previously.

Define $t = e^x$, thus $x = \ln t$, for $t > 0$. (Similarly, let $|t| = e^x$, thus $x = \ln |t|$, for $t < 0$.)

It follows that $\dfrac{dy}{dx} = \dfrac{dy}{dt}\dfrac{dt}{dx} = \dfrac{dy}{dt}e^x = \dfrac{dy}{dt}t$, and

$$\frac{d^2y}{dx^2} = \frac{d}{dx}\left(\frac{dy}{dt}t\right) = \frac{d^2y}{dt^2}\frac{dt}{dx}t + \frac{dy}{dt}\frac{dt}{dx} = \frac{d^2y}{dt^2}t^2 + \frac{dy}{dt}t.$$

That is,

$$t\frac{dy}{dt} = \frac{dy}{dx}, \text{ and}$$

$$t^2\frac{d^2y}{dt^2} = \frac{d^2y}{dx^2} - \frac{dy}{dt}t = \frac{d^2y}{dx^2} - \frac{dy}{dx}.$$

Therefore, in terms of x, equation (**) becomes

$$\left(\frac{d^2y}{dx^2} - \frac{dy}{dx}\right) + \alpha\frac{dy}{dx} + \beta\, y = 0.$$

Or,

$$\frac{d^2y}{dx^2} + (\alpha - 1)\frac{dy}{dx} + \beta\, y = 0.$$

The equation now has constant coefficients, which can be solved using its characteristic equation $r^2 + (\alpha - 1)r + \beta = 0$.

Depending on the number and type of the roots of the characteristic equation, we have:

Case I: There are two distinct real roots r_1 and r_2.

$$y = C_1 e^{r_1 x} + C_2 e^{r_2 x} = C_1 e^{r_1 \ln t} + C_2 e^{r_2 \ln t} = C_1 t^{r_1} + C_2 t^{r_2} .$$

Case II: There are two complex conjugate roots $r = \lambda \pm \mu i, \qquad \mu > 0.$

$$\begin{aligned} y &= C_1 e^{\lambda x} \cos \mu x + C_2 e^{\lambda x} \sin \mu x \\ &= C_1 e^{\lambda \ln t} \cos(\mu \ln t) + C_2 e^{\lambda \ln t} \sin(\mu \ln t) \\ &= C_1 t^{\lambda} \cos(\mu \ln t\ ) + C_2 t^{\lambda} \sin(\mu \ln t). \end{aligned}$$

Case III: There is a repeated real root r.

$$y = C_1 e^{rx} + C_2 x e^{rx} = C_1 t^{r} + C_2 t^{r} \ln t .$$

As can be seen, the two methods arrive at the identical results.

Summary

Given a second order Euler equation

$$t^2 y'' + \alpha t y' + \beta y = 0, \qquad t > 0.$$

Solve its characteristic equation $r^2 + (\alpha - 1)r + \beta = 0$. The general solution depends on the type of roots obtained:

1. When there are two distinct real roots r_1, r_2.

$$y = C_1 t^{r_1} + C_2 t^{r_2}.$$

2. When there are two complex conjugate roots $r = \lambda \pm \mu i$.

$$y = C_1 t^{\lambda} \cos(\mu \ln t) + C_2 t^{\lambda} \sin(\mu \ln t).$$

3. When there is one repeated real root r.

$$y = C_1 t^r + C_2 t^r \ln t.$$

With $t = 0$ being the only discontinuity of $p(t)$ and $q(t)$, when $t_0 > 0$, in each case above there is always a unique solution valid everywhere on $(0, \infty)$ for any pair of initial conditions $y(t_0) = y_0$ and $y'(t_0) = y'_0$. When $t_0 < 0$, replace every t in each formula above by $|t|$, and a unique solution valid everywhere on $(-\infty, 0)$ can always be found.

Higher Order Linear Equations with Constant Coefficients

The solutions of linear differential equations with constant coefficients of the third order or higher can be found in similar ways as the solutions of second order linear equations. For an n-th order homogeneous linear equation with constant coefficients:

$$a_n y^{(n)} + a_{n-1} y^{(n-1)} + \ldots + a_2 y'' + a_1 y' + a_0 y = 0, \qquad a_n \neq 0.$$

It has a general solution of the form

$$\boxed{y = C_1 y_1 + C_2 y_2 + \ldots + C_{n-1} y_{n-1} + C_n y_n}$$

where $y_1, y_2, \ldots, y_{n-1}, y_n$ are any n linearly independent solutions of the equation. (Thus, they form a set of fundamental solutions of the differential equation.) The linear independence of those solutions can be determined by their Wronskian, i.e., $\boldsymbol{W}(y_1, y_2, \ldots, y_{n-1}, y_n)(t) \neq 0$.

Note 1: In order to determine the n unknown coefficients C_i, each n-th order equation requires a set of n initial conditions in an initial value problem: $y(t_0) = y_0$, $y'(t_0) = y'_0$, $y''(t_0) = y''_0$, and $y^{(n-1)}(t_0) = y^{(n-1)}{}_0$.

Note 2: The Wronskian $\boldsymbol{W}(y_1, y_2, \ldots, y_{n-1}, y_n)(t)$ is defined to be the determinant of the following $n \times n$ matrix

$$\begin{bmatrix} y_1 & y_2 & \cdots & \cdots & y_n \\ y'_1 & y'_2 & \cdots & \cdots & y'_n \\ y''_1 & y''_2 & \cdots & \cdots & y''_n \\ \vdots & \vdots & & & \vdots \\ y_1^{(n-1)} & y_2^{(n-1)} & \cdots & \cdots & y_n^{(n-1)} \end{bmatrix}.$$

Such a set of linearly independent solutions, and therefore, a general solution of the equation, can be found by first solving the differential equation's characteristic equation:

$$\boxed{a_n r^n + a_{n-1} r^{n-1} + \ldots + a_2 r^2 + a_1 r + a_0 = 0.}$$

This is a polynomial equation of degree n, therefore, it has n real and/or complex roots (not necessarily distinct). Those necessary n linearly independent solutions can then be found using the four rules below.

(i). If r is a distinct real root, then $y = e^{rt}$ is a solution.

(ii). If $r = \lambda \pm \mu i$ are distinct complex conjugate roots, then $y = e^{\lambda t}\cos \mu t$ and $y = e^{\lambda t}\sin \mu t$ are solutions.

(iii). If r is a real root appearing k times, then $y = e^{rt}$, $y = te^{rt}$, $y = t^2 e^{rt}$, … , and $y = t^{k-1} e^{rt}$ are all solutions.

(iv). If $r = \lambda \pm \mu i$ are complex conjugate roots each appears k times, then

$$\begin{array}{ll} y = e^{\lambda t}\cos \mu t, & y = e^{\lambda t}\sin \mu t, \\ y = te^{\lambda t}\cos \mu t, & y = te^{\lambda t}\sin \mu t, \\ y = t^2 e^{\lambda t}\cos \mu t, & y = t^2 e^{\lambda t}\sin \mu t, \\ \vdots & \vdots \end{array}$$

$y = t^{k-1} e^{\lambda t}\cos \mu t$, and $y = t^{k-1} e^{\lambda t}\sin \mu t$,

are all solutions.

Example: $y^{(4)} - y = 0$

The characteristic equation is $r^4 - 1 = (r^2 + 1)(r + 1)(r - 1) = 0$, which has roots $r = 1, -1, i, -i$. Hence, the general solution is

$$y = C_1 e^{t} + C_2 e^{-t} + C_3 \cos t + C_4 \sin t.$$

Example: $y^{(5)} - 3y^{(4)} + 3y^{(3)} - y'' = 0$

The characteristic equation is $r^5 - 3r^4 + 3r^3 - r^2 = r^2(r - 1)^3 = 0$, which has roots $r = 0$ (a double root), and 1 (a triple root). Hence, the general solution is

$$y = C_1 e^{0t} + C_2 t e^{0t} + C_3 e^{t} + C_4 t e^{t} + C_5 t^2 e^{t}$$

$$= C_1 + C_2 t + C_3 e^{t} + C_4 t e^{t} + C_5 t^2 e^{t}.$$

Example: $y^{(4)} + 4y^{(3)} + 8y'' + 8y' + 4y = 0$

The characteristic equation is $r^4 + 4r^3 + 8r^2 + 8r + 4 = (r^2 + 2r + 2)^2 = 0$, which has roots $r = -1 \pm i$ (repeated). Hence, the general solution is

$$y = C_1 e^{-t} \cos t + C_2 e^{-t} \sin t + C_3 t e^{-t} \cos t + C_4 t e^{-t} \sin t.$$

Example: What is a 4th order homogeneous linear equation whose general solution is

$$y = C_1 e^{t} + C_2 e^{2t} + C_3 e^{3t} + C_4 e^{4t}\ ?$$

The solution implies that $r = 1, 2, 3$, and 4 are the four roots of the characteristic equation. Therefore, $r - 1$, $r - 2$, $r - 3$, and $r - 4$ are its factors. Consequently, the characteristic equation is

$$(r-1)(r-2)(r-3)(r-4) = 0$$

$$r^4 - 10r^3 + 35r^2 - 50r + 24 = 0$$

Hence, an equation is

$$y^{(4)} - 10y^{(3)} + 35y'' - 50y' + 24y = 0.$$

Note: The above answer is not unique. Every nonzero constant multiple of the above equation also has the same general solution. However, the indicated equation is the only equation in the standard form that has the given general solution.

Exercises B-4.1:

1 – 8 Find the general solution of each equation.

1. $y^{(3)} + 25y' = 0$

2. $y^{(3)} + 27y = 0$

3. $y^{(4)} - 18y'' + 81y = 0$

4. $y^{(4)} - 3y'' - 4y = 0$

5. $y^{(4)} + 32y'' + 256y = 0$

6. $y^{(5)} + 5y^{(4)} + 10y^{(3)} + 10y'' + 5y' + y = 0$

7. $y^{(5)} + 2y^{(4)} + 5y^{(3)} = 0$

8. $y^{(6)} - y = 0$

9 – 12 Solve each initial value problem.

9. $y^{(3)} + 4y'' - 5y' = 0,\quad y(0) = 4,\quad y'(0) = -7,\quad y''(0) = 23$

10. $y^{(3)} + 3y'' + 3y' + y = 0,\quad y(0) = 7,\quad y'(0) = -7,\quad y''(0) = 11$

11. $y^{(4)} - 10y'' + 9y = 0,\quad y(0) = 5,\ y'(0) = -1,\ y''(0) = 21,\ y^{(3)}(0) = -49$

12. $y^{(4)} + 13y'' + 36y = 0,\quad y(0) = 0,\ y'(0) = -3,\ y''(0) = 5,\ y^{(3)}(0) = -3$

13. Same as #10, except with initial conditions $y(6561) = 7$, $y'(6561) = -7$, $y''(6561) = 11$.

14. Same as #12, except with initial conditions $y(247) = 0$, $y'(247) = -3$, $y''(247) = 5$, $y^{(3)}(247) = -3$

15. Find a 3rd order homogeneous linear equation which has a particular solution $y = e^{t} - 2e^{-2t} + 5te^{-2t}$.

16. Find a 5th order homogeneous linear equation whose general solution is
$$y = C_1 e^{-t} + C_2 t e^{-t} + C_3 t^2 e^{-t} + C_4 \cos t + C_5 \sin t.$$

17. Find a 6th order homogeneous linear equation whose general solution is $y = C_1 + C_2 t + C_3 e^{-2t}\cos t + C_4 e^{-2t}\sin t + C_5 t e^{-2t}\cos t + C_6 t e^{-2t}\sin t$.

[Hint: the polynomial, with leading coefficient 1, that has complex conjugate roots $\lambda \pm \mu i$ has the form $r^2 - 2\lambda r + (\lambda^2 + \mu^2)$.]

18. Find a 6th order homogeneous linear equation whose general solution is $y = C_1 \cos 2t + C_2 \sin 2t + C_3 t\cos 2t + C_4 t\sin 2t + C_5 t^2 \cos 2t + C_6 t^2 \sin 2t$.

Answers B-4.1:

1. $y = C_1 + C_2 \cos 5t + C_3 \sin 5t$
2. $y = C_1 e^{-3t} + C_2 e^{\frac{3}{2}t} \cos \frac{3\sqrt{3}}{2} t + C_3 e^{\frac{3}{2}t} \sin \frac{3\sqrt{3}}{2} t$
3. $y = C_1 e^{3t} + C_2 e^{-3t} + C_3 t e^{3t} + C_4 t e^{-3t}$
4. $y = C_1 e^{2t} + C_2 e^{-2t} + C_3 \cos t + C_4 \sin t$
5. $y = C_1 \cos 4t + C_2 \sin 4t + C_3 t \cos 4t + C_4 t \sin 4t$
6. $y = C_1 e^{-t} + C_2 t e^{-t} + C_3 t^2 e^{-t} + C_4 t^3 e^{-t} + C_5 t^4 e^{-t}$
7. $y = C_1 + C_2 t + C_3 t^2 + C_4 e^{-t} \cos 2t + C_5 e^{-t} \sin 2t.$
8. $y = C_1 e^{t} + C_2 e^{-t} + C_3 e^{\frac{1}{2}t} \cos \frac{\sqrt{3}}{2} t + C_4 e^{\frac{1}{2}t} \sin \frac{\sqrt{3}}{2} t + C_5 e^{\frac{-1}{2}t} \cos \frac{\sqrt{3}}{2} + C_6 e^{\frac{-1}{2}t} \sin \frac{\sqrt{3}}{2} t$
9. $y = 5 - 2e^{t} + e^{-5t}$
10. $y = 7e^{-t} + 2t^2 e^{-t}$
11. $y = 4e^{t} - e^{-t} + 2e^{-3t}$
12. $y = \cos 2t - 3\sin 2t - \cos 3t + \sin 3t$
13. $y = 7e^{-t+6561} + 2(t - 6561)^2 e^{-t+6561}$
14. $y = \cos 2(t - 247) - 3\sin 2(t - 247) - \cos 3(t - 247) + \sin 3(t - 247)$
15. $y^{(3)} + 3y'' - 4y = 0$
16. $y^{(5)} + 3y^{(4)} + 4y^{(3)} + 4y'' + 3y' + y = 0$
17. $y^{(6)} + 8y^{(5)} + 26y^{(4)} + 40y^{(3)} + 25y'' = 0$
18. $y^{(6)} + 12y^{(4)} + 48y'' + 64y = 0$

Lastly, here is an example of an application of a very simple 4th order nonhomgeneous linear equation that might be familiar to many engineering students.

The static deflection of a uniform beam

Consider a horizontal beam of length *L*, of uniform cross section and made of homogeneous material, acted upon by a vertical force. The beam is positioned along the *x*-axis, with its left end at the origin. The deflection of the beam (its vertical displacement relative to the horizontal axis) at any point *x* is given by, according to the *Euler–Bernoulli beam equation*[*],

$$EI\,u^{(4)} = W(x), \qquad 0 < x < L.$$

The positive constants *E* and *I* are, respectively, the *Young's modulus* of elasticity of the material of the beam, and the beam's cross-sectional moment of inertia about the horizontal axis (i.e., the *second moment of the cross-sectional area*). The value of the product *EI* is a measurement of the beam's stiffness. The forcing function *W* describes the load/force that the beam bears per unit length. If the beam is not bearing external load, then $W(x) = w$, which is the weight-density of the beam itself (acting downwards, which is the positive direction per our usual convention).

The equation then becomes

$$EI\,u^{(4)} = w.$$

It is a (very) simple 4th order nonhomgeneous linear equation. It could be solved simply by integrating both sides four times with respect to *x*. However it is certainly more illustrative for our purpose to solve it using the general procedure that we have learned, namely by characteristic equation and undetermined coefficients methods to obtain both parts of the solution of this nonhomogeneous linear equation, in the form of $u = u_c + U$.

[*] The equation is originally $\frac{d^2}{dx^2}(EI\frac{d^2u}{dx^2}) = W(x)$. When *EI* is constant, it simplifies to $EI\frac{d^4u}{dx^4} = W(x)$.

Its characteristic equation is $EIr^4 = 0$, which has zero as a (quadruple) root. The complementary solution is, therefore, $u_c = C_1 + C_2x + C_3x^2 + C_4x^3$.

The particular solution can be found using the method of Undetermined Coefficients that we have already learned. What form would the particular solution of the equation take?

The general solution of this deflection equation is

$$u(x) = C_1 + C_2\,x + C_3\,x^2 + C_4\,x^3 + \frac{w}{24EI}x^4.$$

The graph of the deflection function is called the *deflection curve* or *elastic curve* of the beam.

Another interesting aspect of this problem is that this equation does not come with initial conditions. Instead, it comes paired with four boundary conditions – describing the physical conditions at the two ends, at $x = 0$ and $x = L$, of the beam. For example, if the beam is securely embedded into walls on both ends, the deflection function above must satisfy the boundary conditions:

$$u(0) = u'(0) = u(L) = u'(L) = 0.$$

In this case the four conditions tell us that the displacement, $u(x)$, is zero at both ends, and that the slope of the beam, $u'(x)$, is also zero at the two ends – hence the beam is securely fixed at both ends (by embedding into the walls). Notice that there are four conditions, which are necessary to solve the four unknown coefficients present in the general solution of this fourth order equation.

Being a fourth order equation, the boundary conditions in a beam problem frequently involve u'' and u'''. Physically, u'' represents the *bending moment* and u''' represents the *shear force* experienced by the beam at a given point.

For a *simply supported beam*, as another example, where a beam is either pinned or hinged at both ends (having no displacement nor resistance to rotation at the two ends), the required boundary conditions are, therefore,

$$u(0) = u''(0) = u(L) = u''(L) = 0.$$

We will study simple boundary value problems, in a quite different context, later in the course.

Exercises B-4.2:

1. *Deflection curves of uniformly loaded beams* Based on the earlier calculation, a beam bearing a uniformly distributed load of density w has a deflection curve in the form

$$u(x) = C_1 + C_2\, x + C_3\, x^2 + C_4\, x^3 + \frac{w}{24EI} x^4 .$$

Find the deflection curve of each set of boundary conditions below.
(i) Beam with both ends embedded: $u(0) = u'(0) = u(L) = u'(L) = 0$.
(ii) Simply supported beam: $u(0) = u''(0) = u(L) = u''(L) = 0$.
(iii) Cantilever beam (left end embedded, right end free): $u(0) = u'(0) = u''(L) = u'''(L) = 0$.

2. Consider a beam of length L bearing a load that is proportional to the distance from the left endpoint, i.e. $W(x) = wx$, for some positive constant w.
(i) Find the general solution of its deflection curve.
(ii) Find the deflection curve given $u(0) = u'(0) = u(L) = u'(L) = 0$.
(iii) Find the deflection curve given $u(0) = u''(0) = u(L) = u''(L) = 0$.
(iv) Find the deflection curve given $u(0) = u'(0) = u''(L) = u'''(L) = 0$.

3. Consider a cantilever beam of length L bearing a load that is proportional to the distance from the right endpoint, i.e. $W(x) = w(L - x)$, for some positive constant w. Find its deflection curve given that $u(0) = u'(0) = u''(L) = u'''(L) = 0$.

Answers B-4.2:

1. (i) $u(x) = \dfrac{w}{24EI}\left(L^2x^2 - 2Lx^3 + x^4\right)$

   (ii) $u(x) = \dfrac{w}{24EI}\left(L^3x - 2Lx^3 + x^4\right)$

   (iii) $u(x) = \dfrac{w}{24EI}\left(6L^2x^2 - 4Lx^3 + x^4\right)$

2. (i) $u(x) = C_1 + C_2\,x + C_3\,x^2 + C_4\,x^3 + \dfrac{w}{120EI}x^5$

   (ii) $u(x) = \dfrac{w}{120EI}\left(2L^3x^2 - 3L^2x^3 + x^5\right)$

   (iii) $u(x) = \dfrac{w}{360EI}\left(7L^4x - 10L^2x^3 + 3x^5\right)$

   (iv) $u(x) = \dfrac{w}{120EI}\left(20L^3x^2 - 10L^2x^3 + x^5\right)$

3. The general solution is

$$u(x) = C_1 + C_2\,x + C_3\,x^2 + C_4\,x^3 + \frac{wL}{24EI}x^4 - \frac{w}{120EI}x^5.$$

The cantilever beam's deflection curve is

$$u(x) = \frac{w}{120EI}\left(10L^3x^2 - 10L^2x^3 + 5Lx^4 - x^5\right).$$

Application of Graph Theory in Electrical Network

Berdewad O. K.[1], Dr. Deo S. D.[2]

[1]Department of Mathematics, NES College, Bhadrawati, Dist. Chandrapur, India

[2]Gondwana University, Gadachiroli, MS, India

Abstract: ***Graph theory is helpful in various practical problems solving in circuit or network analysis and data structure. It leads to graph practically not possible to analyze without the aid of computer. In electrical engineering the word is used for edge, node for vertex and loop for circuit. An electrical network is the set of electronic components i.e. resistors, inductors and capacitors etc. Electric network analysis and synthesis are the study of network topology. Electric network problem can be represented by drawing graphs. In this paper, we present a circuit network in the concept of graph theory application and how to apply graph theory to model the circuit network.***

Keywords: Graph theory, adjacency matrix, electrical circuit and analysis

1. Introduction

A connected graph without closed path i.e. tree was implemented by G.Kirchhoff in 1847 and he employed graph theoretical concept in the calculation of currents in network or circuits and was improved upon J.C.Maxwell in 1892.[4] Ever since, graph theory has been applied in electrical network analysis .An electrical network is a collection of components and device interconnected electrically .The network components are idealized of physical device and system, in order to for them to represent several properties, they must obey the Kirchhoff's law of currents and voltage.[1]A graph representation of electrical network in terms of line segments or arc called edges or branches and points called vertices or terminals.

2. Basic Definition of Graph Theory

Graphs are amenable for pictorial representation of a system using two basic components vertex and edges. A vertex is represented by a dot and an edge is represented by line segment connecting the dots associated with the edge. If the edges of a graph direct one vertex to the other vertex, then the graph is called as a directed graph. Otherwise graph is called an undirected graph. [2] Formally, a graph G= (V, E) contains a finite set V= (v_1, v_2,…..,v_n) of elements called vertices and a finite set (e_1 ,e_2,…..,e_m) of elements called edges. In an undirected graph G= (V, E), the edges are unordered pairs, and each edge e_1 in E is associated with two vertices v_1 and v_2, and it is written as either e1= (v_1,v_2) or e1= (v_2, v_1).But, In a directed graph, each edge e_1 in E is associated with an ordered pairs of vertices (v_1, v_2) and it is denoted the directed edge e_1 from v_1 to v_2. Two vertices v_1 and v_2 of a graph are adjacent, if there is an edge, v_1v_2 connecting them, then vertices are them considered incident to the edge v_1v_2 . [5]

3. Analysis of Electrical Circuit

Ohm's law states that for an edge 'e',the current flowing across that edge Ie is given by $i_e = \frac{pc}{rc}$ = pc.ce

We see that this means that i{uv} = -i{vu} and the negative current as positive currents flowing the different way. The weight of an edge as the conductance of that edge, which denote Ce for a given edge e .The resistance of an edge r_e is defined as $r_e = \frac{1}{ce}$.[3]

Both the resistance and conductance are independent of edge such as r(vu) = r(uv) and c(uv) = c(vu).

4. Kirchhoff's Circuit Law

Kirchhoff's voltage law states that for a closed loop SV=0 or SV rise is equal to SV drops.[1] The total resistance of 'n' resistors in series is RT= R1+R2+R3+………+Rn and the total power are
PT= P1+P2+P3+…….Pn

In series, So that the same current flows through all the components but a different potential voltage can exist across every one. In parallel, so that the same potential difference exists across every components but each component may carry a different current.

Representation of circuit and its graph:
A graph model is used to represented circuit network inn graph by tracing the nodes of the circuit and edges contain in circuit.

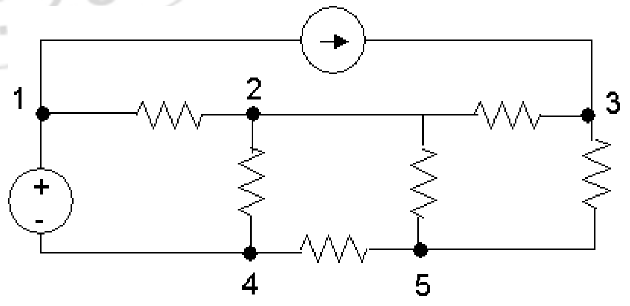


Figure 1

Here is the graph of the circuit,

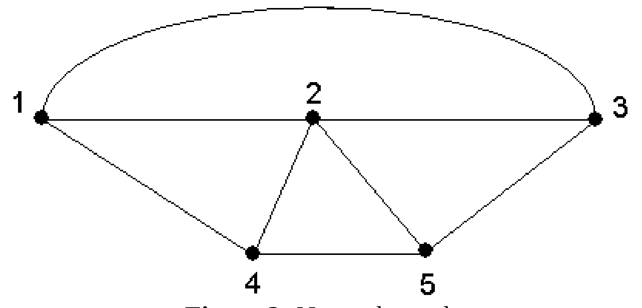


Figure 2: Network graph

A circuit is a path which ends at the vertex it begins. An electric circuit is a closed loop formed by source, wires, load, and a switch, when switch is turned on the electrical circuit is complete and current flows from negative terminals of the power source. An electrical circuit is categories in to three type namely series, parallel and series and parallel circuit. The representation of graph in circuit network are one of the type of representation of graph in which the current flows in circuit and present the linking of connection between resistors series and parallel connection are determined in the circuit.[4] The representation is

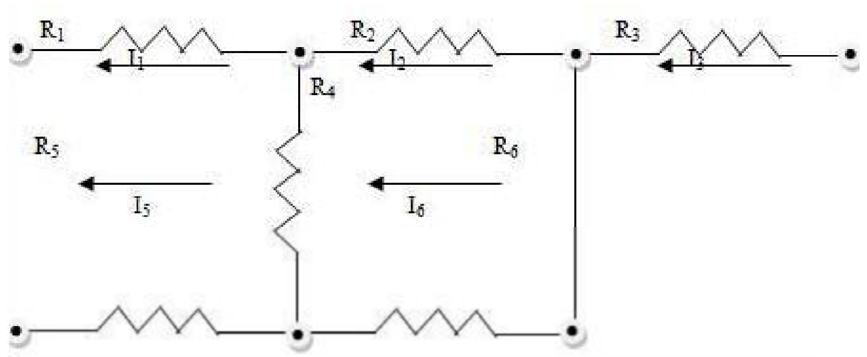


Figure 3

The schematic figure of the electric circuit is as follows,

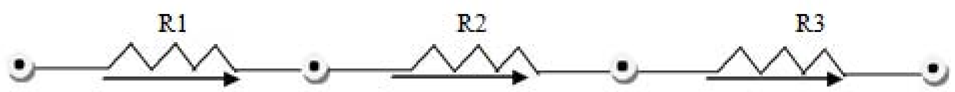


Figure 4

The electrical features of individual network components can be representing suitably in the form of primitive network matrix that describe the performance of interconnected network.

5. Graph Representation of Matrix

A graph can actually be represented using matrices method the two of the most widely used matrices for graph representation is adjacency and incidence matrices. An adjacency matrix is a square matrix in which each row and column is represented by a vertex [5].
Consider figure 4, as an example it has three vertices V={ R1, R2,R3} this mean that the square matrix must be 3x3 let each row and column is represented by each of the six vertices in V.

	R1	R2	R3
R1	0	1	0
R2	1	0	1
R3	0	1	0
R4	0	0	1
R5	0	0	0
R6	0	1	1

The adjacency matrix of 3×3 matrix square matrix represented as follows

$$G = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

6. Conclusion

In this research we focus on the application of graph theory to electrical network analysis and matrix approach as an electrical network analysis. Graph theory is a very interesting topic in mathematics due to numerous applications in various fields especially in computer and electrical engineering. We use the graph theory concept and techniques that we have developed to study electrical networks. Thus, graph theory has more practical application particulars in solving electric network.

References

[1] A. Sudhakaran, Electrical circuit analysis, Tata Mc Grow-Hill Pvt ltd.
[2] B.Bollobas, Modern Graph Theory, Springer 1998.
[3] F.Kirchoff's,Ubar die Auflosung der Gleichungen,auf welche man bei der Untersuchung der linearen Verteilung galvanischer Strome gefuhrt wird,Ann.Phys.chem. 72(1847),497-508
[4] Introductory Graph Theory for Electrical and Electronics Engineers,IEEE MULTIDISCIPLINARY ENGINEERING EDUCATION MAGAZINE.
[5] Narasih Deo, Graph theory & its Application to computer science.

1 Ordinary Differential Equations—Separation of Variables

1.1 Introduction

Calculus is fundamentally important for the simple reason that almost everything we study is subject to change. In many if not most such problems, the problem is modeled by an equation that involves derivations. Such an equation is called a *differential equation.*

Differential equations take many forms but one of the simplest examples is

$$\frac{dy}{dx} = 6x.$$

The equation is formed using two variables x and y. The variable x is known as the independent variable and the variable y as the dependent variable.

The aim is to get an equation for y in terms of x, i.e. of the form $y = f(x)$; which of course can be solved by integration:

$$\begin{aligned} \frac{dy}{dx} &= 6x \\ \int dy &= \int 6x \; dx \\ y &= 3x^2 + C. \end{aligned}$$

Therefore the *general* solution of $\frac{dy}{dx} = 6x$ is $y = 3x^2 + C$ where C is an arbitrary constant.

Hence we need a *boundary* condition (typically in the form of an *initial condition* $y(0) = \textit{something}$) in order to obtain a unique solution.

For example, suppose that we specify the boundary condition that $y = 4$ when $x = 1$, written $y(1) = 4$. Then $4 = 3(1) + c \Rightarrow c = 4 - 3 = 1$. Therefore we get the unique solution $y = 3x^2 + 1$.

Before we look at different types of differential equations (DE), we introduce some terminology.

<u>Order</u>

The *order* of a DE is the order of the highest derivative in the equation:

For example, give the order of the following DE's:

(i) $\frac{dy}{dx} = 2y$

(ii) $\frac{d^2y}{dx^2} - \left(\frac{dy}{dx}\right)^4 + y = 0$

(iii) $\frac{d^4y}{dx^4} = x^2y$.

The DE's are:

(i) First order;

(ii) Second order;

(iii) Fourth order.

Linear

A DE is said to be *linear* if the dependent variable and its derivatives occur to the first power only and if there are no products involving the dependent variable and/or its derivatives.

Example. Which of the following DEs are linear?

a)	$\frac{dy}{dx} = x^2$	Yes.
b)	$\frac{dy}{dx} + 2y = \cos x$	Yes.
c)	$y\frac{dy}{dx} = x^3$	No.
d)	$\frac{dy}{dx} + 4y^2x = \sin x$	No.
e)	$\sin x\frac{dy}{dx} + y\cos x = \sin x$	Yes.

1.2 First-order ODEs (Ordinary differential equations)

These are differential equations involving just one variable and its derivatives, such as,

$$(a)\ \frac{dy}{dx} = y, \qquad (b)\ \frac{dy}{dx} = 2x + 4 \qquad (c)\ \frac{dy}{dx} = 2\cos 2x \text{ and } (d)\ \frac{dy}{dx} = 2\cos 2x + y^2.$$

The examples (a,b,c) are all fairly straightforward to solve, although there is a slight twist in (a), so I will leave that one until later. We will also see how to solve examples like (d) later in this course.

(b) $\frac{dy}{dx} = 2x + 4.$

By integration $\int dy = \int (2x + 4)\ dx$ and so $y = x^2 + 4x + A$.

(c) $\frac{dy}{dx} = 2\cos 2x.$

By integration $\int dy = \int 2\cos(2x)\ dx$. Therefore $y = 2[\frac{1}{2}\sin(2x)] + c$ and hence $y = \sin(2x) + c$.

In more complicated examples it might be necessary to use substitution in the integration step.

For example, suppose that $\frac{dy}{dx} = \cos x \sin^2 x$ and we have the boundary condition that $y\left(\frac{\pi}{2}\right) = \frac{4}{3}$.

Then by integration,

$$\int dy = \int \cos x \sin^2 x\, dx$$

Use the substitution $u = sin\, x$. Then $\frac{du}{dx} = \cos x$, and so, $\int dy = \int u^2 du$ from which we get $y = \frac{1}{3}u^3 + C$.

It is then necessary to rewrite the equation in terms of x. Therefore $y = \frac{1}{3}\sin^3 x + C$.

But $y = \frac{4}{3}$ when $x = \frac{\pi}{2}$. Thus $\frac{4}{3} = \frac{1}{3}\sin^3\left(\frac{\pi}{2}\right) + C$, and so,

$$\frac{4}{3} - \frac{1}{3} = C \quad \Rightarrow \quad C = 1.$$

Hence the unique solution is $y = \frac{1}{3}\sin^3 x + 1$.

1.3 Separation of variables

First order ODEs (and higher order ODEs for that matter) fall into various categories. Separation of variables forms a general category which are straightforward to solve.

For separation of variables we require that the equation can be (re)written in the form:

$$\frac{dy}{dx} = f(x)g(y)$$

where $f(x)$ is only a function of x and $g(y)$ is only a function of y.

Then the general solution of the first order ODE

$$\frac{dy}{dx} = f(x)g(y)$$

is given by

$$\int \frac{1}{g(y)}\, dy = \int f(x)\, dx.$$

Often we need to rewrite the equation so that it is in the correct form.

For example, the ODE

$$y\frac{dy}{dx} - 3 = x$$

can be rewritten as $y\frac{dy}{dx} = x + 3$ and then as

$$\frac{dy}{dx} = \frac{x+3}{y} = f(x)g(y)$$

where $f(x) = x + 3$ and $g(y) = \frac{1}{y}$.

Therefore the equation $y\dfrac{dy}{dx} - 3 = x$ can be solved using separation of variables.

Examples. Which of the following ODEs can be solved by separation of variables?

a) $\frac{dy}{dx} = \cos x \sin y$ Yes.

b) $\sin y \times \frac{dy}{dx} + x^2 = 0$ Yes since $\dfrac{dy}{dx} = -\dfrac{x^2}{\sin y}$.

c) $\frac{dy}{dx} = x^2 + y$ No.

General Approach

The general approach to solving a first order ODE using separation of variables is as follows:

a) Rewrite (if necessary) the equation in the required form:

$$\frac{dy}{dx} = f(x)g(y)$$

b) Find the general solution for

$$\int \frac{1}{g(y)}\ dy = \int f(x)\ dx$$

c) If boundary conditions are given, solve to find the unique solution.

Example. Solve the ODE

$$\frac{1}{y^2}\frac{dy}{dx} + x^2 = 0$$

subject to the initial condition $y(0) = 2$.

a) $\dfrac{dy}{dx} = -x^2y^2$.

b) Thus $\displaystyle\int y^{-2}\ dy = \int -x^2 dx$ and so $-y^{-1} = -\frac{1}{3}x^3 + c$.

c) Use the boundary condition $y(0) = 2$, to obtain $c = -\frac{1}{2}$ and hence

$$y^{-1} = \frac{1}{3}x^3 + \frac{1}{2}.$$

It is quite often the case (as is true here) that one has an implicit function of y rather than an explicit one. We *could* rewrite the solution as $y = \dfrac{1}{\frac{1}{3}x^3 + \frac{1}{2}}$ but I do not think this is any nicer than the previous expression.

Here is a particular ODE that turns up a lot:

1.3.1 Important Fact

The ODE $\frac{dy}{dt} = k(y-b)$ *has general solution*

$$y = b + Ce^{kt} \qquad \textit{where } C \textit{ is a constant that can take any real value.}$$

Reason: There is a slight subtlety here so let us work it out carefully. As usual we can separate variables and get

$$\int \frac{dy}{y-b} = \int k dt,$$

from which $\ln|y-b| = kt + B$, for some constant B and hence $|y-b| = e^{kt+B}$. Therefore,

$$y - b = \pm e^{kt+B} = \pm e^B e^{kt} = Ce^{kt},$$

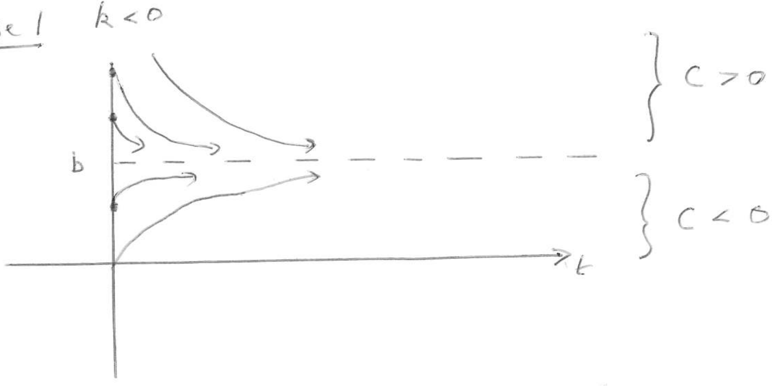
where $C = \pm e^B$ can take any real value. QED

Exercise: Check that the function $y = b + Ce^{kt}$ does indeed satisfy $\frac{dy}{dt} = k(y-b)$.

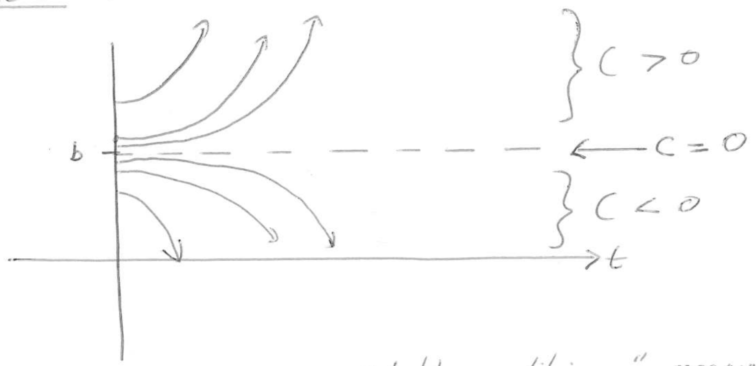
It is illustrative to see what happens to our solution y as the parity of k and the value of C are varied. The relevant sketches appear on the next page.

Solutions to $\frac{dy}{dt} = k(y-b)$ (thus $y = b + Ce^{kt}$)

Case 1 $k < 0$



Case 2 $k > 0$



In Case 1 $y = b$ is a "stable equilibrium" meaning that if you change y slightly it returns to $y = b$

In Case 2 $y = b$ is an "unstable equilibrium" in the sense that if I change y slightly to $b \pm \varepsilon$ then however small — but positive — is my ε, as time increases y shoots off ~~to~~ $\pm\infty$

1.4 Newton's Law of Cooling

Problem. *Sherlock Holmes finds a body at* $1am$ *with temperature* $30°C$*. An hour later the body has temperature* $25°C$*. If the room temperature is* $10°C$*, when did the person die?*

The basic fact we need to solve this problem is:

Newton's Law of Cooling: *The rate of change of temperature of a body is proportional to the difference between the temperature of the body and the ambient temperature.*

So, returning to our problem, we let T denote the temperature of our body at time t. From Newton's Law of Cooling 1.4, we get the rate of change in T; that is $\frac{dT}{dt}$, is proportional to $T - 10$. Written mathematically:

$$\frac{dT}{dt} = k(T - 10) \qquad \textit{for some constant } k. \tag{1.1}$$

As an aside, note that here k will be negative since the temperature will decrease. You could also write $\frac{dT}{dt} = -k(T - 10)$, with $k > 0$. It obviously does not matter which way we do it!

So, now I can apply (1.3.1) to (1.1) to get

$$y = 10 + Ce^{kt}.$$

Now I should decide my units for t. Certainly I should measure t in hours, since that is the question is naturally phrased. More importantly, it is best to take $t = 0$ to be $1am$, since that will make it easiest to apply our initial conditions. Thus, at $t = 0$ we have $T = 30$ and so $30 = 10 + Ce^0$ or $C = 20$. Thus $y = 10 + 20e^{kt}$. Next we have $T(1) = 25$ from which $25 = 10 + 20e^k$ or $e^k = 15/20$ and $k = -0.29$; thus

$$y = 10 + 20e^{-0.29t}.$$

Note that we have indeed found that k is negative, which fits with our intuition and suggests we are on the right track.

Now finally we can solve the problem: the person died when the body temperature was 37; thus when $37 = 10 + 20e^{0.29t}$. In other words $e^{0.29t} = 27/20$ and $t = \frac{\ln(27/20)}{0.29} = -1.03$.

In other words the person died at time $t = -1.03$ or (just before) midnight.

Another example—interest payments: Suppose that you are paying interest on your student loan at a rate of $5\%pa$, compounded continuously (where pa means per year). So, if the amount of the loan is

£$y(t)$ at time t (in years) then $\dfrac{dy}{dt} = \dfrac{5}{100}y$. (Do you see why this is true?) This has solution $y = Ce^{t/20}$. In other words, the amount you owe grows exponentially.

Now lets make the question harder.

Question: *Again you are paying* 5% *interest, compounded continuously but suppose you also pay it off at a continuous rate of of* £500*pa. If you took out a loan of* £3,000 *how quickly will you pay it off?*

Answer: Now there are two changes in y. As before, you are paying interest at a rate of 5% which gives a contribution to $\dfrac{dy}{dt}$ of $1/20y$. But now you are also decreasing y by 500 each year. Thus

$$\frac{dy}{dt} = \frac{1}{20}y - 500.$$

If we rewrite this in the form $\dfrac{dy}{dt} = \dfrac{1}{20}(y - 10,000)$, then we can apply (1.3.1) and we get

$$y = 10,000 + Ce^{t/20}.$$

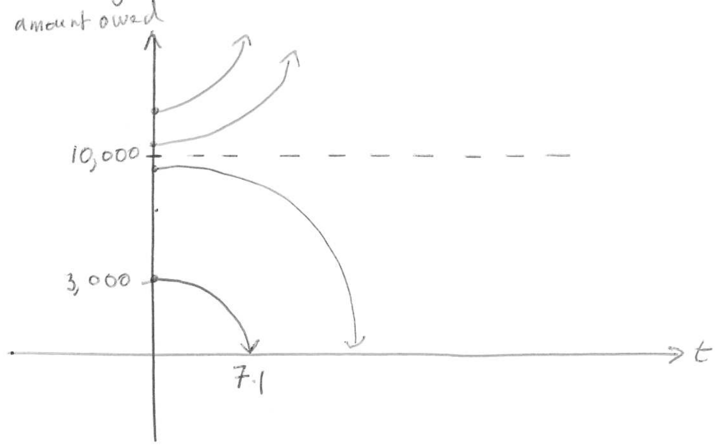
From $y(0) = 3000$ we get $C = -7,000$ and so

$$y = 10,000 - 7,000e^{t/20}.$$

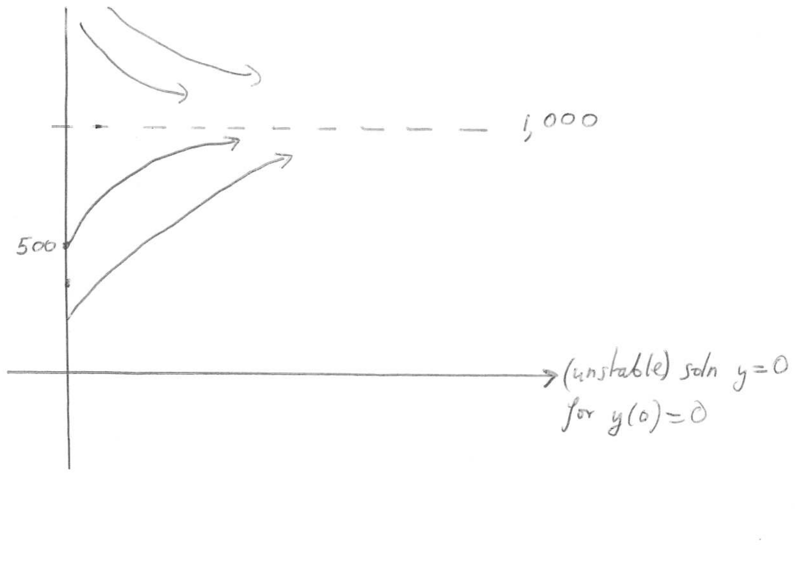
Finally the loan is paid off when $y = 0$ or $e^{t/20} = 10/7$, which gives $t = 20\ln(10/7) = 7.13$. So, you pay if off in 7.13 years.

You can of course repeat this question for different values of the amount $y(0)$ you borrow. The sketches are given on the next page. For $y(0) < 10,000$ it decreases exponentially, but for $y(0) > 10,000$ it increases exponentially.

Interest on your loan



Population growth for differing values of y(0)



Example: Population density. (i) First a rather general question. *Consider the population density $y(t)$ of a certain population of animals at time t. The rate of change of $y(t)$ depends upon two constraints: First the excess of birth rate over death rate; this is proportional to the number of animals present. Secondly extra deaths due to overcrowding is proportional to the square of the number of animals present. Write down a differential equation that models this.*

Answer: Note that the number of animals is proportional to the density of animals, so the question tells us that the birth rate is also proportional to $y(t)$; this gives a contribution to $\frac{dy}{dt}$ of the form αy for some α. Similarly the excess death rate gives a contribution to $\frac{dy}{dt}$ of the form βy^2 for some β. Thus the equation we want is

$$\frac{dy}{dt} = \alpha y + \beta y^2.$$

Comment: We actually know that $\alpha > 0$ (since extra births increase the population) and $\beta < 0$ (since extra deaths decrease the population. Fortunately we do not need to put in $\pm$ signs as they will always come out in the wash.

(ii) A more explicit version: *Suppose in the above equation that $\alpha = 1,000$, $\beta = -1$ and $y(0) = 500$. Find a formula for the population density $y(t)$ and sketch your solution.*

Answer: We now have the equation $\frac{dy}{dt} = 1,000y - y^2$ and we can separate variables to give

$$\int \frac{dy}{1,000y - y^2} = \int dt. \tag{1.2}$$

To solve the LHS we need to use partial fractions; so write

$$\frac{1}{1,000y - y^2} = \frac{1}{y(1,000 - y)} = \frac{A}{y} + \frac{B}{1000 - y} = \frac{A(1000 - y) + yB}{y(1000 - y)}.$$

From this we obtain $1 = 1000A - yA + yB$ and so $A = 10^{-3} = B$. Thus

$$\int \frac{dy}{1,000y - y^2} = \int \frac{10^{-3}}{y} dy + \int \frac{10^{-3}}{1000 - y} dy = 10^{-3} \ln |y| - 10^{-3} \ln |10^3 - y| = 10^{-3} \ln \left| \frac{y}{1000 - y} \right|.$$

Hence the solution to (1.2) is

$$10^{-3} \ln \left(\frac{y}{1000 - y} \right) = \int dt = t + C,$$

for some constant C. Substituting in $y(0) = 500$ gives $C = 10^{-3} \ln(500/500) = 0$. Therefore,

$$10^{-3} ln \left(\frac{y}{1000 - y} \right) = \int dt = t.$$

Equivalently, $\frac{y}{1000-y} = e^{1000t}$. If you want you can solve this equation for y, giving

$$y = \frac{1000e^{1000y}}{1+e^{1000y}},$$

but I do not think that this is much nicer. The sketch (for a range of different initial conditions) is given on page 10.

Exercise: *Solve the equation from part (i) of the Population Density Question, for $\beta = -1$ but arbitrary α and arbitrary initial conditions. You should find that $y = \frac{\alpha A e^{\alpha t}}{(1+Ae^{\alpha t})}$. Here, A will depend upon the initial conditions. Whatever they are, you will see that $y(t) \to \alpha$ as $t \to \infty$.*

So the same rough sketch applies as for the explicit case.

Example: *Suppose that the height of a wave satisfies the following ODE:*

$$\frac{dy}{dt} = -ky\cos(t)$$

where $k > 0$.

Suppose that initially (i.e. at time $t = 0$) the wave is 2 units high and at time $t = \frac{\pi}{2}$ the wave is 1 unit high. Find an expression for the height of the wave for all time.

Solution. Separation of variables gives

$$\int \frac{dy}{y} = \int -k\cos t \; dt$$

Thus $\ln(y) = -k\sin t + c$ which has solution

$$y = \exp(-k\sin t + c) = e^c \exp(-k\sin t).$$

At time $t = 0$, $\sin t = 0$. Therefore $2 = e^c e^{-k(0)}$ and so, $2 = e^c$. Thus

$$y(t) = 2\exp(-k\sin t)$$

Next at time $t = \frac{\pi}{2}$, $\sin t = 1$. Therefore $1 = 2e^{-k}$ and so, $\frac{1}{2} = e^{-k}$ which can be solved to give $k = -\ln\left(\frac{1}{2}\right)$.

Thus $k = \ln(2) = 0.693$ and the unique solution for the wave is

$$y(t) = 2\exp(-0.693\sin t).$$

Order and degree of a differential equation

Order of a differential equation

order of a differential equation is defined as the order of the highest order derivative of the dependent variable with respect to the independent variable involved in the given differential equation.

Consider the following differential equations:

$$\frac{dy}{dx} = e^x \qquad \dots (6)$$

$$\frac{d^2y}{dx^2} + y = 0 \qquad \dots (7)$$

$$\left(\frac{d^3y}{dx^3}\right) + x^2\left(\frac{d^2y}{dx^2}\right)^3 = 0 \qquad \dots (8)$$

The equations (6), (7) and (8) involve the highest derivative of first, second and third order respectively. Therefore, the order of these equations are 1, 2 and 3 respectively.

Degree of a differential equation

To study the **degree of a differential equation**, the key point is that the differential equation must be a polynomial equation in derivatives, i.e., y′, y″, y‴ etc. Consider the following differential equations:

$$\frac{d^3y}{dx^3} + 2\left(\frac{d^2y}{dx^2}\right)^2 - \frac{dy}{dx} + y = 0 \qquad \dots (9)$$

$$\left(\frac{dy}{dx}\right)^2 + \left(\frac{dy}{dx}\right) - \sin^2 y = 0 \qquad \dots (10)$$

$$\frac{dy}{dx} + \sin\left(\frac{dy}{dx}\right) = 0 \qquad \dots (11)$$

We observe that equation (9) is a polynomial equation in y‴, y″ and y′, equation (10) is a polynomial equation in y′ (not a polynomial in y though). Degree of such differential equations can be defined. But equation (11) is not a polynomial equation in y′ and degree of such a differential equation can not be defined.

By the degree of a differential equation, when it is a polynomial equation in derivatives, we mean the highest power (positive integral index) of the highest order derivative involved in the given differential equation.

In view of the above definition, one may observe that differential equations (6), (7), (8) and (9) each are of degree one, equation (10) is of degree two while the degree of differential equation (11) is not defined.

NOTE: Order and degree (if defined) of a differential equation are always positive integers.

Example Find the order and degree, if defined, of each of the following differential equations:

(i) $\frac{dy}{dx} - \cos x = 0$ (ii) $xy\frac{d^2y}{dx^2} + x\left(\frac{dy}{dx}\right)^2 - y\frac{dy}{dx} = 0$

(iii) $y''' + y^2 + e^{y'} = 0$

Solution (i) The highest order derivative present in the differential equation is

$$\frac{dy}{dx}$$

, so its order is one. It is a polynomial equation in y′ and the highest power raised to

$$\frac{dy}{dx}$$ is one, so its degree is one.

(ii) The highest order derivative present in the given differential equation is

$$\frac{d^2y}{dx^2}$$, so its order is two. It is a

polynomial equation in $\frac{d^2y}{dx^2}$ and

$$\frac{dy}{dx}$$

and the highest power raised

to $\frac{d^2y}{dx^2}$ is one, so its degree is one.

(iii) The highest order derivative present in the differential equation is y''' , so its order is three. The given differential equation is not a polynomial equation in its derivatives and so its degree is not defined.

Working Rules For Finding Complementary Function of Linear Differential Equation

Case 1: -

If the roots are unequal ($m = m_1, m_2, m_3$) then the complementary function is

C.F $= c_1e^{m1x} + c_2e^{m2x} + c_3e^{m3x}$

Case 2: -

If the roots are equal ($m = m_1, m_1, m_1$) then the complementary function is

C.F $= (c_1 + c_2x + c_3x^2)\ e^{m1x}$

Case 3: -

If the roots are complex ($m = a \pm ib$) then the complementary function is

C.F $= e^{ax}\ (c_1\cos bx + c_2 \sin bx)$, $c_1e^{ax} \cos(bx + c_2)$ or, $c_1e^{ax} \sin (bx + c_2)$

And if the two equal part of complex roots ($m = a \pm ib, a \pm ib$) then the complementary function is

C.F $= e^{ax}\ \{(c_1 + c_2x) \cos bx + (c_3 + c_4x) \sin bx\}$

Case 4: -

If the roots are "$a \pm \sqrt{b}$" then the complementary function is

C.F $= e^{ax}\ (c_1\cos x\sqrt{b} + c_2 \sin x\sqrt{b})$, $c_1\ e^{ax} \cosh (x\sqrt{b} + c_2)$ or, $c_1e^{ax} \sinh (x\sqrt{b} + c_2)$

$\boxed{\textbf{Partial derivatives}}$

Notice: this material must <u>not</u> be used as a substitute for attending the lectures

0.1 Recall: ordinary derivatives

If y is a function of x then $\frac{dy}{dx}$ is the **derivative** meaning the gradient (slope of the graph) or the rate of change with respect to x.

0.2 Functions of 2 or more variables

Functions which have more than one variable arise very commonly. Simple examples are

- formula for the area of a triangle $A = \frac{1}{2}bh$ is a function of the two variables, base b and height h
- formula for electrical resistors in parallel:

$$R = \left(\frac{1}{R_1} + \frac{1}{R_2} + \frac{1}{R_3}\right)^{-1}$$

  is a function of three variables R_1, R_2 and R_3, the resistances of the individual resistors.

Let's talk about functions of two variables here. You should be used to the notation $y = f(x)$ for a function of one variable, and that the graph of $y = f(x)$ is a curve. For functions of two variables the notation simply becomes

$$z = f(x, y)$$

where the two **independent** variables are x and y, while z is the **dependent** variable. The graph of something like $z = f(x, y)$ is a **surface** in three-dimensional space. Such graphs are usually quite difficult to draw by hand.
Since $z = f(x, y)$ is a function of two variables, if we want to differentiate we have to decide whether we are differentiating with respect to x or with respect to y (the answers are different). A special notation is used. We use the symbol ∂ instead of d and introduce the **partial derivatives** of z, which are:

- $\dfrac{\partial z}{\partial x}$ is read as "partial derivative of z (or f) with respect to x", and means differentiate with respect to x holding y constant
- $\dfrac{\partial z}{\partial y}$ means differentiate with respect to y holding x constant

Another common notation is the subscript notation:

$$z_x \quad \text{means} \quad \frac{\partial z}{\partial x}$$

$$z_y \quad \text{means} \quad \frac{\partial z}{\partial y}$$

Note that we cannot use the dash $'$ symbol for partial differentiation because it would not be clear what we are differentiating with respect to.

0.3 Example

Calculate $\frac{\partial z}{\partial x}$ and $\frac{\partial z}{\partial y}$ when $z = x^2 + 3xy + y - 1$.

Solution. To find $\frac{\partial z}{\partial x}$ treat y as a constant and differentiate with respect to x. We have $z = x^2 + 3xy + y - 1$ so

$$\frac{\partial z}{\partial x} = 2x + 3y$$

Similarly

$$\frac{\partial z}{\partial y} = 3x + 1$$

0.4 Example

Calculate $\frac{\partial z}{\partial x}$ and $\frac{\partial z}{\partial y}$ when $z = 1 - x - \frac{1}{2}y$. Interpret your answers and draw the graph.

Solution. The graph of $z = 1-x-\frac{1}{2}y$ is a plane passing through the points $(x, y, z) = (1, 0, 0)$, $(0, 2, 0)$ and $(0, 0, 1)$. The partial derivatives are:

$$\frac{\partial z}{\partial x} = -1, \quad \frac{\partial z}{\partial y} = -\tfrac{1}{2}$$

Interpretation: $\frac{\partial z}{\partial x}$ is the slope you will notice if you walk on the surface in a direction keeping your y coordinate fixed. $\frac{\partial z}{\partial y}$ is the slope you will notice if you walk on the surface in such a direction that your x coordinate remains the same. There are, of course, many other directions you could walk, and the slope you will notice when walking in some other direction can be worked out knowing both $\frac{\partial z}{\partial x}$ and $\frac{\partial z}{\partial y}$. It's like when you walk on a mountain, there are many directions you could walk and each one will have its own slope.

0.5 Other examples of evaluating partial derivatives

(i) $z = \ln(x^2 - y)$. Then $\frac{\partial z}{\partial x} = \frac{2x}{x^2 - y}$ and $\frac{\partial z}{\partial y} = \frac{-1}{x^2 - y}$. [To deduce these results we used the fact that if $y = \ln f(x)$ then $\frac{dy}{dx} = \frac{f'(x)}{f(x)}$].

(ii) $z = x \cos y + ye^x$. Then $\frac{\partial z}{\partial x} = \cos y + ye^x$ and $\frac{\partial z}{\partial y} = -x \sin y + e^x$.

(iii) $z = y \sin xy$. Then $\frac{\partial z}{\partial x} = y(y \cos xy) = y^2 \cos xy$ and $\frac{\partial z}{\partial y} = yx \cos xy + \sin xy$. For the second result we used the product rule.

(iv) If $x^2 + y^2 + z^2 = 1$ find the rate at which z is changing with respect to y at the point $(\frac{2}{3}, \frac{1}{3}, \frac{2}{3})$. *Solution.* We have $z = (1 - x^2 - y^2)^{1/2}$. We want $\frac{\partial z}{\partial y}$ when

$(x, y) = (\frac{2}{3}, \frac{1}{3})$. But

$$\frac{\partial z}{\partial y} = \tfrac{1}{2}(1 - x^2 - y^2)^{-1/2}(-2y) = -\frac{y}{(1 - x^2 - y^2)^{1/2}}$$

Putting in $(x, y) = (\frac{2}{3}, \frac{1}{3})$ gives

$$\frac{\partial z}{\partial y} = -\frac{1/3}{(1 - (2/3)^2 - (1/3)^2)^{1/2}} = -\tfrac{1}{2}.$$

0.6 Functions of 3 or more variables

The general notation would be something like

$$w = f(x, y, z)$$

where x, y and z are the independent variables. For example, $w = x\sin(y + 3z)$. Partial derivatives are computed similarly to the two variable case. For example, $\partial w/\partial x$ means differentiate with respect to x holding both y and z constant and so, for this example, $\partial w/\partial x = \sin(y + 3z)$. Note that a function of three variables does not have a graph.

0.7 Second order partial derivatives

Again, let $z = f(x, y)$ be a function of x and y.

- $\dfrac{\partial^2 z}{\partial x^2}$ means the second derivative with respect to x holding y constant
- $\dfrac{\partial^2 z}{\partial y^2}$ means the second derivative with respect to y holding x constant
- $\dfrac{\partial^2 z}{\partial x \partial y}$ means differentiate first with respect to y and then with respect to x.

The "mixed" partial derivative $\dfrac{\partial^2 z}{\partial x \partial y}$ is as important in applications as the others. It is a general result that

$$\frac{\partial^2 z}{\partial x \partial y} = \frac{\partial^2 z}{\partial y \partial x}$$

i.e. you get the same answer whichever order the differentiation is done.

0.8 Example

Let $z = 4x^2 - 8xy^4 + 7y^5 - 3$. Find all the first and second order partial derivatives of z.

Solution.

$$\begin{aligned}
\frac{\partial z}{\partial x} &= 8x - 8y^4 \\
\frac{\partial z}{\partial y} &= -8x(4y^3) + 35y^4 = -32xy^3 + 35y^4 \\
\frac{\partial^2 z}{\partial x^2} &= \frac{\partial}{\partial x}\left(\frac{\partial z}{\partial x}\right) = 8 \\
\frac{\partial^2 z}{\partial y^2} &= \frac{\partial}{\partial y}\left(\frac{\partial z}{\partial y}\right) \\
&= \frac{\partial}{\partial y}(-32xy^3 + 35y^4) = -32x(3y^2) + 140y^3 \\
&= -96xy^2 + 140y^3 \\
\frac{\partial^2 z}{\partial x \partial y} &= \frac{\partial}{\partial x}\left(\frac{\partial z}{\partial y}\right) = \frac{\partial}{\partial x}(-32xy^3 + 35y^4) = -32y^3 \\
\frac{\partial^2 z}{\partial y \partial x} &= \frac{\partial}{\partial y}\left(\frac{\partial z}{\partial x}\right) = \frac{\partial}{\partial y}(8x - 8y^4) = -32y^3
\end{aligned}$$

0.9 Example

Find all the first and second order partial derivatives of the function $z = \sin xy$.
Solution.

$$\begin{aligned}
\frac{\partial z}{\partial x} &= y \cos xy \\
\frac{\partial z}{\partial y} &= x \cos xy \\
\frac{\partial^2 z}{\partial x^2} &= -y^2 \sin xy \\
\frac{\partial^2 z}{\partial y^2} &= -x^2 \sin xy \\
\frac{\partial^2 z}{\partial x \partial y} &= \frac{\partial}{\partial x}\left(\frac{\partial z}{\partial y}\right) = \frac{\partial}{\partial x}(x \cos xy) = x(-y \sin xy) + \cos xy = -xy \sin xy + \cos xy \\
\frac{\partial^2 z}{\partial y \partial x} &= \frac{\partial}{\partial y}\left(\frac{\partial z}{\partial x}\right) = \frac{\partial}{\partial y}(y \cos xy) = y(-x \sin xy) + \cos xy = -xy \sin xy + \cos xy
\end{aligned}$$

0.10 Subscript notation for second order partial derivatives

If $z = f(x, y)$ then

- z_{xx} means $\dfrac{\partial^2 z}{\partial x^2}$
- z_{yy} means $\dfrac{\partial^2 z}{\partial y^2}$

- z_{xy} means $\dfrac{\partial^2 z}{\partial x \partial y}$ or $\dfrac{\partial^2 z}{\partial y \partial x}$

0.11 Important point

Unlike ordinary derivatives, partial derivatives do **not** behave like fractions, in particular

$$\frac{\partial x}{\partial z} \neq \frac{1}{\partial z / \partial x}$$

0.12 Small changes

Let

$$z = f(x, y)$$

Imagine we change x to $x + \delta x$ and y to $y + \delta y$ with δx and δy very small. We ask: what is the corresponding change in z? The answer is that the change is δz, given by

$$\delta z \approx \frac{\partial z}{\partial x}\,\delta x + \frac{\partial z}{\partial y}\,\delta y \tag{0.1}$$

This formula requires δx and δy to be very small and even then the formula is only an approximate one. However, it becomes more and more exact as $\delta x \to 0$ and $\delta y \to 0$. This fact is sometimes expressed by saying

$$dz = \frac{\partial z}{\partial x}\,dx + \frac{\partial z}{\partial y}\,dy$$

where dx, dy and dz are infinitesimal increments.
Let's give some idea where formula (0.1) comes from. Let's recall the analogous result for a function of one variable and its derivation. For a function of one variable the notation would be $y = g(x)$ and the graph of this is a curve with a gradient dy/dx at each point x. If consider two points on this curve, (x, y) and a neighbouring point $(x + \delta x, y + \delta y)$ then if this neighbouring point is sufficiently close the line joining the two points, which has gradient $\delta y/\delta x$, is a good approximation to the tangent line at (x, y) which has gradient dy/dx. This means that $\delta y/\delta x \approx dy/dx$ so that $\delta y \approx (dy/dx)\delta x$.
We want to generalise this idea to a function $z = f(x, y)$ of two variables, whose graph will be a surface.
In the (x, y) plane let A be the point with coordinates (x, y), let B be the point with coordinates $(x + \delta x, y)$, and C the point with coordinates $(x + \delta x, y + \delta y)$.
The overall change in height, δz, from A to C is given by

$$\delta z = (\text{change in height } A \text{ to } B) + (\text{change in height } B \text{ to } C)$$

In calculating the change in height from A to B we are travelling across the surface from A to B along a curve in which y is held fixed, so by the result for curves,

$$\text{change in height } A \text{ to } B \approx \frac{\partial z}{\partial x}\,\delta x$$

Similarly

$$\text{change in height } B \text{ to } C \approx \frac{\partial z}{\partial y}\,\delta y$$

Therefore

$$\delta z \approx \frac{\partial z}{\partial x}\,\delta x + \frac{\partial z}{\partial y}\,\delta y$$

and we have derived formula (0.1).

0.13 Example

A cylindrical tank is 1 m high and 0.3 m radius. If height is increased by 5 cm and radius by 1 cm what is the effect on volume?
Solution. Let the radius be r and height be h. Then the volume V is given by

$$V = \pi r^2 h$$

so that $\frac{\partial V}{\partial r} = 2\pi r h$ and $\frac{\partial V}{\partial h} = \pi r^2$. Therefore in the notation of the present problem formula (0.1) becomes

$$\begin{aligned}\delta V &\approx \frac{\partial V}{\partial r}\,\delta r + \frac{\partial V}{\partial h}\,\delta h \\ &= 2\pi r h\,\delta r + \pi r^2 h\,\delta h\end{aligned}$$

In our case $r = 0.3$, $h = 1$, $\delta r = 1$ cm $= 0.01$ m, $\delta h = 5$ cm $= 0.05$ m so

$$\delta V \approx 2\pi(0.3)(1)(0.01) + \pi(0.3)^2(0.05) = 0.033 \ \ \text{m}^3$$

0.14 Example

The angle of elevation of the top of a tower is found to be $30^o \pm 0.5^o$ from a point 300 ± 0.1 m from the base. Estimate the towers height.
Solution. One could imagine that this sort of problem would arise when a surveyor is unable to take completely accurate readings and wants to know the likely margin of error.
Let θ be the angle of elevation, h the towers height and x the distance from tower to observer. Then

$$h = x \tan\theta$$

so that $\frac{\partial h}{\partial x} = \tan\theta$ and $\frac{\partial h}{\partial \theta} = x\sec^2\theta$. Therefore

$$\begin{aligned}\delta h &\approx \frac{\partial h}{\partial x}\,\delta x + \frac{\partial h}{\partial \theta}\,\delta\theta \\ &= \tan\theta\,\delta x + x\sec^2\theta\,\delta\theta\end{aligned}$$

Now $\theta = 30^o = \pi/6$ radians and $\delta\theta = 0.5^o = 0.008727$ radians. Also $x = 300$ m and $\delta x = 0.1$ m. Therefore

$$\delta h \approx (\tan\pi/6)(0.1) + 300(\sec^2\pi/6)(0.008727) = 3.55 \text{ m}$$

From $h = x \tan\theta$, we get $h = 173.21$ m. Our conclusion is that the height is 173.21 ± 3.55 m.
NB: If you had not converted degrees to radians your final answer would be wrong.

0.15 Absolute, relative and percentage change

- absolute change is δz
- relative change is $\frac{\delta z}{z}$
- percentage change is $\frac{\delta z}{z} \times 100$

0.16 Example on percentage change

Length and width of a rectangle are measured with errors of at most 3% and 5% respectively. Estimate the maximum percentage error in the area.
Solution. Let x = length, y = width and A = area. Then, of course, $A = xy$. So $\frac{\partial A}{\partial x} = y$ and $\frac{\partial A}{\partial y} = x$. Therefore

$$\begin{aligned} \delta A &\approx \frac{\partial A}{\partial x}\delta x + \frac{\partial A}{\partial y}\delta y \\ &= y\,\delta x + x\,\delta y \end{aligned}$$

We want percentage change in A, which is relative change multiplied by 100 so let's work out relative change first. This is given by

$$\begin{aligned} \frac{\delta A}{A} &\approx \frac{y\delta x}{A} + \frac{x\delta y}{A} \\ &= \frac{\delta x}{x} + \frac{\delta y}{y} \end{aligned}$$

since $A = xy$. What we are told is that

$$-0.03 \leq \frac{\delta x}{x} \leq 0.03 \quad \text{and} \quad -0.05 \leq \frac{\delta y}{y} \leq 0.05$$

What we need to do now is identify the worst case scenario, i.e. the maximum possible value for $\delta A/A$ given the above constraints. This happens when $\delta x/x = 0.03$ and $\delta y/y = 0.05$, giving $\delta A/A = 0.08$. This is relative error, so the (worst) percentage error is 8%.
NB: in some problems the worst case scenario is obtained by setting one of $\delta x/x$ or $\delta y/y$ to be its most negative (rather than most positive) possible value.

0.17 Chain rule for partial derivatives

Recall the chain rule for ordinary derivatives:

$$\text{if } y = f(u) \text{ and } u = g(x) \text{ then } \frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx}$$

In the above we call u the **intermediate variable** and x the **independent variable**. For partial derivatives the chain rule is more complicated. It depends on how many intermediate variables and how many independent variables are present. Below three formulae are given which it is hoped indicate the general points. Essentially, every intermediate variable has to have a term corresponding to it in the right hand side of the chain rule formula. For example in the second one below there are three intermediate variables x, y and z and three terms in the RHS.
Formula 3 below illustrates a case when there are 2 intermediate and 2 independent variables.

(1) if $z = f(x, y)$ and x and y are functions of t ($x = x(t)$ and $y = y(t)$) then z is ultimately a function of t only and

$$\frac{dz}{dt} = \frac{\partial z}{\partial x}\frac{dx}{dt} + \frac{\partial z}{\partial y}\frac{dy}{dt}$$

(2) if $w = f(x, y, z)$ and $x = x(t)$, $y = y(t)$, $z = z(t)$ then w is ultimately a function of t only and

$$\frac{dw}{dt} = \frac{\partial w}{\partial x}\frac{dx}{dt} + \frac{\partial w}{\partial y}\frac{dy}{dt} + \frac{\partial w}{\partial z}\frac{dz}{dt}$$

(3) if $z = f(x, y)$ and $x = x(u, v)$, $y = y(u, v)$ then z is a function of u and v and

$$\frac{\partial z}{\partial u} = \frac{\partial z}{\partial x}\frac{\partial x}{\partial u} + \frac{\partial z}{\partial y}\frac{\partial y}{\partial u}$$
$$\frac{\partial z}{\partial v} = \frac{\partial z}{\partial x}\frac{\partial x}{\partial v} + \frac{\partial z}{\partial y}\frac{\partial y}{\partial v}$$

0.18 Example

Let $z = x^2 y$, $x = t^2$ and $y = t^3$. Calculate dz/dt by (a) the chain rule, (b) expressing z as a function of t and finding dz/dt directly.
Solution. (a) by the chain rule

$$\begin{aligned}
\frac{dz}{dt} &= \frac{\partial z}{\partial x}\frac{dx}{dt} + \frac{\partial z}{\partial y}\frac{dy}{dt} \\
&= (2xy)(2t) + (x^2)(3t^2) \\
&= 4xyt + 3x^2t^2 \\
&= 4t^2t^3t + 3t^4t^2 \\
&= 7t^6
\end{aligned}$$

(b) $z = x^2 y$ and $x = t^2$, $y = t^3$ so $z = t^4 t^3 = t^7$. Differentiating gives $dz/dt = 7t^6$.

It might be tempting to say that approach (b) is clearly easier so why bother with the chain rule? But the fact remains that the chain rule is of fundamental importance in many applications of partial derivatives. We shall see below the use of the chain rule in studying rates of change. And the chain rule is also of importance in the derivation of the partial differential equations that govern many physical processes (eg the Navier Stokes equations of fluid dynamics); in such cases you are not simply playing around with trivial functions but dealing with *unknown* functions.

0.19 Example

Let $w = xy + z$ with $x = \cos t$, $y = \sin t$ and $z = t$. Calculate dw/dt.
Solution.

$$\begin{aligned}
\frac{dw}{dt} &= \frac{\partial w}{\partial x}\frac{dx}{dt} + \frac{\partial w}{\partial y}\frac{dy}{dt} + \frac{\partial w}{\partial z}\frac{dz}{dt} \\
&= y(-\sin t) + x(\cos t) + (1)(1) \\
&= -\sin^2 t + \cos^2 t + 1
\end{aligned}$$

0.20 Example

Let $u = x^2 - 2xy + 2y^3$ with $x = s^2 \ln t$ and $y = 2st^3$. Find $\partial u/\partial s$ and $\partial u/\partial t$.
Solution. This time u is a function of 2 variables x and y, each of which is itself a function of 2 variables s and t.

$$\begin{aligned}
\frac{\partial u}{\partial s} &= \frac{\partial u}{\partial x}\frac{\partial x}{\partial s} + \frac{\partial u}{\partial y}\frac{\partial y}{\partial s} \\
&= (2x - 2y)(2s \ln t) + (-2x + 6y^2)(2t^3) \\
&= (2s^2 \ln t - 4st^3)(2s \ln t) + (-2s^2 \ln t + 24s^2t^6)(2t^3) \\
\frac{\partial u}{\partial t} &= \frac{\partial u}{\partial x}\frac{\partial x}{\partial t} + \frac{\partial u}{\partial y}\frac{\partial y}{\partial t} \\
&= (2x - 2y)\left(\frac{s^2}{t}\right) + (-2x + 6y^2)(6st^2) \\
&= (2s^2 \ln t - 4st^3)\left(\frac{s^2}{t}\right) + (-2s^2 \ln t + 24s^2t^6)(6st^2)
\end{aligned}$$

0.21 Rates of change: an application of the chain rule

We will do some applications of the chain rule to rates of change.
Example. What rate is the area of a rectangle changing if its length is 15 m and increasing at 3 ms^{-1} while its width is 6 m and increasing at 2 ms^{-1}.
Solution. Let x be the length, y the width, A the area and t = time. The information given tells us that

$$\frac{dx}{dt} = 3 \text{ ms}^{-1}, \qquad \frac{dy}{dt} = 2 \text{ ms}^{-1}$$

Obviously $A = xy$. We want dA/dt when $x = 15$ and $y = 6$. This is given by the chain rule as follows:

$$\frac{dA}{dt} = \frac{\partial A}{\partial x}\frac{dx}{dt} + \frac{\partial A}{\partial y}\frac{dy}{dt} = y\frac{dx}{dt} + x\frac{dy}{dt} = (6)(3) + (15)(2) = 48\ \mathrm{m^2s^{-1}}.$$

Example. The height of a tree increases at a rate of 2 ft per year and the radius increases at 0.1 ft per year. What rate is the volume of timber increasing at when the height is 20 ft and the radius is 1.5 ft. (Assume the tree is a circular cylinder).
Solution. The volume V is given by $V = \pi r^2 h$. The chain rule gives

$$\begin{aligned}\frac{dV}{dt} &= \frac{\partial V}{\partial r}\frac{dr}{dt} + \frac{\partial V}{\partial h}\frac{dh}{dt}\\ &= 2\pi r h\,\frac{dr}{dt} + \pi r^2\,\frac{dh}{dt}\end{aligned}$$

We are told that $dh/dt = 2$ ft per year and $dr/dt = 0.1$ ft per year. So, when $h = 20$ and $r = 1.5$,

$$\frac{dV}{dt} = 2\pi(1.5)(20)(0.1) + \pi(1.5)^2(2) = 32.99\ \mathrm{ft^3/year}$$

0.22 The chain rule and implicit differentiation

Suppose we cannot find y explicitly as a function of x, only implicitly through the equation $F(x, y) = 0$ (for example, $F(x, y)$ might be an awkward expression such that $F(x, y) = 0$ cannot in practice be solved to give y in terms of x). We want a formula for dy/dx.
We know that $F(x, y) = 0$ defines y as a function of x, $y = y(x)$, even if we cannot in practice find the expression for y in terms of x. This means that we could write $F(x, y) = 0$ as $F(x, y(x)) = 0$. Differentiating both sides of this, using the chain rule on the left hand side, gives

$$\frac{\partial F}{\partial x}(1) + \frac{\partial F}{\partial y}\frac{dy}{dx} = 0$$

Hence

$$\frac{dy}{dx} = -\frac{\partial F/\partial x}{\partial F/\partial y}$$

As an example of the use of this formula, let us find dy/dx for the function y defined by $x^2 + xy + y^3 - 7 = 0$. Let $F(x, y) = x^2 + xy + y^3 - 7$. Then by the above formula,

$$\frac{dy}{dx} = -\frac{\partial F/\partial x}{\partial F/\partial y} = -\frac{(2x + y)}{x + 3y^2}$$

Alternatively you could deduce this result by using implicit differentiation (a technique which you should know about from previous study). It should, of course, give the same answer.
As an extension of the above idea, let the equation $f(x, y, z) = 0$ define z as a function of x and y, so that x and y are viewed as independent variables. We want

to find $\partial z/\partial x$ and $\partial z/\partial y$. The calculation here is a somewhat subtle one, in which x actually plays the role of both an intermediate variable and an independent one. Differentiating the equation $f(x, y, z) = 0$ with respect to x using the chain rule gives

$$\frac{\partial f}{\partial x}(1) + \frac{\partial f}{\partial y}\frac{\partial y}{\partial x} + \frac{\partial f}{\partial z}\frac{\partial z}{\partial x} = 0$$

Now $\partial y/\partial x$ is, in fact, zero. The reason is that y and x are independent of each other. So

$$\frac{\partial f}{\partial x} + \frac{\partial f}{\partial z}\frac{\partial z}{\partial x} = 0$$

Hence

$$\frac{\partial z}{\partial x} = -\frac{\partial f/\partial x}{\partial f/\partial z}$$

and similarly

$$\frac{\partial z}{\partial y} = -\frac{\partial f/\partial y}{\partial f/\partial z}$$

0.23 Transforming to polars

Let $u = u(x, y)$ be a function of x and y. Let

$$x = r\cos\theta, \qquad y = r\sin\theta$$

Our aim is to show that

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = \frac{\partial^2 u}{\partial r^2} + \frac{1}{r}\frac{\partial u}{\partial r} + \frac{1}{r^2}\frac{\partial^2 u}{\partial \theta^2} \tag{0.2}$$

which is the expression for the Laplacian operator in plane polar coordinates. It is useful for solving, for example, the steady state heat equation in situations with circular geometry.

By the chain rule,

$$\frac{\partial u}{\partial r} = \frac{\partial u}{\partial x}\frac{\partial x}{\partial r} + \frac{\partial u}{\partial y}\frac{\partial y}{\partial r}$$

i.e.

$$\frac{\partial u}{\partial r} = \cos\theta\frac{\partial u}{\partial x} + \sin\theta\frac{\partial u}{\partial y}$$

Differentiating the above expression with respect to r gives

$$\begin{aligned}
\frac{\partial^2 u}{\partial r^2} &= \cos\theta\frac{\partial}{\partial r}\left(\frac{\partial u}{\partial x}\right) + \sin\theta\frac{\partial}{\partial r}\left(\frac{\partial u}{\partial y}\right) \\
&= \cos\theta\left(\frac{\partial^2 u}{\partial x^2}\frac{\partial x}{\partial r} + \frac{\partial^2 u}{\partial x\partial y}\frac{\partial y}{\partial r}\right) + \sin\theta\left(\frac{\partial^2 u}{\partial x\partial y}\frac{\partial x}{\partial r} + \frac{\partial^2 u}{\partial y^2}\frac{\partial y}{\partial r}\right) \\
&= \cos^2\theta\frac{\partial^2 u}{\partial x^2} + \sin\theta\cos\theta\frac{\partial^2 u}{\partial x\partial y} + \sin\theta\cos\theta\frac{\partial^2 u}{\partial x\partial y} + \sin^2\theta\frac{\partial^2 u}{\partial y^2}.
\end{aligned}$$

Also

$$\begin{aligned}\frac{\partial u}{\partial \theta} &= \frac{\partial u}{\partial x}\frac{\partial x}{\partial \theta} + \frac{\partial u}{\partial y}\frac{\partial y}{\partial \theta} \\ &= -r\sin\theta\frac{\partial u}{\partial x} + r\cos\theta\frac{\partial u}{\partial y}\end{aligned}$$

and, after a long calculation,

$$\frac{\partial^2 u}{\partial \theta^2} = r^2\sin^2\theta\frac{\partial^2 u}{\partial x^2} + r^2\cos^2\theta\frac{\partial^2 u}{\partial y^2} - 2r^2\sin\theta\cos\theta\frac{\partial^2 u}{\partial x\partial y} \\ -r\cos\theta\frac{\partial u}{\partial x} - r\sin\theta\frac{\partial u}{\partial y}$$

It follows that

$$\frac{\partial^2 u}{\partial r^2} + \frac{1}{r}\frac{\partial u}{\partial r} + \tfrac{1}{r^2}\frac{\partial^2 u}{\partial \theta^2} = \cos^2\theta\frac{\partial^2 u}{\partial x^2} + 2\sin\theta\cos\theta\frac{\partial^2 u}{\partial x\partial y} + \sin^2\theta\frac{\partial^2 u}{\partial y^2} \\ + \frac{1}{r}\left(\cos\theta\frac{\partial u}{\partial x} + \sin\theta\frac{\partial u}{\partial y}\right) \\ + \frac{1}{r^2}\left(r^2\sin^2\theta\frac{\partial^2 u}{\partial x^2} + r^2\cos^2\theta\frac{\partial^2 u}{\partial y^2} - 2r^2\sin\theta\cos\theta\frac{\partial^2 u}{\partial x\partial y} - r\cos\theta\frac{\partial u}{\partial x} - r\sin\theta\frac{\partial u}{\partial y}\right) \\ = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$$

so that (0.2) is proved.

Permutations

Example

How many ways can four people fill four executive positions?

Solution. For the sake of concreteness, let's name the four people Tom, Rick, Harry, and Mary, and the four executive positions President, Vice President, Treasurer and Secretary. I think you'll agree that the Multiplication Principle yields a straightforward solution to this problem. If we fill the President position first, there are 4 possible people (Tom, Rick, Harry, and Mary). Let's suppose Mary is named the President. Then, since Mary can't fill more than one position at a time, when we fill the Vice President position, there are only 3 possible people (Tom, Rick, and Harry). If Tom is named the Vice President, when we fill the Treasurer position, there are only 2 possible people (Rick and Harry). Finally, if Rick is named Treasurer, when we fill the Secretary position, there is only 1 possible person (Harry). Putting all of this together, the Multiplication Principle tells us that there are:

$$4 \times 3 \times 2 \times 1$$

or 24 possible ways to fill the four positions.

Alright, alright now... enough of these kinds of examples, eh?! The main point of this example is not to see yet another application of the Multiplication Principle, but rather to introduce the counting of the number of **permutations** as a generalization of the Multiplication Principle.

A Generalization of the Multiplication Principle

Suppose there are n positions to be filled with n different objects, in which there are:

- n choices for the 1st position
- $n - 1$ choices for the 2nd position
- $n - 2$ choices for the 3rd position
- ... and ...
- 1 choice for the last position

The Multiplication Principle tells us there are then in general:

$$n \times (n-1) \times (n-2) \times \ldots \times 1 = n!$$

ways of filling the n positions. The symbol $n!$ is read as "n-factorial," and by definition 0! equals 1.

Definition. A **permutation of *n* objects** is an ordered arrangement of the n objects.

We often call such a permutation a "**permutation of *n* objects taken *n* at a time**," and denote it as ${}_nP_n$. That is:

$${}_nP_n = n \times (n-1) \times (n-2) \times \ldots \times 1 = n!$$

Not that it really matters in this situation (since they are the same), but the first subscripted n represents the number of objects you are wanting to arrange, while the second subscripted n represents the number of positions you have for the objects to fill.

Example

The draft lottery of 1969 for military service ranked all 366 days (Jan 1, Jan 2, ..., Feb 29, ..., Dec 31) of the year. The men who were eligible for service whose birthday was selected first were the first to be drafted. Those whose birthday was selected second were the second to be drafted. And so on. How many possible ways can the 366 days be ranked?

Solution. Well, we have 366 objects (days) and 366 positions (1st spot, 2nd spot, ... , 366th spot) to arrange them. Therefore, there are 366! ("366 factorial") ways of ranking the 366 possible birthdays of the eligible men.

Think About It!

What is the probability that your birthday would be ranked first?

(After you've thought of how you'd solve our problem, click on the icon to reveal one possible solution.)

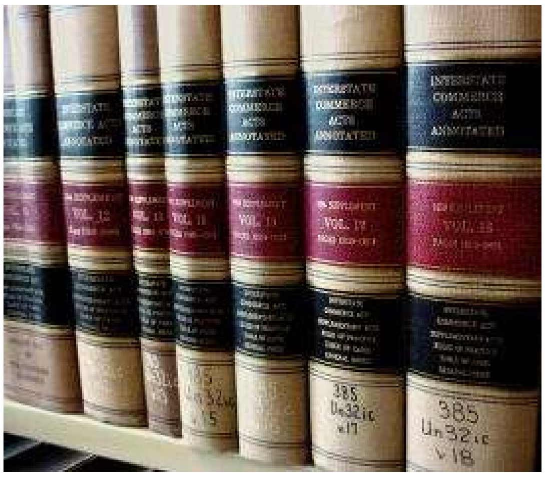
Example

In how many ways can 7 different books be arranged on a shelf?

Solution. We could use the Multiplication Principle to solve this problem. We have seven positions that we can fill with seven books. There are 7 possible books for the first position, 6 possible books for the second position, five possible books for the third position, and so on. The Multiplication Principle tells us therefore that the books can be arranged in:

$$7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1$$

or 5,040 ways. Alternatively, we can use the simple rule for counting permutations. That is, the number of ways to arrange 7 distinct objects is simply ${}_7P_7 = 7! =$ 5,040.



Example

With 6 names in a bag, randomly select a name. How many ways can the 6 names be assigned to 6 job assignments? If we assume that each person can only be assigned to one job, then we must select (or **sample**) the names without replacement. That is, once we select a name, it is set aside and not returned to the bag.

Definition. Sampling without replacement occurs when an object is not replaced after it has been selected.

Solution. If we sample without replacement, the problem reduces to simply determining the number of ways the 6 names can be arranged. We have 6 objects taken 6 at a time, and hence the number of ways is 6! = 720 possible job assignments. In this case, each person is assigned to one and only one job.

What if the 6 names were **sampled with replacement**? That is, once we select a name, it is returned to the bag.

Definition. Sampling with replacement occurs when an object is selected and then replaced before the next object has been selected.

Solution. If we sample with replacement, we have 6 choices for each of the 6 jobs. Applying the Multiplication Principle, there are:

$$6 \times 6 \times 6 \times 6 \times 6 \times 6 = 46{,}656$$

possible job assignments. In this case, each person is allowed to perform more than one job. There's even the possibility that one (rather unlucky) person gets assigned to all six jobs!

The take-home message from this example is that you'll always want to ask yourself whether or not the problem involves sampling with or without replacement. Incidentally, it's not all that different from asking yourself whether or not replication is allowed. Right?

Example

Okay, let's throw a (small) wrench into our work. How many ways can 4 people fill 3 chairs?

> **Solution.** Again, for the sake of concreteness, let's name the four people Tom, Rick, Harry, and Mary and the chairs Left, Middle, and Right. If we fill the Left chair first, there are 4 possible people (Tom, Rick, Harry, and Mary). Let's suppose Tom is selected for the Left chair. Then, since Tom can't sit in more than one chair at a time, when we fill the Middle chair, there are only 3 possible people (Rick, Harry, and Mary). If Rick is selected for the Middle chair, when we fill the Right chair, there are only 2 possible people (Harry and Mary). Putting all of this together, the Multiplication Principle tells us that there are:
>
> $$4 \times 3 \times 2$$
>
> or 24 possible ways to fill the three chairs.

Okay, okay! The main distinction between this example and the first example on this page is that the first example involves arranging all 4 people, whereas this example involves leaving one person out and arranging just 3 of the 4 people. This example allows us to introduce another generalization of the Multiplication Principle, namely the counting of the number of **permutations of *n* objects taken *r* at a time**, where $r \leq n$.

Another Generalization of the Multiplication Principle

Suppose there are r positions to be filled with n different objects, in which there are:

- n choices for the 1st position
- $n - 1$ choices for the 2nd position
- $n - 2$ choices for the 3rd position
- ... and ...
- $n - (r - 1)$ choices for the last position

The Multiplication Principle tells us there are in general:

$$n \times (n - 1) \times (n - 2) \times \ldots \times [n - (r - 1)]$$

ways of filling the r positions. We can easily show that, in general, this quantity equals:

$$\frac{n!}{(n-r)!}$$

Here's how it works:

And, formally:

Definition. A **permutation of *n* objects taken *r* at a time** is an ordered arrangement of n different objects in r positions. The number of such permutations is:

$$_nP_r = \frac{n!}{(n-r)!}$$

The subscripted n represents the number of objects you are wanting to arrange, while the subscripted r represents the number of positions you have for the objects to fill.

Example

An artist has 9 paintings. How many ways can he hang 4 paintings side-by-side on a gallery wall?

A TUTORIAL ON POINTERS AND ARRAYS IN C

by Ted Jensen

Version 1.2 (PDF Version)
Sept. 2003

TABLE OF CONTENTS

PREFACE

This document is intended to introduce pointers to beginning programmers in the C programming language. Over several years of reading and contributing to various conferences on C including those on the FidoNet and UseNet, I have noted a large number of newcomers to C appear to have a difficult time in grasping the fundamentals of pointers. I therefore undertook the task of trying to explain them in plain language with lots of examples.

The first version of this document was placed in the public domain, as is this one. It was picked up by Bob Stout who included it as a file called PTR-HELP.TXT in his widely distributed collection of SNIPPETS. Since that original 1995 release, I have added a significant amount of material and made some minor corrections in the original work.

I subsequently posted an HTML version around 1998 on my website at:

http://pweb.netcom.com/~tjensen/ptr/cpoint.htm

After numerous requests, I've finally come out with this PDF version which is identical to that HTML version cited above, and which can be obtained from that same web site.

Acknowledgements:

There are so many people who have unknowingly contributed to this work because of the questions they have posed in the FidoNet C Echo, or the UseNet Newsgroup comp.lang.c, or several other conferences in other networks, that it would be impossible to list them all. Special thanks go to Bob Stout who was kind enough to include the first version of this material in his SNIPPETS file.

About the Author:

Ted Jensen is a retired Electronics Engineer who worked as a hardware designer or manager of hardware designers in the field of magnetic recording. Programming has been a hobby of his off and on since 1968 when he learned how to keypunch cards for submission to be run on a mainframe. (The mainframe had 64K of magnetic core memory!).

Use of this Material:

note at one of the addresses below informing me of this. I have written this with the hope that it will be useful to others and since I'm not asking any financial remuneration, the only way I know that I have at least partially reached that goal is via feedback from those who find this material useful.

By the way, you needn't be an instructor or teacher to contact me. I would appreciate a note from <u>anyone</u> who finds the material useful, or who has constructive criticism to offer. I'm also willing to answer questions submitted by email at the addresses shown below.

Ted Jensen
Redwood City, California
tjensen@ix.netcom.com
July 1998

INTRODUCTION

If you want to be proficient in the writing of code in the C programming language, you must have a thorough working knowledge of how to use pointers. Unfortunately, C pointers appear to represent a stumbling block to newcomers, particularly those coming from other computer languages such as Fortran, Pascal or Basic.

To aid those newcomers in the understanding of pointers I have written the following material. To get the maximum benefit from this material, I feel it is important that the user be able to run the code in the various listings contained in the article. I have attempted, therefore, to keep all code ANSI compliant so that it will work with any ANSI compliant compiler. I have also tried to carefully block the code within the text. That way, with the help of an ASCII text editor, you can copy a given block of code to a new file and compile it on your system. I recommend that readers do this as it will help in understanding the material.

CHAPTER 1: What is a pointer?

One of those things beginners in C find difficult is the concept of pointers. The purpose of this tutorial is to provide an introduction to pointers and their use to these beginners.

I have found that often the main reason beginners have a problem with pointers is that they have a weak or minimal feeling for variables, (as they are used in C). Thus we start with a discussion of C variables in general.

A variable in a program is something with a name, the value of which can vary. The way the compiler and linker handles this is that it assigns a specific block of memory within the computer to hold the value of that variable. The size of that block depends on the range over which the variable is allowed to vary. For example, on PC's the size of an integer variable is 2 bytes, and that of a long integer is 4 bytes. In C the size of a variable type such as an integer need not be the same on all types of machines.

When we declare a variable we inform the compiler of two things, the name of the variable and the type of the variable. For example, we declare a variable of type integer with the name **k** by writing:

```
    int k;
```

On seeing the "int" part of this statement the compiler sets aside 2 bytes of memory (on a PC) to hold the value of the integer. It also sets up a symbol table. In that table it adds the symbol **k** and the relative address in memory where those 2 bytes were set aside.

Thus, later if we write:

```
    k = 2;
```

we expect that, at run time when this statement is executed, the value 2 will be placed in that memory location reserved for the storage of the value of **k**. In C we refer to a variable such as the integer **k** as an "object".

In a sense there are two "values" associated with the object **k**. One is the value of the integer stored there (2 in the above example) and the other the "value" of the memory location, i.e., the address of **k**. Some texts refer to these two values with the nomenclature ***rvalue*** (right value, pronounced "are value") and ***lvalue*** (left value, pronounced "el value") respectively.

In some languages, the lvalue is the value permitted on the left side of the assignment operator '=' (i.e. the address where the result of evaluation of the right side ends up). The rvalue is that which is on the right side of the assignment statement, the **2** above. Rvalues cannot be used on the left side of the assignment statement. Thus: **2 = k**; is illegal.

Actually, the above definition of "lvalue" is somewhat modified for C. According to K&R II (page 197): [1]

> "An ***object*** is a named region of storage; an ***lvalue*** is an expression referring to an object."

However, at this point, the definition originally cited above is sufficient. As we become more familiar with pointers we will go into more detail on this.

Okay, now consider:

```
int j, k;

 k = 2;
 j = 7;    <-- line 1
 k = j;    <-- line 2
```

In the above, the compiler interprets the **j** in line 1 as the address of the variable **j** (its lvalue) and creates code to copy the value 7 to that address. In line 2, however, the **j** is interpreted as its rvalue (since it is on the right hand side of the assignment operator '='). That is, here the **j** refers to the value ***stored*** at the memory location set aside for **j**, in this case 7. So, the 7 is copied to the address designated by the lvalue of **k**.

In all of these examples, we are using 2 byte integers so all copying of rvalues from one storage location to the other is done by copying 2 bytes. Had we been using long integers, we would be copying 4 bytes.

Now, let's say that we have a reason for wanting a variable designed to hold an lvalue (an address). The size required to hold such a value depends on the system. On older desk top computers with 64K of memory total, the address of any point in memory can be contained in 2 bytes. Computers with more memory would require more bytes to hold an address. Some computers, such as the IBM PC might require special handling to hold a segment and offset under certain circumstances. The actual size required is not too important so long as we have a way of informing the compiler that what we want to store is an address.

Such a variable is called a ***pointer variable*** (for reasons which hopefully will become clearer a little later). In C when we define a pointer variable we do so by preceding its name with an asterisk. In C we also give our pointer a type which, in this case, refers to the type of data stored at the address we will be storing in our pointer. For example, consider the variable declaration:

```
int *ptr;
```

ptr is the name of our variable (just as **k** was the name of our integer variable). The '*' informs the compiler that we want a pointer variable, i.e. to set aside however many bytes is required to store an address in memory. The **int** says that we intend to use our pointer variable to store the address of an integer. Such a pointer is said to "point to" an integer. However, note that when we wrote **int k;** we did not give **k** a value. If this definition is made outside of any function ANSI compliant compilers will initialize it to zero. Similarly, **ptr** has no value, that is we haven't stored an address in it in the above declaration. In this case, again if the declaration is outside of any function, it is initialized to a value guaranteed in such a way that it is guaranteed to not point to any C object or function. A pointer initialized in this manner is called a "null" pointer.

The actual bit pattern used for a null pointer may or may not evaluate to zero since it depends on the specific system on which the code is developed. To make the source code compatible between various compilers on various systems, a macro is used to represent a null pointer. That macro goes under the name NULL. Thus, setting the value of a pointer using the NULL macro, as with an assignment statement such as ptr = NULL, guarantees that the pointer has become a null pointer. Similarly, just as one can test for an integer value of zero, as in **if(k == 0)**, we can test for a null pointer using **if (ptr == NULL)**.

But, back to using our new variable **ptr**. Suppose now that we want to store in **ptr** the address of our integer variable **k**. To do this we use the unary **&** operator and write:

```
    ptr = &k;
```

What the **&** operator does is retrieve the lvalue (address) of **k**, even though **k** is on the right hand side of the assignment operator '=', and copies that to the contents of our pointer ptr. Now, ptr is said to "point to" **k**. Bear with us now, there is only one more operator we need to discuss.

The "dereferencing operator" is the asterisk and it is used as follows:

```
    *ptr = 7;
```

will copy 7 to the address pointed to by **ptr**. Thus if **ptr** "points to" (contains the address of) **k**, the above statement will set the value of **k** to 7. That is, when we use the '*' this way we are referring to the value of that which ptr is pointing to, not the value of the pointer itself.

Similarly, we could write:

```
 printf("%d\n",*ptr);
```

to print to the screen the integer value stored at the address pointed to by **ptr**;.

One way to see how all this stuff fits together would be to run the following program and then review the code and the output carefully.

```
------------ Program 1.1 ---------------------------------

/* Program 1.1 from PTRTUT10.TXT   6/10/97 */
```

```
#include <stdio.h>

int j, k;
int *ptr;

int main(void)
{
    j = 1;
    k = 2;
    ptr = &k;
    printf("\n");
    printf("j has the value %d and is stored at %p\n", j, (void *)&j);
    printf("k has the value %d and is stored at %p\n", k, (void *)&k);
    printf("ptr has the value %p and is stored at %p\n", ptr, (void
*)&ptr);
    printf("The value of the integer pointed to by ptr is %d\n", *ptr);

    return 0;
}
```

Note: We have yet to discuss those aspects of C which require the use of the **(void *)** expression used here. For now, include it in your test code. We'll explain the reason behind this expression later.

To review:

- A variable is declared by giving it a type and a name (e.g. **int k;**)
- A pointer variable is declared by giving it a type and a name (e.g. **int *ptr**) where the asterisk tells the compiler that the variable named **ptr** is a pointer variable and the type tells the compiler what type the pointer is to point to (integer in this case).
- Once a variable is declared, we can get its address by preceding its name with the unary **&** operator, as in **&k**.
- We can "dereference" a pointer, i.e. refer to the value of that which it points to, by using the unary '*' operator as in ***ptr**.
- An "lvalue" of a variable is the value of its address, i.e. where it is stored in memory. The "rvalue" of a variable is the value stored in that variable (at that address).

References for Chapter 1:

1. "The C Programming Language" 2nd Edition
   B. Kernighan and D. Ritchie
   Prentice Hall
   ISBN 0-13-110362-8

CHAPTER 2: Pointer types and Arrays

Okay, let's move on. Let us consider why we need to identify the ***type*** of variable that a pointer points to, as in:

```
     int *ptr;
```

One reason for doing this is so that later, once ptr "points to" something, if we write:

```
    *ptr = 2;
```

the compiler will know how many bytes to copy into that memory location pointed to by **ptr**. If **ptr** was declared as pointing to an integer, 2 bytes would be copied, if a long, 4 bytes would be copied. Similarly for floats and doubles the appropriate number will be copied. But, defining the type that the pointer points to permits a number of other interesting ways a compiler can interpret code. For example, consider a block in memory consisting if ten integers in a row. That is, 20 bytes of memory are set aside to hold 10 integers.

Now, let's say we point our integer pointer **ptr** at the first of these integers. Furthermore lets say that integer is located at memory location 100 (decimal). What happens when we write:

```
    ptr + 1;
```

Because the compiler "knows" this is a pointer (i.e. its value is an address) and that it points to an integer (its current address, 100, is the address of an integer), it adds 2 to **ptr** instead of 1, so the pointer "points to" the **next integer**, at memory location 102. Similarly, were the **ptr** declared as a pointer to a long, it would add 4 to it instead of 1. The same goes for other data types such as floats, doubles, or even user defined data types such as structures. This is obviously not the same kind of "addition" that we normally think of. In C it is referred to as addition using "pointer arithmetic", a term which we will come back to later.

Similarly, since **++ptr** and **ptr++** are both equivalent to **ptr + 1** (though the point in the program when **ptr** is incremented may be different), incrementing a pointer using the unary ++ operator, either pre- or post-, increments the address it stores by the amount sizeof(type) where "type" is the type of the object pointed to. (i.e. 2 for an integer, 4 for a long, etc.).

Since a block of 10 integers located contiguously in memory is, by definition, an array of integers, this brings up an interesting relationship between arrays and pointers.

Consider the following:

```
    int my_array[] = {1,23,17,4,-5,100};
```

Here we have an array containing 6 integers. We refer to each of these integers by means of a subscript to **my_array**, i.e. using **my_array[0]** through **my_array[5]**. But, we could alternatively access them via a pointer as follows:

```
    int *ptr;
    ptr = &my_array[0];       /* point our pointer at the first
                                 integer in our array */
```

And then we could print out our array either using the array notation or by dereferencing our pointer. The following code illustrates this:

```
-----------  Program 2.1  -----------------------------------

/* Program 2.1 from PTRTUT10.HTM   6/13/97 */

#include <stdio.h>

int my_array[] = {1,23,17,4,-5,100};
int *ptr;

int main(void)
{
    int i;
    ptr = &my_array[0];     /* point our pointer to the first
                                        element of the array */
    printf("\n\n");
    for (i = 0; i < 6; i++)
    {
      printf("my_array[%d] = %d   ",i,my_array[i]);   /*<-- A */
      printf("ptr + %d = %d\n",i, *(ptr + i));        /*<-- B */
    }
    return 0;
}
```

Compile and run the above program and carefully note lines A and B and that the program prints out the same values in either case. Also observe how we dereferenced our pointer in line B, i.e. we first added i to it and then dereferenced the new pointer. Change line B to read:

```
    printf("ptr + %d = %d\n",i, *ptr++);
```

and run it again... then change it to:

```
    printf("ptr + %d = %d\n",i, *(++ptr));
```

and try once more. Each time try and predict the outcome and carefully look at the actual outcome.

In C, the standard states that wherever we might use **&var_name[0]** we can replace that with **var_name**, thus in our code where we wrote:

```
    ptr = &my_array[0];
```

we can write:

```
    ptr = my_array;
```

to achieve the same result.

This leads many texts to state that the name of an array is a pointer. I prefer to mentally think "the name of the array is the address of first element in the array". Many beginners (including myself when I was learning) have a tendency to become confused by thinking of it as a pointer. For example, while we can write

```
    ptr = my_array;
```

we cannot write

```
    my_array = ptr;
```

The reason is that while **ptr** is a variable, **my_array** is a constant. That is, the location at which the first element of **my_array** will be stored cannot be changed once **my_array[]** has been declared.

Earlier when discussing the term "lvalue" I cited K&R-2 where it stated:

> "An **object** is a named region of storage; an **lvalue** is an expression referring to an object".

This raises an interesting problem. Since **my_array** is a named region of storage, why is **my_array** in the above assignment statement not an lvalue? To resolve this problem, some refer to **my_array** as an "unmodifiable lvalue".

Modify the example program above by changing

```
    ptr = &my_array[0];
```

to

```
    ptr = my_array;
```

and run it again to verify the results are identical.

Now, let's delve a little further into the difference between the names **ptr** and **my_array** as used above. Some writers will refer to an array's name as a ***constant*** pointer. What do we mean by that? Well, to understand the term "constant" in this sense, let's go back to our definition of the term "variable". When we declare a variable we set aside a spot in memory to hold the value of the appropriate type. Once that is done the name of the variable can be interpreted in one of two ways. When used on the left side of the assignment operator, the compiler interprets it as the memory location to which to move that value resulting from evaluation of the right side of the assignment operator. But, when used on the right side of the assignment operator, the name of a variable is interpreted to mean the contents stored at that memory address set aside to hold the value of that variable.

With that in mind, let's now consider the simplest of constants, as in:

```
    int i, k;
    i = 2;
```

Here, while **i** is a variable and then occupies space in the data portion of memory, **2** is a constant and, as such, instead of setting aside memory in the data segment, it is imbedded directly in the code segment of memory. That is, while writing something like **k = i;** tells the compiler to create code which at run time will look at memory location **&i** to determine the value to be moved to **k**, code created by **i = 2;** simply puts the **2** in the code and there is no referencing of the data segment. That is, both **k** and **i** are objects, but **2** is not an object.

Similarly, in the above, since **my_array** is a constant, once the compiler establishes where the array itself is to be stored, it "knows" the address of **my_array[0]** and on seeing:

```
    ptr = my_array;
```

it simply uses this address as a constant in the code segment and there is no referencing of the data segment beyond that.

This might be a good place explain further the use of the **(void *)** expression used in Program 1.1 of Chapter 1. As we have seen we can have pointers of various types. So far we have discussed pointers to integers and pointers to characters. In coming chapters we will be learning about pointers to structures and even pointer to pointers.

Also we have learned that on different systems the size of a pointer can vary. As it turns out it is also possible that the size of a pointer can vary depending on the data type of the object to which it points. Thus, as with integers where you can run into trouble attempting to assign a long integer to a variable of type short integer, you can run into trouble attempting to assign the values of pointers of various types to pointer variables of other types.

To minimize this problem, C provides for a pointer of type void. We can declare such a pointer by writing:

```
void *vptr;
```

A void pointer is sort of a generic pointer. For example, while C will not permit the comparison of a pointer to type integer with a pointer to type character, for example, either of these can be compared to a void pointer. Of course, as with other variables, casts can be used to convert from one type of pointer to another under the proper circumstances. In Program 1.1. of Chapter 1 I cast the pointers to integers into void pointers to make them compatible with the %p conversion specification. In later chapters other casts will be made for reasons defined therein.

Well, that's a lot of technical stuff to digest and I don't expect a beginner to understand all of it on first reading. With time and experimentation you will want to come back and re-read the first 2 chapters. But for now, let's move on to the relationship between pointers, character arrays, and strings.

CHAPTER 3: Pointers and Strings

The study of strings is useful to further tie in the relationship between pointers and arrays. It also makes it easy to illustrate how some of the standard C string functions can be implemented. Finally it illustrates how and when pointers can and should be passed to functions.

In C, strings are arrays of characters. This is not necessarily true in other languages. In BASIC, Pascal, Fortran and various other languages, a string has its own data type. But in C it does not. In C a string is an array of characters terminated with a binary zero character (written as **'\0'**). To start off our discussion we will write some code which, while preferred for illustrative purposes, you would probably never write in an actual program. Consider, for example:

```
    char my_string[40];

    my_string[0] = 'T';
    my_string[1] = 'e';
    my_string[2] = 'd':
    my_string[3] = '\0';
```

While one would never build a string like this, the end result is a string in that it is an array of characters **terminated with a nul character**. By definition, in C, a string is an array of characters terminated with the nul character. Be aware that "nul" is **not** the same as "NULL". The nul refers to a zero as defined by the escape sequence **'\0'**. That is it occupies one byte of memory. NULL, on the other hand, is the name of the macro used to initialize null pointers. NULL is #defined in a header file in your C compiler, nul may not be #defined at all.

Since writing the above code would be very time consuming, C permits two alternate ways of achieving the same thing. First, one might write:

```
    char my_string[40] = {'T', 'e', 'd', '\0',};
```

But this also takes more typing than is convenient. So, C permits:

```
    char my_string[40] = "Ted";
```

When the double quotes are used, instead of the single quotes as was done in the previous examples, the nul character (**'\0'**) is automatically appended to the end of the string.

In all of the above cases, the same thing happens. The compiler sets aside an contiguous block of memory 40 bytes long to hold characters and initialized it such that the first 4 characters are **Ted\0**.

Now, consider the following program:

```
------------------program 3.1-------------------------------------

/* Program 3.1 from PTRTUT10.HTM   6/13/97 */

#include <stdio.h>

char strA[80] = "A string to be used for demonstration purposes";
char strB[80];

int main(void)
{

    char *pA;      /* a pointer to type character */
    char *pB;      /* another pointer to type character */
    puts(strA);    /* show string A */
    pA = strA;     /* point pA at string A */
    puts(pA);      /* show what pA is pointing to */
    pB = strB;     /* point pB at string B */
    putchar('\n');        /* move down one line on the screen */
    while(*pA != '\0')    /* line A (see text) */
    {
        *pB++ = *pA++;    /* line B (see text) */
    }
    *pB = '\0';           /* line C (see text) */
    puts(strB);           /* show strB on screen */
    return 0;
}

--------- end program 3.1 -------------------------------------
```

In the above we start out by defining two character arrays of 80 characters each. Since these are globally defined, they are initialized to all **'\0'**s first. Then, **strA** has the first 42 characters initialized to the string in quotes.

Now, moving into the code, we declare two character pointers and show the string on the screen. We then "point" the pointer **pA** at **strA**. That is, by means of the assignment statement we copy the address of **strA[0]** into our variable **pA**. We now use **puts()** to show that which is pointed to by **pA** on the screen. Consider here that the function prototype for **puts()** is:

```
    int puts(const char *s);
```

For the moment, ignore the **const**. The parameter passed to **puts()** is a pointer, that is the **value** of a pointer (since all parameters in C are passed by value), and the value of a pointer is the address to which it points, or, simply, an address. Thus when we write **puts(strA);** as we have seen, we are passing the address of **strA[0]**.

Similarly, when we write **puts(pA);** we are passing the same address, since we have set **pA = strA;**

Given that, follow the code down to the **while()** statement on line A. Line A states:

While the character pointed to by **pA** (i.e. ***pA**) is not a nul character (i.e. the terminating **'\0'**), do the following:

Line B states: copy the character pointed to by **pA** to the space pointed to by **pB**, then increment **pA** so it points to the next character and **pB** so it points to the next space.

When we have copied the last character, **pA** now points to the terminating nul character and the loop ends. However, we have not copied the nul character. And, by definition a string in C **must** be nul terminated. So, we add the nul character with line C.

It is very educational to run this program with your debugger while watching **strA**, **strB**, **pA** and **pB** and single stepping through the program. It is even more educational if instead of simply defining **strB[]** as has been done above, initialize it also with something like:

```
    strB[80] = "12345678901234567890123456789012345678901234567890"
```

where the number of digits used is greater than the length of **strA** and then repeat the single stepping procedure while watching the above variables. Give these things a try!

Getting back to the prototype for **puts()** for a moment, the "const" used as a parameter modifier informs the user that the function will not modify the string pointed to by **s**, i.e. it will treat that string as a constant.

Of course, what the above program illustrates is a simple way of copying a string. After playing with the above until you have a good understanding of what is happening, we can proceed to creating our own replacement for the standard **strcpy()** that comes with C. It might look like:

```
   char *my_strcpy(char *destination, char *source)
   {
       char *p = destination;
       while (*source != '\0')
       {
           *p++ = *source++;
       }
       *p = '\0';
       return destination;
   }
```

In this case, I have followed the practice used in the standard routine of returning a pointer to the destination.

Again, the function is designed to accept the values of two character pointers, i.e. addresses, and thus in the previous program we could write:

```
    int main(void)
    {
        my_strcpy(strB, strA);
        puts(strB);
    }
```

I have deviated slightly from the form used in standard C which would have the prototype:

```
    char *my_strcpy(char *destination, const char *source);
```

Here the "const" modifier is used to assure the user that the function will not modify the contents pointed to by the source pointer. You can prove this by modifying the function above, and its prototype, to include the "const" modifier as shown. Then, within the function you can add a statement which attempts to change the contents of that which is pointed to by source, such as:

```
    *source = 'X';
```

which would normally change the first character of the string to an X. The const modifier should cause your compiler to catch this as an error. Try it and see.

Now, let's consider some of the things the above examples have shown us. First off, consider the fact that ***ptr++** is to be interpreted as returning the value pointed to by **ptr** and then incrementing the pointer value. This has to do with the precedence of the operators. Were we to write **(*ptr)++** we would increment, not the pointer, but that which the pointer points to! i.e. if used on the first character of the above example string the 'T' would be incremented to a 'U'. You can write some simple example code to illustrate this.

Recall again that a string is nothing more than an array of characters, with the last character being a **'\0'**. What we have done above is deal with copying an array. It happens to be an array of characters but the technique could be applied to an array of integers, doubles, etc. In those cases, however, we would not be dealing with strings and hence the end of the array would not be marked with a special value like the nul character. We could implement a version that relied on a special value to identify the end. For example, we could copy an array of positive integers by marking the end with a negative integer. On the other hand, it is more usual that when we write a function to copy an array of items other than strings we pass the function the number of items to be copied as well as the address of the array, e.g. something like the following prototype might indicate:

```
    void int_copy(int *ptrA, int *ptrB, int nbr);
```

where **nbr** is the number of integers to be copied. You might want to play with this idea and create an array of integers and see if you can write the function **int_copy()** and make it work.

This permits using functions to manipulate large arrays. For example, if we have an array of 5000 integers that we want to manipulate with a function, we need only pass to that function the address of the array (and any auxiliary information such as nbr above, depending on what we are doing). The array itself does **not** get passed, i.e. the whole array is not copied and put on the stack before calling the function, only its address is sent.

This is different from passing, say an integer, to a function. When we pass an integer we make a copy of the integer, i.e. get its value and put it on the stack. Within the function any manipulation of the value passed can in no way effect the original integer. But, with arrays and pointers we can pass the address of the variable and hence manipulate the values of the original variables.

CHAPTER 4: More on Strings

Well, we have progressed quite a way in a short time! Let's back up a little and look at what was done in Chapter 3 on copying of strings but in a different light. Consider the following function:

```
char *my_strcpy(char dest[], char source[])
{
    int i = 0;
    while (source[i] != '\0')
    {
        dest[i] = source[i];
        i++;
    }
    dest[i] = '\0';
    return dest;
}
```

Recall that strings are arrays of characters. Here we have chosen to use array notation instead of pointer notation to do the actual copying. The results are the same, i.e. the string gets copied using this notation just as accurately as it did before. This raises some interesting points which we will discuss.

Since parameters are passed by value, in both the passing of a character pointer or the name of the array as above, what actually gets passed is the address of the first element of each array. Thus, the numerical value of the parameter passed is the same whether we use a character pointer or an array name as a parameter. This would tend to imply that somehow **source[i]** is the same as ***(p+i)**.

In fact, this is true, i.e wherever one writes **a[i]** it can be replaced with ***(a + i)** without any problems. In fact, the compiler will create the same code in either case. Thus we see that pointer arithmetic is the same thing as array indexing. Either syntax produces the same result.

This is NOT saying that pointers and arrays are the same thing, they are not. We are only saying that to identify a given element of an array we have the choice of two syntaxes, one using array indexing and the other using pointer arithmetic, which yield identical results.

Now, looking at this last expression, part of it.. **(a + i)**, is a simple addition using the **+** operator and the rules of C state that such an expression is commutative. That is **(a + i)** is identical to **(i + a)**. Thus we could write ***(i + a)** just as easily as ***(a + i)**.

But ***(i + a)** could have come from **i[a]** ! From all of this comes the curious truth that if:

```
    char a[20];
    int i;
```

writing

```
    a[3] = 'x';
```

is the same as writing

```
    3[a] = 'x';
```

Try it! Set up an array of characters, integers or longs, etc. and assigned the 3rd or 4th element a value using the conventional approach and then print out that value to be sure you have that working. Then reverse the array notation as I have done above. A good compiler will not balk and the results will be identical. A curiosity... nothing more!

Now, looking at our function above, when we write:

```
    dest[i] = source[i];
```

due to the fact that array indexing and pointer arithmetic yield identical results, we can write this as:

```
    *(dest + i) = *(source + i);
```

But, this takes 2 additions for each value taken on by i. Additions, generally speaking, take more time than incrementations (such as those done using the **++** operator as in **i++**). This may not be true in modern optimizing compilers, but one can never be sure. Thus, the pointer version may be a bit faster than the array version.

Another way to speed up the pointer version would be to change:

```
    while (*source != '\0')
```

to simply

```
    while (*source)
```

since the value within the parenthesis will go to zero (FALSE) at the same time in either case.

At this point you might want to experiment a bit with writing some of your own programs using pointers. Manipulating strings is a good place to experiment. You might want to write your own versions of such standard functions as:

```
    strlen();
    strcat();
    strchr();
```

and any others you might have on your system.

We will come back to strings and their manipulation through pointers in a future chapter. For now, let's move on and discuss structures for a bit.

CHAPTER 5: Pointers and Structures

As you may know, we can declare the form of a block of data containing different data types by means of a structure declaration. For example, a personnel file might contain structures which look something like:

```
    struct tag {
        char lname[20];        /* last name */
        char fname[20];        /* first name */
        int age;               /* age */
        float rate;            /* e.g. 12.75 per hour */
    };
```

Let's say we have a bunch of these structures in a disk file and we want to read each one out and print out the first and last name of each one so that we can have a list of the people in our files. The remaining information will not be printed out. We will want to do this printing with a function call and pass to that function a pointer to the structure at hand. For demonstration purposes I will use only one structure for now. But realize the goal is the writing of the function, not the reading of the file which, presumably, we know how to do.

For review, recall that we can access structure members with the dot operator as in:

```
--------------- program 5.1 ------------------

/* Program 5.1 from PTRTUT10.HTM     6/13/97 */


#include <stdio.h>
#include <string.h>

struct tag {
    char lname[20];      /* last name */
    char fname[20];      /* first name */
    int age;             /* age */
    float rate;          /* e.g. 12.75 per hour */
};

struct tag my_struct;       /* declare the structure my_struct */

int main(void)
{
    strcpy(my_struct.lname,"Jensen");
    strcpy(my_struct.fname,"Ted");
    printf("\n%s ",my_struct.fname);
    printf("%s\n",my_struct.lname);
    return 0;
}

-------------- end of program 5.1 --------------
```

Now, this particular structure is rather small compared to many used in C programs. To the above we might want to add:

```
date_of_hire;                  (data types not shown)
date_of_last_raise;
last_percent_increase;
emergency_phone;
medical_plan;
Social_S_Nbr;
etc.....
```

If we have a large number of employees, what we want to do is manipulate the data in these structures by means of functions. For example we might want a function print out the name of the employee listed in any structure passed to it. However, in the original C (Kernighan & Ritchie, 1st Edition) it was not possible to pass a structure, only a pointer to a structure could be passed. In ANSI C, it is now permissible to pass the complete structure. But, since our goal here is to learn more about pointers, we won't pursue that.

Anyway, if we pass the whole structure it means that we must copy the contents of the structure from the calling function to the called function. In systems using stacks, this is done by pushing the contents of the structure on the stack. With large structures this could prove to be a problem. However, passing a pointer uses a minimum amount of stack space.

In any case, since this is a discussion of pointers, we will discuss how we go about passing a pointer to a structure and then using it within the function.

Consider the case described, i.e. we want a function that will accept as a parameter a pointer to a structure and from within that function we want to access members of the structure. For example we want to print out the name of the employee in our example structure.

Okay, so we know that our pointer is going to point to a structure declared using struct tag. We declare such a pointer with the declaration:

```
struct tag *st_ptr;
```

and we point it to our example structure with:

```
st_ptr = &my_struct;
```

Now, we can access a given member by de-referencing the pointer. But, how do we de-reference the pointer to a structure? Well, consider the fact that we might want to use the pointer to set the age of the employee. We would write:

```
(*st_ptr).age = 63;
```

Look at this carefully. It says, replace that within the parenthesis with that which **st_ptr** points to, which is the structure **my_struct**. Thus, this breaks down to the same as **my_struct.age**.

However, this is a fairly often used expression and the designers of C have created an alternate syntax with the same meaning which is:

```
    st_ptr->age = 63;
```

With that in mind, look at the following program:

```
------------ program 5.2 ---------------------

/* Program 5.2 from PTRTUT10.HTM   6/13/97 */

#include <stdio.h>
#include <string.h>

struct tag{                     /* the structure type */
    char lname[20];             /* last name */
    char fname[20];             /* first name */
    int age;                    /* age */
    float rate;                 /* e.g. 12.75 per hour */
};

struct tag my_struct;           /* define the structure */
void show_name(struct tag *p);  /* function prototype */

int main(void)
{
    struct tag *st_ptr;         /* a pointer to a structure */
    st_ptr = &my_struct;        /* point the pointer to my_struct */
    strcpy(my_struct.lname,"Jensen");
    strcpy(my_struct.fname,"Ted");
    printf("\n%s ",my_struct.fname);
    printf("%s\n",my_struct.lname);
    my_struct.age = 63;
    show_name(st_ptr);          /* pass the pointer */
    return 0;
}

void show_name(struct tag *p)
{
    printf("\n%s ", p->fname);  /* p points to a structure */
    printf("%s ", p->lname);
    printf("%d\n", p->age);
}

-------------------- end of program 5.2 ----------------
```

Again, this is a lot of information to absorb at one time. The reader should compile and run the various code snippets and using a debugger monitor things like **my_struct** and **p** while single stepping through the main and following the code down into the function to see what is happening.

CHAPTER 6: Some more on Strings, and Arrays of Strings

Well, let's go back to strings for a bit. In the following all assignments are to be understood as being global, i.e. made outside of any function, including main().

We pointed out in an earlier chapter that we could write:

```
   char my_string[40] = "Ted";
```

which would allocate space for a 40 byte array and put the string in the first 4 bytes (three for the characters in the quotes and a 4th to handle the terminating **'\0'**).

Actually, if all we wanted to do was store the name "Ted" we could write:

```
   char my_name[] = "Ted";
```

and the compiler would count the characters, leave room for the nul character and store the total of the four characters in memory the location of which would be returned by the array name, in this case **my_name**.

In some code, instead of the above, you might see:

```
   char *my_name = "Ted";
```

which is an alternate approach. Is there a difference between these? The answer is.. yes. Using the array notation 4 bytes of storage in the static memory block are taken up, one for each character and one for the terminating nul character. But, in the pointer notation the same 4 bytes required, **plus** N bytes to store the pointer variable **my_name** (where N depends on the system but is usually a minimum of 2 bytes and can be 4 or more).

In the array notation, **my_name** is short for **&myname[0]** which is the address of the first element of the array. Since the location of the array is fixed during run time, this is a constant (not a variable). In the pointer notation **my_name** is a variable. As to which is the **better** method, that depends on what you are going to do within the rest of the program.

Let's now go one step further and consider what happens if each of these declarations are done within a function as opposed to globally outside the bounds of any function.

```
void my_function_A(char *ptr)
{
    char a[] = "ABCDE"
    .
    .
}
```

```
void my_function_B(char *ptr)
{
    char *cp = "FGHIJ"
    .
    .
}
```

In the case of **my_function_A**, the content, or value(s), of the array **a[]** is considered to be the data. The array is said to be initialized to the values ABCDE. In the case of **my_function_B**, the value of the pointer **cp** is considered to be the data. The pointer has been initialized to point to the string **FGHIJ**. In both **my_function_A** and **my_function_B** the definitions are local variables and thus the string **ABCDE** is stored on the stack, as is the value of the pointer **cp**. The string **FGHIJ** can be stored anywhere. On my system it gets stored in the data segment.

By the way, array initialization of automatic variables as I have done in **my_function_A** was illegal in the older K&R C and only "came of age" in the newer ANSI C. A fact that may be important when one is considering portability and backwards compatibility.

As long as we are discussing the relationship/differences between pointers and arrays, let's move on to multi-dimensional arrays. Consider, for example the array:

```
    char multi[5][10];
```

Just what does this mean? Well, let's consider it in the following light.

```
    char multi[5][10];
```

Let's take the underlined part to be the "name" of an array. Then prepending the **char** and appending the **[10]** we have an array of 10 characters. But, the name **multi[5]** is itself an array indicating that there are 5 elements each being an array of 10 characters. Hence we have an array of 5 arrays of 10 characters each..

Assume we have filled this two dimensional array with data of some kind. In memory, it might look as if it had been formed by initializing 5 separate arrays using something like:

```
    multi[0] = {'0','1','2','3','4','5','6','7','8','9'}
    multi[1] = {'a','b','c','d','e','f','g','h','i','j'}
    multi[2] = {'A','B','C','D','E','F','G','H','I','J'}
    multi[3] = {'9','8','7','6','5','4','3','2','1','0'}
    multi[4] = {'J','I','H','G','F','E','D','C','B','A'}
```

At the same time, individual elements might be addressable using syntax such as:

```
    multi[0][3] = '3'
    multi[1][7] = 'h'
    multi[4][0] = 'J'
```

Since arrays are contiguous in memory, our actual memory block for the above should look like:

```
    0123456789abcdefghijABCDEFGHIJ9876543210JIHGFEDCBA
    ^
    |_____ starting at the address &multi[0][0]
```

Note that I did **not** write **multi[0] = "0123456789"**. Had I done so a terminating **'\0'** would have been implied since whenever double quotes are used a **'\0'** character is appended to the characters contained within those quotes. Had that been the case I would have had to set aside room for 11 characters per row instead of 10.

My goal in the above is to illustrate how memory is laid out for 2 dimensional arrays. That is, this is a 2 dimensional array of characters, NOT an array of "strings".

Now, the compiler knows how many columns are present in the array so it can interpret **multi + 1** as the address of the 'a' in the 2nd row above. That is, it adds 10, the number of columns, to get this location. If we were dealing with integers and an array with the same dimension the compiler would add **10*sizeof(int)** which, on my machine, would be 20. Thus, the address of the **9** in the 4th row above would be **&multi[3][0]** or ***(multi + 3)** in pointer notation. To get to the content of the 2nd element in the 4th row we add 1 to this address and dereference the result as in

```
    *(*(multi + 3) + 1)
```

With a little thought we can see that:

```
    *(*(multi + row) + col)    and
    multi[row][col]            yield the same results.
```

The following program illustrates this using integer arrays instead of character arrays.

```
------------------- program 6.1 ----------------------

/* Program 6.1 from PTRTUT10.HTM   6/13/97*/

#include <stdio.h>
#define ROWS 5
#define COLS 10

int multi[ROWS][COLS];

int main(void)
{
    int row, col;
    for (row = 0; row < ROWS; row++)
    {
        for (col = 0; col < COLS; col++)
        {
            multi[row][col] = row*col;
        }
    }

    for (row = 0; row < ROWS; row++)
    {
        for (col = 0; col < COLS; col++)
        {
            printf("\n%d  ",multi[row][col]);
            printf("%d ",*(*(multi + row) + col));
        }
    }

    return 0;
}
----------------- end of program 6.1 ---------------------
```

Because of the double de-referencing required in the pointer version, the name of a 2 dimensional array is often said to be equivalent to a pointer to a pointer. With a three dimensional array we would be dealing with an array of arrays of arrays and some might say its name would be equivalent to a pointer to a pointer to a pointer. However, here we have initially set aside the block of memory for the array by defining it using array notation. Hence, we are dealing with a constant, not a variable. That is we are talking about a fixed address not a variable pointer. The dereferencing function used above permits us to access any element in the array of arrays without the need of changing the value of that address (the address of **multi[0][0]** as given by the symbol **multi**).

CHAPTER 7: More on Multi-Dimensional Arrays

In the previous chapter we noted that given

```
    #define ROWS 5
    #define COLS 10

    int multi[ROWS][COLS];
```

we can access individual elements of the array **multi** using either:

```
    multi[row][col]
```

or

```
    *(*(multi + row) + col)
```

To understand more fully what is going on, let us replace

```
    *(multi + row)
```

with **X** as in:

```
    *(X + col)
```

Now, from this we see that **X** is like a pointer since the expression is de-referenced and we know that **col** is an integer. Here the arithmetic being used is of a special kind called "pointer arithmetic" is being used. That means that, since we are talking about an integer array, the address pointed to by (i.e. value of) **X + col + 1** must be greater than the address **X + col** by and amount equal to **sizeof(int)**.

Since we know the memory layout for 2 dimensional arrays, we can determine that in the expression **multi + row** as used above, **multi + row + 1** must increase by value an amount equal to that needed to "point to" the next row, which in this case would be an amount equal to **COLS * sizeof(int)**.

That says that if the expression ***(*(multi + row) + col)** is to be evaluated correctly at run time, the compiler must generate code which takes into consideration the value of **COLS**, i.e. the 2nd dimension. Because of the equivalence of the two forms of expression, this is true whether we are using the pointer expression as here or the array expression **multi[row][col]**.

Thus, to evaluate either expression, a total of 5 values must be known:

1. The address of the first element of the array, which is returned by the expression **multi**, i.e., the name of the array.
2. The size of the type of the elements of the array, in this case **sizeof(int)**.
3. The 2nd dimension of the array
4. The specific index value for the first dimension, **row** in this case.
5. The specific index value for the second dimension, **col** in this case.

Given all of that, consider the problem of designing a function to manipulate the element values of a previously declared array. For example, one which would set all the elements of the array **multi** to the value 1.

```
void set_value(int m_array[][COLS])
{
    int row, col;
    for (row = 0; row < ROWS; row++)
    {
        for (col = 0; col < COLS; col++)
        {
            m_array[row][col] = 1;
        }
    }
}
```

And to call this function we would then use:

```
set_value(multi);
```

Now, within the function we have used the values #defined by ROWS and COLS that set the limits on the for loops. But, these #defines are just constants as far as the compiler is concerned, i.e. there is nothing to connect them to the array size within the function. **row** and **col** are local variables, of course. The formal parameter definition permits the compiler to determine the characteristics associated with the pointer value that will be passed at run time. We really don't need the first dimension and, as will be seen later, there are occasions where we would prefer not to define it within the parameter definition, out of habit or consistency, I have not used it here. But, the second dimension must be used as has been shown in the expression for the parameter. The reason is that we need this in the evaluation of **m_array[row][col]** as has been described. While the parameter defines the data type (**int** in this case) and the automatic variables for row and column are defined in the for loops, only one value can be passed using a single parameter. In this case, that is the value of **multi** as noted in the call statement, i.e. the address of the first element, often referred to as a pointer to the array. Thus, the only way we have of informing the compiler of the 2nd dimension is by explicitly including it in the parameter definition.

In fact, in general all dimensions of higher order than one are needed when dealing with multi-dimensional arrays. That is if we are talking about 3 dimensional arrays, the 2nd **and** 3rd dimension must be specified in the parameter definition.

CHAPTER 8: Pointers to Arrays

Pointers, of course, can be "pointed at" any type of data object, including arrays. While that was evident when we discussed program 3.1, it is important to expand on how we do this when it comes to multi-dimensional arrays.

To review, in Chapter 2 we stated that given an array of integers we could point an integer pointer at that array using:

```
    int *ptr;
    ptr = &my_array[0];       /* point our pointer at the first
                                 integer in our array */
```

As we stated there, the type of the pointer variable must match the type of the first element of the array.

In addition, we can use a pointer as a formal parameter of a function which is designed to manipulate an array. e.g.

Given:

```
    int array[3] = {'1', '5', '7'};
    void a_func(int *p);
```

Some programmers might prefer to write the function prototype as:

```
   void a_func(int p[]);
```

which would tend to inform others who might use this function that the function is designed to manipulate the elements of an array. Of course, in either case, what actually gets passed is the value of a pointer to the first element of the array, independent of which notation is used in the function prototype or definition. Note that if the array notation is used, there is no need to pass the actual dimension of the array since we are not passing the whole array, only the address to the first element.

We now turn to the problem of the 2 dimensional array. As stated in the last chapter, C interprets a 2 dimensional array as an array of one dimensional arrays. That being the case, the first element of a 2 dimensional array of integers is a one dimensional array of integers. And a pointer to a two dimensional array of integers must be a pointer to that data type. One way of accomplishing this is through the use of the keyword "typedef". typedef assigns a new name to a specified data type. For example:

```
    typedef unsigned char byte;
```

causes the name **byte** to mean type **unsigned char**. Hence

```
    byte b[10];     would be an array of unsigned characters.
```

Note that in the typedef declaration, the word **byte** has replaced that which would normally be the name of our **unsigned char**. That is, the rule for using **typedef** is that the new name for the data type is the name used in the definition of the data type. Thus in:

```
    typedef int Array[10];
```

Array becomes a data type for an array of 10 integers. i.e. **Array my_arr;** declares **my_arr** as an array of 10 integers and **Array arr2d[5];** makes **arr2d** an array of 5 arrays of 10 integers each.

Also note that **Array *p1d;** makes **p1d** a pointer to an array of 10 integers. Because ***p1d** points to the same type as **arr2d**, assigning the address of the two dimensional array **arr2d** to **p1d**, the pointer to a one dimensional array of 10 integers is acceptable. i.e. **p1d = &arr2d[0];** or **p1d = arr2d;** are both correct.

Since the data type we use for our pointer is an array of 10 integers we would expect that incrementing **p1d** by 1 would change its value by **10*sizeof(int)**, which it does. That is, **sizeof(*p1d)** is 20. You can prove this to yourself by writing and running a simple short program.

Now, while using typedef makes things clearer for the reader and easier on the programmer, it is not really necessary. What we need is a way of declaring a pointer like **p1d** without the need of the **typedef** keyword. It turns out that this can be done and that

```
    int (*p1d)[10];
```

is the proper declaration, i.e. **p1d** here is a pointer to an array of 10 integers just as it was under the declaration using the Array type. Note that this is different from

```
    int *p1d[10];
```

which would make **p1d** the name of an array of 10 pointers to type **int**.

CHAPTER 9: Pointers and Dynamic Allocation of Memory

There are times when it is convenient to allocate memory at run time using **malloc()**, **calloc()**, or other allocation functions. Using this approach permits postponing the decision on the size of the memory block need to store an array, for example, until run time. Or it permits using a section of memory for the storage of an array of integers at one point in time, and then when that memory is no longer needed it can be freed up for other uses, such as the storage of an array of structures.

When memory is allocated, the allocating function (such as **malloc()**, **calloc()**, etc.) returns a pointer. The type of this pointer depends on whether you are using an older K&R compiler or the newer ANSI type compiler. With the older compiler the type of the returned pointer is **char**, with the ANSI compiler it is **void**.

If you are using an older compiler, and you want to allocate memory for an array of integers you will have to cast the char pointer returned to an integer pointer. For example, to allocate space for 10 integers we might write:

```
int *iptr;
iptr = (int *)malloc(10 * sizeof(int));
if (iptr == NULL)

{ .. ERROR ROUTINE GOES HERE .. }
```

If you are using an ANSI compliant compiler, **malloc()** returns a **void** pointer and since a void pointer can be assigned to a pointer variable of any object type, the **(int *)** cast shown above is not needed. The array dimension can be determined at run time and is not needed at compile time. That is, the **10** above could be a variable read in from a data file or keyboard, or calculated based on some need, at run time.

Because of the equivalence between array and pointer notation, once **iptr** has been assigned as above, one can use the array notation. For example, one could write:

```
int k;
for (k = 0; k < 10; k++)
   iptr[k] = 2;
```

to set the values of all elements to 2.

Even with a reasonably good understanding of pointers and arrays, one place the newcomer to C is likely to stumble at first is in the dynamic allocation of multi-dimensional arrays. In general, we would like to be able to access elements of such arrays using array notation, not pointer notation, wherever possible. Depending on the application we may or may not know both dimensions at compile time. This leads to a variety of ways to go about our task.

As we have seen, when dynamically allocating a one dimensional array its dimension can be determined at run time. Now, when using dynamic allocation of higher order arrays, we never need to know the first dimension at compile time. Whether we need to know the higher dimensions depends on how we go about writing the code. Here I will discuss various methods of dynamically allocating room for 2 dimensional arrays of integers.

First we will consider cases where the 2nd dimension is known at compile time.

METHOD 1:

One way of dealing with the problem is through the use of the **typedef** keyword. To allocate a 2 dimensional array of integers recall that the following two notations result in the same object code being generated:

```
    multi[row][col] = 1;      *(*(multi + row) + col) = 1;
```

It is also true that the following two notations generate the same code:

```
    multi[row]             *(multi + row)
```

Since the one on the right must evaluate to a pointer, the array notation on the left must also evaluate to a pointer. In fact **multi[0]** will return a pointer to the first integer in the first row, **multi[1]** a pointer to the first integer of the second row, etc. Actually, **multi[n]** evaluates to a pointer to that array of integers that make up the n-th row of our 2 dimensional array. That is, **multi** can be thought of as an array of arrays and **multi[n]** as a pointer to the n-th array of this array of arrays. Here the word **pointer** is being used to represent an address value. While such usage is common in the literature, when reading such statements one must be careful to distinguish between the constant address of an array and a variable pointer which is a data object in itself.

Consider now:

```
--------------- Program 9.1 --------------------------------

/* Program 9.1 from PTRTUT10.HTM  6/13/97 */

#include <stdio.h>
#include <stdlib.h>

#define COLS 5

typedef int RowArray[COLS];
RowArray *rptr;

int main(void)
{
    int nrows = 10;
    int row, col;
    rptr = malloc(nrows * COLS * sizeof(int));
    for (row = 0; row < nrows; row++)
    {
        for (col = 0; col < COLS; col++)
        {
            rptr[row][col] = 17;
        }
    }

    return 0;
}
------------- End of Prog. 9.1 --------------------------------
```

Here I have assumed an ANSI compiler so a cast on the void pointer returned by **malloc()** is not required. If you are using an older K&R compiler you will have to cast using:

```
    rptr = (RowArray *)malloc(.... etc.
```

Using this approach, **rptr** has all the characteristics of an array name name, (except that rptr is modifiable), and array notation may be used throughout the rest of the program. That also means that if you intend to write a function to modify the array contents, you must use COLS as a part of the formal parameter in that function, just as we did when discussing the passing of two dimensional arrays to a function.

METHOD 2:

In the METHOD 1 above, rptr turned out to be a pointer to type "one dimensional array of COLS integers". It turns out that there is syntax which can be used for this type without the need of **typedef**. If we write:

```
    int (*xptr)[COLS];
```

the variable **xptr** will have all the same characteristics as the variable **rptr** in METHOD 1 above, and we need not use the **typedef** keyword. Here **xptr** is a pointer to an array of integers and the size of that array is given by the **#defined COLS**. The parenthesis placement makes the pointer notation predominate, even though the array notation has higher precedence. i.e. had we written

```
    int *xptr[COLS];
```

we would have defined **xptr** as an array of pointers holding the number of pointers equal to that #defined by COLS. That is not the same thing at all. However, arrays of pointers have their use in the dynamic allocation of two dimensional arrays, as will be seen in the next 2 methods.

METHOD 3:

Consider the case where we do not know the number of elements in each row at compile time, i.e. both the number of rows and number of columns must be determined at run time. One way of doing this would be to create an array of pointers to type **int** and then allocate space for each row and point these pointers at each row. Consider:

```
-------------- Program 9.2 ------------------------------------

/* Program 9.2 from PTRTUT10.HTM   6/13/97 */

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int nrows = 5;     /* Both nrows and ncols could be evaluated */
    int ncols = 10;    /* or read in at run time */
    int row;
    int **rowptr;
    rowptr = malloc(nrows * sizeof(int *));
    if (rowptr == NULL)
    {
        puts("\nFailure to allocate room for row pointers.\n");
        exit(0);
    }

    printf("\n\n\nIndex   Pointer(hex)   Pointer(dec)   Diff.(dec)");

    for (row = 0; row < nrows; row++)
    {
        rowptr[row] = malloc(ncols * sizeof(int));
        if (rowptr[row] == NULL)
        {
            printf("\nFailure to allocate for row[%d]\n",row);
            exit(0);
        }
        printf("\n%d         %p         %d", row, rowptr[row],
rowptr[row]);
        if (row > 0)
        printf("              %d",(int)(rowptr[row] - rowptr[row-1]));
    }

    return 0;
}

--------------- End 9.2 ------------------------------------
```

In the above code **rowptr** is a pointer to pointer to type **int**. In this case it points to the first element of an array of pointers to type **int**. Consider the number of calls to **malloc()**:

```
    To get the array of pointers             1     call
    To get space for the rows                5     calls
                                           -----
                     Total                   6     calls
```

If you choose to use this approach note that while you can use the array notation to access individual elements of the array, e.g. **rowptr[row][col] = 17;**, it does not mean that the data in the "two dimensional array" is contiguous in memory.

You can, however, use the array notation just as if it were a continuous block of memory. For example, you can write:

```
    rowptr[row][col] = 176;
```

just as if rowptr were the name of a two dimensional array created at compile time. Of course **row** and **col** must be within the bounds of the array you have created, just as with an array created at compile time.

If you want to have a contiguous block of memory dedicated to the storage of the elements in the array you can do it as follows:

METHOD 4:

In this method we allocate a block of memory to hold the whole array first. We then create an array of pointers to point to each row. Thus even though the array of pointers is being used, the actual array in memory is contiguous. The code looks like this:

```
------------------ Program 9.3 -----------------------------------

/* Program 9.3 from PTRTUT10.HTM   6/13/97 */

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int **rptr;
    int *aptr;
    int *testptr;
    int k;
    int nrows = 5;     /* Both nrows and ncols could be evaluated */
    int ncols = 8;    /* or read in at run time */
    int row, col;

    /* we now allocate the memory for the array */

    aptr = malloc(nrows * ncols * sizeof(int));
    if (aptr == NULL)
    {
        puts("\nFailure to allocate room for the array");
        exit(0);
    }

    /* next we allocate room for the pointers to the rows */

    rptr = malloc(nrows * sizeof(int *));
    if (rptr == NULL)
    {
        puts("\nFailure to allocate room for pointers");
        exit(0);
    }
    /* and now we 'point' the pointers */

    for (k = 0; k < nrows; k++)
    {
        rptr[k] = aptr + (k * ncols);
    }

    /* Now we illustrate how the row pointers are incremented */
    printf("\n\nIllustrating how row pointers are incremented");
    printf("\n\nIndex   Pointer(hex)  Diff.(dec)");

    for (row = 0; row < nrows; row++)
    {
        printf("\n%d         %p", row, rptr[row]);
        if (row > 0)
        printf("              %d",(rptr[row] - rptr[row-1]));
    }
    printf("\n\nAnd now we print out the array\n");
    for (row = 0; row < nrows; row++)
    {
        for (col = 0; col < ncols; col++)
        {
            rptr[row][col] = row + col;
            printf("%d ", rptr[row][col]);
        }
        putchar('\n');
    }

    puts("\n");

    /* and here we illustrate that we are, in fact, dealing with
       a 2 dimensional array in a contiguous block of memory. */
    printf("And now we demonstrate that they are contiguous in
memory\n");

    testptr = aptr;
    for (row = 0; row < nrows; row++)
    {
        for (col = 0; col < ncols; col++)
        {
            printf("%d ", *(testptr++));
        }
        putchar('\n');
    }

    return 0;
}

------------- End Program 9.3 -----------------
```

Consider again, the number of calls to malloc()

```
    To get room for the array itself      1      call
    To get room for the array of ptrs     1      call
                                        ----
                         Total            2      calls
```

Now, each call to **malloc()** creates additional space overhead since **malloc()** is generally implemented by the operating system forming a linked list which contains data concerning the size of the block. But, more importantly, with large arrays (several hundred rows) keeping track of what needs to be freed when the time comes can be more cumbersome. This, combined with the contiguousness of the data block that permits initialization to all zeroes using **memset()** would seem to make the second alternative the preferred one.

As a final example on multidimensional arrays we will illustrate the dynamic allocation of a three dimensional array. This example will illustrate one more thing to watch when doing this kind of allocation. For reasons cited above we will use the approach outlined in alternative two. Consider the following code:

```
------------------- Program 9.4 -------------------------------------

/* Program 9.4 from PTRTUT10.HTM   6/13/97 */

#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>

int X_DIM=16;
int Y_DIM=5;
int Z_DIM=3;

int main(void)
{
    char *space;
    char ***Arr3D;
    int y, z;
    ptrdiff_t diff;

    /* first we set aside space for the array itself */

    space = malloc(X_DIM * Y_DIM * Z_DIM * sizeof(char));

    /* next we allocate space of an array of pointers, each
       to eventually point to the first element of a
       2 dimensional array of pointers to pointers */

    Arr3D = malloc(Z_DIM * sizeof(char **));

    /* and for each of these we assign a pointer to a newly
       allocated array of pointers to a row */

    for (z = 0; z < Z_DIM; z++)
    {
        Arr3D[z] = malloc(Y_DIM * sizeof(char *));

        /* and for each space in this array we put a pointer to
           the first element of each row in the array space
           originally allocated */
```

```
          for (y = 0; y < Y_DIM; y++)
          {
              Arr3D[z][y] = space + (z*(X_DIM * Y_DIM) + y*X_DIM);
          }
      }

      /* And, now we check each address in our 3D array to see if
         the indexing of the Arr3d pointer leads through in a
         continuous manner */

      for (z = 0; z < Z_DIM; z++)
      {
          printf("Location of array %d is %p\n", z, *Arr3D[z]);
          for ( y = 0; y < Y_DIM; y++)
          {
              printf("  Array %d and Row %d starts at %p", z, y,
Arr3D[z][y]);
              diff = Arr3D[z][y] - space;
              printf("    diff = %d  ",diff);
              printf(" z = %d  y = %d\n", z, y);
          }
      }
      return 0;
}

------------------- End of Prog. 9.4 ----------------------------
```

If you have followed this tutorial up to this point you should have no problem deciphering the above on the basis of the comments alone. There are a couple of points that should be made however. Let's start with the line which reads:

```
    Arr3D[z][y] = space + (z*(X_DIM * Y_DIM) + y*X_DIM);
```

Note that here **space** is a character pointer, which is the same type as **Arr3D[z][y]**. It is important that when adding an integer, such as that obtained by evaluation of the expression **(z*(X_DIM * Y_DIM) + y*X_DIM)**, to a pointer, the result is a new pointer value. And when assigning pointer values to pointer variables the data types of the value and variable must match.

CHAPTER 10: Pointers to Functions

Up to this point we have been discussing pointers to data objects. C also permits the declaration of pointers to functions. Pointers to functions have a variety of uses and some of them will be discussed here.

Consider the following real problem. You want to write a function that is capable of sorting virtually any collection of data that can be stored in an array. This might be an array of strings, or integers, or floats, or even structures. The sorting algorithm can be the same for all. For example, it could be a simple bubble sort algorithm, or the more complex shell or quick sort algorithm. We'll use a simple bubble sort for demonstration purposes.

Sedgewick [1] has described the bubble sort using C code by setting up a function which when passed a pointer to the array would sort it. If we call that function **bubble()**, a sort program is described by bubble_1.c, which follows:

```
/*-------------------- bubble_1.c --------------------*/

/* Program bubble_1.c from PTRTUT10.HTM   6/13/97 */

#include <stdio.h>

int arr[10] = { 3,6,1,2,3,8,4,1,7,2};

void bubble(int a[], int N);

int main(void)
{
    int i;
    putchar('\n');
    for (i = 0; i < 10; i++)
    {
        printf("%d ", arr[i]);
    }
    bubble(arr,10);
    putchar('\n');

    for (i = 0; i < 10; i++)
    {
        printf("%d ", arr[i]);
    }
    return 0;
}

void bubble(int a[], int N)
{
    int i, j, t;
    for (i = N-1; i >= 0; i--)
    {
        for (j = 1; j <= i; j++)
        {
            if (a[j-1] > a[j])
            {
                t = a[j-1];
                a[j-1] = a[j];
                a[j] = t;
            }
        }
    }
}



/*---------------------- end bubble_1.c -----------------------*/
```

The bubble sort is one of the simpler sorts. The algorithm scans the array from the second to the last element comparing each element with the one which precedes it. If the one that precedes it is larger than the current element, the two are swapped so the larger one is closer to the end of the array. On the first pass, this results in the largest element ending up at the end of the array. The array is now limited to all elements except the last and the process repeated. This puts the next largest element at a point preceding the largest element. The process is repeated for a number of times equal to the number of elements minus 1. The end result is a sorted array.

Here our function is designed to sort an array of integers. Thus in line 1 we are comparing integers and in lines 2 through 4 we are using temporary integer storage to store integers. What we want to do now is see if we can convert this code so we can use any data type, i.e. not be restricted to integers.

At the same time we don't want to have to analyze our algorithm and the code associated with it each time we use it. We start by removing the comparison from within the function **bubble()** so as to make it relatively easy to modify the comparison function without having to re-write portions related to the actual algorithm. This results in bubble_2.c:

```
/*---------------------- bubble_2.c -------------------------*/

/* Program bubble_2.c from PTRTUT10.HTM   6/13/97 */

   /* Separating the comparison function */

#include <stdio.h>

int arr[10] = { 3,6,1,2,3,8,4,1,7,2};

void bubble(int a[], int N);
int compare(int m, int n);

int main(void)
{
    int i;
    putchar('\n');
    for (i = 0; i < 10; i++)
    {
        printf("%d ", arr[i]);
    }
    bubble(arr,10);
    putchar('\n');

    for (i = 0; i < 10; i++)
    {
        printf("%d ", arr[i]);
    }
    return 0;
}

void bubble(int a[], int N)

{
    int i, j, t;
    for (i = N-1; i >= 0; i--)
    {
        for (j = 1; j <= i; j++)
        {
            if (compare(a[j-1], a[j]))
            {
                t = a[j-1];
                a[j-1] = a[j];
                a[j] = t;
            }
        }
    }
}

int compare(int m, int n)
{
    return (m > n);
}
/*--------------------- end of bubble_2.c -----------------------*/
```

If our goal is to make our sort routine data type independent, one way of doing this is to use pointers to type void to point to the data instead of using the integer data type. As a start in that direction let's modify a few things in the above so that pointers can be used. To begin with, we'll stick with pointers to type integer.

```
/*----------------------- bubble_3.c -------------------------*/

/* Program bubble_3.c from PTRTUT10.HTM    6/13/97 */

#include <stdio.h>

int arr[10] = { 3,6,1,2,3,8,4,1,7,2};

void bubble(int *p, int N);
int compare(int *m, int *n);

int main(void)
{
    int i;
    putchar('\n');

    for (i = 0; i < 10; i++)
    {
        printf("%d ", arr[i]);
    }
    bubble(arr,10);
    putchar('\n');

    for (i = 0; i < 10; i++)
    {
        printf("%d ", arr[i]);
    }
    return 0;
}

void bubble(int *p, int N)
{
    int i, j, t;
    for (i = N-1; i >= 0; i--)
    {
        for (j = 1; j <= i; j++)
        {
            if (compare(&p[j-1], &p[j]))
            {
                t = p[j-1];
                p[j-1] = p[j];
                p[j] = t;
            }
        }
    }
}

int compare(int *m, int *n)
{
    return (*m > *n);
}

/*------------------ end of bubble3.c -------------------------*/
```

Note the changes. We are now passing a pointer to an integer (or array of integers) to **bubble()**. And from within bubble we are passing pointers to the elements of the array that we want to compare to our comparison function. And, of course we are dereferencing these pointer in our **compare()** function in order to make the actual comparison. Our next step will be to convert the pointers in **bubble()** to pointers to type void so that that function will become more type insensitive. This is shown in bubble_4.

```
/*------------------ bubble_4.c ----------------------------*/

/* Program bubble_4.c from PTRTUT10,HTM   6/13/97 */

#include <stdio.h>
```

```
int arr[10] = { 3,6,1,2,3,8,4,1,7,2};

void bubble(int *p, int N);
int compare(void *m, void *n);

int main(void)
{
    int i;
    putchar('\n');

    for (i = 0; i < 10; i++)
    {
        printf("%d ", arr[i]);
    }
    bubble(arr,10);
    putchar('\n');

    for (i = 0; i < 10; i++)
    {
        printf("%d ", arr[i]);
    }
    return 0;
}

void bubble(int *p, int N)
{
    int i, j, t;
    for (i = N-1; i >= 0; i--)
    {
        for (j = 1; j <= i; j++)
        {
            if (compare((void *)&p[j-1], (void *)&p[j]))
            {
                t = p[j-1];
                p[j-1] = p[j];
                p[j] = t;
            }
        }
    }
}

int compare(void *m, void *n)
{
    int *m1, *n1;
    m1 = (int *)m;
    n1 = (int *)n;
    return (*m1 > *n1);
}

/*------------------ end of bubble_4.c ---------------------*/
```

Note that, in doing this, in **compare()** we had to introduce the casting of the void pointer types passed to the actual type being sorted. But, as we'll see later that's okay. And since what is being passed to **bubble()** is still a pointer to an array of integers, we had to cast these pointers to void pointers when we passed them as parameters in our call to **compare()**.

We now address the problem of what we pass to **bubble()**. We want to make the first parameter of that function a void pointer also. But, that means that within **bubble()** we need to do something about the variable **t**, which is currently an integer. Also, where we use **t = p[j-1];** the type of **p[j-1]** needs to be known in order to know how many bytes to copy to the variable **t** (or whatever we replace **t** with).

Currently, in bubble_4.c, knowledge within **bubble()** as to the type of the data being sorted (and hence the size of each individual element) is obtained from the fact that the first parameter is a pointer to type integer. If we are going to be able to use **bubble()** to sort any type of data, we need to make that pointer a pointer to type **void**. But, in doing so we are going to lose information concerning the size of individual elements within the array. So, in bubble_5.c we will add a separate parameter to handle this size information.

These changes, from bubble4.c to bubble5.c are, perhaps, a bit more extensive than those we have made in the past. So, compare the two modules carefully for differences.

```
/*---------------------- bubble5.c ---------------------------*/

/* Program bubble_5.c from PTRTUT10.HTM    6/13/97 */

#include <stdio.h>
#include <string.h>

long arr[10] = { 3,6,1,2,3,8,4,1,7,2};

void bubble(void *p, size_t width, int N);
int compare(void *m, void *n);

int main(void)
{
    int i;
    putchar('\n');

    for (i = 0; i < 10; i++)
    {
        printf("%d ", arr[i]);
    }
    bubble(arr, sizeof(long), 10);
    putchar('\n');

    for (i = 0; i < 10; i++)
    {
        printf("%ld ", arr[i]);
    }

    return 0;
}

void bubble(void *p, size_t width, int N)
{
    int i, j;
    unsigned char buf[4];
    unsigned char *bp = p;
    for (i = N-1; i >= 0; i--)
    {
        for (j = 1; j <= i; j++)
        {
            if (compare((void *)(bp + width*(j-1)),
                        (void *)(bp + j*width)))  /* 1 */
            {
/*              t = p[j-1];   */
                memcpy(buf, bp + width*(j-1), width);
/*              p[j-1] = p[j];   */
                memcpy(bp + width*(j-1), bp + j*width , width);
/*              p[j] = t;   */
                memcpy(bp + j*width, buf, width);
            }
        }
    }
}

int compare(void *m, void *n)
{
    long *m1, *n1;
    m1 = (long *)m;
    n1 = (long *)n;
    return (*m1 > *n1);
}

/*--------------------- end of bubble5.c ---------------------*/
```

Note that I have changed the data type of the array from **int** to **long** to illustrate the changes needed in the **compare()** function. Within **bubble()** I've done away with the variable **t** (which we would have had to change from type **int** to type **long**). I have added a buffer of size 4 unsigned characters, which is the size needed to hold a long (this will change again in future modifications to this code). The unsigned character pointer ***bp** is used to point to the base of the array to be sorted, i.e. to the first element of that array.

We also had to modify what we passed to **compare()**, and how we do the swapping of elements that the comparison indicates need swapping. Use of **memcpy()** and pointer notation instead of array notation work towards this reduction in type sensitivity.

Again, making a careful comparison of bubble5.c with bubble4.c can result in improved understanding of what is happening and why.

We move now to bubble6.c where we use the same function bubble() that we used in bubble5.c to sort strings instead of long integers. Of course we have to change the comparison function since the means by which strings are compared is different from that by which long integers are compared. And,in bubble6.c we have deleted the lines within **bubble()** that were commented out in bubble5.c.

```
/*--------------------- bubble6.c ---------------------*/
/* Program bubble_6.c from PTRTUT10.HTM   6/13/97 */
```

```
#include <stdio.h>
#include <string.h>

#define MAX_BUF 256

char arr2[5][20] = {  "Mickey Mouse",
                      "Donald Duck",
                      "Minnie Mouse",
                      "Goofy",
                      "Ted Jensen" };

void bubble(void *p, int width, int N);
int compare(void *m, void *n);

int main(void)
{
    int i;
    putchar('\n');

    for (i = 0; i < 5; i++)
    {
        printf("%s\n", arr2[i]);
    }
    bubble(arr2, 20, 5);
    putchar('\n\n');

    for (i = 0; i < 5; i++)
    {
        printf("%s\n", arr2[i]);
    }
    return 0;
}

void bubble(void *p, int width, int N)
{
    int i, j, k;
    unsigned char buf[MAX_BUF];
    unsigned char *bp = p;

    for (i = N-1; i >= 0; i--)
    {
        for (j = 1; j <= i; j++)
        {
          k = compare((void *)(bp + width*(j-1)), (void *)(bp +
j*width));
          if (k > 0)
            {
             memcpy(buf, bp + width*(j-1), width);
             memcpy(bp + width*(j-1), bp + j*width , width);
             memcpy(bp + j*width, buf, width);
            }
        }
    }
}

int compare(void *m, void *n)
```

```
{
    char *m1 = m;
    char *n1 = n;
    return (strcmp(m1,n1));
}

/*------------------ end of bubble6.c ---------------------*/
```

But, the fact that **bubble()** was unchanged from that used in bubble5.c indicates that that function is capable of sorting a wide variety of data types. What is left to do is to pass to **bubble()** the name of the comparison function we want to use so that it can be truly universal. Just as the name of an array is the address of the first element of the array in the data segment, the name of a function decays into the address of that function in the code segment. Thus we need to use a pointer to a function. In this case the comparison function.

Pointers to functions must match the functions pointed to in the number and types of the parameters and the type of the return value. In our case, we declare our function pointer as:

```
   int (*fptr)(const void *p1, const void *p2);
```

Note that were we to write:

```
    int *fptr(const void *p1, const void *p2);
```

we would have a function prototype for a function which returned a pointer to type **int**. That is because in C the parenthesis () operator have a higher precedence than the pointer * operator. By putting the parenthesis around the string (*fptr) we indicate that we are declaring a function pointer.

We now modify our declaration of **bubble()** by adding, as its 4th parameter, a function pointer of the proper type. It's function prototype becomes:

```
    void bubble(void *p, int width, int N,
                int(*fptr)(const void *, const void *));
```

When we call the **bubble()**, we insert the name of the comparison function that we want to use. bubble7.c illustrate how this approach permits the use of the same **bubble()** function for sorting different types of data.

```
/*------------------ bubble7.c ------------------*/

/* Program bubble_7.c from PTRTUT10.HTM  6/10/97 */

#include <stdio.h>
#include <string.h>

#define MAX_BUF 256
```

```
long arr[10] = { 3,6,1,2,3,8,4,1,7,2};
char arr2[5][20] = {  "Mickey Mouse",
                      "Donald Duck",
                      "Minnie Mouse",
                      "Goofy",
                      "Ted Jensen" };

void bubble(void *p, int width, int N,
            int(*fptr)(const void *, const void *));
int compare_string(const void *m, const void *n);
int compare_long(const void *m, const void *n);

int main(void)
{
    int i;
    puts("\nBefore Sorting:\n");

    for (i = 0; i < 10; i++)               /* show the long ints */
    {
        printf("%ld ",arr[i]);
    }
    puts("\n");

    for (i = 0; i < 5; i++)                  /* show the strings */
    {
        printf("%s\n", arr2[i]);
    }
    bubble(arr, 4, 10, compare_long);          /* sort the longs */
    bubble(arr2, 20, 5, compare_string);     /* sort the strings */
    puts("\n\nAfter Sorting:\n");

    for (i = 0; i < 10; i++)             /* show the sorted longs */
    {
        printf("%d ",arr[i]);
    }
    puts("\n");

    for (i = 0; i < 5; i++)            /* show the sorted strings */
    {
        printf("%s\n", arr2[i]);
    }
    return 0;
}

void bubble(void *p, int width, int N,
            int(*fptr)(const void *, const void *))
{
    int i, j, k;
    unsigned char buf[MAX_BUF];
    unsigned char *bp = p;

    for (i = N-1; i >= 0; i--)
    {
        for (j = 1; j <= i; j++)
        {
            k = fptr((void *)(bp + width*(j-1)), (void *)(bp +
j*width));
```

```
            if (k > 0)
            {
                memcpy(buf, bp + width*(j-1), width);
                memcpy(bp + width*(j-1), bp + j*width , width);
                memcpy(bp + j*width, buf, width);
            }
        }
    }
}

int compare_string(const void *m, const void *n)
{
    char *m1 = (char *)m;
    char *n1 = (char *)n;
    return (strcmp(m1,n1));
}

int compare_long(const void *m, const void *n)
{
    long *m1, *n1;
    m1 = (long *)m;
    n1 = (long *)n;
    return (*m1 > *n1);
}

/*----------------- end of bubble7.c -----------------*/
```

References for Chapter 10:

1. "Algorithms in C"
   Robert Sedgewick
   Addison-Wesley
   ISBN 0-201-51425-7

EPILOG

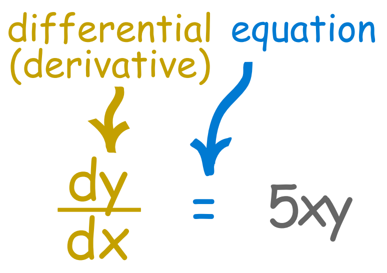
I have written the preceding material to provide an introduction to pointers for newcomers to C. In C, the more one understands about pointers the greater flexibility one has in the writing of code. The above expands on my first effort at this which was entitled ptr_help.txt and found in an early version of Bob Stout's collection of C code SNIPPETS. The content in this version has been updated from that in PTRTUTOT.ZIP included in SNIP9510.ZIP.

I am always ready to accept constructive criticism on this material, or review requests for the addition of other relevant material. Therefore, if you have questions, comments, criticisms, etc. concerning that which has been presented, I would greatly appreciate your contacting me via email me at **tjensen@ix.netcom.com.**

Separation of Variables

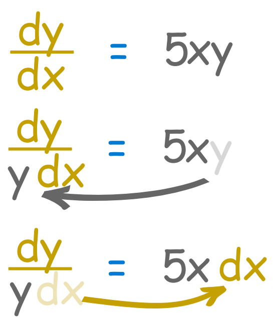
Separation of Variables is a special method to solve some Differential Equations

A Differential Equation is an equation with a function and one or more of its derivatives :



Example: an equation with the function **y** and its derivative $\frac{dy}{dx}$

When Can I Use it?



Separation of Variables can be used when:

All the y terms (including dy) can be moved to one side of the equation, and

All the x terms (including dx) to the other side.

Method

Three Steps:

- **Step 1** Move all the y terms (including dy) to one side of the equation and all the x terms (including dx) to the other side.
- **Step 2** Integrate one side with respect to **y** and the other side with respect to **x**. Don't forget "+ C" (the constant of integration).
- **Step 3** Simplify

Example: Solve this (k is a constant)

$$\frac{dy}{dx} = ky$$

Step 1 Separate the variables by moving all the y terms to one side of the equation and all the x terms to the other side.

Multiply both sides by dx: $dy = ky\ dx$

Divide both sides by y: $\frac{dy}{y} = k\ dx$

Step 2 Integrate both sides of the equation separately:

Put the integral sign in front: $\int \frac{dy}{y} = \int k\ dx$

Integrate left side: $\ln(y) + C = \int k\ dx$

Integrate right side: $\ln(y) + C = kx + D$

C is the constant of integration. And we use D for the other, as it is a different constant.

Step 3 Simplify

We can roll the two constants into one (a=D−C): $\ln(y) = kx + a$

$\mathbf{e^{(\ln(y))} = y}$, so let's take exponents on both sides: $y = e^{kx + a}$

And $\mathbf{e^{kx + a} = e^{kx}\, e^{a}}$ so we get: $y = e^{kx}\, e^{a}$

$\mathbf{e^{a}}$ is just a constant so we replace it with **c** $y = ce^{kx}$

We have solved it:

$$y = ce^{kx}$$

This is a general type of first order differential equation which turns up in all sorts of unexpected places in real world examples.

We used **y** and **x**, but the same method works for other variable names, like this:

Example: Rabbits!

The more rabbits you have the more baby rabbits you will get. Then those rabbits grow up and have babies too! The population will grow faster and faster.

The important parts of this are:

- the population **N** at any time **t**
- the growth rate **r**
- the population's rate of change $\frac{dN}{dt}$

The rate of change at any time equals the growth rate times the population:

$$\frac{dN}{dt} = rN$$

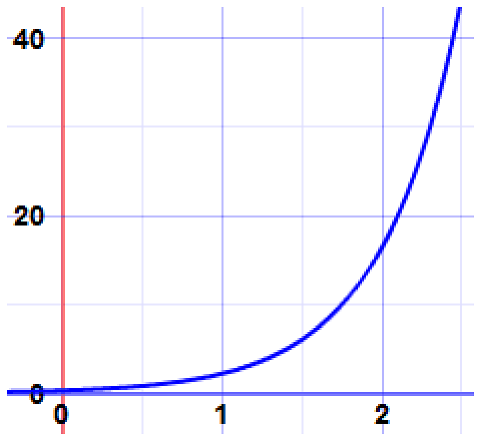
But hey! This is the same as the equation we just solved! It just has different letters:

- N instead of y
- t instead of x
- r instead of k

So we can jump to a solution:

$$N = ce^{rt}$$

And here is an example, the graph of $N = 0.3e^{2t}$:



Exponential Growth

There are other equations that follow this pattern such as continuous compound interest.

More Examples

OK, on to some different examples of separating the variables:

Example: Solve this

$$\frac{dy}{dx} = \frac{1}{y}$$

Step 1 Separate the variables by moving all the y terms to one side of the equation and all the x terms to the other side.

Multiply both sides by dx: $dy = (1/y)\, dx$

Multiply both sides by y: $y\, dy = dx$

Step 2 Integrate both sides of the equation separately:

Put the integral sign in front: $\int y\, dy = \int dx$

Integrate each side: $(y^2)/2 = x + C$

We integrated both sides in the one line, and used just one constant of integration **C.** This saves time, and is perfectly OK as we could have +D on one, +E on the other and just say that C = E−D.

Step 3 Simplify

Multiply both sides by 2: $y^2 = 2(x + C)$

Square root of both sides: $y = \pm\sqrt{(2(x + C))}$

*Note: This is not the same as $y = \sqrt{(2x)} + C$, because the C was added **before** we took the square root. This happens a lot with differential equations. We cannot just add the C at the end of the process. It is added when doing the integration.*

We have solved it:

$$y = \pm\sqrt{(2(x + C))}$$

A harder example:

Example: Solve this

$$\frac{dy}{dx} = \frac{2xy}{1 + x^2}$$

Step 1 Separate the variables

Multiply both sides by dx, divide both sides by y: $\frac{1}{y}dy = \frac{2x}{1 + x^2}dx$

Step 2 Integrate both sides of the equation separately:

Put the integral sign in front: $\int \frac{1}{y}dy = \int \frac{2x}{1 + x^2}dx$

The left side is a simple logarithm, the right side can be integrated using substitution:

Let $\mathbf{u = 1 + x^2}$, so $\mathbf{du = 2x\ dx}$ $\int \frac{1}{y}dy = \int \frac{1}{u}du$

Integrate: $\ln(y) = \ln(u) + C$

Then we make $\mathbf{C = \ln(k)}$: $\ln(y) = \ln(u) + \ln(k)$

So we can get this: $y = uk$

Now put $u = 1 + x^2$ back again: $y = k(1 + x^2)$

Step 3 Simplify

It is already as simple as can be. We have solved it:

$$y = k(1 + x^2)$$

An even harder example: the famous ***Verhulst Equation***

Example: Rabbits Again!

Remember our growth Differential Equation:

$$\frac{dN}{dt} = rN$$

Well, that growth can't go on forever as they will soon run out of available food.

A guy called Verhulst included **k** (the maximum population the food can support) to get:

$$\frac{dN}{dt} = rN(1-N/k)$$

The Verhulst Equation

Can this be solved?

Yes, with the help of one trick ...

Step 1 Separate the variables

Multiply both sides by dt: $dN = rN(1-N/k)\ dt$

Divide both sides by $N(1-N/k)$: $\frac{1}{N(1-N/k)} dN = r\ dt$

Step 2 Integrate

Put the integral sign in front: $\int \frac{1}{N(1-N/k)} dN = \int r\ dt$

Hmmm... the left side looks hard to integrate. In fact it can be done, with a little trick.

We start with this: $$\frac{1}{N(1-N/k)}$$

Multiply top and bottom by k: $$\frac{k}{N(k-N)}$$

Now here is the trick, add **N** and **−N** to the top (see Partial Fractions): $$\frac{N+k-N}{N(k-N)}$$

and split it into two fractions: $$\frac{N}{N(k-N)} + \frac{k-N}{N(k-N)}$$

Simplify each fraction: $$\frac{1}{k-N} + \frac{1}{N}$$

They can be integrated separately now, like this:

$$\int \frac{1}{k-N} dN + \int \frac{1}{N} dN = \int r\, dt$$

Integrate: $$-\ln(k-N) + \ln(N) = rt + C$$

Done!

*(Why did that become **minus** ln(k−N)? Because we are integrating with respect to N.)*

Step 3 Simplify

Negative of all terms: $$\ln(k-N) - \ln(N) = -rt - C$$

Combine ln(): $$\ln((k-N)/N) = -rt - C$$

Now take exponents on both sides: $$(k-N)/N = e^{-rt-C}$$

Separate the powers of e: $$(k-N)/N = e^{-rt}\, e^{-C}$$

$\mathbf{e^{-C}}$ is a constant, we can replace it with **A:** $$(k-N)/N = Ae^{-rt}$$

We are getting close! Just a little more algebra to get N on its own:

Separate the fraction terms: $$(k/N)-1 = Ae^{-rt}$$

Add 1 to both sides: $k/N = 1 + Ae^{-rt}$

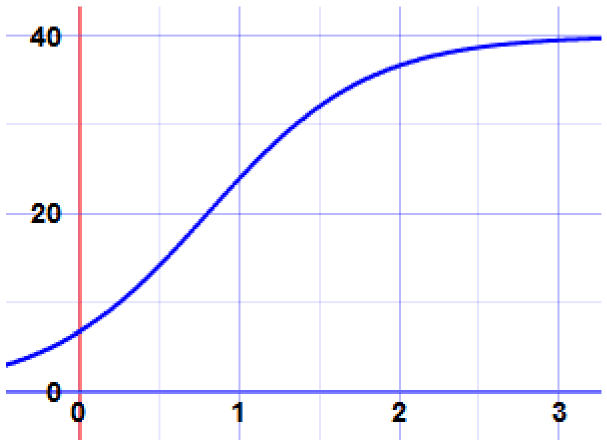
Divide both by k: $1/N = (1 + Ae^{-rt})/k$

Reciprocal of both sides: $N = k/(1 + Ae^{-rt})$

And we have our solution:

$$N = \frac{k}{1 + Ae^{-rt}}$$

And here is an example, the graph of $\frac{40}{1 + 5e^{-2t}}$:



It starts rising exponentially,
then flattens out as it reaches k=40

Constant-Coefficient Linear Differential Equations

Math 240 — Calculus III

Summer 2013, Session II

Monday, August 5, 2013

Agenda

1. Homogeneous constant-coefficient linear differential equations

2. Nonhomogeneous constant-coefficient linear differential equations

Introduction

Last week we found solutions to the linear differential equation

$$y'' + y' - 6y = 0$$

of the form $y(x) = e^{rx}$. In fact, we found all solutions.
This technique will often work. If $y(x) = e^{rx}$ then

$$y'(x) = re^{rx}, \quad y''(x) = r^2e^{rx}, \quad \ldots, \quad y^{(n)}(x) = r^ne^{rx}.$$

So if $r^n + a_1r^{n-1} + \cdots + a_{n-1}r + a_n = 0$ then $y(x) = e^{rx}$ is a solution to the linear differential equation

$$y^{(n)} + a_1y^{(n-1)} + \cdots + a_{n-1}y' + a_ny = 0.$$

Today we'll develop this approach more rigorously.

The auxiliary polynomial

Consider the homogeneous linear differential equation

$$y^{(n)} + a_1 y^{(n-1)} + \cdots + a_{n-1} y' + a_n y = 0$$

with *constant coefficients* a_i. Expressed as a linear differential operator, the equation is $P(D)y = 0$, where

$$P(D) = D^n + a_1 D^{n-1} + \cdots + a_{n-1} D + a_n.$$

Definition

A linear differential operator with constant coefficients, such as $P(D)$, is called a **polynomial differential operator**. The polynomial

$$P(r) = r^n + a_1 r^{n-1} + \cdots + a_{n-1} r + a_n$$

is called the **auxiliary polynomial**, and the equation $P(r) = 0$ the **auxiliary equation**.

The auxiliary polynomial

Example

The equation $y'' + y' - 6y = 0$ has auxiliary polynomial

$$P(r) = r^2 + r - 6.$$

Examples

Give the auxiliary polynomials for the following equations.

1. $y'' + 2y' - 3y = 0$	$r^2 + 2r - 3$
2. $(D^2 - 7D + 24)y = 0$	$r^2 - 7r + 24$
3. $y''' - 2y'' - 4y' + 8y = 0$	$r^3 - 2r^2 - 4r + 8$

The roots of the auxiliary polynomial will determine the solutions to the differential equation.

Polynomial differential operators commute

The key fact that will allow us to solve constant-coefficient linear differential equations is that polynomial differential operators commute.

Theorem

If $P(D)$ and $Q(D)$ are polynomial differential operators, then

$$P(D)Q(D) = Q(D)P(D).$$

Proof.

For our purposes, it will suffice to consider the case where P and Q are linear. $\mathcal{Q}.\mathcal{E}.\mathcal{D}.$

Commuting polynomial differential operators will allow us to turn a root of the auxiliary polynomial into a solution to the corresponding differential equation.

Linear polynomial differential operators

In our example,

$$y'' + y' - 6y = 0,$$

with auxiliary polynomial

$$P(r) = r^2 + r - 6,$$

the roots of $P(r)$ are $r = 2$ and $r = -3$. An equivalent statement is that $r - 2$ and $r + 3$ are linear factors of $P(r)$.

The functions $y_1(x) = e^{2x}$ and $y_2(x) = e^{-3x}$ are solutions to

$$y_1' - 2y_1 = 0 \quad \text{and} \quad y_2' + 3y_2 = 0,$$

respectively.

Theorem

The general solution to the linear differential equation

$$y' - ay = 0$$

is $y(x) = ce^{ax}$.

Theorem

Suppose $P(D)$ and $Q(D)$ are polynomial differential operators

$$P(D)y_1 = 0 = Q(D)y_2.$$

If $L = P(D)Q(D)$, then

$$Ly_1 = 0 = Ly_2.$$

Proof.

$$P(D)Q(D)y_2 = P(D)\Big(Q(D)y_2\Big) = P(D)0 = 0$$

$$\begin{aligned} P(D)Q(D)y_1 &= Q(D)P(D)y_1 \\ &= Q(D)\Big(P(D)y_1\Big) = Q(D)0 = 0 \qquad \mathcal{Q.E.D.} \end{aligned}$$

Example

The theorem implies that, since

$$(D-2)y_1 = 0 \quad \text{and} \quad (D+3)y_2 = 0,$$

the functions $y_1(x) = e^{2x}$ and $y_2(x) = e^{-3x}$ are solutions to

$$y'' + y' - 6y = (D^2 + D - 6)y = (D-2)(D+3)y = 0.$$

Linear polynomial differential operators

Furthermore, solutions produced from different roots of the auxiliary polynomial are independent.

Example

If $y_1(x) = e^{2x}$ and $y_2(x) = e^{-3x}$, then

$$\begin{aligned} W[y_1, y_2](x) &= \begin{vmatrix} e^{2x} & e^{-3x} \\ 2e^{2x} & -3e^{-3x} \end{vmatrix} \\ &= e^{-x} \begin{vmatrix} 1 & 1 \\ 2 & -3 \end{vmatrix} = -5e^{-x} \neq 0. \end{aligned}$$

Distinct linear factors

If we can factor the auxiliary polynomial into distinct linear factors, then the solutions from each linear factor will combine to form a fundamental set of solutions.

Example

Determine the general solution to $y'' - y' - 2y = 0$.

The auxiliary polynomial is

$$P(r) = r^2 - r - 2 = (r-2)(r+1).$$

Its roots are $r_1 = 2$ and $r_2 = -1$. The functions $y_1(x) = e^{2x}$ and $y_2(x) = e^{-x}$ satisfy

$$(D-2)y_1 = 0 = (D+1)y_2.$$

Therefore, y_1 and y_2 are solutions to the original equation. Since we have 2 solutions to a 2^{nd} degree equation, they constitute a fundamental set of solutions; the general solution is

$$y(x) = c_1 e^{2x} + c_2 e^{-x}.$$

Multiple roots

What can go wrong with this process? The auxiliary polynomial could have a multiple root. In this case, we would get one solution from that root, but not enough to form the general solution. Fortunately, there are more.

Theorem

The differential equation $(D-r)^m y = 0$ *has the following* m *linearly independent solutions:*

$$e^{rx},\ xe^{rx},\ x^2e^{rx},\ \ldots,\ x^{m-1}e^{rx}.$$

Proof.

Check it. $\mathcal{Q}.\mathcal{E}.\mathcal{D}.$

Multiple roots

Example

Determine the general solution to $y'' + 4y' + 4y = 0$.

1. The auxiliary polynomial is $r^2 + 4r + 4$.
2. It has the multiple root $r = -2$.
3. Therefore, two linearly independent solutions are
$$y_1(x) = e^{-2x} \quad \text{and} \quad y_2(x) = xe^{-2x}.$$
4. The general solution is
$$y(x) = e^{-2x}(c_1 + c_2 x).$$

Complex roots

What happens if the auxiliary polynomial has complex roots? Can we recover real solutions? Yes!

Theorem

If $P(D)y = 0$ is a linear differential equation with real constant coefficients and $(D-r)^m$ is a factor of $P(D)$ with $r = a+bi$ and $b \neq 0$, then

1. *$P(D)$ must also have the factor $(D-\overline{r})^m$,*
2. *this factor contributes the complex solutions*
$$e^{(a\pm bi)x},\ xe^{(a\pm bi)x},\ \ldots,\ x^{m-1}e^{(a\pm bi)x},$$
3. *the real and imaginary parts of the complex solutions are linearly independent real solutions*
$$x^k e^{ax}\cos bx \quad \text{and} \quad x^k e^{ax}\sin bx$$
for $k = 0, 1, \ldots, m-1$.

Complex roots

Example

Determine the general solution to $y'' + 6y' + 25y = 0$.

1. The auxiliary polynomial is $r^2 + 6r + 25$.
2. Its has roots $r = -3 \pm 4i$.
3. Two independent real-valued solutions are
$$y_1(x) = e^{-3x} \cos 4x \quad \text{and} \quad y_2(x) = e^{-3x} \sin 4x.$$
4. The general solution is
$$y(x) = e^{-3x}(c_1 \cos 4x + c_2 \sin 4x).$$

Segue

We have now learned how to solve homogeneous linear differential equations

$$P(D)y = 0$$

when $P(D)$ is a polynomial differential operator. Now we will try to solve nonhomogeneous equations

$$P(D)y = F(x).$$

Recall that the solutions to a nonhomogeneous equation are of the form

$$y(x) = y_c(x) + y_p(x),$$

where y_c is the general solution to the associated homogeneous equation and y_p is a particular solution.

Overview

The technique proceeds from the observation that, if we know a polynomial differential operator $A(D)$ so that

$$A(D)F = 0,$$

then applying $A(D)$ to the nonhomogeneous equation

$$P(D)y = F \tag{1}$$

yields the homogeneous equation

$$A(D)P(D)y = 0. \tag{2}$$

A particular solution to (1) will be a solution to (2) that is not a solution to the associated homogeneous equation $P(D)y = 0$.

Example

Determine the general solution to

$$(D+1)(D-1)y = 16e^{3x}.$$

1. The associated homogeneous equation is $(D+1)(D-1)y = 0$. It has the general solution $y_c(x) = c_1e^x + c_2e^{-x}$.
2. Recognize the nonhomogeneous term $F(x) = 16e^{3x}$ as a solution to the equation $(D-3)y = 0$.
3. The differential equation
$$(D-3)(D+1)(D-1)y = 0$$
has the general solution $y(x) = c_1e^x + c_2e^{-x} + c_3e^{3x}$.
4. Pick the **trial solution** $y_p(x) = c_3e^{3x}$. Substituting it into the original equation forces us to choose $c_3 = 2$.
5. Thus, the general solution is
$$y(x) = y_c(x) + y_p(x) = c_1e^x + c_2e^{-x} + 2e^{3x}.$$

Annihilators and the method of undetermined coefficients

This method for obtaining a particular solution to a nonhomogeneous equation is called the **method of undetermined coefficients** because we pick a trial solution with an unknown coefficient. It can be applied when

1. the differential equation is of the form
$$P(D)y = F(x),$$
where $P(D)$ is a polynomial differential operator,
2. there is another polynomial differential operator $A(D)$ such that
$$A(D)F = 0.$$

A polynomial differential operator $A(D)$ that satisfies $A(D)F = 0$ is called an **annihilator** of F.

Finding annihilators

Functions that can be annihilated by polynomial differential operators are exactly those that can arise as solutions to constant-coefficient homogeneous linear differential equations. We have seen that these functions are

1. $F(x) = cx^k e^{ax}$,
2. $F(x) = cx^k e^{ax} \sin bx$,
3. $F(x) = cx^k e^{ax} \cos bx$,
4. linear combinations of 1–3.

If the nonhomogeneous term is one of 1–3, then it can be annihilated by something of the form $A(D) = (D - r)^{k+1}$, with $r = a$ in 1 and $r = a + bi$ in 2 and 3. Otherwise, annihilators can be found by taking successive derivatives of F and looking for linear dependencies.

Example

Determine the general solution to

$$(D-4)(D+1)y = 16xe^{3x}.$$

1. The general solution to the associated homogeneous equation $(D-4)(D+1)y = 0$ is $y_c(x) = c_1e^{4x} + c_2e^{-x}$.
2. An annihilator for $16xe^{3x}$ is $A(D) = (D-3)^2$.
3. The general solution to $(D-3)^2(D-4)(D+1)y = 0$ includes y_c and the terms c_3e^{3x} and c_4xe^{3x}.
4. Using the trial solution $y_p(x) = c_3e^{3x} + c_4xe^{3x}$, we find the values $c_3 = -3$ and $c_4 = -4$.
5. The general solution is
$$y(x) = y_c(x) + y_p(x) = c_1e^{4x} + c_2e^{-x} - 3e^{3x} - 4xe^{3x}.$$

Example

Determine the general solution to

$$(D-2)y = 3\cos x + 4\sin x.$$

1. The associated homogeneous equation, $(D-2)y=0$, has the general solution $y_c(x) = c_1 e^{2x}$.
2. Look for linear dependencies among derivatives of $F(x) = 3\cos x + 4\sin x$. Discover the annihilator $A(D) = D^2+1$.
3. The general solution to $(D^2+1)(D-2)y=0$ includes y_c and the additional terms $c_2\cos x + c_3 \sin x$.
4. Using the trial solution $y_p(x) = c_2 \cos x + c_3 \sin x$, we obtain values $c_2 = -2$ and $c_3 = -1$.
5. The general solution is

$$y(x) = c_1 e^{2x} - 2\cos x - \sin x.$$

DIFFERENTIAL EQUATIONS 6

Many physical problems, when formulated in mathematical forms, lead to **differential equations.** Differential equations enter naturally as models for many phenomena in economics, commerce, engineering etc. Many of these phenomena are complex in nature and very difficult to understand. But when they are described by differential equations, it is easy to analyse them. For example, if the rate of change of cost for x outputs is directly proportional to the cost, then this phenomenon is described by the differential equation,

$\frac{dC}{dx} = k\,C$, where C is the cost and k is constant. The solution of this differential equation is

$C = C_0\,e^{kx}$ where $C = C_0$ when $x = 0$.

6.1 FORMATION OF DIFFERENTIAL EQUATIONS

A **Differential Equation** is one which involves one or more independent variables, a dependent variable and one or more of their differential coefficients.

There are two types of differential equations:

(i) **Ordinary differential equations** involving only one independent variable and derivatives of the dependent variable with respect to the independent variable.

(ii) **Partial differential equations** which involve more than one independent variable and partial derivatives of the dependent variable with respect to the independent variables.

The following are a few examples for differential equations:

1) $\left(\frac{dy}{dx}\right)^2 - 3\frac{dy}{dx} + 2y = e^x$ 2) $\frac{d^2y}{dx^2} - 5\frac{dy}{dx} + 3y = 0$

3) $\left\{1+\left(\frac{dy}{dx}\right)^2\right\}^{\frac{3}{2}} = k\frac{d^2y}{dx^2}$ 4) $x\frac{\partial u}{\partial x} + y\frac{\partial u}{\partial y} = 0$

5) $\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = 0$ 6) $\frac{\partial^2 z}{\partial x^2} + \frac{\partial^2 z}{\partial y^2} = x + y$

(1), (2) and (3) are ordinary differential equations and

(4), (5) and (6) are partial differential equations.

In this chapter we shall study ordinary differential equations only.

6.1.1 Order and Degree of a Differential Equation

The order of the derivative of the highest order present in a differential equation is called the **order** of the differential equation.

For example, consider the differential equation

$$x^2\left(\frac{d^2y}{dx^2}\right)^3 + 3\left(\frac{d^3y}{dx^3}\right)^2 + 7\frac{dy}{dx} - 4y = 0$$

The orders of $\frac{d^3y}{dx^3}$, $\frac{d^2y}{dx^2}$ and $\frac{dy}{dx}$ are 3, 2 and 1 respectively. So the highest order is 3. Thus the order of the differential equation is 3.

The degree of the derivative of the highest order present in a differential equation is called the **degree** of the differential equation. Here the differential coefficients should be free from the radicals and fractional exponents.

Thus the degree of

$$x^2\left(\frac{d^2y}{dx^2}\right)^3 + 3\left(\frac{d^3y}{dx^3}\right)^2 + 7\frac{dy}{dx} - 4y = 0 \quad \text{is } 2$$

Example 1

Write down the order and degree of the following differential equations.

(i) $\left(\frac{dy}{dx}\right)^3 - 4\left(\frac{dy}{dx}\right) + y = 3e^x$ **(ii)** $\left(\frac{d^2y}{dx^2}\right)^3 + 7\left(\frac{dy}{dx}\right)^4 = 3\sin x$

(iii) $\frac{d^2x}{dy^2} + a^2x = 0$ **(iv)** $\left(\frac{dy}{dx}\right)^2 - 3\frac{d^3y}{dx^3} + 7\frac{d^2y}{dx^2} + 4\left(\frac{dy}{dx}\right) - \log x = 0$

(v) $\sqrt{1+\left(\frac{dy}{dx}\right)^2} = 4x$ **(vi)** $\left[1+\left(\frac{dy}{dx}\right)^2\right]^{\frac{2}{3}} = \frac{d^2y}{dx^2}$

(vii) $\frac{d^2y}{dx^2} - \sqrt{\frac{dy}{dx}} = 0$ **(viii)** $\sqrt{1+x^2} = \frac{dy}{dx}$

Solution :

The order and the degree respectively are,

(i) 1 ; 3	(ii) 2 ; 3	(iii) 2 ; 1	(iv) 3 ; 1
(v) 1 ; 2	(vi) 2 ; 3	(vii) 2 ; 2	(viii) 1 ; 1

Note

Before ascertaining the order and degree in (v), (vi) & (vii) we made the differential coefficients free from radicals and fractional exponents.

6.1.2 Family of curves

Sometimes a family of curves can be represented by a single equation. In such a case the equation contains an arbitrary constant c. By assigning different values for c, we get a family of curves. In this case c is called the **parameter** or **arbitrary constant** of the family.

Examples

(i) $y = mx$ represents the equation of a family of straight lines through the origin , where m is the parameter.

(ii) $x^2 + y^2 = a^2$ represents the equation of family of concentric circles having the origin as centre, where a is the parameter.

(iii) $y = mx + c$ represents the equation of a family of straight lines in a plane, where m and c are parameters.

6.1.3 Formation of Ordinary Differential Equation

Consider the equation $y = mx + \lambda$ ---------(1)
where m is a constant and λ is the parameter.

This represents one parameter family of parallel straight lines having same slope *m*.

Differentiating (1) with respect to x, we get, $\frac{dy}{dx} = m$

This is the differential equation representing the above family of straight lines.

Similarly for the equation $y = Ae^{5x}$, we form the differential equation $\frac{dy}{dx} = 5y$ by eliminating the arbitrary constant A.

The above functions represent one-parameter families. Each family has a differential equation. To obtain this differential equation differentiate the equation of the family with respect to x, treating the parameter as a constant. If the derived equation is free from parameter then the derived equation is the differential equation of the family.

Note

(i) The differential equation of a two parameter family is obtained by differentiating the equation of the family twice and by eliminating the parameters.

(ii) In general, the order of the differential equation to be formed is equal to the number of arbitrary constants present in the equation of the family of curves.

Example 2

Form the differential equation of the family of curves $y = A\cos 5x + B\sin 5x$ where A and B are parameters.

Solution :

Given $y = A\cos 5x + B\sin 5x$

$$\frac{dy}{dx} = -5A\sin 5x + 5B\cos 5x$$

$$\frac{d^2y}{dx^2} = -25\ (\text{A} \cos 5x) - 25\ (\text{B} \sin 5x) = -25y$$

$$\therefore \quad \frac{d^2y}{dx^2} + 25y = 0.$$

Example 3

Form the differential equation of the family of curves $y = ae^{3x} + be^{x}$ where a and b are parameters.

Solution :

$$y = ae^{3x} + be^{x} \qquad \text{------------(1)}$$

$$\frac{dy}{dx} = 3ae^{3x} + be^{x} \qquad \text{------------(2)}$$

$$\frac{d^2y}{dx^2} = 9ae^{3x} + be^{x} \qquad \text{------------(3)}$$

$$(2) - (1) \Rightarrow \frac{dy}{dx} - y = 2ae^{3x} \qquad \text{------------(4)}$$

$$(3) - (2) \Rightarrow \frac{d^2y}{dx^2} - \frac{dy}{dx} = 6ae^{3x} = 3\left(\frac{dy}{dx} - y\right) \qquad \text{[using (4)]}$$

$$\Rightarrow \frac{d^2y}{dx^2} - 4\frac{dy}{dx} + 3\text{y} = 0$$

Example 4

Find the differential equation of a family of curves given by $y = a \cos (mx + b)$, a and b being arbitrary constants.

Solution :

$$y = a \cos (mx + b) \qquad \text{------------(1)}$$

$$\frac{dy}{dx} = -ma \sin (mx + b)$$

$$\frac{d^2y}{dx^2} = -m^2a \cos (mx + b) = -m^2y \qquad \text{[using (1)]}$$

$$\therefore \quad \frac{d^2y}{dx^2} + m^2y = 0 \text{ is the required differential equation.}$$

Example 5

Find the differential equation by eliminating the arbitrary constants *a* and *b* from $y = a\ \tan x + b \sec x$.

Solution :

$$y = a \tan x + b \sec x$$

Multiplying both sides by $\cos x$ we get,

$$y \cos x = a \sin x + b$$

Differentiating with respect to x we get

$$y(-\sin x) + \frac{dy}{dx}\cos x = a \cos x$$

$$\Rightarrow \quad -y \tan x + \frac{dy}{dx} = a \qquad \text{-----------}(1)$$

Differentiating (1) with respect to x, we get

$$\frac{d^2y}{dx^2} - \frac{dy}{dx}\tan x - y\sec^2 x = 0$$

EXERCISE 6.1

1) Find the order and degree of the following :

(i) $x^2\frac{d^2y}{dx^2} - 3\frac{dy}{dx} + y = \cos x$ (ii) $\frac{d^3y}{dx^3} - 3\left(\frac{d^2y}{dx^2}\right)^2 + 5\frac{dy}{dx} = 0$

(iii) $\frac{d^2y}{dx^2} - \sqrt{\frac{dy}{dx}} = 0$ (iv) $\left(1+\frac{d^2y}{dx^2}\right)^{\frac{1}{2}} = \frac{dy}{dx}$

(v) $\left(1+\frac{dy}{dx}\right)^{\frac{1}{3}} = \frac{d^2y}{dx^2}$ (vi) $\sqrt{1+\frac{d^2y}{dx^2}} = x\frac{dy}{dx}$

(vii) $\left(\frac{d^2y}{dx^2}\right)^{\frac{3}{2}} = \left(\frac{dy}{dx}\right)^2$ (viii) $3\frac{d^2y}{dx^2} + 5\left(\frac{dy}{dx}\right)^3 - 3y = e^x$

(ix) $\frac{d^2y}{dx^2} = 0$ (x) $\left(\frac{d^2y}{dx^2}+1\right)^{\frac{2}{3}} = \left(\frac{dy}{dx}\right)^{\frac{1}{3}}$

2) Find the differential equation of the following

(i) $y = mx$ (ii) $y = cx - c + c^2$

(iii) $y = mx + \frac{a}{m}$, where m is arbitrary constant

(iv) $y = mx + c$ where m and c are arbitrary constants.

3) Form the differential equation of family of rectangular hyperbolas whose asymptotes are the coordinate axes.

4) Find the differential equation of all circles $x^2 + y^2 + 2gx = 0$ which pass through the origin and whose centres are on the x-axis.

5) Form the differential equation of $y^2 = 4a(x + a)$, where a is the parameter.

6) Find the differential equation of the family of curves $y = ae^{2x} + be^{3x}$ where a and b are parameters.

7) Form the differential equation for $y = a \cos 3x + b \sin 3x$ where a and b are parameters.

8) Form the diffrential equation of $y = ae^{bx}$ where a and b are the arbitrary constants.

9) Find the differential equation for the family of concentric circles $x^2 + y^2 = a^2$, a is the paramter.

6.2 FIRST ORDER DIFFERENTIAL EQUATIONS

6.2.1 Solution of a differential equation

A **solution** of a differential equation is an explicit or implicit relation between the variables which satisfies the given differential equation and does not contain any derivatives.

If the solution of a differential equation contains as many arbitrary constants of integration as its order, then the solution is said to be the **general solution** of the differential equation.

The solution obtained from the general solution by assigning particular values for the arbitrary constants, is said to be a **particular solution** of the differential equation.

For example,

Differential equation	General solutuion	Particular solution
(i) $\frac{dy}{dx} = \sec^2 x$	$y = \tan x + c$ (c is arbitrary constant)	$y = \tan x - 5$
(ii) $\frac{dy}{dx} = x^2 + 2x$	$y = \frac{x^3}{3} + x^2 + c$	$y = \frac{x^3}{3} + x^2 + 8$
(iii) $\frac{d^2y}{dx^2} - 9y = 0$	$y = Ae^{3x} + Be^{-3x}$	$y = 5e^{3x} - 7e^{-3x}$

6.2.2 Variables Separable

If it is possible to re-arrange the terms of the first order and first degree differential equation in two groups, each containing only one variable, the variables are said to be separable.

When variables are separated, the differentail equation takes the form $f(x)\,dx + g(y)\,dy = 0$ in which $f(x)$ is a function of x only and $g(y)$ is a function of y only.

Then the general solution is

$$\int f(x)\,dx + \int g(y)\,dy = c \qquad (c \text{ is a constant of integration})$$

For example, consider $x\frac{dy}{dx} - y = 0$

$$x\frac{dy}{dx} = y \quad \Rightarrow \quad \frac{dy}{y} = \frac{dx}{x} \qquad \text{(separating the variables)}$$

$\Rightarrow \quad \int \frac{dy}{y} = \int \frac{dx}{x} + k$ where k is a constant of integration.

$\Rightarrow \quad \log y = \log x + k.$

The value of k varies from $-\infty$ to ∞.

This general solution can be expressed in a more convenient form by assuming the constant of integration to be $\log c$. This is possible because $\log c$ also can take all values between $-\infty$ and ∞ as k does. By this assumption, the general solution takes the form

$$\log y - \log x = \log c \Rightarrow \log\left(\frac{y}{x}\right) = \log c$$

i.e. $\frac{y}{x} = c \quad \Rightarrow \quad y = cx$

which is an elegant form of the solution of the differential equation.

Note

(i) When y is absent, the general form of first order linear differential equation reduces to $\frac{dy}{dx} = f(x)$ and therefore the solution is $y = \int f(x)\, dx + c$

(ii) When x is absent , it reduces to $\frac{dy}{dx} = g(y)$

and in this case, the solution is $\int \frac{dy}{g(y)} = \int dx + c$

Example 6

Solve the differential equation $xdy + ydx = 0$

Solution :

$xdy + ydx = 0$, dividing by xy we get

$$\frac{dy}{y} + \frac{dx}{x} = 0. \quad \text{Then } \int \frac{dy}{y} + \int \frac{dx}{x} = c_1$$

$\therefore$ $\log y + \log x = \log c \quad \Rightarrow \quad xy = c$

Note

(i) $xdy + ydx = 0 \quad \Rightarrow d(xy) = 0 \Rightarrow xy = c$, a constant.

(ii) $d(\frac{x}{y}) = \frac{ydx - xdy}{y^2} \quad \therefore \int \frac{ydx - xdy}{y^2} = \int d\,(\frac{x}{y}) + c = \frac{x}{y} + c$

Example 7

Solve $\frac{dy}{dx} = e^{3x+y}$

Solution :

$$\frac{dy}{dx} = e^{3x} e^{y} \quad \Rightarrow \quad \frac{dy}{e^{y}} = e^{3x}\, dx$$

$$\int e^{-y}\, dy = \int e^{3x}\, dx + c$$

$\Rightarrow \quad -e^{-y} = \frac{e^{3x}}{3} + c \quad \Rightarrow \quad \frac{e^{3x}}{3} + e^{-y} = \text{c}$

Example 8

Solve $(x^2 - ay)\,dx = (ax - y^2)dy$

Solution :

Writing the equation as

$x^2dx + y^2dy = a(xdy + ydx)$

$\Rightarrow \quad x^2dx + y^2dy = a\, d(xy)$

$\therefore \quad \int x^2\, dx + \int y^2\, dy = \text{a}\int d\ (xy) + \text{c}$

$\Rightarrow \quad \dfrac{x^3}{3} + \dfrac{y^3}{3} = a(xy) + c$

Hence the general solution is $x^3 + y^3 = 3axy + c$

Example 9

Solve $y(1+x^2)^{\frac{1}{2}}\, dy + x\sqrt{1+y^2}\, dx = 0$

Solution :

$y\sqrt{1+x^2}\ dy + x\sqrt{1+y^2}\ dx = 0$ [dividing *by* $\sqrt{1+x^2}\ \sqrt{1+y^2}$]

$\Rightarrow \quad \dfrac{y}{\sqrt{1+y^2}}\, dy + \dfrac{x}{\sqrt{1+x^2}}\, dx = 0$

$\therefore \quad \int\dfrac{y}{\sqrt{1+y^2}}\, dy + \int\dfrac{x}{\sqrt{1+x^2}}\, dx = \text{c}_1$

$\therefore \quad \dfrac{1}{2}\int t^{-\frac{1}{2}}\, dt + \dfrac{1}{2}\int u^{-\frac{1}{2}}\, du = \text{c}$

Put $1+y^2 = t$
$2ydy = dt$
put $1+x^2 = u$
$2xdx = du$

i.e. $\quad t^{\frac{1}{2}} + u^{\frac{1}{2}} = c$ or $\sqrt{1+y^2} + \sqrt{1+x^2} = c$

Note : This problem can also be solved by using

$$\int [\, f(x)]^n\, f'(x)\, dx = \frac{[\,f(x)]^{n+1}}{n+1}$$

Example 10

Solve $(\sin x + \cos x)\, dy + (\cos x - \sin x)\, dx = 0$

Solution :

The given equation can be written as

$$dy + \frac{\cos x - \sin x}{\sin x + \cos x}\,dx = 0$$

$$\Rightarrow \int dy + \int \frac{\cos x - \sin x}{\sin x + \cos x}\,dx = c$$

$$\Rightarrow \quad y + \log(\sin x + \cos x) = c$$

Example 11

Solve $x\dfrac{dy}{dx} + \cos y = 0$**, given** $y = \dfrac{\pi}{4}$ **when** $x = \sqrt{2}$

Solution :

$x\,dy = -\cos y\,dx$

$\therefore \quad \int \sec y\,dy = -\int \frac{dx}{x} + k$, where k is a constant of integration.

$\log(\sec y + \tan y) + \log x = \log c$, where $k = \log c$

or $\quad x(\sec y + \tan y) = c.$

When $x = \sqrt{2}$, $y = \frac{\pi}{4}$, we have

$\sqrt{2}\,(\sec\frac{\pi}{4} + \tan\frac{\pi}{4}) = c$ or $c = \sqrt{2}\,(\sqrt{2} + 1) = 2 + \sqrt{2}$

$\therefore \quad$ The particular solution is $x\,(\sec y + \tan y) = 2 + \sqrt{2}$

Example 12

The marginal cost function for producing x units is MC = 23+16x - 3x^2 and the total cost for producing 1 unit is Rs.40. Find the total cost function and the average cost function.

Solution :

Let C(x) be the total cost function where x is the number of units of output. Then

$$\frac{dC}{dx} = MC = 23 + 16x - 3x^2$$

$\therefore \int \frac{dC}{dx}\,dx = \int (23+16x - 3x^2)dx + k$

$C = 23x + 8x^2 - x^3 + k$, where k is a constant

At $x = 1$, $C(x) = 40$ (given)

$23(1) + 8(1)^2 - 1^3 + k = 40 \Rightarrow k = 10$

$\therefore$ Total cost function $C(x) = 23x + 8x^2 - x^3 + 10$

Average cost function $= \frac{\text{Total cost function}}{x}$

$= \frac{23x+8x^2-x^3+10}{x}$

Average cost function $= 23 + 8x - x^2 + \frac{10}{x}$

Example 13

What is the general form of the demand equation which has a constant elasticity of -1 ?

Solution :

Let x be the quantity demanded at price p. Then the elasticity is given by

$\eta_d = \frac{-p}{x}\frac{dx}{dp}$

Given $\frac{-p}{x}\frac{dx}{dp} = -1 \Rightarrow \frac{dx}{x} = \frac{dp}{p} \Rightarrow \int \frac{dx}{x} = \int \frac{dp}{p} + \log k$

$\Rightarrow \log x = \log p + \log k$, where k is a constant.

$\Rightarrow \log x = \log kp \Rightarrow x = kp \Rightarrow p = \frac{1}{k}x$

i.e. $p = cx$, where $c = \frac{1}{k}$ is a constant

Example 14

The relationship between the cost of operating a warehouse and the number of units of items stored in it is given by $\frac{dC}{dx} = ax + b$, where C is the monthly cost of operating the warehouse and x is the number of units of items in storage. Find C as a function of x if $C = C_0$ when $x = 0$.

Solution :

Given $\frac{dC}{dx} = ax + b \quad \therefore \; dC = (ax + b)\, dx$

$\int dC = \int (ax + b)\, dx + k$, ($k$ is a constant)

$\Rightarrow \quad C = \frac{ax^2}{2} + bx + k,$

when $x = 0$, $C = C_0 \;\therefore (1) \Rightarrow \quad C_0 = \frac{a}{2}(0) + b(0) + k$

$\Rightarrow \; k = C_0$

Hence the cost function is given by

$C = \frac{a}{2}x^2 + bx + C_0$

Example 15

The slope of a curve at any point is the reciprocal of twice the ordinate of the point. The curve also passes through the point (4, 3). Find the equation of the curve.

Solution :

Slope of the curve at any point P(x, y) is the slope of the tangent at P(x, y)

$\therefore \; \frac{dy}{dx} = \frac{1}{2y} \quad \Rightarrow \; 2y\,dy = dx$

$\int 2y\, dy = \int dx + c \quad \Rightarrow \quad y^2 = x + c$

Since the curve passes through (4, 3), we have

$9 = 4 + c \;\Rightarrow\; c = 5$

$\therefore$ Equation of the curve is $y^2 = x + 5$

EXERCISE 6.2

1) Solve (i) $\frac{dy}{dx} + \sqrt{\frac{1-y^2}{1-x^2}} = 0$ (ii) $\frac{dy}{dx} = \frac{1+y^2}{1+x^2}$

(iii) $\frac{dy}{dx} = \frac{y+2}{x-1}$ (iv) $x\sqrt{1+y^2} + y\sqrt{1+x^2}\,\frac{dy}{dx} = 0$

2) Solve (i) $\frac{dy}{dx} = e^{2x-y} + x^3 e^{-y}$ (ii) $(1-e^x)\sec^2 y\, dy + 3e^x \tan y\, dx = 0$

3) Solve (i) $\frac{dy}{dx} = 2xy + 2ax$ (ii) $x(y^2+1)\, dx + y(x^2+1)\, dy = 0$

(iii) $(x^2 - yx^2)\frac{dy}{dx} + y^2 + xy^2 = 0$

4) Solve (i) $xdy + ydx + 4\sqrt{1-x^2y^2}\, dx = 0$ (ii) $ydx - xdy + 3x^2y^2e^{x^3}dx = 0$

5) Solve (i) $\frac{dy}{dx} = \frac{y^2+4y+5}{x^2-2x+2}$ (ii) $\frac{dy}{dx} + \frac{y^2+y+1}{x^2+x+1} = 0$

6) Find the equation of the curve whose slope at the point (x, y) is $3x^2 + 2$, if the curve passes through the point (1, -1)

7) The gradient of the curve at any point (x, y) is proportional to its abscissa. Find the equation of the curve if it passes through the points (0, 0) and (1, 1)

8) Solve : $\sin^{-1}x\, dy + \frac{y}{\sqrt{1-x^2}}\, dx = 0$, given that $y = 2$ when $x = \frac{1}{2}$

9) What is the general form of the demand equation which has an elasticity of $-n$?

10) What is the general form of the demand equation which has an elasticity of $-\frac{1}{2}$?

11) The marginal cost function for producing x units is MC = e^{3x+7}. Find the total cost function and the average cost function, given that the cost is zero when there is no production.

6.2.3 Homogeneous differential equations

A differential equation in x and y is said to be **homogeneous** if it can be defined in the form

$\frac{dy}{dx} = \frac{f(x,y)}{g(x,y)}$ where $f(x, y)$ and $g(x, y)$ are homogeneous functions of the same degree in x and y.

$$\frac{dy}{dx} = \frac{xy}{x^2+y^2}, \quad \frac{dy}{dx} = \frac{x^2+y^2}{2xy}, \quad \frac{dy}{dx} = \frac{x^2y}{x^3+y^3}$$

and $\quad \frac{dy}{dx} = \frac{\sqrt{x^2 - y^2} + y}{x}$

are some examples of first order homogeneous differential equations.

6.2.4 Solving first order homogeneous differential equations

If we put $y = vx$ then $\frac{dy}{dx} = v + x\frac{dv}{dx}$ and the differential equation reduces to variables sepaerable form. The solution is got by replacing $\frac{y}{x}$ for v after the integration is over.

Example 16

Solve the differential equation $(x^2 + y^2)dx = 2xydy$

Solution :

The given differential equation can be written as

$$\frac{dy}{dx} = \frac{x^2 + y^2}{2xy} \qquad \text{-------------} (1)$$

This is a homogeneous differential equation

Put $y = vx \quad \therefore \frac{dy}{dx} = v + x\frac{dv}{dx}$ ------------- (2)

Substituting (2) in (1) we get,

$$v + x\frac{dv}{dx} = \frac{x^2 + v^2x^2}{2x(vx)} = \frac{1+v^2}{2v}$$

$$x\frac{dv}{dx} = \frac{1+v^2}{2v} - v \Rightarrow x\frac{dv}{dx} = \frac{1-v^2}{2v}$$

Now, separating the variables,

$$\frac{2v}{1-v^2}dv = \frac{dx}{x} \quad \text{or} \quad \int\frac{-2v}{1-v^2} = \int\frac{-dx}{x} + c_1$$

$\log(1 - v^2) = -\log x + \log c \quad [\int\frac{f'(x)}{f(x)}dx = \log f(x)]$

or $\quad \log(1 - v^2) + \log x = \log c \Rightarrow (1 - v^2)x = c$

Replacing v by $\frac{y}{x}$, we get

$$\left(1-\frac{y^2}{x^2}\right)x = c \quad \text{or} \quad x^2 - y^2 = cx$$

Example 17

Solve : $(x^3 + y^3)dx = (x^2y + xy^2)\, dy$

Solution :

The given equation can be written as

$$\frac{dy}{dx} = \frac{x^3+y^3}{x^2y+xy^2} \qquad \text{-------------- (1)}$$

Put $y = vx \;\therefore\; \frac{dy}{dx} = v + x\frac{dv}{dx}$

$\Rightarrow \quad v + x\frac{dv}{dx} = \frac{1+v^3}{v+v^2}$

$\Rightarrow \quad x\frac{dv}{dx} = \frac{1+v^3}{v+v^2} - v = \frac{1-v^2}{v(v+1)} = \frac{(1-v)(1+v)}{v(v+1)}$

$$\int\frac{v}{1-v}\,dv = \int\frac{1}{x}\,dx + c$$

$\Rightarrow \quad \int\frac{-v}{1-v}\,dv = -\int\frac{1}{x}\,dx + c \quad \text{or} \quad \int\frac{(1-v)-1}{1-v}\,dv = -\int\frac{1}{x}\,dx + c$

$\Rightarrow \quad \int\left(1+\frac{(-1)}{1-v}\right)dv = -\int\frac{1}{x}\,dx + c$

$\therefore \quad v + \log(1-v) = -\log x + c$

Replacing v by $\frac{y}{x}$, we get $\frac{y}{x} + \log(x - \text{y}) = c$

Example 18

Solve $x\frac{dy}{dx} = y - \sqrt{x^2+y^2}$

Solution :

Now, $\frac{dy}{dx} = \frac{y-\sqrt{x^2+y^2}}{x}$ ------------(1)

Put $y = vx \quad \therefore \frac{dy}{dx} = v + x\frac{dv}{dx}$

(1) $\Rightarrow$ $v + x\dfrac{dv}{dx} = \dfrac{vx-\sqrt{x^2+v^2x^2}}{x} = v - \sqrt{1+v^2}$

$\therefore$ $x\dfrac{dv}{dx} = -\sqrt{1+v^2}$ or $= \dfrac{dv}{\sqrt{1+v^2}} = -\dfrac{dx}{x}$

$\Rightarrow$ $\displaystyle\int\frac{dv}{\sqrt{1+v^2}} = -\int\frac{dx}{x} + c_1$

$\Rightarrow$ $\log (v + \sqrt{1+v^2}) = -\log x + \log c$

$\log x\ (v + \sqrt{1+v^2}) = \log c$

or $x\ (v + \sqrt{1+v^2}) = c$

i.e. $x\left[\dfrac{y}{x}+\sqrt{1+\dfrac{y^2}{x^2}}\right] = c$ or $y + \sqrt{x^2+y^2} = c$

Example 19

Solve $(x + y)\, dy + (x - y)dx = 0$

Solution :

The equation is $\dfrac{dy}{dx} = -\left(\dfrac{x-y}{x+y}\right)$ ------------ (1)

Put $y = vx$ $\therefore$ $\dfrac{dy}{dx} = v + x\dfrac{dv}{dx}$

we get $v + x\dfrac{dv}{dx} = -\dfrac{x-vx}{x+vx}$ or $v + x\dfrac{dv}{dx} = -\dfrac{1-v}{1+v}$

i.e. $x\dfrac{dv}{dx} = -\left(\dfrac{1-v}{1+v}+v\right)$ or $x\dfrac{dv}{dx} = \dfrac{-(1-v+v+v^2)}{1+v}$

$\therefore$ $\dfrac{1+v}{1+v^2}dv = -\dfrac{1}{x}dx$ or

$\displaystyle\int\frac{dv}{1+v^2}dv + \frac{1}{2}\int\frac{2v}{1+v^2}dv = \int-\frac{1}{x}\ dx + c$

$\tan^{-1}v + \dfrac{1}{2}\log(1 + v^2) = -\log x + c$

i.e. $\tan^{-1}\left(\dfrac{y}{x}\right) + \dfrac{1}{2}\log\left(\dfrac{x^2+y^2}{x^2}\right) = -\log x + c$

$$\tan^{-1}\left(\frac{y}{x}\right) + \frac{1}{2}\log(x^2+y^2) - \frac{1}{2}\log x^2 = -\log x + c$$

i.e. $$\tan^{-1}\left(\frac{y}{x}\right) + \frac{1}{2}\log(x^2+y^2) = c$$

Example 20

The net profit p and quantity x satisfy the differential equation $\frac{dp}{dx} = \frac{2p^3 - x^3}{3xp^2}$.

Find the relationship between the net profit and demand given that $p = 20$ when $x = 10$.

Solution :

$$\frac{dp}{dx} = \frac{2p^3 - x^3}{3xp^2} \qquad \text{-------------(1)}$$

is a differential equation in x and p of homogeneous type

Put $p = vx \quad \therefore \quad \frac{dp}{dx} = v + x\frac{dv}{dx}$

(1) $\Rightarrow \quad v + x\frac{dv}{dx} = \frac{2v^3-1}{3v^2} \Rightarrow x\frac{dv}{dx} = \frac{2v^3-1}{3v^2} - v$

$$\Rightarrow \quad x\frac{dv}{dx} = -\left[\frac{1+v^3}{3v^2}\right]$$

$$\frac{3v^2}{1+v^3}dv = -\frac{dx}{x} \quad \therefore \quad \int\frac{3v^2}{1+v^3}dv = -\int\frac{dx}{x} = k$$

$\Rightarrow \quad \log(1+v^3) = -\log x + \log k$, where k is a constant

$\log(1+v^3) = \log\frac{k}{x}$ i.e. $1 + v^3 = \frac{k}{x}$

Replacing v *by* $\frac{p}{x}$, we get

$\Rightarrow \quad x^3 + p^3 = kx^2$

But when $x = 10$, it is given that $p = 20$

$\therefore \quad (10)^3 + (20)^3 = k(10)^2 \Rightarrow k = 90 \quad \therefore x^3+p^3 = 90x^2$

$p^3 = x^2(90 - x)$ is the required relationship.

Example 21

The rate of increase in the cost C of ordering and holding as the size q of the order increases is given by the differential equation

$\dfrac{dC}{dq} = \dfrac{C^2 + 2Cq}{q^2}$**. Find the relationship between C and q if C = 1 when q = 1.**

Solution :

$$\frac{dC}{dq} = \frac{C^2 + 2Cq}{q^2} \qquad \text{------------(1)}$$

This is a homogeneous equation in C and q

Put $C = vq \quad \therefore \quad \dfrac{dC}{dq} = v + q\dfrac{dv}{dq}$

$(1) \Rightarrow v + q\dfrac{dv}{dq} = \dfrac{v^2q^2 + 2vq^2}{q^2} = v^2 + 2v$

$\Rightarrow \quad q\dfrac{dv}{dq} = v^2 + v = v\,(v + 1) \quad \Rightarrow \quad \dfrac{dv}{v(v+1)} = \dfrac{dq}{q}$

$\Rightarrow \quad \displaystyle\int \frac{(v+1)-v}{v(v+1)}\,dv = \int \frac{dq}{q} + k$, k is a constant

$\Rightarrow \quad \displaystyle\int \frac{dv}{v} - \int \frac{dv}{v+1} = \int \frac{dq}{q} + \log k,$

$\Rightarrow \quad \log v - \log (v + 1) = \log q + \log k$

$\Rightarrow \quad \log \dfrac{v}{v+1} = \log qk \quad$ or $\quad \dfrac{v}{v+1} = kq$

Replacing v by $\dfrac{C}{q}$ we get, $C = kq(C + q)$

when C = 1 and q = 1

$C = kq(C + q) \quad \Rightarrow \quad k = \dfrac{1}{2}$

$\therefore \quad C = \dfrac{q(C+q)}{2}$ is the relation between C and q

Example 22

The total cost of production y and the level of output x are related to the marginal cost of production by the equation $(6x^2 + 2y^2)\, dx - (x^2 + 4xy)\, dy = 0$. What is the relation between total cost and output if $y = 2$ when $x = 1$?

Solution :

Given $(6x^2 + 2y^2)\, dx = (x^2 + 4xy)\, dy$

$\therefore \dfrac{dy}{dx} = \dfrac{6x^2+2y^2}{x^2+4xy}$ ------------(1)

is a homogeneous equation in x and y.

Put $y = vx \therefore \dfrac{dy}{dx} = v + x\dfrac{dv}{dx}$

(1) $\Rightarrow$ $v + x\dfrac{dv}{dx} = \dfrac{6x^2+2y^2}{x^2+4xy}$ or $\dfrac{1+4v}{6-v-2v^2}dv = \dfrac{1}{x}dx$

$\therefore$ $-\int\dfrac{-1-4v}{6-v-2v^2}dv = \int\dfrac{1}{x}dx + k$, where k is a constant

$\Rightarrow$ $-\log(6-v-2v^2) = \log x + \log k = \log kx$

$\Rightarrow$ $\dfrac{1}{6-v-2v^2} = kx$

$\Rightarrow$ $x = c(6x^2 - xy - 2y^2)$ where $c = \dfrac{1}{k}$ and $v = \dfrac{y}{x}$

when $x = 1$ and $y = 2$, $1 = c(6 - 2 - 8) \Rightarrow c = -\dfrac{1}{4}$

$\Rightarrow$ $4x = (2y^2 + xy - 6x^2)$

EXERCISE 6.3

1) Solve the following differential equations

(i) $\dfrac{dy}{dx} = \dfrac{y}{x} - \dfrac{y^2}{x^2}$ (ii) $2\dfrac{dy}{dx} = \dfrac{y}{x} - \dfrac{y^2}{x^2}$

(iii) $\dfrac{dy}{dx} = \dfrac{xy-2y^2}{x^2-3xy}$ (iv) $x(y-x)\dfrac{dy}{dx} = y^2$

(v) $\dfrac{dy}{dx} = \dfrac{y^2-2xy}{x^2-2xy}$ (vi) $\dfrac{dy}{dx} = \dfrac{xy}{x^2-y^2}$

(vii) $(x + y)^2\, dx = 2x^2\, dy$ (viii) $x\frac{dy}{dx} = y + \sqrt{x^2 + y^2}$

2) The rate of increase in the cost C of ordering and holding as the size q of the order increases is given by the differential eqation $\frac{dC}{dq} = \frac{C^2 + q^2}{2Cq}$. Find the relatinship between C and q if $C = 4$ when $q = 2$.

3) The total cost of production y and the level of output x are related to the marginal cost of production by the equation $\frac{dy}{dx} = \frac{24x^2 - y^2}{xy}$. What is the total cost function if $y = 4$ when $x = 2$?

6.2.5 First order linear differential equation

A first order differential equation is said to be **linear** when the dependent variable and its derivatives occur only in first degree and no product of these occur.

An equation of the form $\frac{dy}{dx} + Py = Q$,

where P and Q are functions of x only, is called a **first order linear differential equation.**

For example,

(i) $\frac{dy}{dx} + 3y = x^3$; here $P = 3$, $Q = x^3$

(ii) $\frac{dy}{dx} + y \tan x = \cos x$, $P = \tan x$, $Q = \cos x$

(iii) $\frac{dy}{dx}x - 3y = xe^x$, $P = -\frac{3}{x}$, $Q = e^x$

(iv) $(1 + x^2)\frac{dy}{dx} + xy = (1+x^2)^3$, $P = \frac{x}{1+x^2}$, $Q = (1 + x^2)^2$

are first order linear differential equations.

6.2.6 Integrating factor (I.F)

A given differential equation may not be integrable as such. But it may become integrable when it is multiplied by a function.

Such a function is called the **integrating factor (I.F).** Hence an integrating factor is one which changes a differential equation into one which is directly integrable.

Let us show that $e^{\int \mathrm{P}dx}$ is the integrating factor

for $\frac{dy}{dx} + \mathrm{P}y = \mathrm{Q}$ ---------(1)

where P and Q are function of x.

$$\text{Now, } \frac{d}{dx}(ye^{\int \mathrm{P}dx}) = \frac{dy}{dx}e^{\int \mathrm{P}dx} + y\frac{d}{dx}(e^{\int \mathrm{P}dx})$$

$$= \frac{dy}{dx}e^{\int \mathrm{P}dx} + ye^{\int \mathrm{P}dx}\frac{d}{dx}\int \mathrm{P}dx$$

$$= \frac{dy}{dx}e^{\int \mathrm{P}dx} + ye^{\int \mathrm{P}dx}\,\mathrm{P} = (\frac{dy}{dx} + \mathrm{P}y)e^{\int \mathrm{P}dx}$$

When (1) is multiplied by $e^{\int \mathrm{P}dx}$,

it becomes $(\frac{dy}{dx} + \mathrm{P}y)\, e^{\int \mathrm{P}dx} = \mathrm{Q}e^{\int \mathrm{P}dx}$

$\Rightarrow \frac{d}{dx}(ye^{\int \mathrm{P}dx}) = \mathrm{Q}\, e^{\int \mathrm{P}dx}$

Integrating this, we have

$ye^{\int \mathrm{P}dx} = \int \mathrm{Q}\, e^{\int \mathrm{P}dx}\, dx + \mathrm{c}$ -------------(2)

So $e^{\int p dx}$ is the integrating factor of the differential equation.

Note

(i) $e^{\log f(x)} = f(x)$ when $f(x) > 0$

(ii) If Q = 0 in $\frac{dy}{dx} + \mathrm{P}y = \mathrm{Q}$, then the general solution is y (I.F) = c, where c is a constant.

(iii) For the differential equation $\frac{dx}{dy} + \mathrm{P}x = \mathrm{Q}$ where P and Q are functions of y alone, the **(I.F)** is $e^{\int \mathrm{P}dy}$ and the solution is

$$x\,(\text{I.F}) = \int Q\,(\text{I.F})\, dy + c$$

Example 23

Solve the equation $(1 - x^2)\dfrac{dy}{dx} - xy = 1$

Solution :

The given equation is $(1-x^2)\dfrac{dy}{dx} - xy = 1$

$\Rightarrow \dfrac{dy}{dx} - \dfrac{x}{1-x^2}\,y = \dfrac{1}{1-x^2}$

This is of the form $\dfrac{dy}{dx} + \text{P}y = \text{Q}$,

where $\text{P} = \dfrac{-x}{1-x^2}$; $\text{Q} = \dfrac{1}{1-x^2}$

$\text{I.F} = e^{\int \text{P}dx} = e^{\int \frac{-x}{1-x^2}dx} = \sqrt{1-x^2}$

The general solution is,

$y\,(\text{I.F}) = \int \text{Q}\,(\text{I.F})dx + c$

$y\sqrt{1-x^2} = \int \dfrac{1}{1-x^2}\sqrt{1-x^2}\,dx + c$

$= \int \dfrac{dx}{\sqrt{1-x^2}} + c$

$y\sqrt{1-x^2} = \sin^{-1}x + c$

Example 24

Solve $\dfrac{dy}{dx} + ay = e^x$ **(where** $a \neq -1$**)**

Solution :

The given equation is of the form $\dfrac{dy}{dx} + \text{P}y = \text{Q}$

Here $\text{P} = a$; $\text{Q} = e^x$

$\therefore$ $\text{I.F} = e^{\int \text{P}dx} = e^{ax}$

The general solution is

$y\,(\text{I.F}) = \int \text{Q}\,(\text{I.F})dx + c$

$$\Rightarrow \quad y\, e^{ax} = \int e^{x} e^{ax}\, dx + c = \int e^{(a+1)x}\, dx + c$$

$$y\, e^{ax} = \frac{e^{(a+1)x}}{a+1} + c$$

Example 25

Solve $\cos x \dfrac{dy}{dx} + y \sin x = 1$

Solution :

The given equation can be reduced to

$$\frac{dy}{dx} + y\frac{\sin x}{\cos x} = \frac{1}{\cos x} \text{ or } \frac{dy}{dx} + y \tan x = \sec x$$

Here P = $\tan x$; Q = $\sec x$

$$\text{I.F} = e^{\int \tan x\, dx} = e^{\log \sec x} = \sec x$$

The general solution is

$$y\,(\text{I.F}) = \int \text{Q}\,(\text{I. F}))\, dx + c$$

$$y \sec x = \int \sec^2 x\, dx + c$$

$$\therefore \quad y \sec x = \tan x + c$$

Example 26

A bank pays interest by treating the annual interest as the instantaneous rate of change of the principal. A man invests Rs.50,000 in the bank deposit which accrues interest, 6.5% per year compounded continuously. How much will he get after 10 years? (Given : $e^{.65}$ =1.9155)

Solution :

Let P(t) denotes the amount of money in the account at time t. Then the differential equation governing the growth of money is

$$\frac{dP}{dt} = \frac{6.5}{100}P = 0.065P \Rightarrow \int \frac{dP}{P} = \int (0.065)\, dt + c$$

$$\log_e P = 0.065t + c \quad \therefore \quad P = e^{0.065t}\, e^{c}$$

$P = c_1 e^{0.065t}$ -------------(1)

At $t = 0$, $P = 50000$.

$(1) \Rightarrow 50000 = c_1 e^0$ or $c_1 = 50000$

$\therefore \quad P = 50000\, e^{0.065t}$

At $t = 10$, $P = 50000\, e^{0.065 \times 10} = 50000\, e^{0.65}$

$= 50000 \times (1.9155) = \text{Rs.}95{,}775.$

Example 27

Solve $\dfrac{dy}{dx} + y \cos x = \dfrac{1}{2} \sin 2x$

Solution :

Here $P = \cos x$; $Q = \dfrac{1}{2}\sin 2x$

$\int P\, dx = \int \cos x\, dx = \sin x$

$\text{I.F} = e^{\int P dx} = e^{\sin x}$

The general solution is

$$\begin{aligned} y\,(\text{I.F}) &= \int Q(\text{I.F})\, dx + c \\ &= \int \tfrac{1}{2}\sin 2x .\, e^{\sin x}\, dx + c \\ &= \int \sin x \cos x .\, e^{\sin x}\, dx + c \\ &= \int t\, e^t\, dt + c = e^t (t-1) + c \\ &= e^{\sin x}(\sin x - 1) + c \end{aligned}$$

Let $\sin x = t$,
then $\cos x\, dx = dt$

Example 28

A manufacturing company has found that the cost C of operating and maintaining the equipment is related to the length m of intervals between overhauls by the equation

$m^2 \dfrac{dC}{dm} + 2mC = 2$ and $C = 4$ when $m = 2$. Find the relationship between C and m.

Solution :

Given $m^2 \frac{dC}{dm} + 2mC = 2$ or $\frac{dC}{dm} + \frac{2C}{m} = \frac{2}{m^2}$

This is a first order linear differential equation of the form

$\frac{dy}{dx} + Py = Q$, where $P = \frac{2}{m}$; $Q = \frac{2}{m^2}$

$\text{I.F} = e^{\int P dm} = e^{\int \frac{2}{m} dm} = e^{\log m^2} = m^2$

General solution is

$$C\,(\text{I.F}) = \int Q\,(\text{I.F})\, dm + k \qquad \text{where } k \text{ is a constant}$$

$$Cm^2 = \int \frac{2}{m^2} m^2\, dm + k$$

$$Cm^2 = 2m + k$$

When $C = 4$ and $m = 2$, we have

$16 = 4 + k \quad \Rightarrow \quad k = 12$

$\therefore$ The relationship between C and m is

$Cm^2 = 2m + 12 = 2(m + 6)$

Example 29

Equipment maintenance and operating costs C are related to the overhaul interval x by the equation

$$x^2 \frac{dC}{dx} - 10xC = -10 \text{ with } C = C_0 \text{ when } x = x_0.$$

Find C as a function of x.

Solution :

$x^2 \frac{dC}{dx} - 10xC = -10$ or $\frac{dC}{dx} - \frac{10C}{x} = -\frac{10}{x^2}$

This is a first order linear differential equation.

$$P = -\frac{10}{x} \quad \text{and} \quad Q = -\frac{10}{x^2}$$

$$\int P\, dx = \int -\frac{10}{x} dx = -10 \log x = \log \left(\frac{1}{x^{10}}\right)$$

$$\text{I.F} = e^{\int P dx} = e^{\log\left(\frac{1}{x^{10}}\right)} = \frac{1}{x^{10}}$$

General solution is

$$C(\text{I.F}) = \int Q\,(\text{I.F})\,dx + k, \text{ where } k \text{ is a constant.}$$

$$\frac{C}{x^{10}} = \int \frac{-10}{x^2}\left(\frac{1}{x^{10}}\right)dx + k \quad \text{or} \quad \frac{C}{x^{10}} = \frac{10}{11}\left(\frac{1}{x^{11}}\right) + k$$

when $C = C_0 \quad x = x_0$

$$\frac{C_0}{x_0^{10}} = \frac{10}{11}\left(\frac{1}{x_0^{11}}\right) + k \Rightarrow k = \frac{C}{x_0^{10}} - \frac{10}{11x_0^{11}}$$

$\therefore$ The solution is

$$\frac{C}{x^{10}} = \frac{10}{11}\left(\frac{1}{x^{11}}\right) + \left[\frac{C}{x_0^{10}} - \frac{10}{11x_0^{11}}\right]$$

$$\Rightarrow \quad \frac{C}{x^{10}} - \frac{C}{x_0^{10}} = \frac{10}{11}\left(\frac{1}{x^{11}} - \frac{1}{x_0^{11}}\right)$$

EXERCISE 6.4

1) Solve the following differential equations

(i) $\frac{dy}{dx} + y \cot x = \operatorname{cosec} x$

(ii) $\frac{dy}{dx} - \sin 2x = y \cot x$

(iii) $\frac{dy}{dx} + y \cot x = \sin 2x$

(iv) $\frac{dy}{dx} + y \cot x = 4x \operatorname{cosec} x$, if $y = 0$ when $x = \frac{\pi}{2}$

(v) $\frac{dy}{dx} - 3y \cot x = \sin 2x$ and if $y = 2$ when $x = \frac{\pi}{2}$

(vi) $x\frac{dy}{dx} - 3y = x^2$

(vii) $\frac{dy}{dx} + \frac{2xy}{1+x^2} = \frac{1}{(1+x^2)^2}$ given that $y = 0$ when $x = 1$

(viii) $\frac{dy}{dx} - y \tan x = e^x \sec x$

(ix) $\log x \frac{dy}{dx} + \frac{y}{x} = \sin 2x$

2) A man plans to invest some amount in a small saving scheme with a guaranteed compound interest compounded continuously at the rate of 12 percent for 5 years. How much should he invest if he wants an amount of Rs.25000 at the end of 5 year period. ($e^{-0.6} = 0.5488$)

3) Equipment maintenance and operating cost C are related to the overhaul interval x by the equation $x^2 \frac{dC}{dx} - (b-1)Cx = -ba$, where a, b are constants and $C = C_0$ when $x = x_0$. Find the relationship between C and x.

4) The change in the cost of ordering and holding C as quantity q is given by $\frac{dC}{dq} = a - \frac{C}{q}$ where a is a constant.

Find C as a function of q if $C = C_0$ when $q = q_0$

6.3 SECOND ORDER LINEAR DIFFERENTIAL EQUATIONS WITH CONSTANT COEFFICIENTS

The general form of linear and second order differential equation with constant coefficients is

$$a\frac{d^2y}{dx^2} + b\frac{dy}{dx} + cy = f(x).$$

We shall consider the cases where

(i) $f(x) = 0$ and $f(x) = Ke^{\lambda x}$

For example,

(i) $3\frac{d^2y}{dx^2} - 5\frac{dy}{dx} + 6y = 0$ (or) $3y'' - 5y' + 6y = 0$

(ii) $\frac{d^2y}{dx^2} - 4\frac{dy}{dx} + 3y = e^{5x}$ (or) $(D^2 - 4D + 3)y = e^{5x}$

(iii) $\frac{d^2y}{dx^2} + \frac{dy}{dx} - y = 7$ (or) $(D^2 + D - 1)y = 7$

are second order linear differential equations.

6.3.1 Auxiliary equations and Complementary functions

For the differential equation, $a\frac{d^2y}{dx^2} + b\frac{dy}{dx} + cy = f(x)$, $am^2 + bm + c = 0$ is said to be the **auxiliary equation**. This is a quadratic equation in m. According to the nature of the roots m_1 and m_2 of auxiliary equation we write the complementary function (C.F) as follows.

Nature of roots	Complementary function
(i) Real and unequal ($m_1 \neq m_2$)	$Ae^{m_1x} + Be^{m_2x}$
(ii) Real and equal ($m_1 = m_2 = m$ say)	$(Ax + B)\,e^{mx}$
(iii) Complex roots ($\alpha \pm i\beta$)	$e^{\alpha x}(A\cos\beta x + B\sin\beta x)$
(In all the cases, A and B are arbitrary constants)	

6.3.2 Particular Integral (P.I)

Consider $(aD^2 + bD + c)y = e^{\lambda x}$

Let $f(D) = aD^2 + bD + c$

Case 1 : If $f(\lambda) \neq 0$ then λ is not a root of the auxiliary equation $f(m) = 0$.

Rule: $\text{P.I} = \frac{1}{f(D)}e^{\lambda x} = \frac{1}{f(\lambda)}e^{\lambda x}$.

Case 2 : If $f(\lambda) = 0$, λ satisfies the auxiliary equation $f(m) = 0$. Then we proceed as follows.

(i) Let the auxiliary equation have two distinct roots m_1 and m_2 and let $\lambda = m_1$.

Then $f(m) = a(m - m_1)(m - m_2) = a(m - \lambda)(m - m_2)$

Rule : $\text{P.I} = \frac{1}{a(D-\lambda)(D-m_2)}e^{\lambda x} = \frac{1}{a(\lambda - m_2)}\,xe^{\lambda x}$

(ii) Let the auxiliary equation have two equal roots each equal to λ. i.e. $m_1 = m_2 = \lambda$.

$\therefore\ f(m) = a\,(m - \lambda)^2$

Rule : $\therefore$ P.I $= \dfrac{1}{a(D-\lambda)^2} e^{\lambda x} = \dfrac{1}{a}\dfrac{x^2}{2!} e^{\lambda x}$

6.3.3 The General solution

The general solution of a second order linear differential equation is y = Complementary function (C.F) + Particular integral (P.I)

Example 30

Solve $3\dfrac{d^2y}{dx^2} - 5\dfrac{dy}{dx} + 2y = 0$

Solution :

The auxiliary equation is $3m^2 - 5m + 2 = 0$

$\Rightarrow\quad (3m - 2)(m - 1) = 0$

The roots are $m_1 = \dfrac{2}{3}$ and $m_2 = 1$ (Real and distinct)

$\therefore$ The complementary function is

$$\text{C.F} = Ae^{\frac{2}{3}x} + Be^{x}$$

The general solution is

$$y = Ae^{\frac{2}{3}x} + Be^{x}$$

Example 31

Solve $(16D^2 - 24D + 9)y = 0$

Solution :

The auxiliary equation is $16m^2 - 24m + 9 = 0$

$$(4m - 3)^2 = 0 \Rightarrow m = \frac{3}{4}, \frac{3}{4}$$

The roots are real and equal

$\therefore$ The C.F is $(Ax + B)e^{\frac{3}{4}x}$

The general solution is $y = (Ax + B)e^{\frac{3}{4}x}$

Example 32

Solve $(D^2 - 6D + 25)y = 0$

Solution :

The auxiliary equation is $m^2 - 6m + 25 = 0$

$$\Rightarrow \quad m = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

$$= \frac{6 \pm \sqrt{36 - 100}}{2} = \frac{6 \pm 8i}{2} = 3 \pm 4i$$

The roots are complex and is of the form

$\alpha \pm i\beta$ with $\alpha = 3$ and $\beta = 4$

C.F $= e^{\alpha x}(A \cos \beta x + B \sin \beta x)$

$= e^{3x}(A \cos 4x + B \sin 4x)$

The general solution is

$y = e^{3x}(A \cos 4x + B \sin 4x)$

Example 33

Solve $\frac{d^2y}{dx^2} - 5\frac{dy}{dx} + 6y = e^{5x}$

Solution :

The auxiliary equation is $m^2 - 5m + 6 = 0 \Rightarrow m = 3, 2$

$\therefore$ Complementary function C. F $= Ae^{3x} + Be^{2x}$

$$\text{P. I} = \frac{1}{D^2 - 5D + 6} e^{5x} = \frac{1}{6} e^{5x}$$

$\therefore$ The general solution is

y = C.F + P. I

$$y = Ae^{3x} + Be^{2x} + \frac{e^{5x}}{6}$$

Example 34

Solve $\frac{d^2y}{dx^2} + 4\frac{dy}{dx} + 4y = 2e^{-3x}$

Solution :

The auxiliary equation is $m^2 + 4m + 4 = 0 \Rightarrow m = -2, -2$

$\therefore$ Complementary function is C. F = $(Ax + B)e^{-2x}$

$$\text{P. I} = \frac{1}{D^2+4D+4}2e^{-3x}$$

$$= \frac{1}{(-3)^2+4(-3)+4}\ 2e^{-3x} = 2e^{-3x}$$

$\therefore$ The general solution is

y = C.F + P. I

$y = (Ax + B)\,e^{-2x} + 2e^{-3x}$

Example 35

Solve $\frac{d^2y}{dx^2} - 2\frac{dy}{dx} + 4y = 5 + 3e^{-x}$

Solution :

The auxiliary equation is $m^2 - 2m + 4 = 0$

$$\Rightarrow \quad m = \frac{2\pm\sqrt{4-16}}{2} = \frac{2\pm i2\sqrt{3}}{2} = 1 \pm i\sqrt{3}$$

C.F = $e^x(A\cos\sqrt{3}x + B\sin\sqrt{3}x)$

$$\text{P. I}_1 = \frac{1}{D^2-2D+4}5\,e^{0x} = \frac{1}{4}5\,e^{0x} = \frac{5}{4}$$

$$\text{P.I}_2 = \frac{1}{D^2-2D+4}3\,e^{-x}$$

$$= \frac{1}{(-1)^2-2(-1)+4}\ 3e^{-x} = \frac{3e^{-x}}{7}$$

$\therefore$ The general solution is

y = C.F + P. I$_1$ + P.I$_2$

$y = e^x(A\cos\sqrt{3}x + B\sin\sqrt{3}x) + \frac{5}{4} + \frac{3}{7}e^{-x}$

Example 36

Solve $(4D^2 - 8D + 3)y = e^{\frac{1}{2}x}$

Solution :

The auxiliary equation is $4m^2 - 8m + 3 = 0$

$m_1 = \frac{3}{2}, \; m_2 = \frac{1}{2}$

C.F $= Ae^{\frac{3}{2}x} + Be^{\frac{1}{2}x}$

$$\text{P. I} = \frac{1}{4D^2 - 8D + 3} e^{\frac{1}{2}x} = \frac{1}{4(D - \frac{3}{2})(D - \frac{1}{2})} e^{\frac{1}{2}x}$$

$$= \frac{1}{4(\frac{1}{2} - \frac{3}{2})(D - \frac{1}{2})} e^{\frac{1}{2}x} = \frac{x}{-4} e^{\frac{1}{2}x}$$

$\therefore$ The general solution is

$y = \text{C.F.} + \text{P. I}$

$$y = Ae^{\frac{3}{2}x} + Be^{\frac{1}{2}x} - \frac{x}{4} e^{\frac{x}{2}}$$

Example 37

Solve : $(D^2 + 10D + 25)y = \frac{5}{2} + e^{-5x}$

Solution :

The auxiliary equation is $m^2 + 10m + 25 = 0$

$\Rightarrow \quad (m + 5)^2 = 0$

$\Rightarrow \quad m = -5, \; -5$

$\therefore$ C.F $= (Ax + B)\, e^{-5x}$

$$\text{P. I}_1 = \frac{1}{D^2 + 10D + 25} \frac{5}{2} e^{0x} = \frac{1}{25} \left(\frac{5}{2}\right) = \frac{1}{10}$$

$$\text{P.I}_2 = \frac{1}{D^2 + 10D + 25} e^{-5x} = \frac{1}{(D + 5)^2} e^{-5x}$$

$$= \frac{x^2}{2!} e^{-5x} = \frac{x^2}{2} (e^{-5x})$$

$\therefore$ The general solution is

y = C.F + P. I_1 + P.I_2

$y = (Ax + B)\, e^{-5x} + \frac{1}{10} + \frac{x^2}{2}\, e^{-5x}$

Example 38

Suppose that the quantity demanded

$Q_d = 42 - 4p - 4\frac{dp}{dt} + \frac{d^2p}{dt^2}$ and quantity supplied $Q_s = -6 + 8p$ where p is the price. Find the equilibrium price for market clearance.

Solution :

For market clearance, the required condition is $Q_d = Q_s$.

$\Rightarrow \quad 42 - 4p - 4\frac{dp}{dt} + \frac{d^2p}{dt^2} = -6 + 8p$

$\Rightarrow \quad 48 - 12p - 4\frac{dp}{dt} + \frac{d^2p}{dt^2} = 0$

$\Rightarrow \quad \frac{d^2p}{dt^2} - 4\frac{dp}{dt} - 12p = -48$

The auxiliary equation is $m^2 - 4m - 12 = 0$

$\Rightarrow \quad m = 6\ ,\ -2$

C.F. = $Ae^{6t} + Be^{-2t}$

P. I = $\frac{1}{D^2 - 4D - 12}(-48)\, e^{0t} = \frac{1}{-12}(-48) = 4$

$\therefore$ The general solution is

p = C.F. + P. I

$p = Ae^{6t} + Be^{-2t} + 4$

EXERCISE 6.5

1) Solve :

(i) $\frac{d^2y}{dx^2} - 10\frac{dy}{dx} + 24y = 0$ (ii) $\frac{d^2y}{dx^2} + \frac{dy}{dx} = 0$

(iii) $\frac{d^2y}{dx^2} + 4y = 0$ (iv) $\frac{d^2y}{dx^2} + 4\frac{dy}{dx} + 4y = 0$

2) Solve :

(i) $(3D^2 + 7D - 6)y = 0$ (ii) $(4D^2 - 12D + 9)y = 0$

(iii) $(3D^2 - D + 1)y = 0$

3) Solve :

(i) $(D^2 - 13D + 12)\, y = e^{-2x} + 5e^x$

(ii) $(D^2 - 5D + 6)\, y = e^{-x} + 3e^{-2x}$

(iii) $(D^2 - 14D + 49)\, y = 3 + e^{7x}$

(iv) $(15D^2 - 2D - 1)\, y = e^{\frac{x}{3}}$

4) Suppose that $Q_d = 30 - 5P + 2\dfrac{dP}{dt} + \dfrac{d^2P}{dt^2}$ and $Q_s = 6 + 3P$. Find the equilibrium price for market clearance.

EXERCISE 6.6

Choose the correct answer

1) The differential equation of straight lines passing through the origin is

(a) $x\dfrac{dy}{dx} = y$ (b) $\dfrac{dy}{dx} = \dfrac{x}{y}$ (c) $\dfrac{dy}{dx} = 0$ (d) $x\dfrac{dy}{dx} = \dfrac{1}{y}$

2) The degree and order of the differential equation

$\dfrac{d^2y}{dx^2} - 6\sqrt{\dfrac{dy}{dx}} = 0$ are

(a) 2 and 1 (b) 1 and 2 (c) 2 and 2 (d) 1 and 1

3) The order and degree of the differential equation

$\left(\dfrac{dy}{dx}\right)^2 - 3\dfrac{d^3y}{dx^3} + 7\dfrac{d^2y}{dx^2} + \dfrac{dy}{dx} = x + \log x$ are

(a) 1 and 3 (b) 3 and 1 (c) 2 and 3 (d) 3 and 2

4) The order and degree of $\left[1 + \left(\dfrac{dy}{dx}\right)^2\right]^{\frac{2}{3}} = \dfrac{d^2y}{dx^2}$ are

(a) 3 and 2 (b) 2 and 3 (c) 3 and 3 (d) 2 and 2

5) The solution of $x\,dy + y\,dx = 0$ is

(a) $x + y = c$ (b) $x^2 + y^2 = c$ (c) $xy = c$ (d) $y = cx$

6) The solution of $x\,dx + y\,dy = 0$ is

(a) $x^2 + y^2 = c$ (b) $\frac{x}{y} = c$ (c) $x^2 - y^2 = c$ (d) $xy = c$

7) The solution of $\frac{dy}{dx} = e^{x-y}$ is

(a) $e^y e^x = c$ (b) y = log ce^x

(c) $y = \log(e^x + c)$ (d) $e^{x+y} = c$

8) The solution of $\frac{dp}{dt} = ke^{-t}$ (k is a constant) is

(a) $c - \frac{k}{e^t} = p$ (b) $p = ke^t + c$

(c) $t = \log \frac{c-p}{k}$ (d) $t = \log_c p$

9) In the differential equation $(x^2 - y^2)\,dy = 2xy\,dx$, if we make the subsititution $y = vx$ then the equation is transformed into

(a) $\frac{1+v^2}{v+v^3}\,dv = \frac{dx}{x}$ (b) $\frac{1-v^2}{v(1+v^2)}\,dv = \frac{dx}{x}$

(c) $\frac{dv}{v^2-1} = \frac{dx}{x}$ (d) $\frac{dv}{1+v^2} = \frac{dx}{x}$

10) When $y = vx$ the differential equation $x\frac{dy}{dx} = y + \sqrt{x^2+y^2}$ reduces to

(a) $\frac{dv}{\sqrt{v^2-1}} = \frac{dx}{x}$ (b) $\frac{v\,dv}{\sqrt{v^2+1}} = \frac{dx}{x}$

(c) $\frac{dv}{\sqrt{v^2+1}} = \frac{dx}{x}$ (d) $\frac{v\,dv}{\sqrt{1-v^2}} = \frac{dx}{x}$

11) The solution of the equation of the type $\frac{dy}{dx} + \mathrm{P}y = 0$, (P is a function of x) is given by

(a) $y e^{\int \mathrm{P}dx} = c$ (b) $y\int \mathrm{P}dx = c$

(c) $x e^{\int \mathrm{P}dx} = y$ (d) $y = cx$

12) The solution of the equation of the type $\frac{dx}{dy}$ + Px = Q (P and Q are functions of y) is

(a) $y = \int Q\, e^{\int P dx} dy + c$ (b) $y e^{\int P dx} = \int Q\, e^{\int P dx} dx + c$

(c) $x e^{\int P dy} = \int Q\, e^{\int P dy} dy + c$ (d) $x e^{\int P dy} = \int Q\, e^{\int P dx} dx + c$

13) The integrating factor of $x\frac{dy}{dx} - y = e^x$ is

(a) $\log x$ (b) $e^{\frac{-1}{x}}$ (c) $\frac{1}{x}$ (d) $\frac{-1}{x}$

14) The integrating factor of $(1 + x^2)\frac{dy}{dx} + xy = (1 + x^2)^3$ is

(a) $\sqrt{1+x^2}$ (b) $\log(1 + x^2)$ (c) $e^{\tan^{-1}x}$ (d) $\log^{(\tan^{-1}x)}$

15) The integrating factor of $\frac{dy}{dx} + \frac{2y}{x} = x^3$ is

(a) $2\log x$ (b) e^{x^2} (c) $3\log(x^2)$ (d) x^2

16) The complementary function of the differential equation $(D^2 - D)\,y = e^x$ is

(a) $A + B\,e^x$ (b) $(Ax + B)e^x$ (c) $A + Be^{-x}$ (d) $(A+Bx)e^{-x}$

17) The complementary function of the differential equation $(D^2 - 2D + 1)y = e^{2x}$ is

(a) $Ae^x + Be^{-x}$ (b) $A + Be^x$ (c) $(Ax + B)e^x$ (d) $A+Be^{-x}$

18) The particular integral of the differential equation $\frac{d^2y}{dx^2} - 5\frac{dy}{dx} + 6y = e^{5x}$ is

(a) $\frac{e^{5x}}{6}$ (b) $\frac{xe^{5x}}{2!}$ (c) $6e^{5x}$ (d) $\frac{e^{5x}}{25}$

19) The particular integral of the differential equation $\frac{d^2y}{dx^2} - 6\frac{dy}{dx} + 9y = e^{3x}$ is

(a) $\frac{e^{3x}}{2!}$ (b) $\frac{x^2e^{3x}}{2!}$ (c) $\frac{xe^{3x}}{2!}$ (d) $9e^{3x}$

20) The solution of $\frac{d^2y}{dx^2} - y = 0$ is

(a) $(A + B)e^x$ (b) $(Ax + B)e^{-x}$ (c) $Ae^x + \frac{B}{e^x}$ (d) $(A+Bx)e^{-x}$

INTERPOLATION AND FITTING A STRAIGHT LINE 7

7.1 INTERPOLATION

Interpolation is the art of reading between the lines in a table. It means insertion or filling up intermediate values of a function from a given set of values of the function. The following table represents the population of a town in the decennial census.

Year	:	1910	1920	1930	1940	1950
Population (in thousands)	:	12	15	20	27	39

Then the process of finding the population for the year 1914, 1923, 1939, 1947 etc. with the help of the above data is called **interpolation.** The process of finding the population for the year 1955, 1960 etc. is known as extrapolation.

The following assumptions are to be kept in mind for interpolation :

(i) The value of functions should be either in increasing order or in decreasing order.

(ii) The rise or fall in the values should be uniform. In other words that there are no sudden jumps or falls in the value of function during the period under consideration.

The following methods are used in interpolation :

1) Graphic method, 2) Algebraic method

7.1.1 Graphic method of interpolation

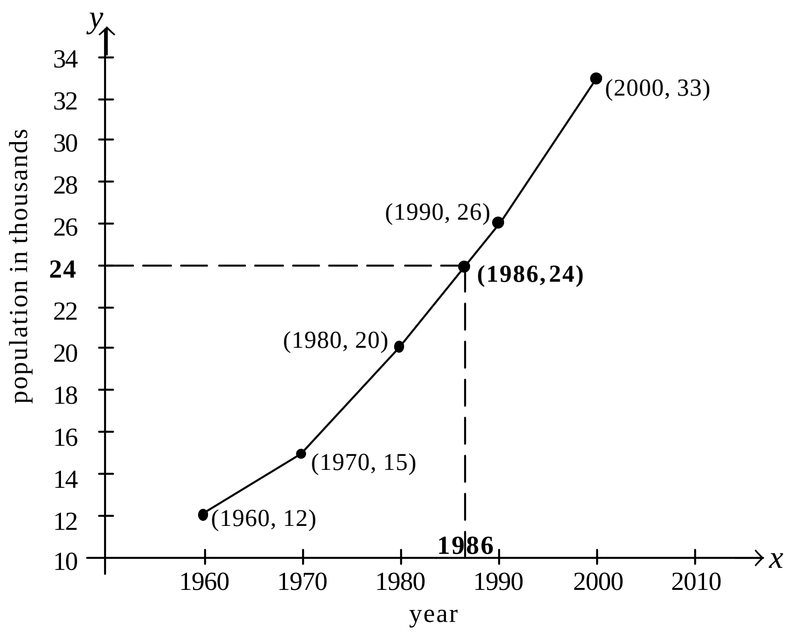
Let $y = f(x)$, then we can plot a graph between different values of x and corresponding values of y. From the graph we can find the value of y for given x.

Example 1

From the following data, estimate the population for the year 1986 graphically.

Year :	1960	1970	1980	1990	2000
Population : (in thousands)	**12**	**15**	**20**	**26**	**33**

Solution :



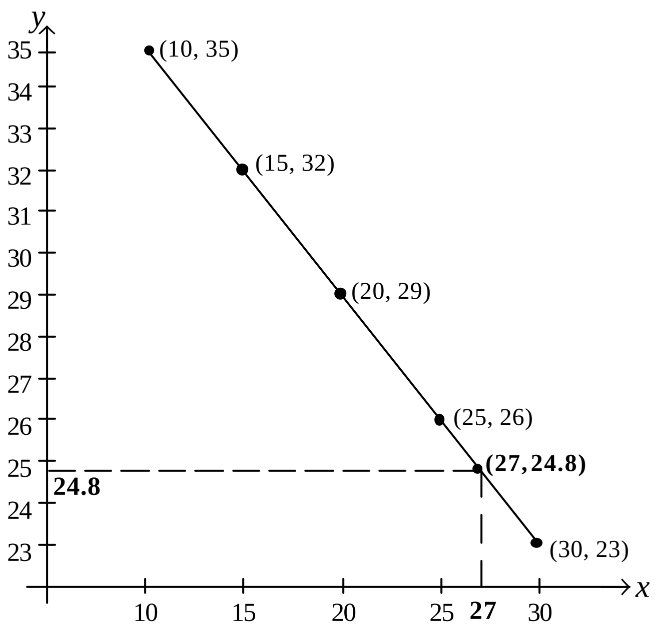
From the graph, it is found that the population for 1986 was 24 thousands

Example 2

Using graphic method, find the value of *y* when *x* = 27, from the following data.

x :	**10**	**15**	**20**	**25**	**30**
y :	**35**	**32**	**29**	**26**	**23**

Solution :



The value of y when $x = 27$ is 24.8

7.1.2 Algebraic methods of interpolation

The mathematical methods of interpolation are many. Of these we are going to study the following methods:

(i) Finite differences

(ii) Gregory-Newton's formula

(iii) Lagrange's formula

7.1.3 Finite differences

Consider the arguments $x_0, x_1, x_2, \dots x_n$ and the entries $y_0, y_1, y_2, \dots, y_n$. Here $y = f(x)$ is a function used in interpolation.

Let us assume that the x-values are in the increasing order and equally spaced with a space-length h.

Then the values of x may be taken to be x_0, $x_0 + h$, $x_0 + 2h$, ... $x_0 + nh$ and the function assumes the values $f(x_0)$, $f(x_0+h)$, $f(x_0 + 2h)$, ..., $f(x_0 + nh)$

Forward difference operator

For any value of x, the forward difference operator Δ(delta) is defined by

$$\Delta f(x) = f(x+h) - f(x).$$

In particular, $\Delta y_0 = \Delta f(x_0) = f(x_0+h) - f(x_0) = y_1 - y_0$

$\Delta f(x)$, $\Delta[f(x+h)]$, $\Delta[f(x+2h)]$, ... are the first order differences of $f(x)$.

Consider $\Delta^2 f(x) = \Delta[\Delta\{f(x)\}]$

$$= \Delta[f(x+h) - f(x)]$$
$$= \Delta[f(x+h)] - \Delta[f(x)]$$
$$= [f(x+2h) - f(x+h)] - [f(x+h) - f(x)]$$
$$= f(x+2h) - 2f(x+h) + f(x).$$

$\Delta^2 f(x)$, $\Delta^2 [f(x+h)]$, $\Delta^2 [f(x+2h)]$ ... are the second order differences of $f(x)$.

In a similar manner, the higher order differences $\Delta^3 f(x)$, $\Delta^4 f(x)$,...$\Delta^n f(x)$, ... are all defined.

Backward difference operator

For any value of x, the backward difference operator ∇(nabla) is defined by

$$\nabla f(x) = f(x) - f(x-h)$$

In particular, $\nabla y_n = \nabla f(x_n) = f(x_n) - f(x_n - h) = y_n - y_{n-1}$

$\nabla f(x)$, $\nabla[f(x+h)]$, $\nabla[f(x+2h)]$, ... are the first order differences of $f(x)$.

Consider $\nabla^2 f(x) = \nabla[\nabla\{f(x)\}] = \nabla[f(x) - f(x-h)]$

$$= \nabla[\mathrm{f}(x)] - \nabla[f(x-h)]$$
$$= f(x) - 2f(x-h) + f(x-2h)$$

$\nabla^2 f(x)$, $\nabla^2 [f(x+h)]$, $\nabla^2 [f(x+2h)]$ ... are the second order differences of f(x).

In a similar manner the higher order backward differences $\nabla^3 f(x)$, $\nabla^4 f(x)$,...$\nabla^n f(x)$, ... are all defined.

Shifting operator

For any value of x, the shifting operator E is defined by

$$E[f(x)] = f(x+h)$$

In particular, $E(y_0) = E[f(x_0)] = f(x_0+h) = y_1$

Further, $E^2 [f(x)] = E[E\{f(x)] = E[f(x+h)] = f(x+2h)$

Similarly $E^3[f(x)] = f(x+3h)$

In general $E^n [f(x)] = f(x+nh)$

The relation between D and E

We have $\Delta f(x) = f(x+h) - f(x)$

$= E\, f(x) - f(x)$

$\Delta f(x) = (E - 1) f(x)$

$\Rightarrow \quad \Delta = E - 1$

i.e. $\quad E = 1 + \Delta$

Results

1) The differences of constant function are zero.

2) If $f(x)$ is a polynomial of the nth degree in x, then the nth difference of $f(x)$ is constant and $\Delta^{n+1} f(x) = 0$.

Example 3

Find the missing term from the following data.

x	**:**	**1**	**2**	**3**	**4**
f(x)	**:**	**100**	**--**	**126**	**157**

Solution :

Since three values of $f(x)$ are given, we assume that the polynomial is of degree two.

Hence third order differences are zeros.

$\Rightarrow \quad \Delta^3 [f(x_0)] = 0$

or $\quad \Delta^3(y_0) = 0$

$\therefore \quad (E - 1)^3 y_0 = 0 \qquad (\Delta = E - 1)$

$(E^3 - 3E^2 + 3E - 1)\, y_0 = 0$

$\Rightarrow \quad y_3 - 3y_2 + 3y_1 - y_0 = 0$

$157 - 3(126) + 3y_1 - 100 = 0$

$\therefore \quad y_1 = 107$

i.e. the missing term is 107

Example 4

Estimate the production for 1962 and 1965 from the following data.

Year :	**1961**	**1962**	**1963**	**1964**	**1965**	**1966**	**1967**
Production: (in tons)	**200**	**--**	**260**	**306**	**--**	**390**	**430**

Solution :

Since five values of $f(x)$ are given, we assume that polynomial is of degree four.

Hence fifth order diferences are zeros.

$\therefore \quad \Delta^5 [f(x_0)] = 0$

i.e. $\quad \Delta^5 (y_0) = 0$

$\therefore \quad (E - 1)^5 (y_0) = 0$

i.e. $\quad (E^5 - 5E^4 + 10E^3 - 10E^2 + 5E - 1)\, y_0 = 0$

$y_5 - 5y_4 + 10y_3 - 10y_2 + 5y_1 - y_0 = 0$

$390 - 5y_4 + 10(306) - 10(260) + 5y_1 - 200 = 0$

$\Rightarrow \quad y_1 - y_4 = -130$ --------------(1)

Since fifth order differences are zeros, we also have

$\Delta^5 [f(x_1)] = 0$

i.e. $\Delta^5 (y_1) = 0$

i.e. $(E - 1)^5 y_1 = 0$

$(E^5 - 5E^4 + 10E^3 - 10E^2 + 5E - 1)y_1 = 0$

$y_6 - 5y_5 + 10y_4 - 10y_3 + 5y_2 - y_1 = 0$

$430 - 5(390) + 10y_4 - 10(306) + 5(260) - y_1 = 0$

$\Rightarrow \quad 10y_4 - y_1 = 3280$ ------------(2)

By solving the equations (1) and (2) we get,

$y_1 = 220$ and $y_4 = 350$

$\therefore$ The productions for 1962 and 1965 are 220 tons and 350 tons respectively.

7.1.4 Derivation of Gregory - Newton's forward formula

Let the function $y = f(x)$ be a polynomial of degree n which assumes $(n+1)$ values $f(x_0), f(x_1), f(x_2)... f(x_n)$, where $x_0, x_1, x_2, ... x_n$ are in the increasing order and are equally spaced.

Let $x_1 - x_0 = x_2 - x_1 = x_3 - x_2 = ... = x_n - x_{n-1} = h$ (a positive quantity)

Here $f(x_0) = y_0, \quad f(x_1) = y_1, ... f(x_n) = y_n$

Now $f(x)$ can be written as,

$$f(x) = a_0 + a_1 (x - x_0) + a_2(x-x_0)(x-x_1) + ... + a_n(x-x_0)(x-x_1)... (x-x_{n-1}) \quad \text{----------------(1)}$$

When $x = x_0$, (1) implies

$f(x_0) = a_0$ or $a_0 = y_0$

When $x = x_1$, (1) $\Rightarrow$

$f(x_1) = a_0 + a_1 (x_1 - x_0)$

i.e. $y_1 = y_0 + a_1 h$

$\therefore \quad a_1 = \dfrac{y_1 - y_0}{h} \quad \Rightarrow \quad a_1 = \dfrac{\Delta y_0}{h}$

When $x = x_2$, (1) $\Rightarrow$

$f(x_2) = a_0 + a_1(x_2 - x_0) + a_2(x_2 - x_0)(x_2 - x_1)$

$$y_2 = y_0 + \frac{\Delta y_0}{h}(2h) + a_2 (2h)(h)$$

$$2h^2 a_2 = y_2 - y_0 - 2\Delta y_0$$

$$= y_2 - y_0 - 2(y_1 - y_0)$$

$$= y_2 - 2y_1 + y_0 = \Delta^2 y_0$$

$$\therefore \quad a_2 = \frac{\Delta^2 y_0}{2!\, h^2}$$

In the same way we can obtain

$$a_3 = \frac{\Delta^3 y_0}{3!\, h^3}, \quad a_4 = \frac{\Delta^4 y_0}{4!\, h^4}, \ldots, a_n = \frac{\Delta^n y_0}{n!\, h^n}$$

substituting the values of a_0, a_1, ..., a_n in (1) we get

$$f(x) = y_0 + \frac{\Delta y_0}{h}(x - x_0) + \frac{\Delta^2 y_0}{2!\, h^2}(x - x_0)(x - x_1) + \ldots$$

$$+ \frac{\Delta^n y_0}{n!\, h^n}(x - x_0)(x - x_1) \ldots (x - x_{n-1}) \quad \text{---------(2)}$$

Denoting $\frac{x - x_0}{h}$ by u, we get

$$x - x_0 = hu$$

$$x - x_1 = (x - x_0) - (x_1 - x_0) = hu - h = h(u-1)$$

$$x - x_2 = (x - x_0) - (x_2 - x_0) = hu - 2h = h(u-2)$$

$$x - x_3 = h(u - 3)$$

In general

$$x - x_{n-1} = h\{u - (n-1)\}$$

Thus (2) becomes,

$$f(x) = y_0 + \frac{u}{1!}\Delta y_0 + \frac{u(u-1)}{2!}\Delta^2 y_0 + \ldots$$

$$+ \frac{u(u-1)(u-2)\ldots(u-\overline{n-1})}{n!}\Delta^n y_0$$

where $u = \frac{x - x_0}{h}$. This is the Gregory-Newton's forward formula.

Example 5

Find y when $x = 0.2$ given that

x	**:**	**0**	**1**	**2**	**3**	**4**
y	**:**	**176**	**185**	**194**	**202**	**212**

Solution :

0.2 lies in the first interval (x_0, x_1) i.e. (0, 1). So we can use Gregory-Newton's forward interpolation formula. Since five values are given, the interpolation formula is

$$y = y_0 + \frac{u}{1!}\Delta y_0 + \frac{u(u-1)}{2!}\Delta^2 y_0 + \frac{u(u-1)(u-2)}{3!}\Delta^3 y_0$$

$$+\frac{u(u-1)(u-2)(u-3)}{4!}\Delta^4 y_0 \text{ where } u = \frac{x-x_0}{h}$$

Here $h = 1$, $x_0 = 0$ and $x = 0.2$

$$\therefore u = \frac{0.2-0}{1} = 0.2$$

The forward difference table :

x	y	Dy	D²y	D³y	D⁴y
0	**176**				
		9			
1	185		**0**		
		9		**-1**	
2	194		-1		**4**
		8		3	
3	202		2		
		10			
4	212				

$$\therefore \quad y = \mathbf{176} + \frac{0.2}{1!}(\mathbf{9}) + \frac{0.2(0.2-1)}{2!}(\mathbf{0})$$

$$+ \frac{(0.2)(0.2-1)(0.2-2)}{3!}(\mathbf{-1}) + \frac{(0.2)(0.2-1)(0.2-2)(0.2-3)}{4!}(\mathbf{4})$$

$$= 176 + 1.8 - 0.048 - 0.1344$$

$$= 177.6176$$

i.e. when $x = 0.2$, $y = 177.6176$

Example 6

If $y_{75} = 2459, y_{80} = 2018, y_{85} = 1180$ and $y_{90} = 402$ find y_{82}.

Solution :

We can write the given data as follows:

x	:	75	80	85	90
y	:	2459	2018	1180	402

82 lies in the interval (80, 85). So we can use Gregory-Newton's forward interpolation formula. Since four values are given, the interpolation formula is

$$y = y_0 + \frac{u}{1!}\Delta y_0 + \frac{u(u-1)}{2!}\Delta^2 y_0 + \frac{u(u-1)(u-2)}{3!}\Delta^3 y_0$$

where $u = \dfrac{x - x_0}{h}$

Here $h = 5, x_0 = 75$ $x = 82$

$\therefore u = \dfrac{82 - 75}{5} = \dfrac{7}{5} = 1.4$

The forward difference table :

x	**y**	**Dy**	**D^2y**	**D^3y**
75	**2459**			
		-441		
80	2018		**-397**	
		-838		**457**
85	1180		60	
		-778		
90	402			

$$\therefore \quad y = 2459 + \frac{1.4}{1!}(-441) + \frac{1.4(1.4-1)}{2!}(-397) + \frac{1.4(1.4-1)(1.4-2)}{3!}(457)$$

$$= 2459 - 617.4 - 111.6 - 25.592$$

$y = 1704.408$ when $x = 82$

Example 7

From the following data calculate the value of $e^{1.75}$

x	**:**	**1.7**	**1.8**	**1.9**	**2.0**	**2.1**
e^x	**:**	**5.474**	**6.050**	**6.686**	**7.389**	**8.166**

Solution :

Since five values are given, the interpolation formula is

$$y_x = y_0 + \frac{u}{1!}\Delta y_0 + \frac{u(u-1)}{2!}\Delta^2 y_0 + \frac{u(u-1)(u-2)}{3!}\Delta^3 y_0 + \frac{u(u-1)(u-2)(u-3)}{4!}\Delta^4 y_0$$

where $u = \dfrac{x-x_0}{h}$

Here $h = 0.1$, $x_0 = 1.7$ $x = 1.75$

$$\therefore u = \frac{1.75-1.7}{0.1} = \frac{0.05}{0.1} = 0.5$$

The forward difference table :

x	y	Δy	$\Delta^2 y$	$\Delta^3 y$	$\Delta^4 y$
1.7	**5.474**				
		0.576			
1.8	6.050		**0.060**		
		0.636		**0.007**	
1.9	6.686		0.067		0
		0.703		0.007	
2.0	7.389		0.074		
		0.777			
2.1	8.166				

$$\therefore y = 5.474 + \frac{0.5}{1!}(0.576) + \frac{0.5(0.5-1)}{2!}(0.06) + \frac{0.5(0.5-1)(0.5-2)}{3!}(0.007)$$

$$= 5.474 + 0.288 - 0.0075 + 0.0004375$$

$\therefore y = 5.7549375$ when $x = 1.75$

Example 8

From the data, find the number of students whose height is between 80cm. and 90cm.

Height in cms x	**:**	**40-60**	**60-80**	**80-100**	**100-120**	**120-140**
No. of students y	**:**	**250**	**120**	**100**	**70**	**50**

Solution :

The difference table

	x	y	Δy	$\Delta^2 y$	$\Delta^3 y$	$\Delta^4 y$
Below	60	**250**				
			120			
Below	80	370		**-20**		
			100		**-10**	
Below	100	470		-30		**20**
			70		10	
Below	120	540		-20		
			50			
Below	140	590				

Let us calculate the number of students whose height is less than 90cm.

Here $x = 90$ $\quad u = \dfrac{x - x_0}{h} = \dfrac{90-60}{20} = 1.5$

$$y(90) = 250 + (1.5)(120) + \frac{(1.5)(1.5-1)}{2!}(-20)$$

$$+ \frac{(1.5)(1.5-1)(1.5-2)}{3!}(-10) + \frac{(1.5)(1.5-1)(1.5-2)(1.5-3)}{4!}(20)$$

$= 250 + 180 - 7.5 + 0.625 + 0.46875$

$= 423.59 \simeq 424$

Therefore number of students whose height is between

80cm. and 90cm. is $y(90) - y(80)$

i.e. $424 - 370 = 54$.

Example 9

Find the number of men getting wages between Rs.30 and Rs.35 from the following table

Wages	**x**	**:**	**20-30**	**30-40**	**40-50**	**50-60**
No. of men	**y**	**:**	**9**	**30**	**35**	**42**

Solution :

The difference table

	x	y	Δy	$\Delta^2 y$	$\Delta^3 y$
Under	30	**9**			
			30		
Under	40	39		**5**	
			35		**2**
Under	50	74		7	
			42		
Under	60	116			

Let us calculate the number of men whose wages is less than Rs.35.

For $x = 35$, $u = \dfrac{x - x_0}{h} = \dfrac{35-30}{10} = 0.5$

By Newton's forward formula,

$$y(35) = 9 + \frac{(0.5)}{1}(30) + \frac{(0.5)(0.5-1)}{2!}(5) + \frac{(0.5)(0.5-1)(0.5-2)}{3!}(2)$$

$$= 9 + 15 - 0.6 + 0.1$$

$$= 24 \text{ (approximately)}$$

Therefore number of men getting wages between

Rs.30 and Rs.35 is $y(35) - y(30)$ i.e. $24 - 9 = 15$.

7.1.5 Gregory-Newton's backward formula

Let the function $y = f(x)$ be a polynomial of degree n which assumes $(n+1)$ values $f(x_0), f(x_1), f(x_2), \ldots, f(x_n)$ where $x_0, x_1, x_2, \ldots, x_n$ are in the increasing order and are equally spaced.

Let $x_1 - x_0 = x_2 - x_1 = x_3 - x_2 = \ldots x_n - x_{n-1} = h$ (a positive quantity)

Here $f(x)$ can be written as

$$f(x) = a_0 + a_1(x-x_n) + a_2(x-x_n)(x-x_{n-1}) + \dots$$
$$+ a_n(x-x_n)(x-x_{n-1}) \dots (x-x_1) \quad \text{-----------(1)}$$

When $x = x_n$, (1) $\Rightarrow$

$$f(x_n) = a_0 \quad \text{or} \quad a_0 = y_n$$

When $x = x_{n-1}$, (1) $\Rightarrow$

$$f(x_{n-1}) = a_0 + a_1(x_{n-1}-x_n)$$

or $\quad y_{n-1} = y_n + a_1(-h)$

or $\quad a_1 = \dfrac{y_n - y_{n-1}}{h} \Rightarrow a_1 = \dfrac{\nabla y_n}{h}$

When $x = x_{n-2}$, (1) $\Rightarrow$

$$f(x_{n-2}) = a_0 + a_1(x_{n-2} - x_n) + a_2(x_{n-2} - x_n)(x_{n-2} - x_{n-1})$$

$$y_{n-2} = y_n + \frac{\nabla y_n}{h}(-2h) + a_2(-2h)(-h)$$

$$2h^2 a_2 = (y_{n-2} - y_n) + 2\nabla y_n$$
$$= y_{n-2} - y_n + 2(y_n - y_{n-1})$$
$$= y_{n-2} - 2y_{n-1} + y_n = \nabla^2 y_n$$

$$\therefore a_2 = \frac{\nabla^2 y_n}{2!h^2}$$

In the same way we can obtain

$$a_3 = \frac{\nabla^3 y_n}{3!h^3} \;, \quad a_4 = \frac{\nabla^4 y_n}{4!h^4} \dots a_n = \frac{\nabla^n y_n}{n!}$$

$$\therefore \quad f(x) = y_n + \frac{\nabla y_n}{h}(x-x_n) + \frac{\nabla^2 y_n}{2!h^2}(x-x_n)(x-x_{n-1}) + \dots$$
$$+ \frac{\nabla^n y_n}{n!}(x-x_n)(x-x_{n-1}) \dots (x-x_1) \quad \text{------------(2)}$$

Further, denoting $\dfrac{x-x_n}{h}$ by u, we get

$$x-x_n = h_u$$

$x-x_{n-1} = (x-x_n)\ (x_n-x_{n-1}) = hu + h = h(u+1)$

$x-x_{n-2} = (x-x_n)\ (x_n-x_{n-2}) = hu + 2h = h(u+2)$

$x-x_{n-3} = h(u+3)$

In general

$x-x_{n-k} = h(u+k)$

Thus (2) becomes,

$$f(x) = y_n + \frac{u}{1!}\nabla y_n + \frac{u(u+1)}{2!}\nabla^2 y_n + \ldots$$

$$+ \frac{u(u+1)\ldots\{u+(n-1)\}}{n!}\nabla^n y_n \text{ where } u = \frac{x-x_n}{h}$$

This is the Gregory-Newton's backward formula.

Example 10

Using Gregory-Newton's formula estimate the population of town for the year 1995.

Year	**x**	**:**	**1961**	**1971**	**1981**	**1991**	**2001**
Population	**y**	**:**	**46**	**66**	**81**	**93**	**101**

(in thousands)

Solution :

1995 lies in the interval (1991, 2001). Hence we can use Gregory-Newton's backward interpolation formula. Since five values are given, the interpolation formula is

$$y = y_4 + \frac{u}{1!}\nabla y_4 + \frac{u(u+1)}{2!}\nabla^2 y_4 + \frac{u(u+1)(u+2)}{3!}\nabla^3 y_4$$

$$+ \frac{u(u+1)(u+2)(u+3)}{4!}\nabla^4 y_4 \text{ where } u = \frac{x-x_4}{h}$$

Here $h = 10$, $x_4 = 2001$ $x = 1995$

$$\therefore\ u = \frac{1995-2001}{10} = -0.6$$

The backward difference table :

x	y	∇y	$\nabla^2 y$	$\nabla^3 y$	$\nabla^4 y$
1961	46				
		20			
1971	66		-5		
		15		2	
1981	81		-3		**-3**
		12		**-1**	
1991	93		**-4**		
		8			
2001	**101**				

$$\therefore \quad y = 101 + \frac{(-0.6)}{1!}(8) + \frac{(-0.6)(-0.6+1)}{2!}(-4)$$

$$+ \frac{(-0.6)(-0.6+1)(-0.6+2)}{3!}(-1) +$$

$$\frac{(-0.6)(-0.6+1)(-0.6+2)(-0.6+3)}{4!}(-3)$$

$$= 101-4.8+0.48+0.056+0.1008 \quad \therefore \quad y = 96.8368$$

i.e. the population for the year 1995 is 96.837 thousands.

Example 11

From the following table, estimate the premium for a policy maturing at the age of 58

Age	**x**	**:**	**40**	**45**	**50**	**55**	**60**
Premium	**y**	**:**	**114.84**	**96.16**	**83.32**	**74.48**	**68.48**

Solution :

Since five values are given, the interpolation formula is

$$y = y_4 + \frac{u}{1!}\nabla y_4 + \ldots + \frac{u(u+1)(u+2)(u+3)}{4!}\nabla^4 y_4$$

where $u = \frac{58-60}{5} = -0.4$

The backward difference table :

x	y	∇y	$\nabla^2 y$	$\nabla^3 y$	$\nabla^4 y$
40	114.84				
		-18.68			
45	96.16		5.84		
		-12.84		-1.84	
50	83.32		4.00		**0.68**
		-8.84		**-1.16**	
55	74.48		**2.84**		
		-6.00			
60	**68.48**				

$$\therefore \quad y = 68.48 + \frac{(-0.4)}{1!}(-6) + \frac{(-0.4)(0.6)}{2}(2.84)$$
$$+\frac{(-0.4)(0.6)(1.6)}{6}(-1.16) + \frac{(-0.4)(0.6)(1.6)(2.6)}{24}(0.68)$$
$$= 68.48 + 2.4 - 0.3408 + 0.07424 - 0.028288$$
$$\therefore \quad y = 70.5851052 \quad \text{i.e. } y \simeq 70.59$$

$\therefore$ Premium for a policy maturing at the age of 58 is 70.59

Example 12

From the following data, find y when $x = 4.5$

x	**:**	**1**	**2**	**3**	**4**	**5**
y	**:**	**1**	**8**	**27**	**64**	**125**

Solution :

Since five values are given, the interpolation formula is

$$y = y_4 + \frac{u}{1!}\nabla y_4 + \ldots + \frac{u(u+1)(u+2)(u+3)}{4!}\nabla^4 y_4$$

where $u = \dfrac{x - x_4}{h}$

Here $u = \dfrac{4.5-5}{1} = -0.5$

The backward difference table :

x	***y***	**Ñ*y***	**Ñ^2*y***	**Ñ^3*y***	**Ñ^4*y***
1	1				
		7			
2	8		12		
		19		6	
3	27		18		**0**
		37		**6**	
4	64		**24**		
		61			
5	**125**				

$$\therefore \; y = 125 + \frac{(-0.5)}{1}(61) + \frac{(-0.5)(0.5)}{2}(24) + \frac{(-0.5)(0.5)(1.5)}{6}(6)$$

$\therefore \; y = 91.125$ when $x = 4.5$

7.1.6 Lagrange's formula

Let the function $y = f(x)$ be a polynomial of degree n which assumes $(n + 1)$ values $f(x_0)$, $f(x_1)$, $f(x_2)$ ...$f(x_n)$ corresponding to the arguments x_0, x_1, x_2, ... x_n (not necessarily equally spaced).

Here $f(x_0) = y_0$, $f(x_1) = y_1$, ..., $f(x_n) = y_n$.

Then the Lagrange's formula is

$$f(x) = y_0 \frac{(x-x_1)(x-x_2)...(x-x_n)}{(x_0-x_1)(x_0-x_2)...(x_0-x_n)}$$
$$+ y_1 \frac{(x-x_0)(x-x_2)...(x-x_n)}{(x_1-x_0)(x_1-x_2)...(x_1-x_n)}$$
$$+ \dots + y_n \frac{(x-x_0)(x-x_1)...(x-x_{n-1})}{(x_n-x_0)(x_n-x_1)...(x_n-x_{n-1})}$$

Example 13

Using Lagrange's formula find the value of y when $x = 42$ from the following table

x	**:**	**40**	**50**	**60**	**70**
y	**:**	**31**	**73**	**124**	**159**

Solution :

By data we have

$x_0 = 40$, $x_1 = 50$, $x_2 = 60$, $x_3 = 70$ and $x = 42$

$y_0 = 31$, $y_1 = 73$, $y_2 = 124$, $y_3 = 159$

Using Lagrange's formula, we get

$$y = y_0 \frac{(x-x_1)(x-x_2)(x-x_3)}{(x_0-x_1)(x_0-x_2)(x_0-x_3)}$$
$$+ y_1 \frac{(x-x_0)(x-x_2)(x-x_3)}{(x_1-x_0)(x_1-x_2)(x_1-x_3)}$$
$$+ y_2 \frac{(x-x_0)(x-x_1)(x-x_3)}{(x_2-x_0)(x_2-x_1)(x_2-x_3)}$$

$$+ y_3 \frac{(x-x_0)(x-x_1)(x-x_2)}{(x_3-x_0)(x_3-x_1)(x_3-x_2)}$$

$$y(42) = 31\frac{(-8)(-18)(-28)}{(-10)(-20)(-30)} + 73\frac{(2)(-18)(-28)}{(10)(-10)(-20)}$$

$$+124\frac{(2)(-8)(-28)}{(20)(10)(-10)} + 159\frac{(2)(-8)(-18)}{(30)(20)(10)}$$

$$= 20.832 + 36.792 - 27.776 + 7.632$$

$$y = 37.48$$

Example 14

Using Lagrange's formula find y when $x = 4$ from the following table

x	**:**	**0**	**3**	**5**	**6**	**8**
y	**:**	**276**	**460**	**414**	**343**	**110**

Solution :

Given

$x_0 = 0$, $x_1 = 3, x_2 = 5, x_3 = 6, x_4 = 8$ and $x = 4$

$y_0 = 276, y_1 = 460, y_2 = 414, y_3 = 343, y_4 = 110$

Using Lagrange's formula

$$y = y_0\frac{(x-x_1)(x-x_2)(x-x_3)(x-x_4)}{(x_0-x_1)(x_0-x_2)(x_0-x_3)(x_0-x_4)}$$

$$+ y_1\frac{(x-x_0)(x-x_2)(x-x_3)(x-x_4)}{(x_1-x_0)(x_1-x_2)(x_1-x_3)(x_1-x_4)}$$

$$+ y_2\frac{(x-x_0)(x-x_1)(x-x_3)(x-x_4)}{(x_2-x_0)(x_2-x_1)(x_2-x_3)(x_2-x_4)}$$

$$+ y_3\frac{(x-x_0)(x-x_1)(x-x_2)(x-x_4)}{(x_3-x_0)(x_3-x_1)(x_3-x_2)(x_3-x_4)}$$

$$+ y_4\frac{(x-x_0)(x-x_1)(x-x_2)(x-x_3)}{(x_4-x_0)(x_4-x_1)(x_4-x_2)(x_4-x_3)}$$

$$= 276\frac{(1)(-1)(-2)(-4)}{(-3)(-5)(-6)(-8)} + 460\frac{(4)(-1)(-2)(-4)}{(3)(-2)(-3)(-5)}$$

$$+ 414\frac{(4)(1)(-2)(-4)}{(5)(2)(-1)(-3)} + 343\frac{(4)(1)(-1)(-4)}{(6)(3)(1)(-2)}$$

$$+ 110\frac{(4)(1)(-1)(-2)}{(8)(5)(3)(2)}$$

$$= -3.066 + 163.555 + 441.6 - 152.44 + 3.666$$

$$y = 453.311$$

Example 15

Using Lagrange's formula find $y(11)$ from the following table

x	**:**	**6**	**7**	**10**	**12**
y	**:**	**13**	**14**	**15**	**17**

Solution :

Given

$x_0 = 6,\ x_1 = 7, x_2 = 10, x_3 = 12$ and $x = 11$

$y_0 = 13, y_1 = 14, y_2 = 15, y_3 = 17$

Using Lagrange's formula

$$= 13\frac{(4)(1)(-1)}{(-1)(-4)(-6)} + 14\frac{(5)(1)(-1)}{(1)(-3)(-5)}$$

$$+ 15\frac{(5)(4)(-1)}{(4)(3)(-2)} + 17\frac{(5)(4)(1)}{(6)(5)(2)}$$

$$= 2.1666 - 4.6666 + 12.5 + 5.6666$$

$$y = 15.6666$$

EXERCISE 7.1

1) Using Graphic method, find the value of y when $x = 42$, from the following data.

x	:	20	30	40	50
y	:	51	43	34	24

2) The population of a town is as follows.

Year	x :	1940	1950	1960	1970	1980	1990
Population (in lakhs)	y :	20	24	29	36	46	50

Estimate the population for the year 1976 graphically

3) From the following data, find f(3)

x	:	1	2	3	4	5
f(x)	:	2	5	-	14	32

4) Find the missing term from the following data.

x	:	0	5	10	15	20	25
y	:	7	11	14	--	24	32

5) From the following data estimate the export for the year 2000

Year	x	:	1999	2000	2001	2002	2003
Export (in tons)	y	:	443	–	369	397	467

6) Using Gregory-Newton's formula, find y when $x = 145$ given that

x	:	140	150	160	170	180
y	:	46	66	81	93	101

7) Using Gregory-Newton's formula, find y(8) from the following data.

x	:	0	5	10	15	20	25
y	:	7	11	14	18	24	32

8) Using Gregory-Newton's formula, calculate the population for the year 1975

Year	:	1961	1971	1981	1991	2001
Population	:	98572	132285	168076	198690	246050

9) From the following data find the area of a circle of diameter 96 by using Gregory-Newton's formula

Diameter	x	:	80	85	90	95	100
Area	y	:	5026	5674	6362	7088	7854

10) Using Gregory-Newton's formula, find y when $x = 85$

x	:	50	60	70	80	90	100
y	:	184	204	226	250	276	304

11) Using Gregory-Newton's formula, find $y(22.4)$

x	:	19	20	21	22	23
y	:	91	100	110	120	131

12) From the following data find $y(25)$ by using Lagrange's formula

x	:	20	30	40	50
y	:	512	439	346	243

13) If $f(0) = 5$, $f(1) = 6$, $f(3) = 50$, $f(4) = 105$, find $f(2)$ by using Lagrange's formula

14) Apply Lagrange's formula to find y when $x = 5$ given that

x	:	1	2	3	4	7
y	:	2	4	8	16	128

7.2 FITTING A STRAIGHT LINE

A commonly occurring problem in many fields is the necessity of studying the relationship between two (or more) variables.

For example the weight of a baby is related to its age ; the price of a commodity is related to its demand ; the maintenance cost of a car is related to its age.

7.2.1 Scatter diagram

This is the simplest method by which we can represent diagramatically a bivariate data. Suppose x and y denote respectively the age and weight of an adult male, then consider a sample of n individuals with ages $x_1, x_2, x_3, \dots x_n$ and the corresponding weights as $y_1, y_2, y_3, \dots y_n$. Plot the points

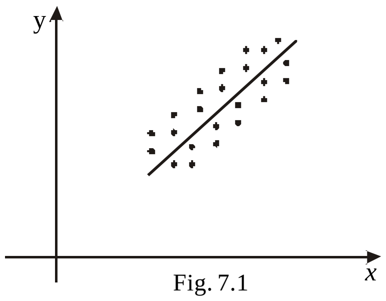


Fig. 7.1

$(x_1, y_1), (x_2, y_2), (x_3, y_3), \ldots (x_n, y_n)$ on a rectangular co-ordinate system. The resulting set of points in a graph is called a **scatter diagram.**

From the scatter diagram it is often possible to visualize a smooth curve approximating the data. Such a curve is called an approximating curve. In the above figure, the data appears to be well approximated by a straight line and we say that a linear relationship exists between the two variables.

7.2.2 Principle of least squares

Generally more than one curve of a given type will appear to fit a set of data. In constructing lines it is necessary to agree on a definition of a "best fitting line".

Consider the data points $(x_1, y_1), (x_2, y_2), (x_3, y_3), \ldots (x_n, y_n)$. For a given value of x, say x_1, in general there will be a difference between the value y, and the corresponding value as determined from the curve C (in Fig. 7.2)

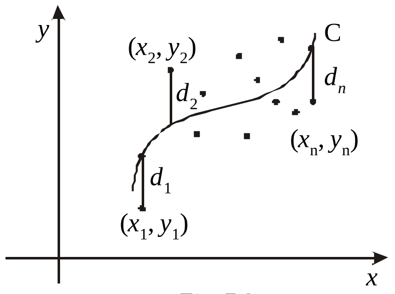


Fig. 7.2

We denote this difference by d_1, which is referred to as a deviation or error. Here d_1 may be positive, negative or zero. Similarly corresponding to the values $x_2, x_3, \ldots x_n$ we obtain the deviations $d_2, d_3, \ldots d_n$.

A measure of the "goodness of fit" of the curve to the set of data is provided by the quantities $d_1^2, d_2^2, \ldots d_n^2$.

Of all the curves approximating a given set of data points, the curve having the property that $d_1^2 + d_2^2 + d_3^2 + \ldots + d_n^2$ is a minimum is the best fitting curve. If the approximating curve is a straight line then such a line is called the "line of best fit".

7.2.3 Derivation of normal equations by the principle of least squares.

Let us consider the fitting of a straight line

$$\boldsymbol{y = ax + b} \qquad \text{------------}\mathbf{(1)}$$

to set of n points (x_1, y_1), (x_2, y_2), ... (x_n, y_n).

For the different values of a and b equation (1) represents a family of straight lines. Our aim is to determine a and b so that the line (1) is the line of best fit.

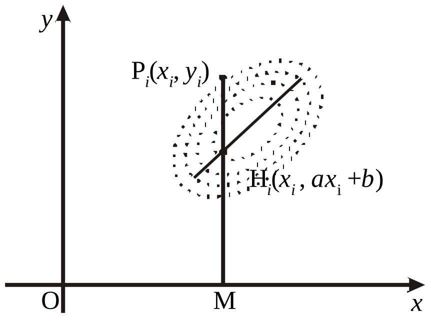


Fig. 7.3

Now a and b are determined by applying principle of least squares.

Let P_i (x_i, y_i) be any point in the scatter diagram. Draw P_iM perpendicular to x-axis meeting the line (1) in H_i. The x-coordinate of H_i is x_i. The ordinate of H_i is $ax_i + b$.

$$P_iH_i = P_iM - H_iM$$

$$= y_i - (ax_i + b) \text{ is the deviation for } y_i.$$

According to the principle of least squares, we have to find a and b so that

$$E = \sum_{i=1}^{n} P_iH_i^2 = \sum_{i=1}^{n} [y_i - (ax_i + b)]^2 \text{ is minimum.}$$

For maxima or minima, the partial derivatives of E with respect to a and b should vanish separately.

$$\therefore \quad \frac{\partial E}{\partial a} = 0 \Rightarrow -2\sum_{i=1}^{n} x_i[y_i - (ax_i + b)] = 0$$

$$\boldsymbol{a\sum_{i=1}^{n} x_i^2 + b\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} x_i y_i} \qquad \text{-------------}\mathbf{(2)}$$

$$\frac{\partial E}{\partial b} = 0 \Rightarrow -2\sum_{i=1}^{n} [y_i - (ax_i + b)] = 0$$

i.e., $\Sigma y_i - a\Sigma x_i - nb = 0$

Þ $\quad \boldsymbol{a \overset{n}{\underset{i=1}{\text{Ó}}} x_i + nb = \overset{n}{\underset{i=1}{\text{Ó}}} y_i}$ **-------------(3)**

(2) and (3) are known as the **normal equations.** Solving the normal equations we get a and b.

Note

The normal equations for the line of best fit of the form

$\boldsymbol{y = a + bx}$ are

$\boldsymbol{na + b\text{S}x_i = \text{S}y_i}$

$\boldsymbol{a\text{S}x_i + b\text{S}x_i^2 = \text{S}x_iy_i}$

Example 16

Fit a straight line to the following

Sx = 10, Sy = 19, Sx^2 = 30, Sxy = 53 and n = 5.

Solution :

The line of best fit is $y = ax + b$

$\Sigma y = a\Sigma x + nb$

$\Sigma xy = a\Sigma x^2 + b\Sigma x$

$\Rightarrow \quad 10a + 5b = 19$ ------------(1)

$30a + 10b = 53$ ------------(2)

Solving (1) and (2) we get, $a = 1.5$ and $b = 0.8$

The line of best fit is $y = 1.5x + 0.8$

Example 17

In a straight line of best fit find x-intercept when Sx = 10, Sy=16.9, Sx^2 = 30, Sxy = 47.4 and n = 7.

Solution :

The line of best fit is $y = ax + b$

The normal equations are

$\Sigma y = a\Sigma x + nb$

$\Sigma xy = a\Sigma x^2 + b\Sigma x$

$\Rightarrow \quad 10a + 7b = 16.9 \quad$ -----------(1)

$30a + 10b = 47.4 \quad$ -----------(2)

Solving (1) and (2) we get,

$a = 1.48$ and $b = 0.3$

The line of best fit is $y = 1.48x + 0.3$

$\therefore$ The x-intercept of the line of best fit is $-\dfrac{0.3}{1.48}$

Example 18

Fit a straight line for the following data.

x	**:**	**0**	**1**	**2**	**3**	**4**
y	**:**	**1**	**1**	**3**	**4**	**6**

Solution :

The line of best fit is $y = ax + b$

The normal equations are

$a\Sigma x + nb = \Sigma y \quad$ ------------(1)

$a\Sigma x^2 + b\Sigma x = \Sigma xy \quad$ ------------(2)

Now from the data

x	y	x^2	xy
0	1	0	0
1	1	1	1
2	3	4	6
3	4	9	12
4	6	16	24
10	**15**	**30**	**43**

By substituting these values in (1) and (2) we get,

$10a + 5b = 15 \quad$ ------------(3)

$30a + 10b = 43 \quad$ ------------(4)

Solving (3) and (4) we get, $a = 1.3$ and $b = 0.4$

The line of best fit is $y = 1.3x + 0.4$.

Example 19

Fit a straight line to the following data:

x :	**4**	**8**	**12**	**16**	**20**	**24**
y :	**7**	**9**	**13**	**17**	**21**	**25**

Solution :

Take the origin at $\frac{12+16}{2} = 14$

Let $u_i = \frac{x_i - 14}{2}$ Here $n = 6$

The line of best fit is $y = au + b$

The normal equations are $a\Sigma u + nb = \Sigma y$ ---------(1)

$a\Sigma u^2 + b\Sigma u = \Sigma uy$ ----------(2)

x	*y*	*u*	*u*²	*uy*
4	7	-5	25	-35
8	9	-3	9	-27
12	13	-1	1	-13
16	17	1	1	17
20	21	3	9	63
24	25	5	25	125
Total	**92**	**0**	**70**	**130**

On substituting the values in the normal equation (1) and (2)

$a = 1.86$ and $b = 15.33$

The line of best fit is $y = 1.86\left(\frac{x-14}{2}\right)+15.33 = 0.93x + 2.31$

Example 20

Fit a straight line to the following data.

x :	**100**	**200**	**300**	**400**	**500**	**600**
y :	**90.2**	**92.3**	**94.2**	**96.3**	**98.2**	**100.3**

Solution :

Let $u_i = \dfrac{x_i - 350}{50}$ and $v_i = y_i - 94.2$ Here $n = 6$.

The line of best fit is $v = au + b$

The normal equations are $a\Sigma u + nb = v$ ----------(1)

$a\Sigma u^2 + b\Sigma u = \Sigma uv$----------(2)

x	***y***	***u***	***v***	$\boldsymbol{u^2}$	**uv**
100	90.2	-5	-4	25	20
200	92.3	-3	-1.9	9	5.7
300	94.2	-1	0	1	0
400	96.3	1	2.1	1	2.1
500	98.2	3	4	9	12
600	100.3	5	6.1	25	30.5
Total		**0**	**63**	**70**	**70.3**

Substituting the values in (1) and (2) we get

$a = 1.0043$ and $b = 1.05$

The line of best fit is $v = 1.0043\,u + 1.05$

$\Rightarrow \quad y = 0.02x + 88.25$

EXERCISE 7.2

1) Define a scatter diagram.

2) State the principle of least squares.

3) Fit the line of best fit if $\Sigma x = 75$, $\Sigma y = 115$, $\Sigma x^2 = 1375$, $\Sigma xy = 1875$, and $n = 6$.

4) In a line of best fit find the slope and the y intercept if $\Sigma x = 10$, $\Sigma y = 25$, $\Sigma x^2 = 30$, $\Sigma xy = 90$ and $n = 5$.

5) Fit a straight line $y = ax + b$ to the following data by the method of least squares.

x	0	1	3	6	8
y	1	3	2	5	4

6) A group of 5 students took tests before and after training and obtained the following scores.

Scores before training	3	4	4	6	8
Scores after training	4	5	6	8	10

Find by the method of least squares the straight line of best fit

7) By the method of least squares find the best fitting straight line to the data given below:

x :	100	120	140	160	180	200
y :	0.45	0.55	0.60	0.70	0.80	0.85

8) Fit a straight line to the data given below. Also estimate the value y at $x = 3.5$

x :	0	1	2	3	4
y :	1	1.8	3.3	4.5	6.3

9) Find by the method of least squares, the line of best fit for the following data.

Depth of water applied (in cm)	x :	0	12	24	36	48
Average yield (tons / acre)	y :	35	55	65	80	90

10) The following data show the advertising expenses (expressed as a percentage of total expenses) and the net operating profits (expressed as a percentage of total sales) in a random sample of six drug stores.

Advertising expenses	0.4	1.0	1.3	1.5	2.0	2.8
Net operating profits	1.90	2.8	2.9	3.6	4.3	5.4

Fit a line of best fit.

11) The following data is the number of hours which ten students studied for English and the scores obtained by them in the examinations.

Hours studied	x :	4	9	10	12	14	22
Test score	y :	31	58	65	68	73	91

(i) Fit a straight line $y = ax + b$

(ii) Predict the score of the student who studied for 17 hours.

EXERCISE 7.3

Choose the correct answer

1) $\Delta f(x)$ =

   (a) $f(x+h)$ (b) $f(x)–f(x+h)$

   (c) $f(x+h)$–f(x) (d) $f(x)–f(x–h)$

2) $E^2 f(x)$ =

   (a) $f(x+h)$ (b) $f(x+2h)$ (c) $f(2h)$ (d) $f(2x)$

3) E =

   (a) $1+\Delta$ (b) $1 - \Delta$ (c) $\nabla + 1$ (d) $\nabla - 1$

4) $\nabla f(x+3h)$ =

   (a) $f(x+2h)$ (b) $f(x+3h)$-$f(x+2h)$

   (c) $f(x+3h)$ (d) $f(x+2h) - f(x - 3h)$

5) When $h = 1$, $\Delta(x^2)$ =

   (a) $2x$ (b) $2x - 1$ (c) $2x+1$ (d) 1

6) The normal equations for estimating a and b so that the line $y = ax + b$ may be the line of best fit are

   (a) $a\Sigma x_i^2 + b\Sigma x_i = \Sigma x_i y_i$ and $a\Sigma x_i + nb = \Sigma y_i$

   (b) $a\Sigma x_i + b\Sigma x_i^2 = \Sigma x_i y_i$ and $a\Sigma x_i^2 + nb = \Sigma y_i$

   (c) $a\Sigma x_i + nb = \Sigma x_i y_i$ and $a\Sigma x_i^2 + b\Sigma x_i = \Sigma y_i$

   (d) $a\Sigma x_i^2 + nb = \Sigma x_i y_i$ and $a\Sigma x_i + b\Sigma x_i = \Sigma y_i$

7) In a line of best fit $y = 5.8\ (x\text{-}1994) + 41.6$ the value of y when $x = 1997$ is

   (a) 50 (b) 54 (c) 59 (d) 60

8) Five data relating to x and y are to be fit in a straight line. It is found that $\Sigma x = 0$ and $\Sigma y = 15$. Then the y-intercept of the line of best fit is,

   (a) 1 (b) 2 (c) 3 (d) 4

9) The normal equations of fitting a straight line $y = ax + b$ are $10a + 5b = 15$ and $30a + 10b = 43$. The slope of the line of best fit is

(a) 1.2 (b) 1.3 (c) 13 (d) 12

10) The normal equations obtained in fitting a straight line $y = ax + b$ by the method of least squares over n points (x, y) are $4 = 4a + b$ and $\Sigma xy = 120a + 24b$. Then $n =$

(a) 30 (b) 5 (c) 6 (d) 4

PROBABILITY DISTRIBUTIONS 8

8.1 RANDOM VARIABLE AND PROBABILITY FUNCTION

Random variable

A **random variable** is a real valued function defined on a sample space S and taking values in $(-\infty , \infty)$

8.1.1 Discrete Random Variable

A random variable X is said to be discrete if it assumes only a finite or an infinite but countable number of values.

Examples

(i) Consider the experiment of tossing a coin twice. The sample points of this experiment are s_1 = (H, H), s_2 = (H, T), s_3 = (T, H) and s_4 = (T, T).

Random variable X denotes the number of heads obtained in the two tosses.

Then $X(s_1) = 2$ $\quad X(s_2) = 1$

$X(s_3) = 1$ $\quad X(s_4) = 0$

$R_X = \{0, 1, 2\}$

where s is the typical element of the sample space, $X(s)$ represents the real number which the random variable X associates with the outcome s.

R_X, the set of all possible values of X, is called the **range** space X.

(ii) Consider the experiment of rolling a pair of fair dice once. Then sample space

$$S = \{(1,1)\ (1,2)\ \ldots\ldots\ldots\ldots(1,6)$$
$$\vdots \quad \vdots \quad \vdots$$
$$(6,1)\ (6,2)\ \ldots\ldots\ldots\ldots(6,6)\}$$

Let the random variable X denote the sum of the scores on the two dice. Then $R_X = \{2, 3, 4, \ldots\ldots, 12\}$.

(iii) Consider the experiment of tossing of 3 coins simultaneously.

Let the random variable X be Number of heads obtained in this experiment.

Then

S = {HHH, HHT, HTT, TTT, TTH, THH, HTH, THT}

$R_X = \{0, 1, 2, 3\}$

(iv) Suppose a random experiment consists of throwing 4 coins and recording the number of heads.

Then $R_X = \{0, 1, 2, 3, 4\}$

The number of printing mistakes in each page of a book and the number of telephone calls received by the telephone operator of a firm, are some other examples of discrete random variable.

8.1.2 Probability function and Probability distribution of a Discrete random variable

Let X be a discrete random variable assuming values x_1, x_2, x_3... If there exists a function p denoted by $p(x_i) = P[X = x_i]$ such that

(i) $p(x_i) \geq 0$ for $i = 1, 2, \ldots$

(ii) $\sum\limits_i p(x_i) = 1$

then p is called as the **probability function** or **probability mass function (p.m.f)** of X.

The collection of all pairs (x_i, $p(x_i)$) is called the probability distribution of X.

Example 1

Consider the experiment of tossing two coins. Let X be a random variable denoting the number of heads obtained.

X	**:**	**0**	**1**	**2**
p(x_i)	**:**	$\frac{1}{4}$	$\frac{1}{2}$	$\frac{1}{4}$

Is p(x_i) a p.m.f ?

Solution :

(i) $p(x_i) > 0$ for all i

(ii) $\Sigma p(x_i) = p(0) + p(1) + p(2)$

$$= \frac{1}{4} + \frac{1}{2} + \frac{1}{4} = 1$$

Hence $p(x_i)$ is a p.m.f.

Example 2

Consider the discrete random variable X as the sum of the numbers that appear, when a pair of dice is thrown. The probability distribution of X is

X	**:**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
***p*(x_i)**	**:**	$\frac{1}{36}$	$\frac{2}{36}$	$\frac{3}{36}$	$\frac{4}{36}$	$\frac{5}{36}$	$\frac{6}{36}$	$\frac{5}{36}$	$\frac{4}{36}$	$\frac{3}{36}$	$\frac{2}{36}$	$\frac{1}{36}$

Is p(x_i) a p.m.f?

Solution :

$p(x_i) > 0$ for all i

(ii) $\Sigma p(x_i) = \frac{1}{36} + \frac{2}{36} + \frac{3}{36} + ...+ \frac{1}{36} = 1$

Hence $p(x_i)$ is a p.m.f.

8.1.3 Cumulative Distribution function : (c.d.f.)

Let X be a discrete random variable. The function F(x) is said to be the cumulative distribution function (c.d.f.) of the random variable X if

$$F(x) = P(X \le x)$$

$$= \sum_i p(x_i)$$ where the sum is taken over i such that $x_i \le x$.

Remark : $P(a < X \le b) = F(b) - F(a)$

Example 3

A random variable X has the following probability function :

Values of X, x :	**-2**	**-1**	**0**	**1**	**2**	**3**
$p(x)$:	**0.1**	***k***	**0.2**	**2*k***	**0.3**	***k***

(i) Find the value of *k*

(ii) Construct the c.d.f. of X

Solution :

(i) Since $\sum_i p(x_i) = 1$,

$p(-2) + p(-1) + p(0) + p(1) + p(2) + p(3) = 1$

$0.1 + k + 0.2 + 2k + 0.3 + k = 1$

$0.6 + 4k = 1 \Rightarrow 4k = 1 - 0.6$

$4k = 0.4 \qquad \therefore k = \frac{.4}{4} = 0.1$

Hence the given probability function becomes,

x	:	-2	-1	0	1	2	3
$p(x)$	:	0.1	0.1	0.2	0.2	0.3	0.1

(ii) Cumulative distribution function $F(x) = P(X \le x)$

x	$F(x) = P(X \le x)$
-2	$F(-2) = P(X \le -2) = 0.1$
-1	$F(-1) = P(X \le -1) = P(X = -2) + P(X = -1)$
	$= 0.1 + 0.1 = 0.2$
0	$F(0) = P(X \le 0) = P(X=-2) + P(X=-1) + P(X = 0)$
	$= 0.1 + 0.1 + 0.2 = 0.4$
1	$F(1) = P(X \le 1) = 0.6$
2	$F(2) = P(X \le 2) = 0.9$
3	$F(3) = P(X \le 3) = 1$

$$\begin{aligned} F(x) &= 0 \text{ if } x < -2 \\ &= .1 \text{ if } -2 \le x < -1 \\ &= .2 \text{ if } -1 \le x < 0 \\ &= .4 \text{ if } 0 \le x < 1 \\ &= .6 \text{ if } 1 \le x < 2 \\ &= 0.9, \text{ if } 2 \le x < 3 \\ &= 1 \text{ if } x \ge 3 \end{aligned}$$

Example 4

For the following probability distribution of X

X	**:**	**0**	**1**	**2**	**3**
***p*(*x*)**	**:**	$\frac{1}{6}$	$\frac{1}{2}$	$\frac{3}{10}$	$\frac{1}{30}$

Find (i) P(X $\le$ 1) (ii) P(X $\le$ 2) (iii) P(0< X < 2)

Solution :

(i) $P(X \le 1) = P(X = 0) + P(X = 1)$

$= p(0) + p(1)$

$= \frac{1}{6} + \frac{1}{2} = \frac{4}{6} = \frac{2}{3}$

(ii) $P(X \leq 2) = P(X = 0) + P(X = 1) + P(X = 2)$

$$= \frac{1}{6} + \frac{1}{2} + \frac{3}{10} = \frac{29}{30}$$

Aliter $P(X \leq 2)$ can also be obtained as

$$P(X \leq 2) = 1 - P(X > 2)$$

$$= 1 - P(X = 3) = 1 - \frac{1}{30} = \frac{29}{30}$$

(iii) $P(0 < X < 2) = P(X = 1) = \frac{1}{2}$

8.1.4 Continuous Random Variable

A random variable X is said to be continuous if it takes a continuum of values. i.e. if it takes all possible values between certain defined limits.

For example,

(i) The amount of rainfall on a rainy day.

(ii) The height of individuals. (iii) The weight of individuals.

8.1.5 Probability function

A function f is said to be the probability density function (p.d.f) of a continuous random variable X if the following conditions are satisfied

(i) $f(x) \geq 0$ for all x (ii) $\int_{-\infty}^{\infty} f(x)\,dx = 1$

Remark :

(i) The probability that the random variable X lies in the interval (a, b) is given by $P(a < X < b) = \int_a^b f(x)\,dx$.

(ii) $P(X = a) = \int_a^a f(x)\,dx = 0$

(iii) $P(a \leq X \leq b) = P(a \leq X < b) = P(a < X \leq b) = P(a < X < b)$

8.1.6 Continuous Distribution function

If X is a continuous random variable with p.d.f. $f(x)$, then the function $F_X(x) = P(X \leq x)$

$$= \int_{-\infty}^{x} f(t)\, dt$$

is called the distribution function (d.f.) or cumulative distribution function (c.d.f) of the random variable X.

Properties : The cumulative distribution function has the following properties.

(i) $\underset{x \to -\infty}{\mathrm{L}t}\, F(x) = 0$ i.e. $F(-\infty) = 0$

(ii) $\underset{x \to \infty}{\mathrm{L}t}\, F(x) = 1$ i.e. $F(\infty) = 1$

(iii) Let F be the c.d.f. of a continuous random variable X with p.d.f f. Then $f(x) = \frac{d}{dx} F(x)$ for all x at which F is differentiable.

Example 5

A continuous random variable X has the following p.d.f.

$$f(x) = \begin{cases} k(2-x) & \text{for } 0 < x < 2 \\ 0 & \text{otherwise} \end{cases}$$

Determine the value of k.

Solution :

If $f(x)$ be the p.d.f., then $\int_{-\infty}^{\infty} f(x)\, dx = 1$

$$\int_{-\infty}^{0} f(x)\, dx + \int_{0}^{2} f(x)\, dx + \int_{2}^{\infty} f(x)\, dx = 1$$

$$\Rightarrow \quad 0 + \int_{0}^{2} f(x)\, dx + 0 = 1$$

$$\Rightarrow \quad \int_{0}^{2} k(2-x)\, dx = 1$$

$$k\left(\int_0^2 2dx - xdx\right) = 1 \qquad \therefore \qquad k = \frac{1}{2}$$

Hence $f(x) = \begin{cases} \frac{1}{2}(2-x) & \text{for } 0 < x < 2 \\ 0 & \text{otherwise} \end{cases}$

Example 6

Verify that

$$f(x) = \begin{cases} \mathbf{3x^2} & \textbf{for } \mathbf{0 < x < 1} \\ \mathbf{0} & \textbf{otherwise} \end{cases}$$

is a p.d.f and evaluate the following probabilities

(i) P(X $\leq \frac{1}{3}$) **(ii) P($\frac{1}{3} \leq$ X $\leq \frac{1}{2}$)**

Solution :

Clearly $f(x) \geq 0$ for all x and hence one of the conditions for p.d.f is satisfied.

$$\int_{-\infty}^{\infty} f(x)\, dx = \int_0^1 f(x)\, dx = \int_0^1 3x^2\, dx = 1$$

$\therefore$ The other condition for p.d.f is also satisfied.

Hence the given function is a p.d.f

(i) P(X $\leq \frac{1}{3}$) $= \int_{-\infty}^{\frac{1}{3}} f(x)\, dx$ $\qquad$ P(X $\leq x$) $= \int_{-\infty}^{x} f(t)\, dt$

$$= \int_0^{\frac{1}{3}} 3x^2\, dx = \frac{1}{27}$$

(ii) P($\frac{1}{3} \leq$ X $\leq \frac{1}{2}$) $= \int_{\frac{1}{3}}^{\frac{1}{2}} f(x)\, dx$

$$= \int_{\frac{1}{3}}^{\frac{1}{2}} 3x^2\, dx = \frac{1}{8} - \frac{1}{27} = \frac{19}{216}$$

Example 7

Given the p.d.f of a continuous random variable X as follows

$$f(x) = \begin{cases} \boldsymbol{kx(1-x)} & \textbf{for } \boldsymbol{0 < x < 1} \\ \boldsymbol{0} & \textbf{otherwise} \end{cases}$$

Find *k* and c.d.f

Solution :

If X is a continuous random variable with p.d.f $f(x)$ then

$$\int_{-\infty}^{\infty} f(x)\,dx = 1$$

$$\int_{0}^{1} k\,x\,(1-x)\,dx = 1$$

$$k\left[\frac{x^2}{2} - \frac{x^3}{3}\right]_0^1 = 1 \qquad \therefore\; k = 6$$

Hence the given p.d.f becomes,

$$f(x) = \begin{cases} 6x(1-x) & \text{for } 0 < x < 1 \\ 0 & \text{otherwise} \end{cases}$$

To find c.d.f F(x)

$$F(x) = 0 \qquad \text{for } x \le 0$$

$$F(x) = P(X \le x) = \int_{-\infty}^{x} f(x)\,dx$$

$$= \int_{0}^{x} 6x(1-x)dx = 3x^2 - 2x^3 \qquad \text{for } 0 < x < 1$$

$$F(x) = 1 \qquad \text{for } x \ge 1$$

$\therefore$ The c.d.f of X is as follows.

$$\begin{aligned} F(x) &= 0 && \text{for } x \le 0 \\ &= 3x^2 - 2x^3 && \text{for } 0 < x < 1. \\ &= 1 && \text{for } x \ge 1 \end{aligned}$$

Example 8

Suppose that the life in hours of a certain part of radio tube is a continuous random variable X with p.d.f is given by

$$\mathbf{f}(x) = \begin{cases} \dfrac{\mathbf{100}}{x^2}, & \textbf{when } x \geq \mathbf{100} \\ \mathbf{0} & \textbf{elsewhere} \end{cases}$$

(i) What is the probability that all of three such tubes in a given radio set will have to be replaced during the first of 150 hours of operation?

(ii) What is the probability that none of three of the original tubes will have to be replaced during that first 150 hours of operation?

Solution :

(i) A tube in a radio set will have to be replaced during the first 150 hours if its life is < 150 hours. Hence, the required probability '*p*' that a tube is replaced during the first 150 hours is,

$$p = P(X \leq 150) = \int_{100}^{150} f(x)\,dx$$

$$= \int_{100}^{150} \frac{100}{x^2}\,dx = \frac{1}{3}$$

∴ The probability that all three of the original tubes will have to replaced during the first 150 hours $= p^3 = \left(\frac{1}{3}\right)^3 = \frac{1}{27}$

(ii) The probability that a tube is not replaced during the first 150 hours of operation is given by

$$P(X > 150) = 1 - P(X \leq 150) = 1 - \frac{1}{3} = \frac{2}{3}$$

∴ the probability that none of the three tubes will be replaced during the 150 hours of operation $= \left(\frac{2}{3}\right)^3 = \frac{8}{27}$

EXERCISE 8.1

1) Which of the following set of functions define a probability space on S = $[x_1, x_2, x_3]$?

(i) $p(x_1) = \frac{1}{3}$ $p(x_2) = \frac{1}{2}$ $p(x_3) = \frac{1}{4}$

(ii) $p(x_1) = \frac{1}{3}$ $p(x_2) = \frac{1}{6}$ $p(x_3) = \frac{1}{2}$

(iii) $p(x_1) = 0$ $p(x_2) = \frac{1}{3}$ $p(x_3) = \frac{2}{3}$

(iv) $p(x_1) = p(x_2) = \frac{2}{3}$ $p(x_3) = \frac{1}{3}$

2) Consider the experiment of throwing a single die. The random variable X represents the score on the upper face and assumes the values as follows:

X	:	1	2	3	4	5	6
$p(x_i)$	:	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{6}$

Is $p(x_i)$ a p.m.f?

3) A random variable X has the following probability distribution.

Values of X, x :	0	1	2	3	4	5	6	7	8
$p(x)$:	a	$3a$	5a	$7a$	$9a$	$11a$	$13a$	$15a$	$17a$

(i) Determine the value of a

(ii) Find $P(X < 3)$, $P(X > 3)$ and $P(0< X < 5)$

4) The following function is a probability mass function - Verify.

$$p(x) = \begin{cases} \frac{1}{3} & \text{for } x=1 \\ \frac{2}{3} & \text{for } x=2 \\ 0 & \text{otherwise} \end{cases}$$

Hence find the c.d.f

5) Find k if the following function is a probability mass function.

$$p(x) = \begin{cases} \frac{k}{6} & \text{for } x = 0 \\ \frac{k}{3} & \text{for } x = 2 \\ \frac{k}{2} & \text{for } x = 4 \\ 0 & \text{otherwise} \end{cases}$$

6) A random variable X has the following probability distribution

values of X, x :	-2	0	5
$p(x)$:	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{2}$

Evaluate the following probabilities

(a) $P(X \leq 0)$ (b) $P(X < 0)$ (c) $P(0 \leq X \leq 10)$

7) A random variable X has the following probability function

Values of X, x :	0	1	2	3
$p(x)$:	$\frac{1}{16}$	$\frac{3}{8}$	k	$\frac{5}{16}$

(i) Find the value of k (ii) Construct the c.d.f. of X

8) A continuous random variable has the following p.d.f

$f(x) = kx^2, \quad 0 \leq x \leq 10$

$= 0$ otherwise.

Determine k and evaluate (i) $P(.2 \leq X \leq 0.5)$ (ii) $P(X \leq 3)$

9) If the function $f(x)$ is defined by

$f(x) = ce^{-x}, \quad 0 \leq x < \infty$. Find the value of c.

10) Let X be a continuous random variable with p.d.f.

$$f(x) = \begin{cases} ax, & 0 < x \leq 1 \\ a, & 1 \leq x \leq 2 \\ -ax + 3a, & 2 \leq x \leq 3 \\ 0 & \text{otherwise} \end{cases}$$

(i) Determine the constant a

(ii) Compute $P(X \leq 1.5)$

11) Let X be the life length of a certain type of light bulbs in hours. Determine 'a' so that the function

$f(x) = \frac{a}{x^2}, \quad 1000 \leq x \leq 2000$

$= 0$ otherwise.

may be the probability density function.

12) The kms. X in thousands which car owners get with a certain kind of tyre is a random variable having p.d.f.

$$f(x) = \frac{1}{20} e^{-\frac{x}{20}}, \quad \text{for } x > 0$$
$$= 0 \quad \text{for } x \leq 0$$

Find the probabilities that one of these tyres will last

(i) atmost 10,000 kms

(ii) anywhere from 16,000 to 24,000 kms

(iii) atleast 30,000 kms.

8.2 MATHEMATICAL EXPECTATION

The concept of Mathematical expectation plays a vital role in statistics. Expected value of a random variable is a weighted average of all the possible outcomes of an experiment.

If X is a discrete random value which can assume values $x_1, x_2, \dots x_n$ with respective probabilities $p(x_i) = P[X = x_i]$; $i = 1, 2 \dots n$ then its **mathematical expectation** is defined as

$$E(X) = \sum_{i=1}^{n} x_i\, p(x_i), \qquad \left(\text{Here } \sum_{i=1}^{n} p(x_i) = 1\right)$$

If X is a continuous random variable with probability density function $f(x)$, then

$$E(X) = \int_{-\infty}^{\infty} x\, f(x)\, dx$$

Note

E(X) is also known as the mean of the random variable X.

Properties

1) $E(c) = c$ where c is constant

2) $E(X + Y) = E(X) + E(Y)$

3) $E(aX + b) = aE(X) + b$ where a and b are constants.

4) $E(XY) = E(X)\,E(Y)$ if X and Y are independent

Note

The above properties holds good for both discrete and continous random variables.

Variance

Let X be a random variable. Then the **Variance of X**, denoted by Var(X) or σ^2_x is

$$\text{Var}(X) = \sigma^2_x = E[X - E(X)]^2$$
$$= E(X^2) - [E(X)]^2$$

The positive square root of Var(X) is called the **Standard Deviation** of X and is denoted by σ_x.

Example 9

A multinational bank is concerned about the waiting time (in minutes) of its customer before they would use ATM for their transaction. A study of a random sample of 500 customers reveals the following probability distribution.

X	**:**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
***p*(*x*)**	**:**	**.20**	**.18**	**.16**	**.12**	**.10**	**.09**	**.08**	**.04**	**.03**

Calculate the expected value of waiting time, X, of the customer

Solution :

Let X denote the waiting time (in minutes) per customer.

X	:	0	1	2	3	4	5	6	7	8
$p(x)$	:	.20	.18	.16	.12	.10	.09	.08	.04	.03

Then $E(X) = \Sigma x\, p(x)$

$$= (0 \times .2) + (1 \times 0.18) + ... + (8 \times 0.03) = 2.71$$

The expected value of X is equal to 2.71 minutes. Thus the average waiting time of a customer before getting access to ATM is 2.71 minutes.

Example 10

Find the expected value of the number of heads appearing when two fair coins are tossed.

Solvtion :

Let X be the random variable denoting the number of heads.

Possible values of X	:	0	1	2
Probabilities $p(x_i)$	:	$\frac{1}{4}$	$\frac{1}{2}$	$\frac{1}{4}$

The Expected value of X is

$$E(X) = x_1\, p(x_1) + x_2\, p(x_2) + x_3\, p(x_3)$$

$$= 0\left(\frac{1}{4}\right) + 1\left(\frac{1}{2}\right) + 2\left(\frac{1}{4}\right) = 1$$

Therefore, the expected number of heads appearing in the experiment of tossing 2 fair coins is 1.

Example 11

The probability that a man fishing at a particular place will catch 1, 2, 3, 4 fish are 0.4, 0.3, 0.2 and 0.1 respectively. What is the expected number of fish caught?

Solution :

Possible values of X	:	1	2	3	4
Probabilities $p(x_i)$	:	0.4	0.3	0.2	0.1

$$\therefore \quad E(X) = \sum_i x_i\, p(x_i)$$

$$= x_1\, p(x_1) + x_2\, p(x_2) + x_2\, p(x_3) + x_4\, p(x_4)$$

$$= 1\,(.4) + 2(.3) + 3(.2) + 4(.1)$$

$$= .4 + .6 + .6 + .4 = 2$$

Example 12

A person receives a sum of rupees equal to the square of the number that appears on the face when a balance die is tossed. How much money can he expect to receive?

Solution :

Random variable X: as square of the number that can appear on the face of a die. Thus

possible values of	X	:	1^2	2^2	3^2	4^2	5^2	6^2
probabilities	$p(x_i)$	:	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{6}$

The Expected amount that he receives,

$$E(X) = 1^2\left(\frac{1}{6}\right) + 2^2\left(\frac{1}{6}\right) + \ldots + 6^2\left(\frac{1}{6}\right)$$

$$= \text{Rs.}\frac{91}{6}$$

Example 13

A player tosses two fair coins. He wins Rs.5 if two heads appear, Rs. 2 if 1 head appears and Rs.1 if no head occurs. Find his expected amount of gain.

Solution :

Consider the experiment of tossing two fair coins. There are four sample points in the sample space of this experiment.

i.e. S = {HH, HT, TH, TT}

Let X be the random variable denoting the amount that a player wins associated with the sample point.

Thus,

Possible values of X (Rs.)		:	5	2	1
Probabilities	$p(x_i)$	:	$\frac{1}{4}$	$\frac{1}{2}$	$\frac{1}{4}$

$$E(X) = 5\left(\frac{1}{4}\right) + 2\left(\frac{1}{2}\right) + 1\left(\frac{1}{4}\right)$$

$$= \frac{5}{4} + 1 + \frac{1}{4} = \frac{10}{4} = \frac{5}{2}$$

$$= \text{Rs. } 2.50$$

Hence expected amount of winning is Rs.2.50

Example 14

A random variable X has the probability function as follows :

values of X	**:**	**-1**	**0**	**1**
probability	**:**	**0.2**	**0.3**	**0.5**

Evaluate (i) E(3X +1) (ii) E(X^2) (iii) Var(X)

Solution :

X	:	–1	0	1
$p(x_i)$	:	0.2	0.3	0.5

(i) E(3X+1) = 3E(X) + 1

Now E(X) = –1 x 0.2 + 0 x 0.3 + 1 x 0.5

= –1 x 0.2 + 0 + 0.5 = 0.3

E(3X + 1) = 3(0.3) + 1 = 1.9

(ii) $E(X^2) = \Sigma x^2\, p(x)$

$= (-1)^2 \times 0.2 + (0)^2 \times 0.3 + (1)^2 \times 0.5$

= 0.2 + 0 + 0.5 = 0.7

(iii) Var(X) $= E(X^2) - [E(X)]^2$

$= .7 - (.3)^2 = .61$

Example 15

Find the mean, variance and the standard deviation for the following probability distribution

Values of X, *x*	**:**	**1**	**2**	**3**	**4**
probability, *p*(*x*)	**:**	**0.1**	**0.3**	**0.4**	**0.2**

Solution :

Mean $= E(X) = \Sigma x\, p(x)$

= 1(0.1) + 2(0.3) + 3(0.4) + 4(0.2) = 2.7

Variance $= E(X^2) - [E(X)]^2$

Now $E(X^2) = \Sigma x^2\, p(x)$

$= 1^2(0.1) + 2^2(0.3) + 3^2(0.4) + 4^2(0.2) = 8.1$

$\therefore$ Variance $= 8.1 - (2.7)^2$

$= 8.1 - 7.29 = .81$

Standard Deviation $= \sqrt{0.81} = 0.9$

Example 16

Let X be a continuous random variable with p.d.f.

$$f(x) = \begin{cases} \frac{1}{2} & \text{for } -1 < x < 1 \\ 0 & \text{otherwise} \end{cases}$$

Find (i) E(X) (ii) E(X^2) (iii) Var(X)

Solution :

(i) $E(X) = \int_{-\infty}^{\infty} x f(x)\, dx$ (by definition)

$= \frac{1}{2} \int_{-1}^{1} x\, dx = \frac{1}{2}\left[\frac{x^2}{2}\right]_{-1}^{1} = 0$

(ii) $E(X^2) = \int_{-1}^{1} x^2 f(x)\, dx$

$= \int_{-1}^{1} x^2 \frac{1}{2} dx$

$= \frac{1}{2}\left[\frac{x^3}{3}\right]_{-1}^{1} = \frac{1}{3}$

(iii) $\text{Var}(X) = E(X^2) - [E(X)]^2$

$= \frac{1}{3} - 0 = \frac{1}{3}$

EXERCISE 8.2

1) A balanced die is rolled. A person recieves Rs. 10 when the number 1 or 3 or 5 occurs and loses Rs. 5 when 2 or 4 or 6 occurs. How much money can he expect on the average per roll in the long run?

2) Two unbiased dice are thrown. Find the expected value of the sum of the points thrown.

3) A player tossed two coins. If two heads show he wins Rs. 4. If one head shows he wins Rs. 2, but if two tails show he must pay Rs. 3 as penalty. Calculate the expected value of the sum won by him.

4) The following represents the probability distribution of D, the daily demand of a certain product. Evaluate E(D).

D :	1	2	3	4	5
P[D=d] :	0.1	0.1	0.3	0.3	0.2

5) Find E(2X-7) and E(4X + 5) for the following probability distribution.

X :	–3	–2	–1	0	1	2	3
$p(x)$:	. 05	.1	.3	0	.3	.15	.1

6) Find the mean, variance and standard deviation of the following probability distribution.

Values of X :	–3	–2	–1	0	1	2	3
Probability $p(x)$:	$\frac{1}{7}$	$\frac{1}{7}$	$\frac{1}{7}$	$\frac{1}{7}$	$\frac{1}{7}$	$\frac{1}{7}$	$\frac{1}{7}$

7) Find the mean and variance for the following probability distribution.

$$f(x) = \begin{cases} 2e^{-2x}, & x \geq 0 \\ 0 \quad , & x < 0 \end{cases}$$

8.3 DISCRETE DISTRIBUTIONS

We know that the frequency distributions are based on observed data derived from the collected sample information. For example, we may study the marks of the students of a class and formulate a frequency distribution as follows:

Marks	No. of students
0 - 20	10
20-40	12
40-60	25
60-80	15
80-100	18
Total	**80**

The above example clearly shows that the observed frequency distributions are obtained by grouping. Measures like averages, dispersion, correlation, etc. generally provide us a consolidated view of the whole observed data. This may very well be used in formulating certain ideas (inference) about the characteristics of the whole set of data.

Another type of distribution in which variables are distributed according to some definite probability law which can be expressed mathematically are called theoretical probability distribution.

The probability distribution is a total listing of the various values the random variable can take along with the corresponding probabilities of each value. For example; consider the pattern of distribution of machine breakdown in a manufacturing unit. The random variable would be the various values the machine breakdown could assume. The probability corresponding to each value of the breakdown as the relative frequency of occurence of the breakdown. This probability distribution is constructed by the actual breakdown pattern discussed over a period of time.

Theoretical probability distributions are basically of two types

(i) Discrete and (ii) Continuous

In this section, we will discuss theoretical discrete distributions namely, Binomial and Poisson distributions.

8.3.1 Binomial Distribution

It is a distribution associated with repetition of independent trials of an experiment. Each trial has two possible outcomes, generally called success and failure. Such a trial is known as **Bernoulli trial.**

Some examples of Bernoulli trials are :

(i) a toss of a coin (Head or tail)

(ii) the throw of a die (even or odd number)

An experiment consisiting of a repeated number of Bernoulli trials is called a **binomial experiment.** A binomial experiment must possess the following properties:

(i) there must be a fixed number of trials.

(ii) all trials must have identical probabilities of success (p) i.e. if we call one of the two outcomes as "success" and the other as "failure", then the probability p of success remains as constant throughout the experiment.

(iii) the trials must be independent of each other i.e. the result of any trial must not be affected by any of the preceeding trial.

Let X denote the number of successes in 'n' trials of a binomial experiment. Then X follows a binomial distribution with parameters n and p and is denoted by X~B(n, p).

A random variable X is said to follow **Binomial distribution** with parameters n and p, if it assumes only non-negative values and its probability mass function is given by

$$P[X=x] = p(x) = {}^nC_x\, p^x\, q^{n-x}\,;\ x = 0, 1, 2, ..n\ ;\ q = 1- p$$

Remark

(i) $\sum\limits_{x=0}^{n} p(x) = \sum\limits_{x=0}^{n} {}^nC_x\, p^x\, q^{n-x} = (q + p)^n = 1$

(ii) ${}^nC_r = \dfrac{n(n-1)\ldots(n-\overline{r-1})}{1.2.3\ldots r}$

Mean and Variance

For the binomial distribution

Mean = np

Variance $= npq$; Standard Deviation = $\sqrt{npq}$

Example 17

What is the probability of getting exactly 3 heads in 8 tosses of a fair coin.

Solution :

Let p denote the probability of getting head in a toss.

Let X be the number of heads in 8 tosses.

Then $p = \frac{1}{2}$, $q = \frac{1}{2}$ and $n = 8$

Probability of getting exactly 3 heads is

$$P(X = 3) = {}^8c_3 \left(\frac{1}{2}\right)^3 \left(\frac{1}{2}\right)^5$$

$$= \frac{8 \text{ x } 7 \text{ x } 6}{1 \text{ x } 2 \text{ x } 3} \left(\frac{1}{2}\right)^8 = \frac{7}{32}$$

Example 18

Write down the Binomial distribution whose mean is 20 and variance being 4.

Solution :

Given mean, $np = 20$; variance, $npq = 4$

Now $q = \frac{npq}{np} = \frac{4}{20} = \frac{1}{5}$ $\therefore$ $p = 1 - q = \frac{4}{5}$

From np = 20, we have

$$n = \frac{20}{p} = \frac{20}{\frac{4}{5}} = 25$$

Hence the binomial distribution is

$$p(x) = {}^nC_x p^x q^{n-x} = {}^{25}C_x \left(\frac{4}{5}\right)^x \left(\frac{1}{5}\right)^{n-x}, \quad x = 0, 1, 2, \ldots, 25$$

Example 19

On an average if one vessel in every ten is wrecked, find the probability that out of five vessels expected to arrive, atleast four will arrive safely.

Solution :

Let the probability that a vessel will arrive safely, $p = \frac{9}{10}$

Then probability that a vessel will be wrecked, $q = 1-p = \frac{1}{10}$

No. of vessels, $n = 5$

$\therefore$ The probability that atleast 4 out of 5 vessels to arrive safely is

$$P(X \geq 4) = P(X = 4) + P(X = 5)$$

$$= {}^5C_4 \left(\frac{9}{10}\right)^4 \frac{1}{10} + {}^5C_5 \left(\frac{9}{10}\right)^5$$

$$= 5(.9)^4(.1) + (.9)^5 \quad = .91854$$

Example 20

For a binomial distribution with parameters n = 5 and p = .3 find the probabilities of getting (i) atleast 3 successes (ii) atmost 3 successes.

Solution :

Given $n = 5$, $p = .3$ $\therefore$ $q = .7$

(i) The probability of atleast 3 successes

$$P(X \geq 3) = P(X = 3) + P(X = 4) + P(X = 5)$$

$$= {}^5C_3 (0.3)^3 (0.7)^2 + {}^5C_4 (0.3)^4 (0.7) + {}^5C_5 (.3)^5 (7)^0$$

$$= .1631$$

(ii) The probability of atmost 3 successes

$$P(X \leq 3) = P(X = 0) + P(X = 1) + P(X = 2) + P(X = 3)$$

$$= (.7)^5 + {}^5C_1 (.7)^4 (.3) + {}^5C_2 (.7)^3 (.3)^2 + {}^5C_3 (.7)^2 (.3)^3$$

$$= .9692$$

8.3.2 Poisson distribution

Poisson distribution is also a discrete probability distribution and is widely used in statistics. Poisson distribibuition occurs when there are events which do not occur as outcomes of a definite number of trials of an experiment but which occur at random points of time and space wherein our interest lies only in the number of occurences of the event, not in its non-occurances. This distribution is used to describe the behaviour of rare events such as

(i) number of accidents on road

(ii) number of printing mistakes in a book

(iii) number of suicides reported in a particular city.

Poisson distribution is an approximation of binomial distribution when n (number of trials) is large and p, the probability of success is very close to zero with np as constant.

A random variable X is said to follow a **Poisson distribution** with parameter $\lambda > 0$ if it assumes only non-negative values and its probability mass function is given by

$$P[X = x] = p(x) = \frac{e^{-\lambda}\lambda^{x}}{x!}; \quad x = 0, 1, 2, \ldots$$

Remark

It should be noted that

$$\sum_{x=0}^{\infty} P[X = x] = \sum_{x=0}^{\infty} p(x) = 1$$

Mean and Variance

For the of poisson distribution

Mean, $E(X) = \lambda$, Variance, $Var(X) = \lambda$, S.D $= \sqrt{\lambda}$

Note

For poission distribution mean and variance are equal.

Example 21

Find the probability that atmost 5 defective fuses will be found in a box of 200 fuses if experience shows that 2 percent of such fuses are defective. ($e^{-4} = 0.0183$)

Solution :

p = probability that a fuse is defective = $\frac{2}{100}$

$n = 200$

$\therefore\ \lambda = np = \frac{2}{100} \times 200 = 4$

Let X denote the number of defective fuses found in a box.

Then the distribution is given by

$$P[X = x] = p(x) = \frac{e^{-4}4^{x}}{x!}$$

So, probability that atmost 5 defective fuses will be found in a box of 200 fuses

$= P(X \leq 5)$

$= P(X = 0) + P(X = 1) + P(X = 2)$

$\quad + P(X = 3) + P(X =4) + P(X = 5)$

$$= e^{-4} + \frac{e^{-4}4}{1!} + \frac{e^{-4}4^{2}}{2!} + \frac{e^{-4}4^{3}}{3!} + \frac{e^{-4}4^{4}}{4!} + \frac{e^{-4}4^{5}}{5!}$$

$$= e^{-4}\left(1 + \frac{4}{1!} + \frac{4^{2}}{2!} + \frac{4^{3}}{3!} + \frac{4^{4}}{4!} + \frac{4^{5}}{5!}\right)$$

$$=0.0183 \times \frac{643}{15} \quad =0.785$$

Example 22

Suppose on an average 1 house in 1000 in a certain district has a fire during a year. If there are 2000 houses in that district, what is the probability that exactly 5 houses will have fire during the year? (e^{-2} = .13534)

Solution :

p = probability that a house catches fire = $\frac{1}{1000}$

Here $n = 2000 \quad \therefore\ \lambda = np = 2000 \times \frac{1}{1000} = 2$

Let X denote the number of houses that has a fire

Then the distribution is given by $P[X = x] = \frac{e^{-2}2^x}{x!}$, $x = 0, 1, 2, ...$

Probability that exactly 5 houses will have a fire during the year is

$$P[X = 5] = \frac{e^{-2}2^5}{5!}$$

$$= \frac{.13534 \times 32}{120} = .0361$$

Example 23

The number of accidents in a year attributed to taxi drivers in a city follows poisson distribution with mean 3. Out of 1000 taxi drivers, find the approximate number of drivers with

(i) no accident in a year

(ii) more than 3 accidents in a year

Solution :

Here $\lambda = np = 3$

$N = 1000$

Then the distribution is

$P[X = x] = \frac{e^{-3}3^x}{x!}$ where X denotes the number accidents.

(i) P (no accidents in a year) = P(X = 0)

$= e^{-3} = 0.05$

$\therefore$ Number of drivers with no accident = 1000 x 0.05 = 50

(ii) P (that more than 3 accident in a year) = P(X > 3)

$= 1 - P(X \leq 3)$

$$= 1 - \left[e^{-3} + \frac{e^{-3}3^1}{1!} + \frac{e^{-3}3^2}{2!} + \frac{e^{-3}3^3}{3!}\right]$$

$= 1 - e^{-3}[1 + 3 + 4.5 + 4.5]$

$= 1 - e^{-3}(13) = 1 - .65 = .35$

$\therefore$ Number of drivers with more than 3 accidents

$= 1000 \times 0.35 = 350$

EXERCISE 8.3

1) Ten coins are thrown simultaneously. Find the probability of getting atleast 7 heads.

2) In a binomial distribution consisting of 5 independent trials, probabilities of 1 and 2 successes are 0.4096 and 0.2048 respectively. Find the parameter '*p*' of the distribution.

3) For a binomial distribution, the mean is 6 and the standard deviation is $\sqrt{2}$. Write down all the terms of the distribution.

4) The average percentage of failure in a certain examination is 40. What is the probability that out of a group of 6 candidates atleast 4 passed in the examination?

5) An unbiased coin is tossed six times. What is the probability of obtaining four or more heads?

6) It is stated that 2% of razor blades supplied by a manufacturer are defective. A random sample of 200 blades is drawn from a lot. Find the probability that 3 or more blades are defective. ($e^{-4} = .01832$)

7) Find the probability that atmost 5 defective bolts will be found in a box of 200 bolts, if it is known that 2% of such bolts are expected to be defective ($e^{-4} = 0.01832$)

8) An insurance company insures 4,000 people against loss of both eyes in car acidents. Based on previous data, the rates were computed on the assumption that on the average 10 persons in 1,00,000 will have car accidents each year that result in this type of injury. What is the probability that more than 3 of the injured will collect on their policy in a given year? ($e^{-0.4} = 0.6703$)

9) It is given that 3% of the electric bulbs manufactured by a company are defective. Find the probability that a sample of 100 bulbs will contain (i) no defective (ii) exactly one defective. $(e^{-3} = 0.0498)$.

10) Suppose the probability that an item produced by particular machine is defective equals 0.2. If 10 items produced from this machine are selected at random, what is the probability that not more than one defective is found? $(e^{-2} = .13534)$

8.4 CONTINUOUS DISTRIBUTIONS

The binomial and Poisson distributions discussed in the previous section are the most useful theoretical distributions. In order to have mathematical distribution suitable for dealing with quantities whose magnitudes vary continuously like heights and weights of individuals, a continuous distribution is needed. Normal distribution is one of the most widely used continuous distributions.

8.4.1 Normal Distribution

Normal Distribution is considered to be the most important and powerful of all the distributions in statistics. It was first introduced by De Moivre in 1733 in the development of probability. Laplace (1749 - 1827) and Gauss (1827 - 1855) were also associated with the development of Normal distribution.

A random variable X is said to follow a **Normal Distribution** with mean μ and variance σ^2 denoted by $X \sim N(\mu, \sigma^2)$, if its probability density function is given by

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}, \quad -\infty < x < \infty,\ -\infty < \mu < \infty,\ \sigma > 0$$

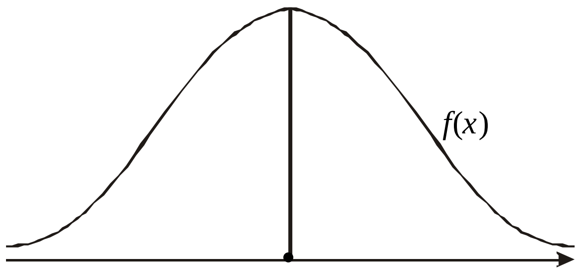
Remark

The parameters μ and σ^2 completely describe the normal distribution. Normal distribution could be also considered as limiting form of binomial distribution under the following conditions:

(i) n, the number of trials is indefinitely large i.e. $n \to \infty$

(ii) neither p nor q is very small.

The graph of the p.d.f of the normal distribution is called the **Normal curve**, and it is given below.



$x = \mu$

Normal probability curve

8.4.2 Properties of Normal Distribution

The following are some of the important properties of the normal curve and the normal distribution.

(i) The curve is "bell - shaped" and symmetric about $x = \mu$

(ii) Mean, Median and Mode of the distribution coincide.

(iii) There is one maximum point of the normal curve which occurs at the mean (μ). The height of the curve declines as we go in either direction from the mean.

(iv) The two tails of the curve extend infinitely and never touch the horizontal (x) axis.

(v) Since there is only one maximum point, the normal curve is unimodal i.e. it has only one mode.

(vi) Since $f(x)$ being the probability, it can never be negative and hence no portion of the curve lies below the x - axis.

(vii) The points of inflection are given by $x = \mu \pm \sigma$

(viii) Mean Deviation about mean

$$\sqrt{\frac{2}{\pi}}\sigma = \frac{4}{5}\sigma$$

(ix) Its mathematical equation is completely determined if the mean and S.D are known i.e. for a given mean μ and S.D σ, there is only one Normal distribution.

(x) Area Property : For a normal distribution with mean μ and S.D σ, the total area under normal curve is 1, and

(a) $P(\mu - \sigma < X < \mu + \sigma) = 0.6826$

i.e. (mean) $\pm 1\sigma$ covers 68.27%;

(b) $P(\mu - 2\sigma < X < \mu + 2\sigma) = 0.9544$

i.e. (mean) $\pm 2\sigma$ covers 95.45% area

(c) $P(\mu - 3\sigma < X < \mu + 3\sigma) = 0.9973$

i.e. (mean) $\pm 3\sigma$ covers 99.73% area

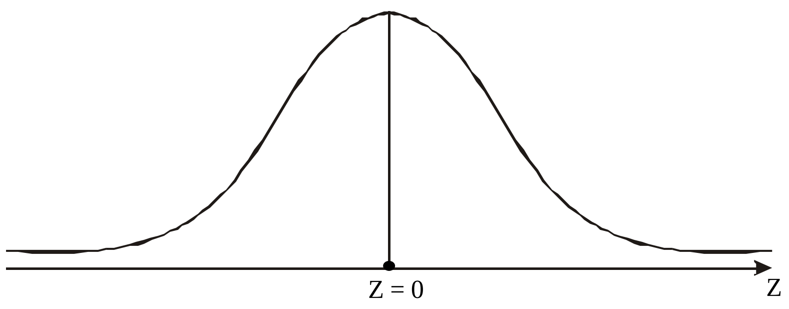
10.4.3 Standard Normal Distribution

A random variable which has a normal distribution with a mean $\mu = 0$ and a standard deviation $\sigma = 1$ is referred to as **Standard Normal Distribution.**

Remark

(i) If $X \sim N(\mu, \sigma^2)$, then $Z = \frac{X-\mu}{\sigma}$ is a standard normal variate with $E(Z) = 0$ and $var(Z) = 1$ i.e. $Z \sim N(0, 1)$.

(ii) It is to be noted that the standard normal distribution has the same shape as the normal distribution but with the special properties of $\mu = 0$ and $\sigma = 1$.



A random variable Z is said to have a **standard normal distribution** if its probability density function is given by

$$\varphi(z) = \frac{1}{\sqrt{2\pi}} e^{-\frac{z^2}{2}} \quad , \qquad -\infty < z < \infty$$

Example 24

What is the probability that Z

(a) lies between 0 and 1.83

(b) is greater than 1.54

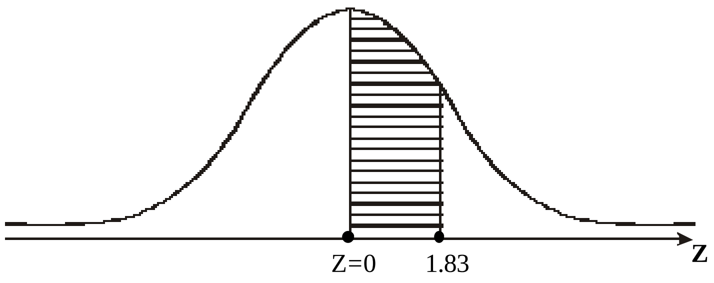
(c) is greater than -0.86

(d) lies between 0.43 and 1.12

(e) is less than 0.77

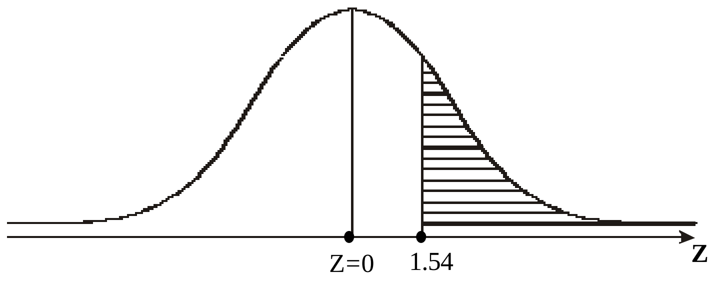
Solution :

(a) **Z lies between 0 and 1.83.**



$P(0 \leq Z \leq 1.83) = 0.4664$ (obtained from the tables directly)

(b) **Z is greater than 1.54 i.e. $P(Z \geq 1.54)$**

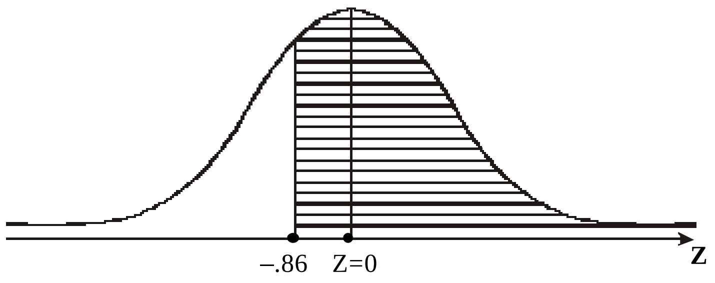


Since the total area to the right of $Z = 0$ is 0.5 and area between $Z = 0$ and 1.54 (from tables) is 0.4382

$P(Z \geq 1.54) = 0.5 - P(0 \leq Z \leq 1.54)$

$= 0.5 - .4382 = .0618$

(c) **Z is greater than −0.86 i.e. P(Z ≥ −0.86)**



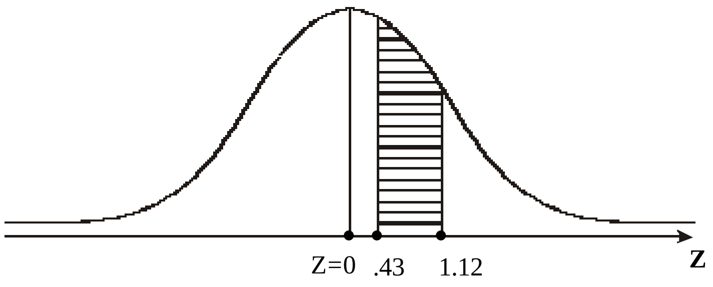
Here the area of interest $P(Z \geq -0.86)$ is represented by the two components.

(i) Area between $Z = -0.86$ and $Z = 0$, which is equal to 0.3051 (from tables)

(ii) $Z > 0$, which is 0.5

$\therefore P(Z \geq -0.86) = 0.3051 + 0.5 = 0.8051$

(d) **Z lies between 0.43 and 1.12**

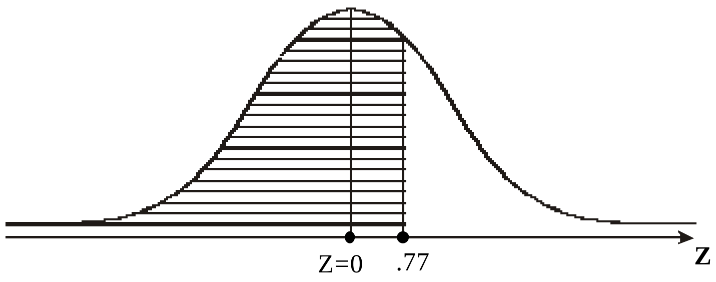


$\therefore P(0.43 \leq Z \leq 1.12) = P(0 \leq Z \leq 1.12) - P(0 \leq Z \leq 0.43)$

$= 0.3686 - 0.1664$ (from tables)

$= 0.2022.$

(e) **Z is less than 0.77**



$P(Z \leq 0.77) = 0.5 + P(0 \leq Z \leq 0.77)$

$= 0.5 + .2794 = .7794$ (from tables)

Example 25

If X is a normal random variable with mean 100 and variance 36

find (i) P(X > 112) (ii) P(X < 106) (iii) P(94 < X < 106)

Solution :

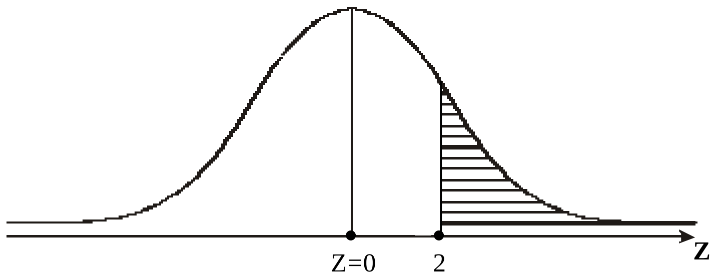
Mean, $\mu = 100$; Variance, $\sigma^2 = 36$; S.D, $\sigma = 6$

Then the standard normal variate Z is given by

$Z = \frac{X-\mu}{\sigma} = \frac{X-100}{6}$

(i) When X = 112, then $Z = \frac{112-100}{6} = 2$

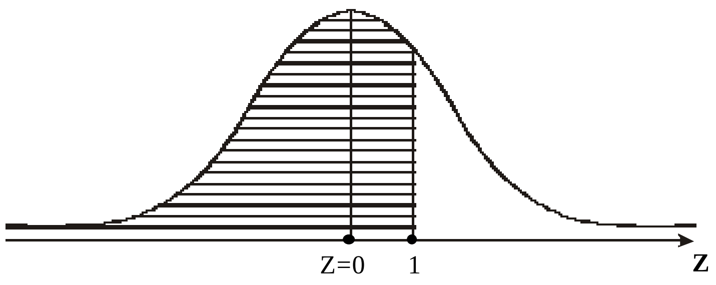
$\therefore P(X > 112) = P(Z > 2)$



$= P(0 \leq Z < \infty) - P(0 \leq Z \leq 2)$

$= 0.5 - 0.4772 = 0.0228$ (from tables)

(ii) For a given value $X = 106$, $Z = \frac{106 - 100}{6} = 1$



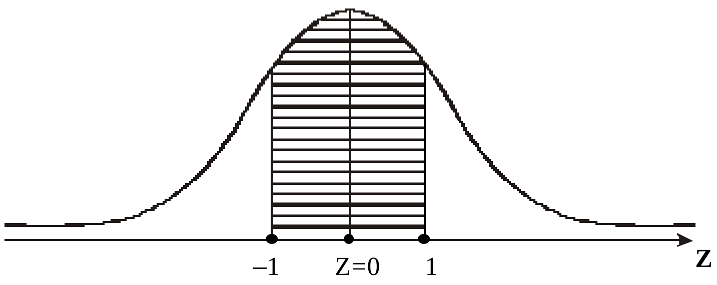
$P(X < 106) = P(Z < 1)$

$= P(-\infty \leq Z \leq 0) + P(0 \leq Z \leq 1)$

$= 0.5 + 0.3413 = 0.8413$ (from tables)

(iii) When $X = 94$, $Z = \frac{94 - 100}{6} = -1$

$X = 106$, $Z = \frac{106 - 100}{6} = +1$



$\therefore$ $P(94 < X < 106) = P(-1 < Z < 1)$

$= P(-1 < Z < 0) + P(0 < Z < 1)$

$= 2\,P(0 < Z < 1)$ (by symmetry)

$= 2\,(0.3413)$

$= 0.6826$

Example 26

In a sample of 1000 candidates the mean of certain test is 45 and S.D 15. Assuming the normality of the distrbution find the following:

(i) How many candidates score between 40 and 60?

(ii) How many candidates score above 50?

(iii) How many candidates score below 30?

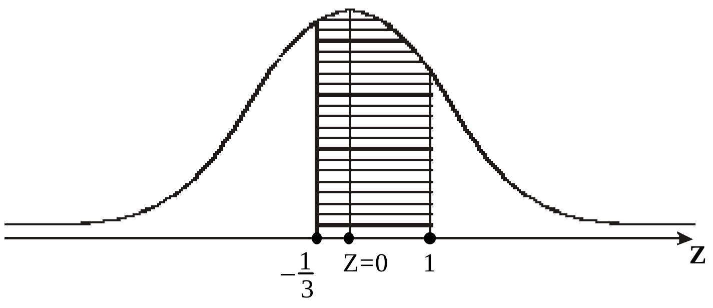
Solution :

Mean = μ = 45 and S.D. = σ = 15

Then $Z = \dfrac{X-\mu}{\sigma} = \dfrac{X-45}{15}$

(i) $P(40 < X < 60) = P(\dfrac{40-45}{15} < Z < \dfrac{60-45}{15})$

$= P(-\dfrac{1}{3} < Z < 1)$



$= P(-\dfrac{1}{3} \le Z \le 0) + P(0 \le Z \le 1)$

$= P(0 \le Z \le 0.33) + P(0 \le Z \le 1)$

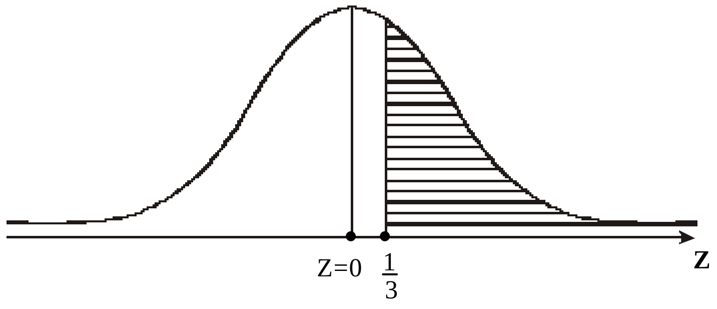
= 0.1293 + 0.3413 (from tables)

P(40 < X < 60) = 0.4706

Hence number of candidates scoring between 40 and 60

= 1000 x 0.4706 = 470.6 $\simeq$ 471

(ii) $P(X > 50) = P(Z > \frac{1}{3})$



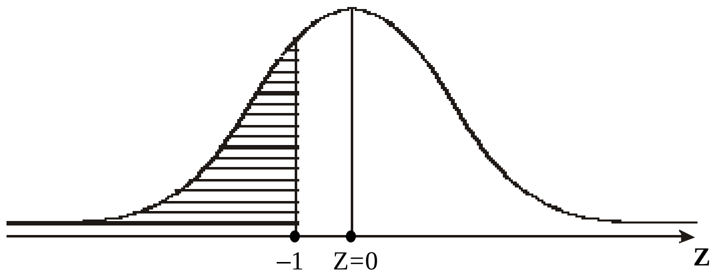
$= 0.5 - P(0 < Z < \frac{1}{3}) = 0.5 - P(0 < Z\ 0.33)$

$= 0.5 - 0.1293 = 0.3707$ (from tables)

Hence number of candidates scoring above 50

$= 1000 \times 0.3707 = 371.$

(iii) $P(X < 30) = P(Z < -1)$



$= 0.5 - P(-1 \leq Z \leq 0)$

$= 0.5 - P(0 \leq Z \leq 1)$ ∵ Symmetry

$= 0.5 - 0.3413 = 0.1587$ (from tables)

∴ Number of candidates scoring less than 30

$= 1000 \times 0.1587 = 159$

Example 27

The I.Q (intelligence quotient) of a group of 1000 school children has mean 96 and the standard deviation 12.

Assuming that the distribution of I.Q among school children is normal, find approximately the number of school children having I.Q.

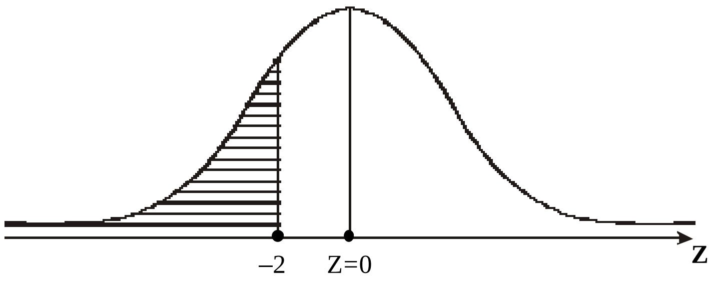
(i) less than 72 (ii) between 80 and 120

Solution :

Given $N = 1000$, $\mu = 96$ and $\sigma = 12$

Then $Z = \frac{X-\mu}{\sigma} = \frac{X-96}{12}$

(i) $P(X < 72) = P(Z < -2)$



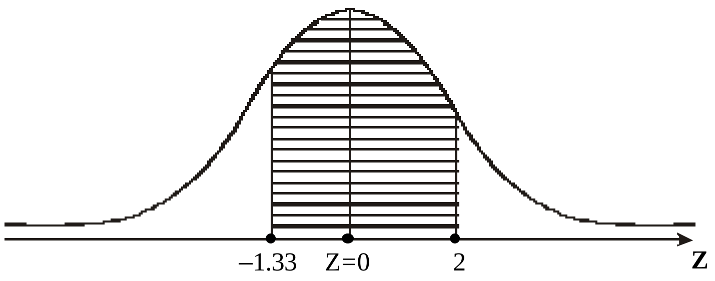
$= P(-\infty < Z \leq 0) - P(-2 \leq Z \leq 0)$

$= P(0 \leq Z < \infty) - P(0 \leq Z \leq 2)$ (By symmetry)

$= 0.5 - 0.4772$ (from tables) $= 0.0228$.

$\therefore$ Number of school children having I.Q less than 72
$= 1000 \times 0.0228 = 22.8 \simeq 23$

(ii) $P(80 < X < 120) = P(-1.33 < Z < 2)$



$= P(-1.33 \leq Z \leq 0) + P(0 \leq Z \leq 2)$

$= P(0 \leq Z \leq 1.33) + P(0 \leq Z \leq 2)$

$= .4082 + .4772$ (from tables)

$= 0.8854$

$\therefore$ Number of school children having I.Q. between 80 and 120

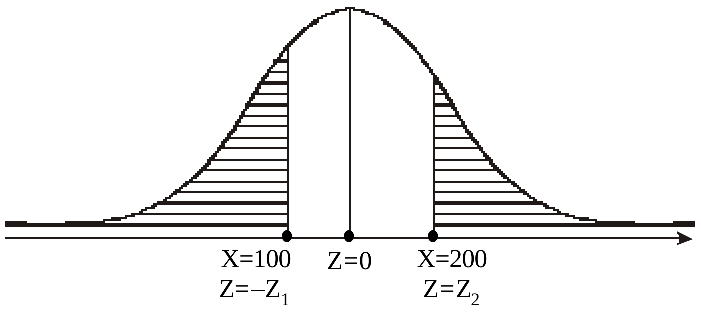
$= 1000 \times .8854 = 885.$

Exercise 28

In a normal distribution 20% of the items are less than 100 and 30% are over 200. Find the mean and S.D of the distribution.

Solution :

Representing the given data diagramtically,



From the diagram

$P(-Z_1 < Z < 0) = 0.3$

i.e. $P(0 < Z < Z_1) = 0.3$

$\therefore$ $Z_1 = 0.84$ (from the normal table)

Hence $-0.84 = \dfrac{100-\mu}{\sigma}$

i.e. $100 - \mu = -0.84\sigma$ ----------(1)

$P(0 < Z < Z_2) = 0.2$

$\therefore$ $Z_2 = 0.525$ (from the normal table)

Hence $\quad 0.525 = \dfrac{200-\mu}{\sigma}$

i.e. $200 - \mu = 0.525\sigma$ ----------(2)

Solving (1) and (2), $\mu = 161.53$

$\sigma = 73.26$

EXERCISE 8.4

1) Find the area under the standard normal curve which lies
   (i) to the right of $Z = 2.70$
   (ii) to the left of $Z = 1.73$

2) Find the area under the standard normal curve which lies
   (i) between $Z = 1.25$ and $Z = 1.67$
   (ii) between $Z = -0.90$ and $Z = -1.85$

3) The distribution of marks obtained by a group of students may be assumed to be normal with mean 50 marks and standard deviation 15 marks. Estimate the proportion of students with marks below 35.

4) The marks in Economics obtained by the students in Public examination is assumed to be approximately normally distributed with mean 45 and S.D 3. A student taking this subject is chosen at random. What is the probability that his mark is above 70?

5) Assuming the mean height of soldiers to be 68.22 inches with a variance 10.8 inches. How many soldiers in a regiment of 1000 would you expect to be over 6 feet tall ?

6) The mean yield for one-acre plot is 663 kgs with a S.D 32 kgs. Assuming normal distribution, how many one-acre plot in a batch of 1000 plots would you expect to have yield (i) over 700 kgs (ii) below 650 kgs.

7) A large number of measurements is normally distributed with a mean of 65.5" and S.D of 6.2". Find the percentage of measurements that fall between 54.8" and 68.8".

8) The diameter of shafts produced in a factory conforms to normal distribution. 31% of the shafts have a diameter less than 45mm. and 8% have more than 64mm. Find the mean and standard deviation of the diameter of shafts.

9) The results of a particular examination are given below in a summary form.

Result	percentage of candidates
1. passed with distinction	10
2. passed	60
3. failed	30

It is known that a candidate gets plucked if he obtained less tham 40 marks out of 100 while he must obtain atleast 75 marks in order to pass with distinction. Determine the mean and the standard deviation of the distribution assuming this to be normal.

EXERCISE 8.5

Choose the correct answer

1) If a fair coin is tossed three times the probability function $p(x)$ of the number of heads x is

(a)

x	0	1	2	3
$p(x)$	$\frac{1}{8}$	$\frac{1}{8}$	$\frac{2}{8}$	$\frac{3}{8}$

(b)

x	0	1	2	3
$p(x)$	$\frac{1}{8}$	$\frac{3}{8}$	$\frac{3}{8}$	$\frac{1}{8}$

(c)

x	0	1	2	3
$p(x)$	$\frac{1}{8}$	$\frac{1}{8}$	$\frac{2}{8}$	$\frac{3}{8}$

(d) none of these

2) If a discrete random variable has the probability mass function as

x	0	1	2	3
$p(x)$	k	$2k$	$3k$	$5k$

then the value of k is

(a) $\frac{1}{11}$ (b) $\frac{2}{11}$ (c) $\frac{3}{11}$ (d) $\frac{4}{11}$

3) If the probability density function of a variable X is defined as $f(x) = Cx(2-x)$, $0 < x < 2$ then the value of C is

(a) $\frac{4}{3}$ (b) $\frac{6}{4}$ (c) $\frac{3}{4}$ (d) $\frac{3}{5}$

4) The mean and variance of a binomial distribution are

(a) $np,\ npq$ (b) $pq,\ npq$ (c) $np,\ \sqrt{npq}$ (d) $np,\ nq$

5) If X~N (μ, σ), the standard Normal variate is distributed as

(a) N(0, 0) (b) N(1, 0) (c) N(0, 1) (d) N(1, 1)

6) The normal distribution curve is

(a) Bimodal (b) Unimodal

(c) Skewed (d) none of these

7) If X is a poission variate with P(X = 1) = P(X = 2), the mean of the Poisson variate is equal to

(a) 1 (b) 2 (c) –2 (d) 3

8) The standard deviation of a Poissson variate is 2, the mean of the poisson variate is

(a) 2 (b) 4 (c) $\sqrt{2}$ (d) $\frac{1}{\sqrt{2}}$

9) The random variables X and Y are independent if

(a) E(X Y) = 1 (b) E(XY) = 0

(c) E(X Y) = E(X) E(Y) (d) E(X+Y) = E(X) + E(Y)

10) The mean and variance of a binomial distribution are 8 and 4 respectively. Then P(X = 1) is equal to

(a) $\frac{1}{2^{12}}$ (b) $\frac{1}{2^{4}}$ (c) $\frac{1}{2^{6}}$ (d) $\frac{1}{2^{10}}$

11) If X~N (μ, σ^2), the points of inflection of normal distribution curve are

(a) $\pm\mu$ (b) $\mu \pm \sigma$ (c) $\sigma \pm \mu$ (d) $\mu \pm 2\sigma$

12) If X~N (μ, σ^2), the maximum probability at the point of inflection of normal distribution is

(a) $\frac{1}{\sqrt{2\pi}}e^{\frac{1}{2}}$ (b) $\frac{1}{\sqrt{2\pi}}e^{-\frac{1}{2}}$ (c) $\frac{1}{\sigma\sqrt{2\pi}}$ (d) $\frac{1}{\sqrt{2\pi}}$

13) If a random variable X has the following probability distribution

X	-1	-2	1	2
$p(x)$	$\frac{1}{3}$	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{3}$

then the expected value of X is

(a) $\frac{3}{2}$ (b) $\frac{1}{6}$ (c) $\frac{1}{2}$ (d) $\frac{1}{3}$

14) If X~N (5, 1), the probability density function for the normal variate X is

(a) $\frac{1}{5\sqrt{2\pi}}e^{-\frac{1}{2}(\frac{x-1}{5})^2}$ (b) $\frac{1}{\sqrt{2\pi}}e^{-\frac{1}{2}(\frac{x-1}{5})^2}$

(c) $\frac{1}{\sqrt{2\pi}}e^{-\frac{1}{2}(x-5)^2}$ (d) $\frac{1}{\sqrt{\pi}}e^{-\frac{1}{2}(x-5)^2}$

15) If X~N (8, 64), the standard normal variate Z will be

(a) $z = \frac{X-64}{8}$ (b) $\frac{X-8}{64}$

(c) $\frac{X-8}{8}$ (d) $\frac{X-8}{\sqrt{8}}$

SAMPLING TECHNIQUES AND STATISTICAL INFERENCE 9

9.1 SAMPLING AND TYPES OF ERRORS

Sampling is being used in our everyday life without knowing about it. For examples, a cook tests a small quantity of rice to see whether it has been well cooked and a grain merchant does not examine each grain of what he intends to purchase, but inspects only a small quantity of grains. Most of our decisions are based on the examination of a few items only.

In a statistical investigation, the interest usually lies in the assessment of general magnitude and the study of variation with respect to one or more characteristics relating to individuals belonging to a group. This group of individuals or units under study is called **population** or **universe.** Thus in statistics, population is an aggregate of objects or units under study. The population may be finite or infinite.

9.1.1 Sampling and sample

Sampling is a method of selecting units for analysis such as households, consumers, companies etc. from the respective population under statistical investigation. The theory of sampling is based on the **principle of statistical regularity.** According to this principle, a moderately large number of items chosen at random from a large group are almost sure on an average to possess the characteristics of the larger group.

A smallest non-divisible part of the population is called a **unit.** A unit should be well defined and should not be ambiguous. For example, if we define unit as a household, then it should be defined that a person should not belong to two households nor should it leave out persons belonging to the population.

A finite subset of a population is called a **sample** and the number of units in a sample is called its **sample size**.

By analysing the data collected from the sample one can draw inference about the population under study.

9.1.2 Parameter and Statistic

The statistical constants of a population like mean (μ), variance (σ^2), proportion (P) are termed as **parameters.** Statistical measures like mean ($\overline{X}$), variance (s^2), proportion (p) computed from the sampled observations are known as **statistics.**

Sampling is employed to throw light on the population parameter. A **statistic** is an estimate based on sample data to draw inference about the population parameter.

9.1.3 Need for Sampling

Suppose that the raw materials department in a company receives items in lots and issues them to the production department as and when required. Before accepting these items, the inspection department inspects or tests them to make sure that they meet the required specifications. Thus

(i) it could inspect all items in the lot or

(ii) it could take a sample and inspect the sample for defectives and then estimate the total number of defectives for the population as a whole.

The first approach is called **complete enumeration (census)**. It has two major disadvantages namely, the time consumed and the cost involved in it.

The second approach that uses sampling has two major advantages. (i) It is significantly less expensive. (ii) It takes least possible time with best possible results.

There are situations that involve destruction procedure where sampling is the only answer. A well-designed statistical sampling methodology would give accurate results and at the same time will result in cost reduction and least time. Thus sampling is the best available tool to decision makers.

9.1.4 Elements of Sampling Plan

The main steps involved in the planning and execution of sample survey are :

(i) Objectives

The first task is to lay down in concrete terms the basic objectives of the survey. Failure to define the objective(s) will clearly undermine the purpose of carrying out the survey itself. For example, if a nationalised bank wants to study savings bank account holders perception of the service quality rendered over a period of one year, the objective of the sampling is, here,to analyse the perception of the account holders in the bank.

(ii) Population to be covered

Based on the objectives of the survey, the population should be well defined. The characteristics concerning the population under study should also be clearly defined. For example, to analyse the perception of the savings bank account holders about the service rendered by the bank, all the account holders in the bank constitute the population to be investigated.

(iii) Sampling frame

In order to cover the population decided upon, there should be some list, map or other acceptable material (called the **frame**) which serves as a guide to the population to be covered. The list or map must be examined to be sure that it is reasonably free from defects. The sampling frame will help us in the selection of sample. All the account numbers of the savings bank account holders in the bank are the sampling frame in the analysis of perception of the customers regarding the service rendered by the bank.

(iv) Sampling unit

For the purpose of sample selection, the population should be capable of being divided up into sampling units. The division of the population into sampling units should be unambiguous. Every element of the population should belong to just one sampling unit.

Each account holder of the savings bank account in the bank, form a unit of the sample as all the savings bank account holders in the bank constitute the population.

(v) Sample selection

The size of the sample and the manner of selecting the sample should be defined based on the objectives of the statistical investigation. The estimation of population parameter along with their margin of uncertainity are some of the important aspects to be followed in sample selection.

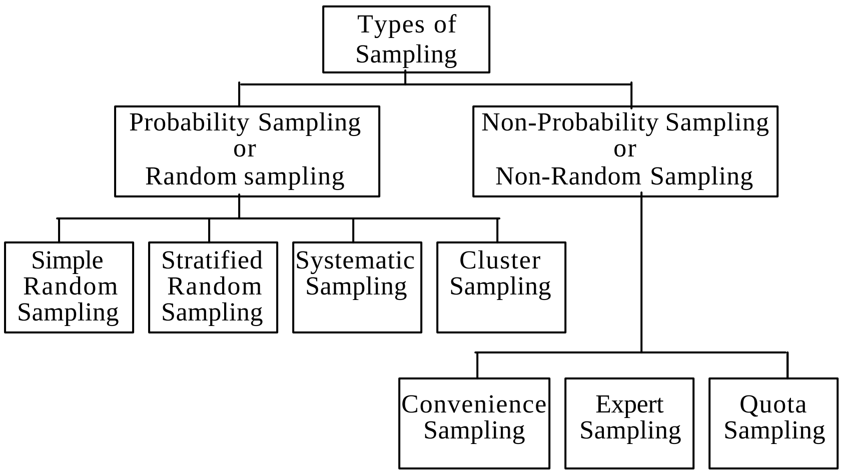
(vi) Collection of data

The method of collecting the information has to be decided, keeping in view the costs involved and the accuracy aimed at. Physical observation, interviewing respondents and collecting data through mail are some of the methods that can be followed in collection of data.

(vii) Analysis of data

The collected data should be properly classified and subjected to an appropriate analysis. The conclusions are drawn based on the results of the analysis.

9.1.5 Types of Sampling



The technique of selecting a sample from a population usually depends on **the nature of the data** and **the type of enquiry**. The procedure of sampling may be broadly classified under the following heads :

(i) Probability sampling or random sampling and
(ii) Non-probability sampling or non-random sampling.

(i) Probability sampling

Probability sampling is a method of sampling that ensures that every unit in the population has a known **non-zero chance** of being included in the sample.

The different methods of random sampling are :

(a) Simple Random Sampling

Simple random sampling is the foundation of probability sampling. It is a special case of probability sampling in which every unit in the population has an **equal chance** of being included in a sample. Simple random sampling also makes the selection of every possible combination of the desired number of units equally likely. Sampling may be done with or without replacement.

It may be noted that when the sampling is with replacement, the units drawn are replaced before the next selection is made.The population size remains constant when the sampling is with replacement.

If one wants to select n units from a population of size N without replacement,then every possible selection of n units must have the same probability. Thus there are ${}^{N}c_{n}$ possible ways to pick up n units from the population of size N. Simple random sampling guarantees that a sample of n units, has the same probability $\frac{1}{{}^{N}c_{n}}$ of being selected.

Example

A bank wants to study the Savings Bank account holders perception of the service quality rendered over a period of one year. The bank has to prepare a complete list of savings bank account holders, called as **sampling frame,** say 500. Now the process involves selecting a sample of 50 out of 500 and interviewing them. This could be achieved in many ways. Two common ways are :

(1) **Lottery method :** Select 50 slips from a box containing well shuffled 500 slips of account numbers without replacement. This method can be applied when the population is small enough to handle.

(2) **Random numbers method :** When the population size is very large, the most practical and inexpensive method of selecting a simple random sample is by using the random number tables.

(b) Stratified Random Sampling

Stratified random sampling involves dividing the population into a number of groups called **strata** in such a manner that the units within a stratum are **homogeneous** and the units between the strata are **hetrogeneous**. The next step involves selecting a simple random sample of appropriate size from each stratum. The sample size in each stratum is usually of (a) equal size, (b) proportionate to the number of units in the stratum.

For example, a marketing manager in a consumer product company wants to study the customer's attitude towards a new product in order to improve the sales. Then three typical cities that will influence the sales will be considered as three strata. The customers within a city are similar but between the cities are vastly different. Selection of the customers for the study from each city has to be a random sample to draw meaningful inference on the whole population.

(c) Systematic sampling

Systematic sampling is a convenient way of selecting a sample. It requires less time and cost when compared to simple random sampling.

In this method, the units are selected from the population at a uniform interval. To facilitate this we arrange the items in numerical, alphabetical, geographical or any other order. When a complete list of the population is available, this method is used.

If we want to select a sample of size n from a population of size N under systematic sampling, frist select an item j at random such that $1 \leq j \leq k$ where $k = \frac{N}{n+1}$ and k is the nearest possible integer. Then $j, j + k, j + 2k, \ldots, j + (n-1) k$ th items constitute a systematic random sample.

For example, if we want to select a sample of 9 students out of 105 students numbered as 1, 2, ..., 105 , select a student among 1, 2, ... , 11 at random (say at 3rd position). Here $k = \frac{105}{10} = 10.5$ and $\therefore$ $k = 11$. Hence students at the positions 3, 14, 25, 36, 47, 58, 69, 80, 91 form a random sample of size 9.

(d) Cluster sampling

Cluster sampling is used when the population is divided into **groups** or **clusters** such that each cluster is a representative of the population.

If a study has to be done to find out the number of children that each family in Chennai has, then the city can be divided into several clusters and a few clusters can be chosen at random. Every family in the chosen clusters can be a sample unit.

In using cluster sampling the following points should be noted

(a) For getting precise results clusters should be as small as possible consistent with the cost and limitations of the survey and

(b) The number of units in each cluster must be more or less equal.

(ii) Non-Probability Sampling

The fundamental difference between probability sampling and non-probability sampling is that in non-probability sampling procedure, the selection of the sample units does not ensure a known chance to the units being selected. In other words the units are selected without using the **principle of probability.** Even though the non-probability sampling has advantages such as reduced cost, speed and convenience in implementation, it lacks accuracy in view of the seleciton bias. Non-probability sampling is suitable for pilot studies and exploratory research

The methods of **non-random sampling** are :

(a) Purposive sampling

In this sampling, the sample is selected with definite purpose in view and the choice of the sampling units depends entirely on the discretion and judgement of the investigator.

For example, if an investigator wants to give the picture that the standard of living has increased in the city of Madurai, he may take the individuals in the sample from the posh localities and ignore the localities where low income group and middle class families live.

(b) Quota sampling

This is a restricted type of purposive sampling. This consists in specifying quotas of the samples to be drawn from different groups and then drawing the required samples from these groups by purposive sampling. Quota sampling is widely used in opinion and market research surveys.

(c) Expert opinion sampling or expert sampling

Expert opinion sampling involves gathering a set of people who have the knowledge and expertise in certain key areas that are crucial to decision making. The advantage of this sampling is that it acts as a support mechanism for some of our decisions in situations where virtually no data are available. The major disadvantage is that even the experts can have prejudices, likes and dislikes that might distort the results.

9.1.6 Sampling and non-sampling errors

The errors involved in the collection of data, processing and analysis of data may be broadly classfied as (i) **sampling errors** and (ii) **non-sampling errors.**

(i) Sampling errors

Sampling errors have their origin in sampling and arise due to the fact that only a part of the population has been used to estimate population parameters and draw inference about the population. Increasing in the sample size usually results in decrease in the sampling error.

Sampling errors are primarily due to some of the following reasons :

(a) **Faulty selection of the sample**

Some of the bias is introduced by the use of defective sampling technique for the selection of a sample in which the investigator deliberately selects a representative sample to obtain certain results.

(b) **Substitution**

If difficulty arise in enumerating a particular sampling unit included in the random sample, the investigators usually substitute a convenient member of the population leading to sampling error.

(c) **Faulty demarcation of sampling units**

Bias due to defective demarcation of sampling units is particularly significant in area surveys such as agricultural experiments. Thus faulty demarcation could cause sampling error.

(ii) Non-sampling errors

The non-sampling errors primarily arise at the stages of observation, classification and analysis of data.

Non-sampling errors can occur at every stage of the planning or execution of census or sample surveys. Some of the more important non-sampling errors arise from the following factors :

(a) **Errors due to faulty planning and definitions**
Sampling error arises due to improper data specification, error in location of units, measurement of characteristics and lack of trained investigators.

(b) **Response errors**

These errors occur as a result of the responses furnished by the respondents.

(c) **Non-response bias**

Non-response biases occur due to incomplete information on all the sampling units.

(d) **Errors in coverage**

These errors occur in the coverage of sampling units.

(e) **Compiling errors**

These errors arise due to compilation such as editing and coding of responses.

EXERCISE 9.1

1) Explain sampling distribution and standard error.
2) Distinguish between the terms parameter and statistic
3) Explain briefly the elements of sampling plan.
4) Discuss probability sampling.
5) Discuss non-probability sampling.
6) Distinguish between sampling and non sampling errors.

9.2 SAMPLING DISTRIBUTIONS

Consider all possible samples of size n which can be drawn from a given population. For each sample we can compute a statistic such as mean, standard deviation, etc. which will vary from sample to sample. The aggregate of various values of the statistic under consideration may be grouped into a frequency distribution. This distribution is known as **sampling distribution** of the statistic. Thus the probability distribution of all the possible values that a sample statistic can take, is called the sampling distribution of the statistic. **Sample mean** and **sample proportion** based on a random sample are examples of sample statistic.

Supposing a Market Research Agency wants to estimate the annual household expenditure on consumer durables from among the population of households (say 50000 households) in Tamil Nadu. The agency can choose fifty different samples of 50 households each. For each of the samples, we can calculate the mean annual expenditure on consumer durables as given in the following table :

Sample No.	**Total expenditure for 50 households**	**Mean Rs.**
1	100000	2000
2	300000	6000
3	200000	4000
4	150000	3000
.	.	.
.	.	.
.	.	.
49	600000	12000
50	400000	8000

The distribution of all the sample means is known as the **sampling distribution of the mean.** The figures Rs. 2000, 6000 ... 8000 are the sampling distribution of the means.

Similarly, the sampling distribution of the **sample variance** and **sample proportion** can also be obtained.

In a sample of n items if n_1 belongs to Category-1 and n-n_1 belongs to the Category-2, then $\frac{n_1}{n}$ is defined as the sample proportion p belonging to the first category and $\frac{n-n_1}{n}$ or (1-p) is the sample proportion of second category. This concept could be extended to k such categories with proportions (say) p_1, p_2, ... , p_k such that $p_1 + p_2 + \ldots + p_k = 1$

We could also arrive at a sampling distribution of a **proportion.** For example, if in a factory producing electrical switches, 15 different samples of 1000 switches are taken for inspection and number of defectives in each sample could be noted. We could find a probability distribtuion of the **proportion** of defective switches.

9.2.1 Sampling distribution of the Mean from normal population

If X_1, X_2, ..., X_n are *n* independent random samples drawn from a normal population with mean μ and standard deviation σ, then the sampling distribution of $\overline{X}$ (the sample mean) follows a normal distribution with mean μ and standard deviation $\frac{\sigma}{\sqrt{n}}$.

It may be noted that

(i) the sample mean $\overline{X} = \frac{\Sigma X_i}{n} = \frac{X_1 + X_2 + ... + X_n}{n}$

Thus $\overline{X}$ is a random variable and will be different every time when a new sample of *n* observations are taken

(ii) $\overline{X}$ is an unbiased estimator of the population mean μ.

i.e. $E(\overline{X}) = \mu$, denoted by $\mu_{\overline{X}} = \mu$.

(iii) the standard deviation of the sample mean $\overline{X}$ is given by

$$\sigma_{\overline{X}} = \frac{\sigma}{\sqrt{n}}$$

For example, consider a sample of weights of four boys from the normal population of size 10000 with replacement. The mean weight of the four boys is worked out. Again take another new sample of four boys from the same population and find the mean weight. If the process is repeated an infinite number of times, the probability distribution of these infinite number of sample means would be sampling distribution of mean.

9.2.2 Central limit theorem

When the samples are drawn from a normal population with mean μ and standard deviation σ, the sampling distribution of the mean is also normal with mean $\mu_{\overline{X}} = \mu$ and standard deviation $\sigma_{\overline{X}} = \frac{\sigma}{\sqrt{n}}$. However the sampling distribution of the mean, when the population is not normal is equally important.

The central limit theorem says that from any given population with mean μ and standard deivation σ, if we draw a random sample of *n* observations, the sampling distribution of the mean will approach a normal distribution with a mean μ and standard deviation $\frac{\sigma}{\sqrt{n}}$ as the sample size increases and becomes large.

In practice a sample size of 30 and above is considered to be large.

Thus the central limit theorem is a hall mark of statistical inference. It permits us to make inference about the population parameter based on random samples drawn from populations that are not neccessarily normally distributed.

9.2.3 Sampling distribution of proportions

Suppose that a population is infinite and that the probability of occurrence of an event, say success, is P. Let Q = 1 – P denote the probability of failure.

Consider all possible samples of size *n* drawn from this population. For each sample, determine the proportion *p* of successes. Applying central limit theorem, if the sample size *n* is large, the distribution of the sample porportion *p* follows a normal distribution with mean $\mu_p = P$ and S.D $\sigma_p = \sqrt{\frac{PQ}{n}}$.

9.2.4 Standard error

The standard deviation of the sampling distribution of a statistic is called the **standard error** of the statistic. The standard deviation of the distribution of the sample means is called the **standard error of the mean**. Likewise, the standard deviation of the distribution of the sample proportions is called the **standard error of the proportion.**

The standard error is popularly known as **sampling error**. Sampling error throws light on the precision and accuracy of the estimate. The standard error is inversely proportional to the sample size i.e. the larger the sample size the samller the standard error.

The standard error measures the dispersion of all possible values of the statistic in repeated samples of a fixed size from a given population. It is used to set up confidence limits for population parameters in tests of significance. Thus the standard errors of sample mean $\overline{X}$ and sample proportion p are used to find confidence limits for the population mean μ and the population proportion P respectively.

Statistic	**Standard error**	**Remarks**
Sample mean $\overline{X}$	$\frac{\sigma}{\sqrt{n}}$	Population size is infinite or sample with replacement.
	$\frac{\sigma}{\sqrt{n}}\sqrt{\frac{N-n}{N-1}}$	Population size N finite or sample without replacement
Sample proportion p	$\sqrt{\frac{PQ}{n}}$	Population size is infinite or sample with replacement.
	$\sqrt{\frac{PQ}{n}}\sqrt{\frac{N-n}{N-1}}$	Population size N finite or sample without replacement

Example 1

A population consists of the five numbers 2, 3, 6, 8, 11. Consider all possible samples of size 2 which can be drawn with replacement from this population. find (i) mean of the population, (i) the standard deviation of the population (iii) the mean of the sample distribution of means and (iv) the standard error of means.

Solution :

(i) The population mean $\mu = \frac{\Sigma x}{N} = \frac{2+3+6+8+11}{5} = 6$

(ii) The variance of the population $\sigma^2 = \frac{1}{N}\Sigma(x - \mu)^2$

$$= \frac{1}{5}\{(2-6)^2 + (3-6)^2 + (6-6)^2 + (8-6)^2 + (11-6)^2\}$$
$$= 10.8$$

$\therefore$ the standard deviation of the population $\sigma = 3.29$

(iii) There are 25 samples of size two which can be drawn with replacement. They are

(2, 2)	(2, 3)	(2, 6)	(2, 8)	(2, 11)
(3, 2)	(3, 3)	(3, 6)	(3, 8)	(3, 11)
(6, 2)	(6, 3)	(6, 6)	(6, 8)	(6, 11)
(8, 2)	(8, 3)	(8, 6)	(8, 8)	(8, 11)
(11, 2)	(11, 3)	(11, 6)	(11, 8)	(11, 11)

The corresponding sample means are

2.0	2.5	4.0	5.0	6.5
2.5	3.0	4.5	5.5	7.0
4.0	4.5	6.0	7.0	8.5
5.0	5.5	7.0	8.0	9.5
6.5	7.0	8.5	9.5	11.0

The mean of sampling distribution of means

$$\mu_{\overline{x}} = \frac{\text{sum of all sample means}}{25} = \frac{150}{25} = 6.0$$

(iv) The variance $\sigma^2_{\overline{x}}$ of the sampling distribution of means is obtained as follows :

$$\sigma^2_{\overline{x}} = \frac{1}{25}\{2-6)^2 + (2.5-6)^2 + \dots + (6.5-6)^2 + \dots$$
$$+ (9.5-6)^2 + (11-6)^2\}$$
$$= \frac{135}{25} = 5.4$$

$\therefore$ the standard error of means $\sigma_{\overline{x}} = \sqrt{5.4} = 2.32$

Example 2

Assume that the monthly savings of 1000 employees working in a factory are normally distributed with mean Rs. 2000 and standard deviation Rs. 50 If 25 samples consisting of 4 employees each are obtained, what would be the mean and standard deviation of the resulting sampling distribution of means if sampling were done (i) with replacement, (ii) without replacement.

Solution :

Given $N = 1000$, $\mu = 2000$, $\sigma = 50$, $n = 4$

(i) *Sampling with replacement*

$$\mu_{\overline{X}} = \mu = 2000$$

$$\sigma_{\overline{X}} = \frac{\sigma}{\sqrt{n}} = \frac{50}{\sqrt{4}} = 25$$

(ii) *Sampling without replacement*

The mean of the sampling distribution of the means is

$$\mu_{\overline{X}} = \mu = 2000$$

The standard deviation of the sampling distribution of means is

$$\sigma_{\overline{X}} = \frac{\sigma}{\sqrt{n}}\sqrt{\frac{N-n}{N-1}}$$

$$= \frac{50}{\sqrt{4}}\sqrt{\frac{100-4}{1000-1}}$$

$$= (25)\sqrt{\frac{996}{999}} = 25\left(\sqrt{.996}\right)$$

$$= (25)(0.9984) = 24.96$$

Example 3

A random sample of size 5 is drawn without replacement from a finite population consisting of 41 units. If the population S.D is 6.25, find the S.E of the sample mean.

Solution :

Population size N = 41

Sample size $n = 5$

Standard deviation of the popultion $\sigma = 6.25$

S.E of sample mean $= \dfrac{\sigma}{\sqrt{n}}\sqrt{\dfrac{N-n}{N-1}}$ (N is finite)

$$= \frac{6.25}{\sqrt{5}}\sqrt{\frac{41-5}{41-1}}$$

$$= \frac{6.25 \text{ x } 6}{\sqrt{5}\ 2\sqrt{10}} = \frac{3 \text{ x } 6.25}{5\sqrt{2}} = 2.65$$

Example 4

The marks obtained by students in an aptitude test are normally distributed with a mean of 60 and a standard deviation of 30. A random sample of 36 students is drawn from this population.

(i) What is the standard error of the sampling mean?

(ii) What is the probability that the mean of a sample of 16 students will be either less than 50 or greater than 80?

[P (0 < Z < 4) = 0.4999]

Solution :

(i) The standard error of the sample mean $\overline{X}$ is given by

$$\sigma_{\overline{X}} = \frac{\sigma}{\sqrt{n}} = \frac{30}{\sqrt{36}} = 5 \qquad \text{(N is not given)}$$

(ii) The random variable $\overline{X}$ follows normal distribution with mean $\mu_{\overline{X}}$ and standard deviation $\dfrac{\sigma}{\sqrt{n}}$.

To find P($\overline{X}$<50 or $\overline{X}$>80).

P($\overline{X}$<50 or $\overline{X}$>80) = P($\overline{X}$<50) + P($\overline{X}$>80)

$$= P\left(\frac{\overline{X}-\mu_{\overline{X}}}{\sigma_{\overline{X}}} < \frac{50-60}{5}\right) + P\left(\frac{\overline{X}-\mu_{\overline{X}}}{\sigma_{\overline{X}}} > \frac{80-60}{5}\right)$$

$= P(Z < -2) + P(Z > 4)$
$= [0.5 - P(0 < Z < 2)] + [0.5 - P(0 < Z < 4)]$
$= (0.5 - 0.4772) + (0.5 - 0.4999)$
$= .02283$, which is the required probability.

Example 5

2% of the screws produced by a machine are defective. What is the probability that in a consignment of 400 such screws, 3% or more will be defective.

Solution :

Here N is not given, but $n = 400$

Population proportion $P = 2\% = 0.02 \therefore Q = 1 - P = 0.98$

The sample size is large

$\therefore$ The sample porportion is normally distributed with mean

$$\mu_p = 0.02 \text{ and S.D} = \sqrt{\frac{PQ}{n}} = \sqrt{\frac{0.02 \times 0.98}{400}} = 0.007$$

Probability that the sample proportion $p \geq 0.03$

= Area under the normal curve to the right of $Z = 1.43$.

$$\left(Z = \frac{p-P}{S.D} = \frac{0.03-0.02}{0.007} = 1.43\right)$$

$\therefore$ required probability = 0.5 – Area between Z = 0 to Z = 1.43

$= 0.5 - 0.4236 = 0.0764$

EXERCISE 9.2

1) A population consists of four numbers 3, 7, 11 and 15. Consider all possible samples of size two which can be drawn with replacement from this population.

Find (i) the population mean

(ii) the population standard deviation.

(iii) the mean of the sampling distribution of mean

(iv) the standard deviation of the sampling distribution of mean.

2) A population consists of four numbers 3, 7, 11 and 15. Consider all possible samples of size two which can be drawn without replacement from this population.

Find (i) the population mean

(ii) the population standard deviation.

(iii) the mean of the sampling distribution of mean

(iv) the standard deviation of the sampling distribution of mean.

3) The weights of 1500 iron rods are normally distributed with mean of 22.4 kgs. and standard deviation of 0.048 kg. If 300 random samples of size 36 are drawn from this population, determine the mean and standard deviation of the sampling distribution of mean when sampling is done (i) with replacement (ii) without replacement.

4) 1% of the outgoing +2 students in a school have joined I.I.T. Madras. What is the probability that in a group of 500 such students 2% or more will be joining I.I.T. Madras.

9.3 ESTIMATION

The technique used for generalising the results of the sample to the population is provided by an important branch of statistics called **statistical inference.** The concept of statistical inference deals with two basic aspects namely (a) **Estimation** and (b) **Testing of hypothesis.**

In statistics, estimation is concerned with making inference about the parameters of the population using information available in the samples. The parameter estimation is very much needed in the decision making process.

The estimation of population parameters such as mean, variance, proportion, etc. from the correspoinding sample statistics is an important function of statistical inference.

9.3.1 Estimator

A sample statistic which is used to estimate a population parameter is known as **estimator.**

A good estimator is one which is as close to the true value of population parameter as possible. A good estimator possesses the following properties:

(i) Unbiasedness

As estimate is said to be unbiased if its expected value is equal to its parameter.

The sample mean $\overline{X} = \frac{1}{n}\Sigma x$ is an unbiased estimator of population mean μ. For a sample of size n, drawn from a population of size N, $s^2 = \frac{1}{n-1} \Sigma(x - \bar{x})^2$ is an unbiased estimator of population variance. Hence s^2 is used in **estimation** and in **testing of hypothesis.**

(ii) Consistency

An estimator is said to be consistent if the estimate tends to approach the parameter as the sample size increases.

(iii) Efficiency

If we have two unbiased estimators for the same population prarameter, the first estimator is said to be more efficient than the second estimator if the standard error of the first estimator is smaller than that of the second estimator for the same sample size.

(iv) Sufficiency

If an estimator possesses all information regarding the parameter, then the estimator is said to be a sufficient estimator.

9.3.2 Point Estimate and Interval Estimate

It is possible to find two types of estimates for a population parameter. They are **point estimate** and **interval estimate.**

Point Estimate

An estimate of a population parameter given by a single number is called a point estimator of the parameter. Mean ($\bar{x}$) and the sample variance $[s^2 = \frac{1}{n-1} \Sigma(x-\bar{x})^2]$ are the examples of point estimates.

A point estimate will rarely coincide with the true population parameter value.

Interval Estimate

An estimate of a population parameter given by two numbers between which the parameter is expected to lie is called an interval estimate of the parameter.

Interval estimate indicates the accuracy of an estimate and is therefore preferable to point estimate. As point estimate provides a single value for the population parameter it may not be suitable in some situation.

For example,

if we say that a distance is measured as 5.28mm, we are giving a point estimate. On the other hand, if we say that the distance is 5.28 ± 0.03 mm i.e. the distance lies between 5.25 and 5.31mm, we are giving an interval estimate.

9.3.3 Confidence Interval for population mean and proportion

The interval within which the unknown value of parameter is expected to lie is called **confidence interval.** The limits so determined are called **confidence limits.**

Confidence intervals indicate the probability that the population parameter lies within a specified range.

Computation of confidence interval

To compute confidence interval we require

(i) the sample statistic,

(ii) the standard error (S.E) of sampling distribution of the statistic

(iii) the degree of accuracy reflected by the Z-value.

If the size of sample is sufficiently large, then the sampling distribution is approximately normal. Therefore, the sample value can be used in estimation of standard error in the place of population value. The Z-distribution is used in case of large samples to estimate the confidence limits.

We give below values of Z corresponding to some confidence levels.

Confidence Levels	99%	98%	96%	95%	80%	50%
Value of Z, Z_c	2.58	2.33	2.05	1.96	1.28	0.674

(i) Confidence interval estimates for means

Let μ and σ be the population mean and standard deviation of the population.

Let $\overline{X}$ and s be the sample mean and standard deviation of the sampling distribution of a statistic.

The confidence limits for m are given below :

Population size	Sample size	confidence limits for μ.
Infinite	*n*	$\overline{X} \pm (Z_C)\frac{s}{\sqrt{n}}$, z_c is the value of *Z* corresponding to confidence levels.
Finite, N	*n*	$\overline{X} \pm (Z_C)\frac{s}{\sqrt{n}} \sqrt{\frac{N-n}{N-1}}$

(ii) Confidence intervals for proportions

If *p* is the proportion of successes in a sample of size *n* drawn from a population with P as its proportion of successes, then the confidence intervals for P are given below :

Population	Sample size	Confidence limits for P
Infinite	*n*	$p \pm (Z_C) \sqrt{\frac{pq}{n}}$
Finite, N	*n*	$p \pm (Z_C) \sqrt{\frac{pq}{n}} \sqrt{\frac{N-n}{N-1}}$

Example 6

Sensing the downward trend in demand for a leather product, the financial manager was considering shifting his company's resources to a new product area. He selected a sample of 10 firms in the leather industry and discovered their earnings (in %) on investment. Find point estimate of the mean and variance of the population from the data given below.

21.0 25.0 20.0 16.0 12.0 10.0 17.0 18.0 13.0 11.0

Solution :

X	$\overline{X}$	X- $\overline{X}$	(X- $\overline{X}$)^2
21.0	16.3	4.7	22.09
25.0	16.3	8.7	75.69
20.0	16.3	3.7	13.69
16.0	16.3	-0.3	0.09
12.0	16.3	-4.3	18.49
10.0	16.3	-6.3	39.69
17.0	16.3	0.7	0.49
18.0	16.3	1.7	2.89
13.0	16.3	-3.3	10.89
11.0	16.3	-5.3	28.09
163.0			**212.10**

Sample mean, $\overline{X} = \frac{\Sigma X}{n} = \frac{163}{10} = 16.3$

Sample variance, $s^2 = \frac{1}{n-1}\Sigma(X-\overline{X})^2$

$= \frac{212.10}{n-1} = 23.5$ (the sample size is small)

Sample standard deviation = $\sqrt{23.5} = 4.85$

Thus the point estimate of mean and of variance of the population from which the samples are drawn are 16.3 and 23.5 respectively.

Example 7

A sample of 100 students are drawn from a school. The mean weight and variance of the sample are 67.45 kg and 9 kg. respectively. Find (a) 95% and (b) 99% confidence intervals for estimating the mean weight of the students.

Solution :

Sample size, $n = 100$

The sample mean, $\overline{X} = 67.45$

The sample variance $s^2 = 9$

The sample standard deviation $s = 3$

Let μ be the population mean.

(a) The 95% confidence limits for μ are given by

$$\overline{X} \pm (Z_c) \frac{s}{\sqrt{n}}$$

$$\Rightarrow \quad 67.45 \pm (1.96) \frac{3}{\sqrt{100}}$$ (Here $Z_c = 1.96$ for 95% confidence level)

$$\Rightarrow \quad 67.45 \pm 0.588$$

Thus the 95% confidence intervals for estimating μ is given by

(66.86, 68.04)

(b) The 99% confidence limits for estimating μ are given by

$$\overline{X} \pm (Z_c) \frac{s}{\sqrt{n}}$$

$$\Rightarrow \quad 67.45 \pm (2.58) \frac{3}{\sqrt{100}}$$ (Here $z_c = 2.58$ for 99% confidence level)

$$\Rightarrow \quad 67.45 \pm 0.774$$

Thus the 99% confidence interval for estimating μ is given by

(66.67, 68.22)

Example 8

A random sample of size 50 with mean 67.9 is drawn from a normal population. If it is known that the standard error of the sample mean is $\sqrt{0.7}$, find 95% confidence interval for the population mean.

Solution :

$n = 50$, sample mean $\overline{X} = 67.9$

95% confidence limits for population mean μ are :

$\overline{X} \underline{+} (Z_c)\{S.E(\overline{X})\}$

$\Rightarrow \quad 67.9 \underline{+} (1.96)\,(\sqrt{0.7})$

$\Rightarrow \quad 67.9 \underline{+} 1.64$

Thus the 95% confidence intervals for estimating μ is given by (66.2, 69.54)

Example 9

A random sample of 500 apples was taken from large consignment and 45 of them were found to be bad. Find the limits at which the bad apples lie at 99% confidence level.

Solution :

We shall find confidence limits for the proportion of bad apples.

Sample size $n = 500$

Proportion of bad apples in the sample = $\frac{45}{500} = 0.09$

p = 0.09

$\therefore$ Proportion of good apples in the sample $q = 1-p = 0.91$.

The confidence limits for the population proportion P of bad apples are given by

$$p \underline{+} (Z_c)\left(\sqrt{\frac{pq}{n}}\right)$$

$\Rightarrow$ $0.09 \pm (2.58)\sqrt{\frac{(.09)(0.91}{500}}$ $\Rightarrow$ 0.09 ± 0.033

The required interval is (0.057, 0.123)

Thus, the bad apples in the consignment lie between 5.7% and 12.3%

Example 10

Out of 1000 TV viewers, 320 watched a particular programme. Find 95% confidence limits for TV viewers who watched this programme.

Solution :

Sample size $n = 1000$

Sample proportion of TV viewers $p = \frac{x}{n} = \frac{320}{1000}$

$= .32$

$\therefore \quad q = 1 - p = .68$

$\text{S.E } (p) = \sqrt{\frac{pq}{n}}$

$= 0.0147$

The 95% confidence limits for population propotion P are given by

$p \pm (1.96)$ S.E $(p) = 0.32 \pm 0.028$

$\Rightarrow$ 0.292 and 0.348

$\therefore$ TV viewers of this programme lie between 29.2% and 34.8%

Example 11

Out of 1500 school students, a sample of 150 selected at random to test the accuracy of solving a problem in business mathematics and of them 10 did a mistake. Find the limits within which the number of students who did the problem wrongly in whole universe of 1500 students at 99% confidence level.

Solution :

Population size, N = 1500

Sample size, n = 150

Sample proportion, $p = \frac{10}{150} = 0.07$

$\therefore \quad q = 1-p = 0.93$

Standard error of p, SE (p) = $\sqrt{\frac{pq}{n}} = 0.02$

The 99% confidence limits for population proportion P are given by

$$p \pm (Z_c) \sqrt{\frac{pq}{n}} \sqrt{\frac{N-n}{N-1}}$$

$$\Rightarrow \quad 0.07 \pm (2.58)(0.02) \sqrt{\frac{1500-150}{1500-1}}$$

$$\Rightarrow \quad 0.07 \pm 0.048$$

$\therefore$ The confidence interval for P is (0.022 , 0.118)

$\therefore$ The number of students who did the problem wrongly in the population of 1500 lies between .022 x 1500 = 33 and .118 x 1500 = 177.

EXERCISE 9.3

1) A sample of five measurements of the diameter of a sphere were recorded by a scientist as 6.33, 6.37, 6.36, 6.32 and 6.37 mm. Determine the point estimate of (a) mean, (b) variance.

2) Measurements of the weights of a random sample of 200 ball bearings made by a certain machine during one week showed mean of 0.824 newtons and a standard deviation of 0.042 newtons. Find (a) 95% and (b) 99% confidence limits for the mean weight of all the ball bearings.

3) A random sample of 50 branches of State Bank of India out of 200 branches in a district showed a mean annual profit of Rs.75 lakhs and a standard deviation of 10 lakhs. Find the 95% confidence limits for the estimate of mean profit of 200 branches.

4) A random sample of marks in mathematics secured by 50 students out of 200 students showed a mean of 75 and a standard deviation of 10. Find the 95% confidence limits for the estimate of their mean marks.

5) Out of 10000 customer's ledger accounts, a sample of 200 accounts was taken to test the accuracy of posting and balancing wherein 35 mistakes were found. Find 95% confidence limits within which the number of defective cases can be expected to lie.

6) A sample poll of 100 voters chosen at random from all voters in a given district indicated that 55% of them were in favour of a particular candidate. Find (a) 95% confidence limits, (b) 99% confidence limits for the proportion of all voters in favour of this candidate.

9.4 HYPOTHESIS TESTING

There are many problems in which, besides estimating the value of a parameter of the population, we must decide whether a statement concerning a parameter is true or false; that is, we must test a hypothesis about a parameter.

To illustrate the general concepts involved in this kind of decision problems, suppose that a consumer protection agency wants to test a manufacturer's claim that the average life time of electric bulbs produced by him is 200 hours. So it instructs a member of its staff to take 50 electric bulbs from the godown of the company and test them for their lifetime continuously with the intention of rejecting the claim if the mean life time of the bulbs is below 180 hours (say); otherwise it will accept the claim.

Thus **hypothesis** is an assumption that we make about an unknown population parameter. We can collect sample data from the population, arrive at the sample statistic and then test if the hypothesis about the population parameter is true.

9.4.1 Null Hypothesis and Alternative Hypothesis

In hypothesis testing, the statement of the hypothesis or assumed value of the population parameter is always stated before we begin taking the sample for analysis.

A statistical statement about the population parameter assumed before taking the sample for possible rejection on the basis of outcome of sample data is known as a **null hypothesis.**

The null hypothesis asserts that there is no diffeence between the sample statistic and population parameter and whatever difference is there is attributable to sampling error.

A hypothesis is said to be **alternative hypothesis** when it is complementary to the null hypothesis.

The null hypothesis and alternative hypothesis are usually denoted by H_0 and H_1 respectively.

For example, if we want to test the null hypothesis that the average height of soldiers is 173 cms, then

$$H_0 : \mu = 173 = \mu_0 \text{ (say)}$$
$$H_1 : \mu \neq 173 \neq \mu_0.$$

9.4.2 Types of Error

For testing the hypothesis, we take a sample from the population, and on the basis of the sample result obtained, we decide whether to accept or reject the hypothesis.

Here, two types of **errors** are possible. A null hypothesis could be rejected when it is true. This is called **Type I error** and the probability of committing type I error is denoted by α.

Alternatively, an error could result by accepting a null hypothesis when it is false. This is known as **Type II error** and the probability of committing type II error is denoted by β.

This is illustrated in the following table :

Actual	Decision based on sampling	Error and its Probability
H_0 is True	Rejecting H_0	Type I error ; $\alpha = P\{H_1 / H_0\}$
H_0 is False	Accepting H_0	Type II error ; $\beta = P\{H_0 / H_1\}$

9.4.3 Critical region and level of significance

A region in the sample space which amounts to rejection of null hypothesis (H_0) is called the **critical region.**

After formulating the null and alternative hypotheses about a population parameter, we take a sample from the population and calculate the value of the relevant statistic, and compare it with the hypothesised population parameter.

After doing this, we have to decide the **criteria** for accepting or rejecting the null hypothesis. These criteria are given as a range of values in the form of an interval, say (a, b), so that if the statistic value falls outside the range, we reject the null hypothesis.

If the statistic value falls within the interval (a, b), then we accept H_0. This criterion has to be decided on the basis of the **level of significance.** A 5% level of signififance means that 5% of the statistical values arrived at from the samples will fall outside this range (a, b) and 95% of the values will be within the range (a, b).

Thus the level of significance is the probability of Type I error α. The levels of significance usually employed in testing of hypothesis are 5% and 1%.

A high significance level chosen for testing a hypothesis would imply that higher is the probability of rejecting a null hypothesis if it is true.

9.4.4 Test of significance

The tests of significance are (a) Test of significance for **large samples** and (b) Test of significance for **small samples.**

For larger sample size (>30), all the distributions like Binomial, Poission etc., are approximated by normal distribution. Thus normal probability curve can be used for testing of hypothesis.

For the test statistic Z (standard normal variate), the critical region at 5% level is given by $|Z| \geq 1.96$ and hence the acceptance region is $|Z| < 1.96$. Where as the critical region for Z at 1% level is $|Z| \geq 2.58$ and the acceptance region is $|Z| < 2.58$.

The testing hypothesis involves five steps :

(i) The formulation of null hypothesis and an alternative hypothesis

(ii) Set up suitable significance level.

(iii) Setting up the statistical test criteria.

(iv) Setting up rejection region for the null hypothesis.

(v) Conclusion.

Example 12

The mean life time of 50 electric bulbs produced by a manufacturing company is estimated to be 825 hours with a standard deviation of 110 hours. If m is the mean life time of all the bulbs produced by the company, test the hypothesis that m = 900 hours at 5% level of significance.

Solution :

Null Hypothesis, $H_0 : \mu = 900$

Alternative Hypothesis, $H_1 : \mu \neq 900$

Test statistic , Z is the standard normal variate.

under H_0, $Z = \dfrac{\overline{X} - \mu}{\frac{\sigma}{\sqrt{n}}}$ where $\overline{X}$ is the sample mean

σ = s.d.of the population

$$= \frac{\overline{X} - \mu}{\frac{s}{\sqrt{n}}} \qquad \text{(For large sample, } \sigma = s\text{)}$$

$$= \frac{825 - 900}{\frac{110}{\sqrt{50}}} = -4.82.$$

$\therefore \; |Z| = 4.82$

Significant level, $\alpha = 0.05$ or 5%

Critical region is $|Z| \geq 1.96$

Acceptance region is $|Z| < 1.96$

The calculated Z is much greater than 1.96.

Decision : Since the calculated value of $|Z| = 4.82$ falls in the critical region, the value of Z is significant at 5% level.

$\therefore$ the null hypothesis is rejected.

$\therefore$ we conclude that the mean life time of the population of electric bulbs cannot be taken as 900 hours.

Example 13

A company markets car tyres. Their lives are normally distributed with a mean of 50000 kilometers and standard deviation of 2000 kilometers. A test sample of 64 tyres has a mean life of 51250 kms. Can you conclude that the sample mean differs significantly from the population mean? (Test at 5% level)

Solution :

Sample size, $n = 64$

Sample mean, $\overline{X} = 51250$

H_0 : population mean $\mu = 50000$

H_1 : $\mu \neq 50000$

Under H_0, the test statistic $Z = \dfrac{\overline{X} - \mu}{\frac{\sigma}{\sqrt{n}}} \sim N(0, 1)$

$$Z = \frac{51250 - 50000}{\frac{2000}{\sqrt{64}}} = 5$$

Since the calculated Z is much greater than 1.96, it is highly significant.

$\therefore$ $H_0 : \mu = 50000$ is rejected.

$\therefore$ The sample mean differs significantly from the population mean

Example 14

A sample of 400 students is found to have a mean height of 171.38 cms. Can it reasonably be regarded as a sample from a large population with mean height of 171.17 cms and standard deviation of 3.3 cms. (Test at 5% level)

Solution :

Sample size, $n = 400$

Sample mean , $\overline{X} = 171.38$

Population mean, $\mu = 171.17$

Sample standard deviation = s.

Population standard deviation, $\sigma = 3.3$

Set $H_0 : \mu = 171.38$

The test statistic, $Z = \frac{\overline{X} - \mu}{\frac{\sigma}{\sqrt{n}}} \sim N(0, 1)$

$= \frac{\overline{X} - \mu}{\frac{\sigma}{\sqrt{n}}}$ since the sample is large, $s = \sigma$

$= \frac{171.38 - 171.17}{\frac{3.3}{\sqrt{400}}}$

$= 1.273$

Since $|Z| = 1.273 < 1.96$, we accept the null hypothesis at 5% level of signifiance.

Thus the sample of 400 has come from the population with mean height of 171.17 cms.

EXERCISE 9.4

1) The mean I.Q of a sample of 1600 children was 99. Is it likely that this was a random sample from a population with mean I.Q 100 and standard deviation 15 ? (Test at 5% level of significance)

2) The income distribution of the population of a village has a mean of Rs.6000 and a variance of Rs.32400. Could a sample of 64 persons with a mean income of Rs.5950 belong to this population?
(Test at both 5% and 1% levels of significance)

3) The table below gives the total income in thousand rupees per year of 36 persons selected randomly from a particular class of people

Income (thousands Rs.)

6.5	10.5	12.7	13.8	13.2	11.4
5.5	8.0	9.6	9.1	9.0	8.5
4.8	7.3	8.4	8.7	7.3	7.4
5.6	6.8	6.9	6.8	6.1	6.5
4.0	6.4	6.4	8.0	6.6	6.2
4.7	7.4	8.0	8.3	7.6	6.7

On the basis of the sample data, can it be concluded that the mean income of a person in this class of people is Rs. 10,000 per year? (Test at 5% level of significance)

4) To test the conjecture of the management that 60 percent employees favour a new bonus scheme, a sample of 150 employees was drawn and their opinion was taken whether they favoured it or not. Only 55 employees out of 150 favoured the new bonus scheme Test the conjecture at 1% level of significance.

EXERCISE 9.5

Choose the correct answer

1) The standard error of the sample mean is

   (a) Type I error (b) Type II error

   (c) Standard deviation of the sampling distribution of the mean

   (d) Variance of the sampling distribution of the mean

2) If a random sample of size 64 is taken from a population whose standard deviation is equal to 32, then the standard error of the mean is

   (a) 0.5 (b) 2 (c) 4 (d) 32

3) The central limit theorem states that the sampling distribution of the mean will approach normal distribution

   (a) As the size of the population increases

   (b) As the sample size increases and becomes larger

   (c) As the number of samples gets larger

   (d) As the sample size decreases

4) The Z-value that is used to establish a 95% confidence interval for the estimation of a population parameter is

   (a) 1.28 (b) 1.65 (c) 1.96 (d) 2.58

5) Probability of rejecting the null hypothesis when it is true is

   (a) Type I error (b) Type II error

   (c) Sampling error (d) Standard error

6) Which of the following statements is true?

   (a) Point estimate gives a range of values

   (b) Sampling is done only to estimate a statistic

   (c) Sampling is done to estimate the population parameter

   (d) Sampling is not possible for an infinite population

7) The number of ways in which one can select 2 customers out of 10 customers is

   (a) 90 (b) 60 (c) 45 (d) 50

APPLIED STATISTICS 10

10.1 LINEAR PROGRAMMING

Linear programming is the general technique of optimum allocation of limited resources such as labour, material, machine, capital etc., to several competing activities such as products, services, jobs, projects, etc., on the basis of given criterion of optimality. The term limited here is used to describe the availability of scarce resources during planning period. The criterion of optimality generally means either performance, return on investment, utility, time, distance etc., The word **linear** stands for the proportional relationship of two or more variables in a model. **Programming** means 'planning' and refers to the process of determining a particular plan of action from amongst several alternatives. It is an extremely useful technique in the decision making process of the management.

10.1.1 Structure of Linear Programming Problem (LPP)

The LP model includes the following three basic elements.

(i) Decision variables that we seek to determine.

(ii) Objective (goal) that we aim to optimize (maximize or minimize)

(iii) Constraints that we need to satisfy.

10.1.2 Formulation of the Linear Programming Problem

The procedure for mathematical formulation of a linear programming consists of the following major steps.

Step 1 : Study the given situation to find the **key decision** to be made

Step 2 : Identify the **variables** involved and designate them by symbols x_j(j = 1, 2 ...)

Step 3 : Express the **feasible alternatives** mathematically in terms of variables, which generally are : $x_j \geq 0$ for all j

Step 4 : Identify the **constraints** in the problem and express them as linear inequalities or equations involving the decision variables.

Step 5 : Identify the **objective function** and express it as a linear function of the decision variables.

10.1.3 Applications of Linear programming

Linear programming is used in many areas. Some of them are

(i) *Transport :* It is used to prepare the distribution plan between source production and destination.

(ii) *Assignment :* Allocation of the tasks to the persons available so as to get the maximum efficiency.

(iii) *Marketing :* To find the shortest route for a salesman who has to visit different locations so as to minimize the total cost.

(iv) *Investment :* Allocation of capital to differerent activities so as to maximize the return and minimize the risk.

(v) *Agriculture :* The allotment of land to different groups so as to maximize the output.

10.1.4 Some useful Definitions

A **feasible solution** is a solution which satisfies all the constraints (including non-negativity) of the problem.

A region which contains all feasible solutions is known as **feasible region**.

A feasible solution which optimizes (maximizes or minimizes) the objective function,is called **optimal solution** to the problem.

Note

Optimal solution need not be unique.

Example 1

A furniture manufacturing company plans to make two products, chairs and tables from its available resources, which consists of 400 board feet of mahogany

timber and 450 man-hours of labour. It knows that to make a chair requires 5 board feet and 10 man-hours and yields a profit of Rs.45, while each table uses 20 board feet and 15 man - hours and has a profit of Rs.80. How many chairs and tables should the company make to get the maximum profit under the above resource constraints? Formulate the above as an LPP.

Solution :

Mathematical Formulation :

The data of the problem is summarised below:

Products	Raw material (per unit)	Labour (per unit)	Profit (per unit)
Chair	5	10	Rs. 45
Table	20	15	Rs. 80
Total availability	400	450	

Step 1 : The key decision to be made is to determine the number of units of chairs and tables to be produced by the company.

Step 2 : Let x_1 designate the number of chairs and x_2 designate the number of tables, which the company decides to produce.

Step 3 : Since it is not possible to produce negative quantities, feasible alternatives are set of values of x_1 and x_2, such that $x_1 \geq 0$ and $x_2 \geq 0$

Step 4 : The constraints are the limited availability of raw material and labour. One unit of chair requires 5 board feet of timber and one unit of table requires 20 board feet of timber. Since x_1 and x_2 are the quantities of chairs and tables, the total requirement of raw material will be $5x_1 + 20x_2$, which should not exceed the available raw material of 400 board feet timber. So, the raw material constraint becomes,

$$5x_1 + 20x_2 \leq 400$$

Similarly, the labour constraint becomes,

$$10x_1 + 15x_2 \leq 450$$

Step 5 : The objective is to maximize the total profit that the company gets out of selling their products, namely chairs, tables. This is given by the linear function.

$$z = 45x_1 + 80x_2.$$

The linear programming problem can thus be put in the following mathematical form.

$$\text{maximize } z = 45x_1 + 80x_2$$

$$\begin{aligned} \text{subject to} \quad 5x_1 + 20x_2 &\leq 400 \\ 10x_1 + 15x_2 &\leq 450 \\ x_1 \geq 0, \ x_2 &\geq 0 \end{aligned}$$

Example 2

A firm manufactures headache pills in two sizes A and B. Size A contains 2 mgs. of aspirin. 5 mgs. of bicarbonate and 1 mg. of codeine. Size B contains 1 mg. of aspirin, 8 mgs. of bicarbonate and 6 mgs. of codeine. It is found by users that it requires atleast 12 mgs. of aspirin, 74 mgs. of bicarbonate and 24 mgs. of codeine for providing immediate relief. It is required to determine the least number of pills a patient should take to get immediate relief. Formulate the problem as a standard LPP.

Solution :

The data can be summarised as follows :

Head ache	per pill		
pills	Aspirin	Bicarbonate	Codeine
Size A	2	5	1
Size B	1	8	6
Minimum requirement	12	74	24

Decision variables :

x_1 = number of pills in size A

x_2 = number of pills in size B

Following the steps as given in (10.1.2) the linear programming problem can be put in the following mathematical format :

$$\begin{aligned} &\text{Maximize} && z = x_1 + x_2 \\ &\text{subject to} && 2x_1 + x_2 \geq 12 \\ &&& 5x_1 + 8x_2 \geq 74 \\ &&& x_1 + 6x_2 \geq 24 \\ &&& x_1 \geq 0, \quad x_2 \geq 0 \end{aligned}$$

10.1.5 Graphical method

Linear programming problem involving two decision variables can be solved by graphical method. The major steps involved in this method are as follows.

Step 1 : State the problem mathematically.

Step 2 : Plot a graph representing all the constraints of the problem and identify the feasible region (solution space). The feasible region is the intersection of all the regions represented by the constraints of the problem and is restricted to the first quadrant only.

Step 3 : Compute the co-ordinates of all the corner points of the feasible region.

Step 4 : Find out the value of the objective function at each corner point determined in step3.

Step 5 : Select the corner point that optimizes (maximzes or minimizes) the value of the objective function. It gives the optimum feasible solution.

Example 3

A company manufactures two products P_1 and P_2. The company has two types of machines A and B for processing

the above products. Product P_1 takes 2 hours on machine A and 4 hours on machine B, whereas product P_2 takes 5 hours on machine A and 2 hours on machine B. The profit realized on sale of one unit of product P_1 is Rs.3 and that of product P_2 is Rs. 4. If machine A and B can operate 24 and16 hours per day respectively, determine the weekly output for each product in order to maximize the profit, through graphical method.

Solution :

The data of the problem is summarized below.

Product	Hours on Machine A	Hours on Machine B	profit (per unit)
P_1	2	4	3
P_2	5	2	4
Max. hours / week	120	80	

Let x_1 be the number of units of P_1 and x_2 be the number of units of P_2 produced. Then the mathematical formulation of the problem is

$$\text{Maximize } z = 3x_1 + 4x_2$$

$$\text{subject to} \quad 2x_1 + 5x_2 \leq 120$$

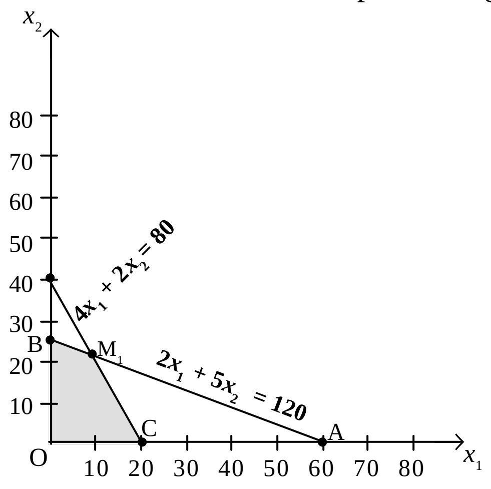
$$4x_1 + 2x_2 \leq 80$$

$$x_1,\ x_2 \geq 0$$

Solution by graphical method

Consider the equation $2x_1 + 5x_2 = 120$, and $4x_1 + 2x_2 = 80$. Clearly (0, 24) and (60, 0) are two points on the line $2x_1 + 5x_2 = 120$. By joining these two points we get the straight line $2x_1 + 5x_2 = 120$. Similarly, by joining the points (20, 0) and (0, 40) we get the sraight line $4x_1 + 2x_2 = 80$. (Fig. 10.1)

Now all the constraints have been represented graphically.



(Fig. 10.1)

The area bounded by all the constrints called feasible region or solution space is as shown in the Fig. 10.1, by the shaded area OCM_1 B

The optimum value of objective function occurs at one of the extreme (corner) points of the feasible region. The coordinates of the extreme points are

0 = (0, 0), C = (20, 0), M_1 = (10, 20), B = (0, 24)

We now compute the z-values correspoinding to extreme points.

Extreme point	coordinates (x_1, x_2)	$z = 3x_1 + 4x_2$
O	(0, 0)	0
C	(20, 0)	60
M_1	(10, 20)	110
B	(0, 24)	96

The optimum solution is that extreme point for which the objective function has the largest (maximum) value. Thus the optimum solution occurs at the point M_1 i.e. $x_1 = 10$ and $x_2 = 20$.

Hence to maximize profit of Rs.110, the company should produce 10 units of P_1 and 20 units of P_2 per week.

Note

In case of maximization problem, the corner point at which the objective function has a maximum value represent the optimal solution. In case of minimization problem, the corner point at which the objective function has a minimum value represents the optimnal solution.

Example 4

Solve graphically :

Minimize $z = 20x_1 + 40x_2$

Subject to $36x_1 + 6x_2 \geq 108$

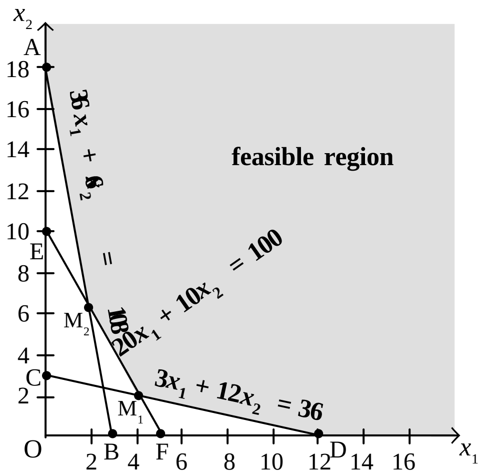
$3x_1 + 12x_2 \geq 36$

$20x_1 + 10x_2 \geq 100$

$x_1, \ x_2 \geq 0$

Solution :

A(0, 18) and B(3, 0) ; C(0, 3) and D(12, 0) ; E(0, 10) and F(5, 0) are the points on the lines $36x_1 + 6x_2 = 108$, $3x_1 + 12x_2 = 36$ and $20x_1 + 10x_2 = 100$ respectively. Draw thte above lines as Fig. 10.2.



(Fig. 10.2)

Now all the constraints of the given problem have been graphed. The area beyond three lines represents the **feasible region** or **solution space**, as shown in the above figure. Any point from this region would satisfy the constraints.

The coordinates of the extreme points of the feasible region are:

A = (0, 18), M_2 = (2, 6), M_1 = (4, 2), D = (12, 0)

Now we compute the z-values corresponding to extreme points.

Extreme point	coordinates (x_1, x_2)	$z = 20x_1 + 40x_2$
A	(0, 18)	720
M_1	(4, 2)	160
M_2	(2, 6)	280
D	(12, 0)	240

The optimum solution is that extreme point for which the objective function has minimum value. Thus optimum solution occurs at the point M_1 i.e. $x_1 = 4$ and $x_2 = 2$ with the objective function value of $z = 160$ $\therefore$ Minimum $z = 160$ at $x_1 = 4$, $x_2 = 2$

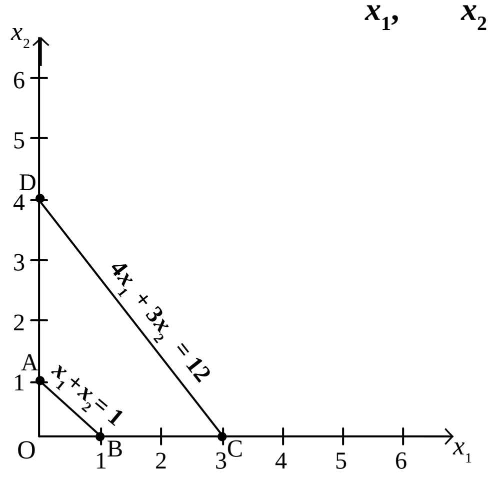
Example 5

Maximize $z = x_1 + x_2$ subject to $x_1 + x_2 \leq 1$

$4x_1 + 3x_2 \geq 12$

$x_1, \quad x_2 \geq 0$

Solution :



(Fig. 10.3)

From the graph, we see that the given problem has no solutiion as the feasible region does not exist.

EXERCISE 10.1

1) A company produces two types of products say type A and B. Profits on the two types of product are Rs.30/- and Rs.40/- per kg. respectively. The data on resources required and availability of resources are given below.

	Requirements		Capacity available per month
	Product A	Product B	
Raw materials (kgs)	60	120	12000
Machining hours / piece	8	5	600
Assembling (man hours)	3	4	500

Formulate this problem as a linear programming problem to maximize the profit.

2) A firm manufactures two products A & B on which the profits earned per unit are Rs.3 and Rs.4 respectively. Each product is processed on two machines M_1 and M_2. Product A requires one minute of processing time on M_1 and two minutes on M_2, while B requires one minute on M_1 and one minute on M_2. Machine M_1 is available for not more than 7 hrs 30 minutes while M_2 is available for 10 hrs during any working day. Formulate this problem as a linear programming problem to maximize the profit.

3) Solve the following, using graphical method

Maximize $z = 45x_1 + 80x_2$

subject to the constraints

$5x_1 + 20x_2 \leq 400$

$10x_1 + 15x_2 \leq 450$

$x_1, x_2 \geq 0$

4) Solve the following, using graphical method

Maximize $z = 3x_1 + 4x_2$

subject to the constraints

$2x_1 + x_2 \leq 40$

$2x_1 + 5x_2 \leq 180$

$x_1, \; x_2 \geq 0$

5) Solve the following, using graphical method

Minimize $z = 3x_1 + 2x_2$

subject to the constraints

$5x_1 + x_2 \geq 10$

$2x_1 + 2x_2 \geq 12$

$x_1 + 4x_2 \geq 12$

$x_1, x_2 \geq 0$

10.2 CORRELATION AND REGRESSION

10.2.1 Meaning of Correlation

The term correlation refers to the degree of relationship between two or more variables. If a change in one variable effects a change in the other variable, the variables are said to be correlated. There are basically three types of correlation, namely positive correlation, negative correlation and zero correlatiion.

Positive correlation

If the values of two variables deviate (change) in the same direction i.e. if the increase (or decrease) in one variable results in a corresponding increase (or decrease) in the other, the correlation between them is said to be positive.

Example

(i) the heights and weights of individuals

(ii) the income and expenditure

(iii) experience and salary.

Negative Correlation

If the values of the two variables constantly deviate (change) in the opposite directions i.e. if the increase (or decrease) in one results in corresponding decrease (or increase) in the other, the correlation between them is said to be negative.

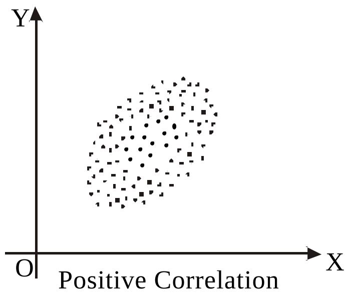
Example

(i) price and demand ,

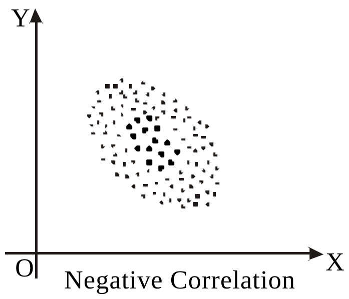
(ii) repayment period and EMI

10.2.2 Scatter Diagram

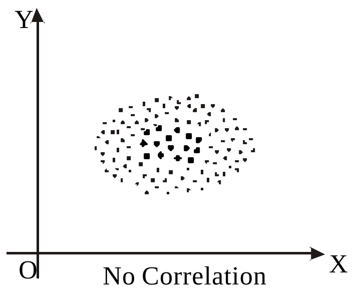
Let (x_1, y_1), (x_2, y_2) ... (x_n, y_n) be the n pairs of observation of the variables x and y. If we plot the values of x along x-axis and the corresponding values of y along y-axis, the diagram so obtained is called a **scatter diagram.** It gives us an idea of relationship between x and y. The types of scatter diagram under simple linear correlation are given below.



Positive Correlation
(Fig. 10.4)



Negative Correlation
(Fig. 10.5)



No Correlation
(Fig. 10.6)

(i) If the plotted points show an upward trend, the correlation will be positive (Fig. 10.4).

(ii) If the plotted points show a downward trend, the correlation will be negative (Fig. 10.5).

(iii) If the plotted point show no trend the variables are said to be uncorrelated (Fig. 10.6).

10.2.3 Co-efficient of Correlation

Karl pearson (1867-1936) a British Biometrician, developed the coefficient of correlation to express the degree of linear relationship between two variables. Correlation co-efficient between two random variables X and Y denoted by $r(\text{X, Y})$, is given by

$$r(\text{X, Y}) = \frac{\text{Cov(X, Y)}}{\text{SD(X) SD(Y)}}$$

where

$$\text{Cov (X, Y)} = \frac{1}{n}\sum_i (\text{X}_i - \overline{\text{X}})(\text{Y}_i - \overline{\text{Y}}) \quad \text{(covariance between X and Y)}$$

$$\text{SD (X)} = \sigma_x = \sqrt{\frac{1}{n}\Sigma_i(\text{X}_i - \overline{\text{X}})^2} \quad \text{(standard deviation of X)}$$

$$\text{SD (Y)} = \sigma_y = \sqrt{\frac{1}{n}\Sigma_i(\text{Y}_i - \overline{\text{Y}})^2} \quad \text{(standard deviationof Y)}$$

Hence the formula to compute Karl Pearson correlation co-efficient is

$$r(\text{X, Y}) = \frac{\frac{1}{n}\Sigma_i(\text{X}_i - \overline{\text{X}})(\text{Y}_i - \overline{\text{Y}})}{\sqrt{\frac{1}{n}\Sigma_i(\text{X}_i - \overline{\text{X}})^2}\sqrt{\frac{1}{n}\Sigma_i(\text{Y}_i - \overline{\text{Y}})^2}}$$

$$= \frac{\Sigma_i(\text{X}_i - \overline{\text{X}})(\text{Y}_i - \overline{\text{Y}})}{\sqrt{\Sigma_i(\text{X}_i - \overline{\text{X}})^2}\sqrt{\Sigma_i(\text{Y}_i - \overline{\text{Y}})^2}} = \frac{\Sigma xy}{\sqrt{\Sigma x^2}\sqrt{\Sigma y^2}}$$

Note

The following formula may also be used to compute correlation co-efficient between the two variables X and Y.

(i) $r(\text{X, Y}) = \dfrac{\text{N}\Sigma\text{XY - }\Sigma\text{X}\,\Sigma\text{Y}}{\sqrt{\text{N}\Sigma\text{X}^2 - (\Sigma\text{X})^2}\sqrt{N\Sigma\text{Y}^2 - (\Sigma\text{Y})^2}}$

(ii) $r(\text{X, Y}) = \dfrac{N\Sigma dxdy - \Sigma dx\,\Sigma dy}{\sqrt{N\Sigma dx^2 - (\Sigma dx)^2}\sqrt{N\Sigma dy^2 - (\Sigma dy)^2}}$

where $dx = x - \text{A}$; $dy = y - \text{B}$ are the deviations from arbitrary values A and B.

10.2.4 Limits for Correlation co-efficient

Correlation co-efficient lies between -1 and +1.
i.e. $-1 \leq r\,(x, y) \leq 1$.

(i) If $r(\text{X , Y}) = +1$ the variables X and Y are said to be perfectly possitively correlated.

(ii) If $r(\text{X , Y}) = -1$ the variables X and Y are said to be perfectly negatively correlated.

(iii) If $r(\text{X , Y}) = 0$ the variables X and Y are said to be uncorrelated.

Example 6

Calculate the correlation co-efficient for the following heights (in inches) of fathers(X) and their sons(Y).

X	**:**	**65**	**66**	**67**	**67**	**68**	**69**	**70**	**72**
Y	**:**	**67**	**68**	**65**	**68**	**72**	**72**	**69**	**71**

Solution :

$$\overline{\text{X}} = \frac{\Sigma\text{X}}{\text{n}} = \frac{544}{8} = 68$$

$$\overline{\text{Y}} = \frac{\Sigma\text{Y}}{\text{n}} = \frac{552}{8} = 69$$

X	Y	$x = X - \overline{X}$	$y = Y - \overline{Y}$	x^2	y^2	xy
65	67	-3	-2	9	4	6
66	68	-2	-1	4	1	2
67	65	-1	-4	1	16	4
67	68	-1	-1	1	1	1
68	72	0	3	0	9	0
69	72	1	3	1	9	3
70	69	2	0	4	0	0
72	71	4	2	16	4	8
544	**552**	**0**	**0**	**36**	**44**	**24**

Karl Pearson Correlation Co-efficient,

$$r(x, y) = \frac{\Sigma xy}{\sqrt{\Sigma x^2}\sqrt{\Sigma y^2}} = \frac{24}{\sqrt{36}\sqrt{44}} = .603$$

Since $r(x, y) = .603$, the variables X and Y are positively correlated. i.e. heights of fathers and their respective sons are said to be positively correlated.

Example 7

Calculate the correlation co-efficient from the data below:

X	**:**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Y	**:**	**9**	**8**	**10**	**12**	**11**	**13**	**14**	**16**	**15**

Solution :

X	Y	X^2	Y^2	XY
1	9	1	81	9
2	8	4	64	16
3	10	9	100	30
4	12	16	144	48
5	11	25	121	55
6	13	36	169	78
7	14	49	196	98
8	16	64	256	128
9	15	81	225	135
45	**108**	**285**	**1356**	**597**

$$r\ (X,Y) = \frac{N\Sigma XY - \Sigma X\,\Sigma Y}{\sqrt{N\Sigma X^2 - (\Sigma X)^2}\sqrt{N\Sigma Y^2 - (\Sigma Y)^2}}$$

$$= \frac{9(597) - (45)\,(108)}{\sqrt{9(285) - (45)^2}\sqrt{9(1356) - (108)^2}} = .95$$

$\therefore$ X and Y are highly positively correlated.

Example 8

Calculate the correlation co-efficient for the ages of husbands (X) and their wives (Y)

X :	**23**	**27**	**28**	**29**	**30**	**31**	**33**	**35**	**36**	**39**
Y :	**18**	**22**	**23**	**24**	**25**	**26**	**28**	**29**	**30**	**32**

Solution :

Let A = 30 and B = 26 then dx = X– A dy = Y–B

X	Y	d_x	d_y	$d^2{}_x$	$d^2{}_y$	d_xd_y
23	18	-7	-8	49	64	56
27	22	-3	-4	9	16	12
28	23	-2	-3	4	9	6
29	24	-1	-2	1	4	2
30	25	0	-1	0	1	0
31	26	1	0	1	0	0
33	28	3	2	9	4	6
35	29	5	3	25	9	15
36	30	6	4	36	16	24
39	32	9	6	81	36	54
		11	**-3**	**215**	**159**	**175**

$$r\ (x, y) = \frac{N\Sigma dxdy - \Sigma dx\ \Sigma dy}{\sqrt{N\Sigma d_x^2 - (\Sigma dx)^2}\sqrt{N\Sigma d_y^2 - (\Sigma dy)^2}}$$

$$= \frac{10(175)-(11)(-3)}{\sqrt{10(215)-(11)^2}\sqrt{10(159)-(-3)^2}}$$

$$= \frac{1783}{1790.8} = 0.99$$

$\therefore$ X and Y are highly positively correlated. i.e. the ages of husbands and their wives have a high degree of correlation.

Example 9

Calculate the correlation co-efficient from the following data

N = 25,	**SX = 125,**	**SY = 100**
SX^2 = 650	**SY^2 = 436,**	**SXY = 520**

Solution :

We know,

$$r = \frac{N\Sigma XY - \acute{O}X\acute{O}Y}{\sqrt{N\Sigma X^2-(\Sigma X)^2}\sqrt{N\Sigma Y^2-(\Sigma Y)^2}}$$

$$= \frac{25(520) - (125)(100)}{\sqrt{25(650)-(125)^2}\sqrt{25(436)-(100)^2}}$$

$$r = -0.667$$

10.2.5 Regression

Sir Francis Galton (1822 - 1911), a British biometrician, defined **regression** in the context of heriditary characteristics. The literal meaning of the word "regression" is **"Stepping back towards the average".**

Regression is a mathematical measure of the average relationship between two or more variables in terms of the original units of the data.

There are two types of variables considered in regression analysis, namely dependent variable and independent variable(s).

10.2.6 Dependent Variable

The variable whose value is to be predicted for a given independent variable(s) is called dependent variable, denoted by Y. For example, if advertising (X) and sales (Y) are correlated, we could estimate the expected sales (Y) for given advertising expenditure (X). So in this case Y is a dependent variable.

10.2.7 Independent Variable

The variable which is used for prediction is called an independent variable. For example, it is possible to estimate the required amount of expenditure (X) for attaining a given amount of sales (Y), when X and Y are correlated. So in this case Y is independent variable. There can be more than one independent variable in regression.

The line of regression is the line which gives the best estimate to the value of one variable for any specific value of the other variable.

Thus the line of Regression is the **line of best fit** and is obtained by the **principle of least squares**. (Refer pages 61 & 62 of Chapter 7). The equation corresponds to the line of regression is also referred to as regression equation.

10.2.8 Two Regression Lines

For the pair of values of (X, Y), where X is an independent variable and Y is the dependent variable the line of regression of Y on X is given by

$$Y - \overline{Y} = b_{yx}(X - \overline{X})$$

where b_{yx} is the regression co-efficient of Y on X and given by $b_{yx} = r\,.\,\frac{\sigma_y}{\sigma_x}$, where r is the correlation co-efficient between X and Y and σ_x and σ_y are the standard deviations of X and Y respectively.

$$\therefore\ b_{yx} = \frac{\Sigma xy}{\Sigma x^2} \quad \text{where } x = X - \overline{X} \text{ and } y = Y - \overline{Y}$$

Similarly when Y is treated as an independent variable and X as dependent variable, the line of regression of X on Y is given by

$$(X - \overline{X}) = b_{xy} (Y - \overline{Y})$$

where $b_{xy} == r \frac{\sigma_x}{\sigma_y} = \frac{\Sigma xy}{\Sigma y^2}$ here $x = X - \overline{X}$; $y = Y - \overline{Y}$

Note

The two regression equations are not reversible or interchangeable because of the simple reason that the basis and assumption for deriving these equations are quite different.

Example 10

Calculate the regression equation of X on Y from the following data.

X	**:**	**10**	**12**	**13**	**12**	**16**	**15**
Y	**:**	**40**	**38**	**43**	**45**	**37**	**43**

Solution :

X	Y	$x=X-\overline{X}$	$y=Y-\overline{Y}$	x^2	y^2	xy
10	40	-3	-1	9	1	3
12	38	-1	-3	1	9	3
13	43	0	2	0	4	0
12	45	-1	4	1	16	-4
16	37	3	-4	9	16	-12
15	43	2	2	4	4	4
78	**246**	**0**	**0**	**24**	**50**	**6**

$$\overline{X} = \frac{\Sigma X}{n} = \frac{78}{6} = 13 \qquad \overline{Y} = \frac{\Sigma Y}{n} = \frac{246}{6} = 41$$

$$b_{xy} = \frac{\Sigma xy}{\Sigma y^2} = \frac{-6}{50} = -0.12$$

$\therefore$ Regression equation of X on Y is $(X - \overline{X}) = b_{xy} (Y - \overline{Y})$

$$X - 13 = -0.12 (Y - 41) \Rightarrow X = 17.92 - 0.12Y$$

Example 11

Marks obtained by 10 students in Economics and Statistics are given below.

Marks in Economics	**:**	**25**	**28**	**35**	**32**	**31**	**36**	**29**	**38**	**34**	**32**
Marks in Statistics	**:**	**43**	**46**	**49**	**41**	**36**	**32**	**31**	**30**	**33**	**39**

Find (i) the regression equation of Y on X

(ii) estimate the marks in statistics when the marks in Economics is 30.

Solution :

Let the marks in Economics be denoted by X and statistics by Y.

X	Y	$x=X-\overline{X}$	$y=Y-\overline{Y}$	x^2	y^2	xy
25	43	-7	5	49	25	-35
28	46	-4	8	16	64	32
35	49	3	11	9	121	33
32	41	0	3	0	9	0
31	36	-1	-2	1	4	2
36	32	4	-6	16	36	-24
29	31	-3	-7	9	49	21
38	30	6	-8	36	64	-48
34	33	2	-5	4	25	-10
32	39	0	1	0	1	0
320	**380**	**0**	**0**	**140**	**398**	**-93**

$$\overline{X} = \frac{\Sigma X}{n} = \frac{320}{10} = 32 \qquad \overline{Y} = \frac{\Sigma Y}{n} = \frac{380}{10} = 38$$

$$b_{yx} = \frac{\Sigma xy}{\Sigma x^2} = \frac{-93}{140} = -0.664$$

(i) Regression equation of Y on X is

$$Y - \overline{Y} = b_{yx}(X - \overline{X})$$

$$Y - 38 = -0.664 (X - 32)$$

$$\Rightarrow \quad Y = 59.25 - 0.664X$$

(ii) To estimate the marks in statistics (Y) for a given marks in the Economics (X), put X = 30, in the above equation we get,

$$Y = 59.25 - 0.664(30)$$
$$= 59.25 - 19.92 = 39.33 \text{ or } 39$$

Example 12

Obtain the two regression equations for the following data.

X	**:**	**4**	**5**	**6**	**8**	**11**
Y	**:**	**12**	**10**	**8**	**7**	**5**

Solution :

The above values are small in magnitude. So the following formula may be used to compute the regression co-efficient.

$$b_{xy} = \frac{N\Sigma XY - \acute{O}X\acute{O}Y}{N\Sigma Y^2 - (\Sigma Y)^2}$$

$$b_{yx} = \frac{N\Sigma XY - \acute{O}X\acute{O}Y}{N\Sigma X^2 - (\Sigma X)^2}$$

X	Y	X^2	Y^2	XY
4	12	16	144	48
5	10	25	100	50
6	8	36	64	48
8	7	64	49	56
11	5	121	25	55
34	**42**	**262**	**382**	**-257**

$$\overline{X} = \frac{\Sigma X}{n} = \frac{34}{5} = 6.8 \qquad \overline{Y} = \frac{\Sigma Y}{n} = \frac{42}{5} = 8.4$$

$$b_{XY} = \frac{5(257) - (34)(42)}{5(382) - (42)^2} = -0.98$$

$$b_{YX} = \frac{5(257) - (34)(42)}{5(262) - (34)^2} = -0.93$$

Regression Equation of X on Y is

$$(X - \overline{X}) = b_{XY}(Y - \overline{Y})$$

$\Rightarrow \quad X - 6.8 = -0.98(Y - 8.4)$

$X = 15.03 - 0.98Y$

Regression equation of Y on X is

$Y - \overline{Y} = b_{YX}(X - \overline{X})$

$Y - 8.4 = -0.93(X - 6.8)$

$\Rightarrow \quad Y = 14.72 - 0.93X$

EXERCISE 10.2

1) Calculate the correlation co-efficient from the following data.

X :	12	9	8	10	11	13	7
Y :	14	8	6	9	11	12	3

2) Find the co-efficient of correlation for the data given below.

X :	10	12	18	24	23	27
Y :	13	18	12	25	30	10

3) From the data given below, find the correlation co-efficient.

X :	46	54	56	56	58	60	62
Y :	36	40	44	54	42	58	54

4) For the data on price (in rupees) and demand (in tonnes) for a commodity, calculate the co-efficient of correlation.

Price (X) :	22	24	26	28	30	32	34	36	38	40
Demand(Y) :	60	58	58	50	48	48	48	42	36	32

5) From the following data, compute the correlation co-efficient.
$N = 11$, $\Sigma X = 117$, $\Sigma Y = 260$, $\Sigma X^2 = 1313$
$\Sigma Y^2 = 6580$, $\Sigma XY = 2827$

6) Obtain the two regression lines from the following

X :	6	2	10	4	8
Y :	9	11	5	8	7

7) With the help of the regression equation for the data given below calculate the value of X when Y = 20.

X :	10	12	13	17	18
Y :	5	6	7	9	13

8) Price indices of cotton (X) and wool (Y) are given below for the 12 months of a year. Obtain the equations of lines of regression between the indices.

X :	78	77	85	88	87	82	81	77	76	83	97	93
Y :	84	82	82	85	89	90	88	92	83	89	98	99

9) Find the two regression equations for the data given below.

X :	40	38	35	42	30
Y :	30	35	40	36	29

10.3 TIME SERIES ANALYSIS

Statistical data which relate to successive intervals or points of time, are referred to as "time series".

The following are few examples of time series.

(i) Quarterly production, Half-yearly production, and yearly production for particular commodity.

(ii) Amount of rainfall over 10 years period.

(iii) Price of a commodity at different points of time.

There is a strong notion that the term " time series" usually refer only to Economical data. But it equally applies to data arising in other natural and social sciences. Here the time sequence plays a vital role and it requires special techniques for its analysis. In analysis of time series, we analyse the past in order to understand the future better.

10.3.1 Uses of analysis of Time Series

(i) It helps to study the past conditions, assess the present achievements and to plan for the future.

(ii) It gives reliable forcasts.

(iii) It provides the facility for comparison.

Thus wherever time related data is given in Economics, Business, Research and Planning, the analysis of time series provides the opportunity to study them in proper perspective.

10.3.2 Components of Time Series

A graphical representation of a Time Series data, generally shows the changes (variations) over time. These changes are known as principal components of Time Series . They are

(i)	Secular trend	(ii) Seasonal variation
(iii)	Cyclical variation	(iv) Irregular variation.

Secular trend (or Trend)

It means the smooth, regular, long-term movement of a series if observed long enough. It is an upward or downward trend. It may increase or decrease over period of time. For example, time series relating to population, price, production, literacy,etc. may show increasing trend and time series relating to birth rate, death rate, poverty may show decreasing trend.

Seasonal Variation

It is a short-term variation. It means a periodic movement in a time series where the period is not longer than one year. A periodic movement in a time series is one which recurs or repeats at regular intervals of time or periods. Following are the examples of seasonal variation.

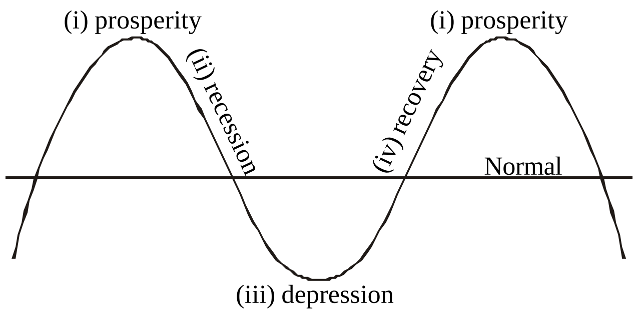
(i) passenger traffic during 24 hours of a day

(ii) sales in a departmental stores during the seven days of a week.

The factors which cause this type of variation are due to climatic changes of the different season, customs and habits of the people. For example more amount of ice creams will be sold in summer and more number of umbrellas will be sold during rainy seasons.

Cyclical Variation

It is also a short-term variation. It means the oscillatory movement in a time series, the period of oscillation being more than a year. One complete period is called a cycle. Business cycle is the suitable example for cyclical variation. There are many time series relating to Economics and Business, which have certain wave-like movements called business cycle. The four phases in business cycle, (i) prosperity (ii) recession (iii) depression (iv) recovery, recur one after another regularly.



(Fig. 10.7)

Irregular Variation

This type of variation does not follow any regularity. These variations are either totally unaccountable or caused by unforeseen events such as wars, floods, fire, strikes etc. Irregular variation is also called as **Erratic Variation.**

10.3.3 Models

In a given time series, some or all the four components, namely secular trend, seasonal variation, cyclical variation and irregular variation may be present. It is important to separate the different components of times series because either our interest may be on a particular component or we may want to study the series after eliminating the effect of a particular component. Though there exist many models, here we consider only two models.

Multiplicative Model

According to this model, it is assumed that there is a multiplicative relationship among the four components. i.e.,

$$y_t = T_t \times S_t \times C_t \times I_t,$$

Where y_t is the value of the variable at time *t*, or observed data at time *t*, T_t is the Secular trend or trend, S_t is the Seasonal variation, C_t is the Cyclical variation and I is the Irregular variation or Erratic variation.

Additive Model

According to this model, it is assumed that y_t be the sum of the four components.

$$y_t = T_t + S_t + C_t + I_t,$$

10.3.4 Measurement of secular trend

The following are the four methods to estimate the secular trend

(i) Graphic method or free - hand method

(ii) Method of Semi - Averages

(iii) Method of Moving Averages

(iv) Method of least squares.

(i) Graphic Method / Free - hand Method

This is the simplest method of studying the trend procedure. Let us take time on the x - axis, and observed data on the y-axis. Mark a point on a graph sheet, corresponding to each pair of time and observed value. After marking all such possible points, draw a straight line which will best fit to the data according to personal judgement.

It is to be noted that the line should be so drawn that it passes between the plotted points in such a manner that the fluctuations in one direction are approximately equal to those in other directions.

When a trend line is fitted by the free hand method an attention should be paid to conform the following conditions.

(i) The number of points above the line is equal to the number of points below the line, as far as possible.

(ii) The sum of the vertical deviations from the trend of the annual observation above the trend should equal the sum of the vertical deviations from the trend of the observations below the trend.

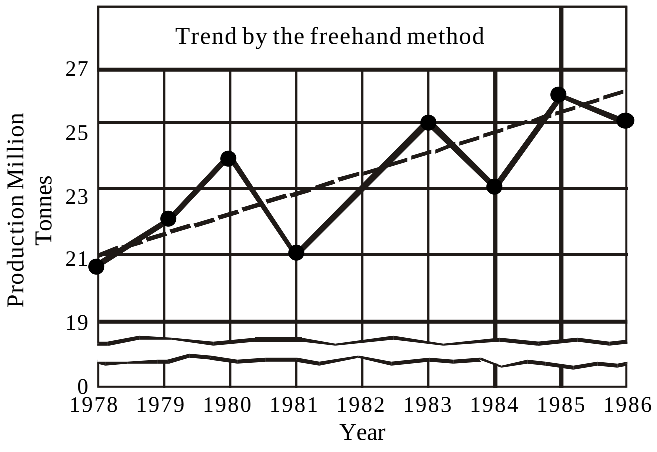
(iii) The sum of the squares of the vertical deviations of the observations from the trend should be as small as possible.

Example 13

Fit a trend line to the following data by the free hand mehtod.

year	**1978**	**1979**	**1980**	**1981**	**1982**
production of steel	**20**	**22**	**24**	**21**	**23**
year	**1983**	**1984**	**1985**	**1986**	
production of steel	**25**	**23**	**26**	**25**	

Solution :



(Fig. 10.8)

Note

(i) The trend line drawn by the free hand method can be extended to predict future values. However, since the free hand curve fitting is too subjective, this method should not be generally used for predictions

(ii) In the above diagram **false base line** (zig-zag) has been used. Generally we use false base line following objectives.

(a) Variations in the data are clearly shown

(b) A large part of the graph is not wasted or space is saved.

(c) The graph provides a better visual communications.

(ii) Method of Semi Averages

This method involves very simple calculations and it is easy to adopt. When this method is used the given data is divided into two equal parts. For example, if we are given data from 1980 to 1999, i.e., over a period of 20 years, the two equal points will be first 10 years, i.e. from 1980 to 1989, and from 1990 to 1999. In case of odd number of years like 7, 11, 13 etc., two equal parts can be made by omitting the middle year. For example, if the data are given for 7 years from 1980 to 1986, the two equal parts would be from 1980 to 1982 and from 1984 to 1986. The middle year 1983 will be omitted.

After dividing the data in two parts, find the arithmetic mean of each part. Thus we get semi averages from which we calculate the annual increase or decrease in the trend.

Example 14

Find trend values to the following data by the method of semi-averages.

Year	**1980**	**1981**	**1982**	**1983**	**1984**	**1985**	**1986**
Sales	**102**	**105**	**114**	**110**	**108**	**116**	**112**

Solution :

No. of years = 7 (odd no.) By omitting the middle year (1983) we have

Year	Sales	Semi total	Semi-average
1980	102		
1981	105	321	107
1982	114		
1983	110		
1984	108		
1985	116	336	112
1986	112		

Difference between middle periods = 1985 – 1981 = 4

Difference between semi averages = 112 - 107 = 5

Annual increase in trend $= \frac{5}{4} = 1.25$

Year	1980	1981	1982	1983	1984	1985	1986
Trend	105.75	107	108.25	109.50	110.75	112	113.25

Example 15

The sales in tonnes of a commodity varied from 1994 to 2001 as given below:

Year	**1994**	**1995**	**1996**	**1997**
Sales	**270**	**240**	**230**	**230**
Year	**1998**	**1999**	**2000**	**2001**
Sales	**220**	**200**	**210**	**200**

Find the trend values by the method of semi-average. Estimate the sales in 2005.

Solution :

Year	Sales	Semi total	Semi average
1994	270		
1995	240		
1996	230	$\longrightarrow$ 970 $\longrightarrow$	242.5
1997	230		
1998	220		
1999	200		
2000	210	$\longrightarrow$ 830 $\longrightarrow$	207.5
2001	200		

Difference between middle periods = 1999.5 - 1995.5 = 4

Decrease in semi averages = 242.5 - 207.5 = 35

Annual decrease in trend $= \frac{35}{4} = 8.75$

Half yearly decrease in trend = 4.375

Year	1994	1995	1996	1997
Sales trend	255.625	246.875	238.125	229.375
Year	1998	1999	2000	2001
Sales trend	220.625	211.875	203.125	194.375

Trend value for the year 2005 = 194.375 – (8.75 x 4) = 159.375

(iii) Method of Moving Averages

This method is simple and flexible algebraic method of measuring trend. The method of Moving Average is a simple device for eliminating fluctuations and obtaining trend values with a fair degree of accuracy. The technique of Moving Average is based on the arithmetic mean but with a distinction. In arithmetic mean we sum all the items and divide the sum by number of items, whereas in Moving Average method there are various averages in one series depending upon the number of years taken in a Moving Average. While applying this method, it is necessary to define a period for Moving Average such as 3 yearly moving average (odd number of years), 4-yearly moving average (even number of years) etc.

Moving Average - Odd number of years (say 3 years)

To find out the trend values by the method of 3-yearly moving averages, the following steps are taken into consideration.

1) Add up the values of the first three years and place the yearly sum against the median year i.e. the 2nd year.
2) Leave the first year item and add up the values of the next three years. i.e. from the 2nd year to 4 th year and place the sum (known as moving total) against the 3rd year.
3) Leave the first two items and add the values of the next three years. i.e. from 3rd year to the 5th year and place the sum (moving total) against the 4th year.
4) This process must be continued till the value of the last item is taken for calculating the moving average.
5) Each 3-yearly moving total must be divided by 3 to get themoving averages. This is our required trend values.

Note

The above 5 steps can be applied to get 5-years, 7-years, 9-years etc., Moving Averages.

Moving Average - Even number of years (say 4 years)

1) Add up the values of the first four years and place the sum against the middle of 2nd and 3rd years.
2) Leave the first year value and add from the 2nd year onwards to the 5th year and write the sum (moving total) against the middle of the 3rd and the 4th items.
3) Leave the first two year values and add the values of the next four years i.e. from the 3rd year to the 6th year. Place the sum (moving total) against the middle of the 4th and the 5th items.
4) This process must be continued till the value of the last item is taken into account.
5) Add the first two 4-years moving total and write the sum against 3rd year.
6) Leave the first 4-year moving total and add the next two 4-year moving total. This sum must be placed against 4th year.
7) This process must be continued till all the four-yearly moving totals are summed up and centred.
8) Divide the 4-years moving total centred by 8 and write the quotient in a new column. These are our required trend values.

Note

The above steps can be applied to get 6-years, 8-years, 10-years etc., Moving Averages.

Example 16

Calculate the 3-yearly Moving Averages of the production figures (in mat. tonnes) given below

Year	**1973**	**1974**	**1975**	**1976**	**1977**	**1978**	**1979**	**1980**
Production	**15**	**21**	**30**	**36**	**42**	**46**	**50**	**56**
Year	**1981**	**1982**	**1983**	**1984**	**1985**	**1986**	**1987**	
Production	**63**	**70**	**74**	**82**	**90**	**05**	**102**	

Solution :

Calculation of 3-yearly Moving Averages

Year	Productiion y	3-yearly Moving total	3-yearly Moving average
1973	15	---	---
1974	21	66	22.00
1975	30	87	29.00
1976	36	108	36.00
1977	42	124	41.33
1978	46	138	46.00
1979	50	152	50.67
1980	56	169	56.33
1981	63	189	63.00
1982	70	207	69.00
1983	74	226	75.33
1984	82	246	82.00
1985	90	267	89.00
1986	95	287	95.67
1987	102	---	---

Example 17

Estimate the trend values using the data given below by taking 4-yearly Moving Average.

Year	**1974**	**1975**	**1976**	**1977**	**1978**	**1979**	**1980**
Value	**12**	**25**	**39**	**54**	**70**	**37**	**105**
Year	**1981**	**1982**	**1983**	**1984**	**1985**	**1986**	**1987**
Value	**100**	**82**	**65**	**49**	**34**	**20**	**7**

Solution :

Year	value	4 year moving total	4 year moving total centered	Two 4-year moving total (Trend values)
1974	12	---	---	---
1975	25		---	---
		$\rightarrow$ 130		
1976	39		318	39.75
		$\rightarrow$ 188		
1977	54		388	48.50
		$\rightarrow$ 200		
1978	70		466	58.25
		$\rightarrow$ 266		
1979	37		578	72.25
		$\rightarrow$ 312		
1980	105		636	79.50
		$\rightarrow$ 324		
1981	100		676	84.50
		$\rightarrow$ 352		
1982	82		648	81.00
		$\rightarrow$ 296		
1983	65		526	65.75
		$\rightarrow$ 230		
1984	49		398	49.75
		$\rightarrow$ 168		
1985	34		278	34.75
		$\rightarrow$ 110		
1986	20		---	---
1987	7		---	---

(iv) Method of Least Squares

The method of least squares is most widely used in practice. The method of least squares may be used to fit a straight line trend.

The straight line trend is generally expressed by an equation

$$y_t = a + bx$$

Where y_t is used to represent the trend values, 'a' is the intercept, 'b' represents slope of the line which is also known as the ratio of growth during a unit of time. The variable x represents the time periods.

In order to determine the values of the unknown constants '*a*' and '*b*' the following equations, known as normal equations, are used.

$$\Sigma y = na + b\Sigma x$$

$$\Sigma xy = a\Sigma x + b\Sigma x^2,$$

where n represents number of observations (years, months or any other period) for which the data are given.

For derivation of normal equations, refer pages 61 & 62 of Chapter 7. To solve the above normal equations and get trend values the following are the computational steps.

Case (i) When the number of years is odd

1) Denote the years as the X variable and its corresponding values as Y.

2) Assume the middle year as the period of origin and take deviations accordingly. Thus $\Sigma X = 0$.

3) Find ΣX^2, ΣY^2 and ΣXY.

4) Substitute ΣX, ΣX^2, ΣY and ΣXY in the above normal equation and the solve it.

   Hence $a = \dfrac{\Sigma Y}{n}$ $b = \dfrac{\Sigma XY}{\Sigma X^2}$

5) Put the values of '*a*' and '*b*' in the equation and solving for each value of X, we get the trend values.

Case (ii) When the number of years is even

In this case, assume the variable X as

$$X = \frac{x - \text{A.M. of two Middle years}}{.5}$$

and all other steps are similar to case (i)

Note

If the time lag between consecutive years is not one assume the variable X as

$$X = \frac{x - \text{A.M. of two Middle years}}{d}$$

where $d = \dfrac{\text{Difference between two consecutive years}}{2}$

Example 18

Below are the given the figures of production (in thousand tonnes) of a sugar factory.

Year	**1980**	**1981**	**1982**	**1983**	**1984**	**1985**	**1986**
Production	**80**	**90**	**92**	**83**	**94**	**99**	**92**

Fit a straight line trend to these figures by the method of least squares and estimate the production in the year of 1990.

Solution :

Given the numbers of years $n = 7$ (odd)

Year x	Production ('000 tonnes) $y = Y$	$X = x - 1983$	X^2	XY
1980	80	-3	9	-240
1981	90	-2	4	-180
1982	92	-1	1	-92
1983	83	0	0	0
1984	94	1	1	94
1985	99	2	4	198
1986	92	3	9	276
	630	**0**	**28**	**56**

The equation of the straight line trend is $Y_t = a + bX$

Substituting the values of ΣX, ΣX^2, ΣXY and n in normal equation, we get

$630 = 7a + b(0) \quad \Rightarrow \; a = 90$

$56 = a(0) + b(28) \quad \Rightarrow \; b = 2$

Hence the equation of the straight line trend is

$$Y_t = 90 + 2X$$

Trend values

For $X = -3$, $\quad Y_t = 90 + 2(-3) = 84$

For $X = -2$, $\quad Y_t = 90 + 2(-2) = 86$

For $X = -2$, $\quad Y_t = 90 + 2(-1) = 88$

For $X = 0$, $\quad Y_t = 90 + 2(0) = 90$

For $X = 1$, $\quad Y_t = 90 + 2(1) = 92$

For $X = 2$, $\quad Y_t = 90 + 2(2) = 94$

For $X = 3$, $\quad Y_t = 90 + 2(3) = 96$

To estimate the production in 1990, substitute X = 7 in the trend equation.

$\therefore\; Y_{1990} = 90 + 2(7) = 104 \times 1000$ tonnes

Example 19

Fit a straight line trend by the method of least squares to the following data. Also predict the earnings for the year 1988.

Year	**1979**	**1980**	**1981**	**1982**	**1983**	**1984**	**1985**	**1986**
Earnings	**38**	**40**	**65**	**72**	**69**	**60**	**87**	**95**

Solution :

Number of years $n = 8$ (even)

year x	Earnings (in lakhs) y = Y	X = $\frac{x-1982.5}{.5}$	X^2	XY
1979	38	-7	49	-266
1980	40	-5	25	-200
1981	65	-3	9	-195
1982	72	-1	1	-72
1983	69	1	1	69
1984	60	3	9	180
1985	87	5	25	435
1986	95	7	49	665
	526	**0**	**168**	**616**

The equation of the straight line trend is $Y_t = a + bX$

Substituting the values of ΣX, ΣX^2, ΣXY, n in Normal equation, we get,

$526 = 8a + b(0)$ -----------(1)
$616 = a(0) + b(168)$ -----------(2)

Solving (1) and (2) we get $a = 65.75$ and $b = 3.667$

Hence the equation of the straight line trend is

$$Y_7 = 65.75 + 3.667X$$

Year	1979	1980	1981	1982
Trend values	40.08	47.415	54.749	62.083

Year	1983	1984	1985	1986
Trend values	69.417	76.751	84.085	91.419

To estimate the earnings in 1988, substitute X = 11 in the trend equation and we get,

$Y_t = 65.75 + 3.667(11) = 106.087$

The estimate earnings for the year 1988 are Rs.106.087 lakhs.

Measurement of seasonal variation

Seasonal variation can be measured by the method of simple average.

Method of simple average

This method is the simple method of obtaining a seasonal Index. In this method the following steps are essential to calculate the Index.

(i) Arrange the data by years, month or quarters as the case may be.

(ii) Compute the totals of each month or each quarter.

(iii) Divide each total by the number of years for which the data are given. This gives seasonal averages (monthly or quarterly)

(iv) Compute average of seasonal averages. This is called grand average.

(v) Seasonal Index for every season (monthly or quarterly) is calculated as follows

$$\text{Seasonal Index (S.I)} = \frac{\text{Seasonal Average}}{\text{Grand Average}} \times 100$$

Note

(i) If the data is given monthwise,

$$\text{Seasonal Index} = \frac{\text{Monthly Average}}{\text{Grand Average}} \times 100$$

(ii) If quarterly data is given,

$$\text{Seasonal Index} = \frac{\text{Quarterly Average}}{\text{Grand Average}} \times 100$$

Example 20

From the data given below calculate Seasonal Indices.

	year				
Quarter	**1984**	**1985**	**1986**	**1987**	**1988**
I	**40**	**42**	**41**	**45**	**44**
II	**35**	**37**	**35**	**36**	**38**
III	**38**	**39**	**38**	**36**	**38**
IV	**40**	**38**	**40**	**41**	**42**

Solution :

	Quarters			
year	**I**	**II**	**III**	**IV**
1984	40	35	38	40
1985	42	37	39	38
1986	41	35	38	40
1987	45	36	36	41
1988	44	38	38	42
Total	**212**	**181**	**189**	**201**
Average	**42.4**	**36.2**	**37.8**	**40.2**

Grand Average = $\frac{42.4+36.2+37.8+40.2}{4}$ = 39.15

Seasonal Index (S. I) = $\frac{\text{Quarterly Average}}{\text{Grand Average}}$ x 100

Hence, S.I for I Quarter $= \frac{42.4}{39.15} \times 100 = 108.30$

S.I for II Quarter $= \frac{36.2}{39.15} \times 100 = 92.54$

S.I for III Quarter $= \frac{37.8}{39.15} \times 100 = 96.55$

S.I for IV Quarter $= \frac{40.2}{39.15} \times 100 = 102.68$

Note

Measurement of cyclical variation, and measurement of irregular variation is beyond the scope of this book.

EXERCISE 10.3

1) Draw a trend line by graphic method (free hand)

year	1995	1996	1997	1998	1999	2000	2001
Production	20	22	25	26	25	27	30

2) Draw a trend line by graphic method

year	1997	1998	1999	2000	2001
Production	20	24	25	38	60

3) Obtain the trend values by the method of Semi-Average

year	1987	1988	1989	1990	1991	1992	1993
Production (in tonnes)	90	110	130	150	100	150	200

[redacted]

year	1993	1994	1995	1996	1997	1998	1999	2000
Netprofit (Re lakhs)	38	39	41	43	40	39	35	25

5) Using three year moving averages determine the trend values for the following data.

year	1983	1984	1985	1986	1987	1988	1989
Production (in tonnes)	21	22	23	25	24	22	25

year	1990	1991	1992
Production (in tonnes)	26	27	26

6) Below are given figures of production (in thousand tonnes) of a sugar factory. Obtain the trend values be 3-year moving average.

year	1980	1981	1982	1983	1984	1985	1986
Production	80	90	92	83	94	99	92

7) Using four yearly moving averages calculate the trend values.

year	1981	1982	1983	1984	1985	1986	1987
Production	464	515	518	467	502	540	557

year	1988	1989	1990
Production	571	586	612

8) Calculate the trend values by four year moving averages method.

Year	1978	1979	1980	1981	1982	1983	1984
Production	614	615	652	678	681	655	717
Year	1985	1986	1987	1988			
Production	719	708	779	757			

9) Given below are the figures of production of a sugar factory.

year	1989	1990	1991	1992	1993	1994	1995
Production (in tonnes)	77	88	94	85	91	98	90

Fit a straight line trend to these figures by the method of least squares and estimate trend values. Also estimate the production for the year 2000.

10) Fit the straight line trend, find the trend values and estimate the net profit in 2002.

Year	1992	1993	1994	1995	1996	1997	1998
Net profit	65	68	59	55	50	52	54
year	1999	2000					
Net profit	50	42					

11) The following data relate to the profit earned by public limited company from 1984 to 1989.

Year	1984	1985	1986	1987	1988	1989
Profit (Rs in 000)	10	12	15	16	18	19

Fit a straight line trend by the method of least squares to the data and tabulate the trend values.

12) Fit a straight line trend and estimate the trend values.

Year	1992	1993	1994	1995	1996	1997	1998	1999
Net profit	47	53	50	46	41	39	40	36

13) Calculate the seasonal indices by the method of simple average for the following data

year	I quarter	II quarter	III quarter	IV quarter
1985	68	62	61	63
1986	65	58	66	61
1987	68	63	63	67

14) Calculate the seasonal indices for the following data by the method of simple Average.

	Quarters			
year	I	II	III	IV
1994	78	66	84	80
1995	76	74	82	78
1996	72	68	80	70
1997	74	70	84	74
1998	76	74	86	82

15) Calculate the seasonal Indices for the following data using average method.

	Quarters			
year	I	II	III	IV
1982	72	68	80	70
1983	76	70	82	74
1984	74	66	84	80
1985	76	74	84	78
1986	78	74	86	82

10.4 INDEX NUMBERS

"An **Index Number** is a single ratio (usually in percentages) which measures the (combined average) change of several variables between two different times, places and situations" - Alva. M. Tuttle.

Index numbers are the devices for measuring differences in the magnitude of a group of related variables, over two different situations or defined as a measure of the average change in a group of related variables over two different situations. For example, the price of commodities at two different places or two different time periods at the same location. We need Index Numbers to compare the cost of living at different times or in different locations.

10.4.1 Classification of Index Numbers

Index Numbers may be classified in terms of what they measure. They are

(i) Price Index Numbers

(ii) Quantity Index Numbers

(iii) Value Index Numbers

(iv) Special purpose Index Numbers

We shall discuss (i) and (ii).

10.4.2 Uses of Index Numbers

(i) Index numbers are used to evolve business policies.

(ii) Index numbers determine the inflation or deflation in economy.

(iii) Index numbers are used to compare intelligence of students in different locations or for different year.

(iv) Index numbers serve as economic barometers.

10.4.3 Method of construction of Index Numbers

Index Numbers are broadly divided into two groups

(i) Unweighted Index

(ii) Weighted Index

We confine our attention to weighted index numbedrs.

10.4.4 Weighted Index Numbers

The method of construction of weighted indices are

(c) Weighted aggregative method

(d) Weighted averages of relatives method

Weighted Aggregative Index Numbers

Let p_1 and p_0 be the prices of the current year and the base year respctively. Let q_1 and q_0 be the quantities of the current year and the base year respectively. The formulae for assigning weights to the items are :

(i) Laspeyre's Price Index

$P_{01}^{L} = \frac{\Sigma p_1 q_0}{\Sigma p_0 q_0}$ x 100 where w = $p_0 q_0$ is the weight assigned to the items and P_{01} is the price index.

(ii) Paasche's price index

$$P_{01}^{P} = \frac{\Sigma p_1 q_1}{\Sigma p_0 q_1} \text{ x } 100$$

Here the weights assigned to the items are the current year quantities i.e. W = $p_0 q_1$

(iii) Fisher's price Index

$$P_{01}^{F} = \sqrt{P_{01}^{L} \text{ x } P_{01}^{P}} = \sqrt{\frac{\Sigma p_1 q_0}{\Sigma p_0 q_0} \text{ x } \frac{\Sigma p_1 q_1}{\Sigma p_0 q_1}} \text{ x } 100$$

Note

Fisher's price index is the geometric mean of Laspeyre's and Paasche's price index numbers.

Example 21

Compute (i) Laspeyre's (ii) Paasche's and (iii) Fisher's Index Numbers for the 2000 from the following :

Commodity	**Price**		**Quantity**	
	1990	**2000**	**1990**	**2000**
A	**2**	**4**	**8**	**6**
B	**5**	**6**	**10**	**5**
C	**4**	**5**	**14**	**10**
D	**2**	**2**	**19**	**13**

Solution :

Commodity	Price		Quantity					
	Base year p_0	Current year p_1	Base year q_0	Current year q_1	p_0q_0	p_1q_0	p_0q_1	p_1q_1
A	2	4	8	6	16	32	12	24
B	5	6	10	5	50	60	25	30
C	4	5	14	10	56	70	40	50
D	2	2	19	13	38	38	26	26
					160	**200**	**103**	**130**

(i) Laspeyre's Index : $P^L_{01} = \frac{\Sigma p_1 q_0}{\Sigma p_0 q_0} \times 100$

$= \frac{200}{160} \times 100 = 125$

(ii) Paasche's Index : $P^P_{01} = \frac{\Sigma p_1 q_1}{\Sigma p_0 q_1} \times 100$

$= \frac{130}{103} \times 100 = 126.21$

(iii) Fisher's Index : $P^F_{01} = \sqrt{P^L_{01} \times P^P_{01}}$

$= 125.6$

Example 22

From the following data, calculate price index number by (a) Laspeyre's method (b) Paasche's method (iii) Fisher's method.

Commodity	**Base year**		**Current year**	
	Price	**Quantity**	**Price**	**Quantity**
A	**2**	**40**	**6**	**50**
B	**4**	**50**	**8**	**40**
C	**6**	**20**	**9**	**30**
D	**8**	**10**	**6**	**20**
E	**10**	**10**	**5**	**20**

Solution :

Commodity	Base year Price p_0	Base year Qty q_0	Current year Price p_1	Current year Qty q_1	p_0q_0	p_1q_0	p_0q_1	p_1q_1
A	2	40	6	50	80	240	100	300
B	4	50	8	40	200	400	160	320
C	6	20	9	30	120	180	180	270
D	8	10	6	20	80	60	160	120
E	10	10	5	20	100	50	200	100
					580	**930**	**800**	**1110**

(i) Laspeyre's Price Index : $P_{01}^{L} = \frac{\Sigma p_1 q_0}{\Sigma p_0 q_0} \text{ x } 100$

$= \frac{930}{580} \text{ x } 100 = 160.34$

(ii) Paasche's Price Index : $P_{01}^{P} = \frac{\Sigma p_1 q_1}{\Sigma p_0 q_1} \text{ x } 100$

$= \frac{1100}{800} \text{ x } 100 = 137.50$

(iii) Fisher's Index : $P_{01}^{F} = \sqrt{P_{01}^{L} \text{ x } P_{01}^{P}} = 148.48$

10.4.5 Test of adequacy for Index Number

Index Numbers are constructed to study the relative changes in prices, quantities, etc. of one time in comparison with another. Several formulae have been suggested for constructing index numbers and one should select the most appropriate one in a given situation. Following are the tests suggested for choosing an appropriate index.

1) Time reversal test

2) Factor reversal test

Time reversal test

It is a test to determine whether a given method will work both ways in time, forward and backward. When the data for any two years are treated by the same method, but with the bases reversed, the two index numbers secured should be reciprocals of each other so that their product is unity. Symbolically the following relation should be satisfied.

$$\therefore \; \mathbf{P_{01} \times P_{10} = 1}$$

(ignoring the factor 100 in each index) where P_{01} is the index for current period 1 on base period 0 and P_{10} is the index for the current period 0 on base period 1.

Factor reversal test

This test holds that the product of a price index and the quantity index should be equal to the corresponding value index. The test is that the change in price multiplied by the change in quantity should be equal to the total change in value.

$$\therefore \; \mathbf{P_{01} \times Q_{01}} = \frac{\Sigma p_1 q_1}{\Sigma p_0 q_0} \text{ (ignoring the factor 100 in each index)}$$

P_{01} gives the relative change in price and Q_{01} gives the relative change in quantity. The total value of a given commodity in a given year is the product of the quantity and the price per unit.

$\dfrac{\Sigma p_1 q_1}{\Sigma p_0 q_0}$ is the ratio of the total value in the current period to the total value in the base period and this ratio is called the **true value ratio.**

Fisher’s index is known as Ideal Index Number since it is the only index number that satisfies both reversal tests.

Example 23

Calculate Fisher's Ideal Index from the following data and verify that it satisfies both Time Reversal and Factor Reversal test

Commodity	**Price**		**Quantity**	
	1985	**1986**	**1985**	**1986**
A	**8**	**20**	**50**	**60**
B	**2**	**6**	**15**	**10**
C	**1**	**2**	**20**	**25**
D	**2**	**5**	**10**	**8**
E	**1**	**5**	**40**	**30**

Solution :

Commodity	1985		1986					
	p_0	q_0	p_1	q_1	p_1q_0	p_0q_0	p_1q_1	p_0q_1
A	8	50	20	60	1000	400	1200	480
B	2	15	6	10	90	30	60	20
C	1	20	2	25	40	20	50	25
D	2	10	5	8	50	20	40	16
E	1	40	5	30	200	40	150	30
					1380	**510**	**1500**	**571**

Fisher's Ideal Index $= \sqrt{\frac{\Sigma p_1 q_0}{\Sigma p_0 q_0} \times \frac{\Sigma p_1 q_1}{\Sigma p_0 q_1}} \times 100$

$= \sqrt{\frac{1380}{510} \times \frac{1500}{571}} \times 100$

$= 2.6661 \times 100 = 266.61$

Time reversal test

Test is satisfied when $P_{01} \times P_{10} = 1$

$$P_{01} = \sqrt{\frac{\Sigma p_1 q_0}{\Sigma p_0 q_0} \times \frac{\Sigma p_1 q_1}{\Sigma p_0 q_1}} = \sqrt{\frac{1380}{510} \times \frac{1500}{571}}$$

$$P_{10} = \sqrt{\frac{\Sigma p_0 q_1}{\Sigma p_1 q_1} \times \frac{\Sigma p_0 q_0}{\Sigma p_1 q_1}} = \sqrt{\frac{571}{1500} \times \frac{510}{1380}}$$

$$P_{01} \times P_{10} = \sqrt{\frac{1380}{510} \times \frac{1500}{571} \times \frac{571}{1500} \times \frac{510}{1380}}$$

$$= \sqrt{1} = 1$$

Hence Fisher's Ideal Index satisfies Time reversal test.

Factor reversal test

Test is satisfied when $P_{01} \times Q_{01} = \frac{\Sigma p_1 q_1}{\Sigma p_0 q_0}$

$$Q_{01} = \sqrt{\frac{\Sigma q_1 p_0}{\Sigma q_0 p_0} \times \frac{\Sigma q_1 p_1}{\Sigma q_0 p_1}} = \sqrt{\frac{571}{510} \times \frac{1500}{1380}}$$

$$\therefore \quad P_{01} \times Q_{01} = \sqrt{\frac{1380}{510} \times \frac{1500}{571} \times \frac{571}{510} \times \frac{1500}{1380}}$$

$$= \frac{1500}{510} = \frac{\Sigma p_1 q_1}{\Sigma p_0 q_0}$$

Hence Fisher's Ideal Index satisfies Factor reversal test.

Example 24

Compute Index Number using Fisher's formula and show that it satisfies time reversal test and factor reversal test.

Commodity	Base year		Current year	
	Price	Quantity	Price	Quantity
A	10	12	12	15
B	7	15	5	20
C	5	24	9	20
D	16	5	14	5

Solution :

Commodity	Base year		Current year					
	p_0	q_0	p_1	q_1	p_1q_0	p_0q_0	p_1q_1	p_0q_1
A	10	12	12	15	144	120	180	150
B	7	15	5	20	75	105	100	140
C	5	24	9	20	216	120	180	100
D	16	5	14	5	70	80	70	80
					505	**425**	**530**	**470**

$$\text{Fisher's Ideal Index} = \sqrt{\frac{\Sigma p_1q_0}{\Sigma p_0q_0} \times \frac{\Sigma p_1q_1}{\Sigma p_0q_1}} \times 100$$

$$= \sqrt{\frac{505}{425} \times \frac{530}{470}} \times 100 = 115.75$$

Time reversal test

Test is satisfied when $P_{01} \times P_{10} = 1$

$$P_{01} = \sqrt{\frac{\Sigma p_1q_0}{\Sigma p_0q_0} \times \frac{\Sigma p_1q_1}{\Sigma p_0q_1}} = \sqrt{\frac{505}{425} \times \frac{530}{470}}$$

$$P_{10} = \sqrt{\frac{\Sigma p_0q_1}{\Sigma p_1q_1} \times \frac{\Sigma p_0q_0}{\Sigma p_1q_0}} = \sqrt{\frac{470}{530} \times \frac{425}{505}}$$

$$P_{01} \times P_{10} = \sqrt{\frac{505}{425} \times \frac{530}{470} \times \frac{470}{530} \times \frac{425}{505}}$$

$$= \sqrt{1} = 1$$

Hence Fisher's Ideal Index satisfies Time Reversal Test.

Factor reversal test

Test is satisfied when $P_{01} \times Q_{01} = \frac{\Sigma p_1q_1}{\Sigma p_0q_0}$

$$Q_{01} = \sqrt{\frac{\Sigma q_1 p_0}{\Sigma q_0 p_0} \times \frac{\Sigma q_1 p_1}{\Sigma q_0 p_1}} = \sqrt{\frac{470}{530} \times \frac{425}{505}}$$

$$P_{01} \times Q_{01} = \sqrt{\frac{505}{425} \times \frac{530}{470} \times \frac{470}{425} \times \frac{530}{505}}$$

$$= \frac{530}{425} = \frac{\Sigma p_1 q_1}{\Sigma p_0 q_0}$$

Hence Fisher's Ideal Index satisfies Factor reversal test.

10.4.6 Cost of Living Index (CLI)

Cost of living index numbers are generally designed to represent the average change over time in the prices paid by the ultimate consumer for a specified quantity of goods and services. Cost of living index number is also known as **Consumer price index number**

It is well known that a given change in the level of prices (retail) affects the cost of living of different classes of people in different manners. The general index number fails to reveal this. Therefore it is essential to construct a cost of living index number which helps us in determining the effect of rise and fall in prices on different classes of consumers living in different areas. It is to be noted that the demand for a higher wage is based on the cost of living index. The wages and salaries in most countries are adjusted in accordance with the cost of living index.

10.4.7 Methods of constructing cost of living index

Cost of living index may be constructed by the following methods.

(i) Aggregate expenditure method or weighted aggregative method

(ii) Family budget method

Aggregate expenditure method

In this method, the quantities of commodities consumed by the particular group in the base year are used as the weights. On the basis of these weights, aggregate (total) expenditure in current year and base year are calculated and the percentage of change is worked out .

$\therefore$ Cost of Living Index (C.L.I) $= \dfrac{\Sigma p_1 q_0}{\Sigma p_0 q_0} \times 100$

This method is the most popular method for constructing cost of living index and the method is same as Laspeyre's price index.

Family budget method

In this method, the value weights obtained by multiplying prices by quantities consumed (i.e. p_0q_0) are taken as weights. To get the cost of living index, find the sum of respective products of price relatives and value weights and then divide this sum by the sum of the value weights.

$\therefore$ Cost of living Index = $\dfrac{\Sigma PV}{\Sigma V}$ where

P = $\dfrac{p_1}{p_0} \times 100$ is the price relative and

V = $p_0 q_0$ is the value weight for each item.

This method is same as the Weighted average of price relative method.

10.4.8 Uses of cost of living index number

(i) The cost of living index number is mainly used in wage negotiations and wage contracts.

(ii) It is used to calculate the dearness allowance for the employees.

Example 25

Calculate the cost of living index by aggregate expenditure method

Commodity	**Quantity 2000**	**Price (Rs) 2000**	**Price (Rs) 2003**
A	**100**	**8**	**12.00**
B	**25**	**6**	**7.50**
C	**10**	**5**	**5.25**
D	**20**	**48**	**52.00**
E	**65**	**15**	**16.50**
F	**30**	**19**	**27.00**

Solution :

Commodity	Quantity 2000 q_0	Price 2000 p_0	Price 2003 p_1	p_1q_0	p_0q_0
A	100	8	12.00	1200.00	800
B	25	6	7.50	187.50	150
C	10	5	5.25	52.50	50
D	20	48	52.00	1040.00	960
E	65	15	16.50	1072.50	975
F	30	19	27.00	810.00	570
				4362.50	**3505**

$$\text{C. L. I} = \frac{\Sigma p_1q_0}{\Sigma p_0q_0} \times 100$$

$$= \frac{4362.50}{3505} \times 100 = 124.46$$

Example 26

Construct the cost of living Index Number for 2003 on the basis of 2000 from the following data using family Budget method.

Items	**Price**		**Weights**
	2000	**2003**	
Food	**200**	**280**	**30**
Rent	**100**	**200**	**20**
Clothing	**150**	**120**	**20**
Fuel & lighting	**50**	**100**	**10**
Miscellaneous	**100**	**200**	**20**

Solution :

Calculation of CLI by family budget method

Items	p_0	p_1	weights V	$P=\frac{p_1}{p_0}x100$	PV
Food	200	280	30	140	4200
Rent	100	200	20	200	4000
Clothing	150	120	20	80	1600
Fuel & Lighting	50	100	10	200	2000
Misc.	100	200	20	200	4000
			100		**15800**

$$\text{Cost of living index} = \frac{\Sigma PV}{\Sigma V} = \frac{15800}{100} = 158$$

Hence, there is 58% increase in cost of living in 1986 compared to 1980.

EXERCISE 10.4

1) Compute (i) Laspeyre's (ii) Paasche's and (iii) Fisher's index Numbers

Commodity	Price		Quantity	
	Base year	Current year	Base year	Current year
A	6	10	50	50
B	2	2	100	120
C	4	6	60	60
D	10	12	30	25

2) Construct the price index number from the following data by applying
(i) Laspeyre's (ii) Paasche's (iii) Fisher's method

Commodity	1999		1998	
	Price	Quantity	Price	Quantity
A	4	6	2	8
B	6	5	5	10
C	5	10	4	14
D	2	13	2	19

3) Compute (a) Laspyre's (b) Paasche's (c) Fisher's method of index numbers for 1990 from the following :

Commodity	Price		Quantity	
	1980	1990	1980	1990
A	2	4	8	6
B	5	6	10	5
C	4	5	14	10
D	2	2	19	13

4) From the following data calculate the price index number by
(a) Laspeyre's method (b) paasche's method
(c) Fisher's method

Commodity	Base year		Current year	
	Price	Quantity	Price	Quantity
A	5	25	6	30
B	10	5	15	4
C	3	40	2	50
D	6	30	8	35

5) Using the following data, construct Fisher's Ideal index and show that it satisfies Factor Reversal test and Time Reversal test.

Commodity	Price		Quantity	
	Base year	Current year	Base year	Current year
A	6	10	50	56
B	2	2	100	120
C	4	6	60	60
D	10	12	30	24
E	8	12	40	36

6) Calculate Fisher's Ideal Index from the following data and show how it satisfies time reversal test and factor reversal test.

Commodity	Base year (1997)		Current year (1998)	
	Price	Quantity	Price	Quantity
A	10	10	12	8
B	8	12	8	13
C	12	12	15	8
D	20	15	25	10
E	5	8	8	8
F	2	10	4	6

7) Construct cost of living index for 2000 taking 1999 as the base year from the following data using Aggregate Expenditure method.

Commodity	Quantity (kg.)	Price	
	1999	1999	2000
A	6	5.75	6.00
B	1	5.00	8.00
C	6	6.00	9.00
D	4	8.00	10.00
E	2	2.00	1.80
F	1	20.00	15.00

8) Calculate the cost of living Index Number using Family Budget method

Commodity	A	B	C	D	E	F	G	H
Quantity in Base year (unit)	20	50	50	20	40	50	60	40
Price in Base year (Rs.)	10	30	40	200	25	100	20	150
Price in current year (Rs)	12	35	50	300	50	150	25	180

9) Calculate the cost of living index number using Family Budget method for the following data taking the base year as 1995

Commodity	Weight	Price (per unit)	
		1995	1996
A	40	16.00	20.00
B	25	40.00	60.00
C	5	0.50	0.50
D	20	5.12	6.25
E	10	2.00	1.50

10) From the data given below, construct a cost of living index number by family budget method for 1986 with 1976 as the base year.

Commodity	P	Q	R	S	T	U
Quantity in 1976 Base year	50	25	10	20	30	40
Price per unit in 1976 (Rs.)	10	5	8	7	9	6
Price per unit in 1986 (Rs.)	6	4	3	8	10	12

10.5 STATISTICAL QUALITY CONTROL (SQC)

Every product manufactured is required for a specific purpose. It means that if the product meets the specifications required for its rightful use, it is of good quality and if not, then the quality of the product is considered to be poor.

It is a well known fact that all repetitive process no matter how carefully arranged are not exactly identical and contain some variability. Even in the manufacture of commodities by highly specialised machines it is not unusual to come across differences between various units of production. For example, in the manufacture of corks, bottles etc. eventhough highly efficient machines are used some difference may be noticed in various units. If the differences are not much, it can be ignored and the product can be passed off as within specifications. But if it is beyond certain limits, the article has to be rejected and the cause of such variation has to be investigated.

10.5.1 Causes for variation

The variation occurs due to two types of causes namely (i) Chance causes (ii) Assignable causes

(i) Chance causes

If the variation occurs due to some inherent pattern of variation and no causes can be assigned to it, it is called **chance** or **random variation.** Chance Variation is tolerable, permissible inevitable and does not materially affect the quality of a product.

(ii) Assignable causes

The causes due to faulty process and procedure are known as assignable causes. the variation due to assignable causes is of non-random nature. Chance causes cannot detected. However assignable causes can be detected and corrected.

10.5.2 Role and advantages of SQC

The role of statistical quality control is to collect and analyse relevant data for the purpose of detecting whether the process is under control or not.

The value of quality control lies in the fact that assignable causes in a process can be quickly detected. Infact the variations are often discovered before the product becomes defective.

SQC is a well accepted and widespread process on the basis of which it is possible to understand the principles and techniques by which decisions are made based on variation.

Statistical quality control is only diagonstic. It tells us whether the standard is being maintained or not. The remedial action rests with the technicians who employ techniques for the maintenance of uniform quality in a continuous flow of manufactured products.

The purpose for which SQC are used are two fold namely, (a) **Process control,** (b) **Product control.**

In process control an attempt is made to find out if a particular process is within control or not. Process control helps in studying the future performance.

10.5.3 Process and Product control

The main objective in any production process is to control and maintain quality of the manufactured product so that it conforms to specified quality standards. In otherwords, we want to ensure that the proportion of defective items in the manufactured product is not too large. This is called **process control** and is achieved through the technique of control charts.

On the otherhand, by **product control** we mean controlling the quality of the product by critical examination at strategic points and this is achieved through **product control plans** pioneered by Dodge and Romig. Product control aims at guaranteeing a certain quality level to the consumer regardless of what quality level is being maintained by the producer. In otherwords, it attempts to ensure that the product marketed by the sale department does not contain a large number of defective items.

10.5.4 Control Charts

The statistical tool applied in process control is the **control chart.** Control charts are the devices to describe the patterns of variation. The control charts were developed by the physicist, Walter A. Stewart of Bell Telephone Company in 1924. He suggested that the control chart may serve, first to define the goal or standard for the process that the management might strive to attain. Secondly, it may be used as an instrument to attain that goal and thirdly, it may serve as a means of judging whether the goal is being achieved. Thus, control chart is an instrument to be used in specification, production and inspection and is the core of statistical quality control.

A control chart is essentially a graphic device for presenting data so as to directly reveal the frequency and extent of variations from established standards or goals. Control charts are simple to construct and easy to interpret and they tell the manager at a glance whether or not the process is in control, i.e. within the tolerance limits.

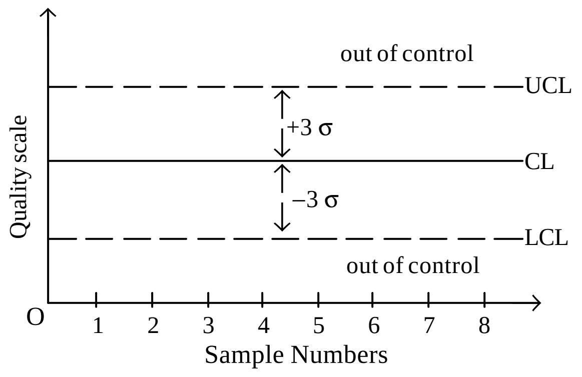
In general a control chart consists of three horizontal lines

(i) A central line to indicate the desired standard or level of the process (CL)

(ii) Upper control limit (UCL) and

(iii) Lower control limit (LCL)

Outline of a control chart



(Fig. 10.9)

From time to time a sample is taken and the data are plotted on the graph. If all the sample points fall within the upper and lower control limits, it is asumed that the process is “in control” and only chance causes are present. When a sample point falls outside the control limits, it is assumed that variations are due to assignable causes.

Types of Control Charts

Broadly speaking, control charts can be divided under two heads.

(i) Control charts of Variables

(ii) Control charts of Attributes

Control charts of variables concern with measurable data on quality characteristics which are usually continous in nature. Such type of data utilises $\overline{X}$ and R chart.

Control charts of attirbutes, namely *c*, *np* and *p* charts concern with the data on quality characteristics, which are not amenable to measurement or attributes (prodcut defective or non defective)

In this chapter, we consider only the control charts of variables, namely $\overline{X}$ **chart** and **R chart.**

R-Chart (Range chart)

The R chart is used to show the variability or dispersion of the quality produced by a given process. R chart is the companion chart to the $\overline{X}$ chart and both are usually required for adequate analysis of the production process under study. The R chart is generally presented along with the $\overline{X}$ chart. The general procedure for constructing the R chart is similar to that for the $\overline{X}$ chart. The required values for constructing the R chart are :

(i) The range of each sample, R.

(ii) The Mean of the sample ranges, $\overline{R}$

(iii) The control limits are set at

$$\text{U.C.L} = D_4\overline{R}$$

$$\text{L.C.L} = D_3\overline{R}$$

The values of D_4 and D_3 can be obtained from tables.

$\overline{X}$ Chart

The $\overline{X}$ chart is used to show the quality averages of the samples drawn from a given process. The following values must first be computed before an $\overline{X}$ chart is constructed.

1) Obtain the mean of each sample $\overline{X}_i$: i = 1, 2 ... *n*

2) Obtain the mean of the sample means

i.e $\overline{\overline{X}} = \dfrac{\overline{X}_1 + \overline{X}_2 + \ldots + \overline{X}_n}{n}$

where n is the total number of observations

3) The control limits are set at

U.C.L. = $\overline{\overline{X}} + A_2\overline{R}$

LCL = $\overline{\overline{X}} - A_2\overline{R}$, where $\overline{R} = \dfrac{\sum_{i=1}^{n} R_i}{n}$, where R_i are the sample ranges.

The values of A_2 for different n can be obtained from the tables.

Example 28

The following data relate to the life (in hours) of 10 samples of 6 electric bulbs each drawn at an interval of one hour from a production process. Draw the control chart for $\overline{X}$ and R and comment.

Sample No.	Life time (in hours)					
1	620	687	666	689	738	686
2	501	585	524	585	653	668
3	673	701	686	567	619	660
4	646	626	572	628	631	743
5	494	984	659	643	660	640
6	634	755	625	582	683	555
7	619	710	664	693	770	534
8	630	723	614	535	550	570
9	482	791	533	612	497	499
10	706	524	626	503	661	754

(Given for n = 6, A_2 = 0.483, D_3 = 0, D_4 = 2.004)

Solution :

Sample No.	Total	Sample Mean $\overline{X}$	Sample Range R
1	4086	681	118
2	3516	586	167
3	3906	651	134
4	3846	641	171
5	4080	680	490
6	3834	639	200
7	3990	665	236
8	3622	604	188
9	3414	569	309
10	3774	629	251
Total		**6345**	**2264**

Central line $\overline{\overline{X}}$ = mean of the sample means = 634.5

$\overline{R}$ = mean of the sample ranges = 226.4

U.C.L. $= \overline{\overline{X}} + A_2\overline{R}$

$= 634.5 + 0.483 \times 226.4$

$= 634.5 + 109.35 = 743.85$

L.C.L. $= \overline{\overline{X}} - A_2\overline{R}$

$= 634.5 - 0.483 \times 226.4$

$= 634.5 - 109.35 = 525.15$

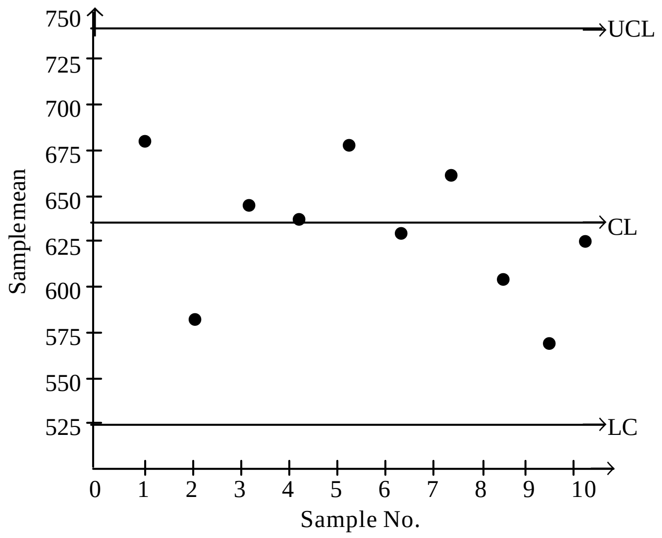
Central line $\overline{R} = 226.4$

U.C.L. $= D_4\overline{R} = 2.004 \times 226.4$

$= 453.7056$

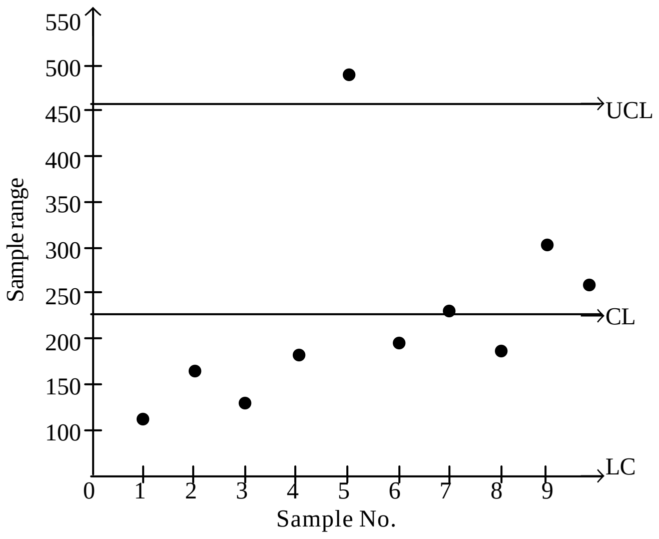
L.C.L. $= D_3\overline{R} = 0 \times 226.4 = 0$

$\overline{X}$ Chart



(Fig. 10.10)

R Chart



(Fig. 10.11)

Conclusion :

Since one of the points of the sample range is outside the UCL of R chart, the process is not in control.

Example 29

The following data shows the value of sample mean $\overline{X}$ and the range R for ten samples of size 5 each. Calculate the values for central line and control limits for mean chart and range chart and determine whether the process is in control

Sample no.	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Mean $\overline{X}$	**11.2**	**11.8**	**10.8**	**11.6**	**11.0**	**9.6**	**10.4**	**9.6**	**10.6**	**10.0**
Range (R)	**7**	**4**	**8**	**5**	**7**	**4**	**8**	**4**	**7**	**9**

(Given for *n* = 5, A_2 = 0.577, D_3 = 0 D_4 = 2.115)

Solution :

Control limits for $\overline{X}$ chart

$$\overline{\overline{X}} = \frac{1}{n}\Sigma\overline{X}$$

$$= \frac{1}{10}(11.2 + 11.8 + 10.8 + ...+ 10.0) = 10.66$$

$$\overline{R} = \frac{1}{n}\Sigma R = \frac{1}{10}(63) = 6.3$$

$$\text{U.C.L} = \overline{\overline{X}} + A_2\overline{R}$$

$$= 10.66 + (0.577 \text{ x } 6.3) = 14.295$$

$$\text{L.C.L.} = \overline{\overline{X}} - A_2\overline{R}$$

$$= 10.66 - (0.577 \text{ x } 6.3) = 7.025$$

CL = Central line = $\overline{\overline{X}}$ = 10.66

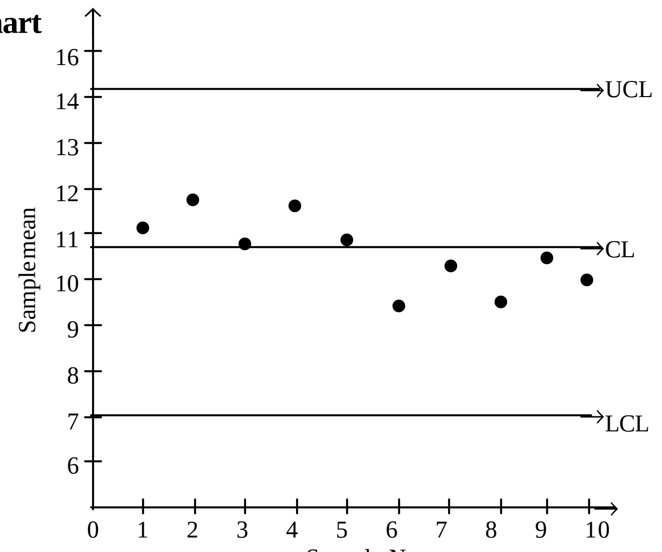
Range chart

$$\text{U.C.L.} = D_4\overline{R} = 2.115 \text{ x } 6.3 = 13.324$$

$$\text{L.C.L.} = D_3\overline{R} = 0$$

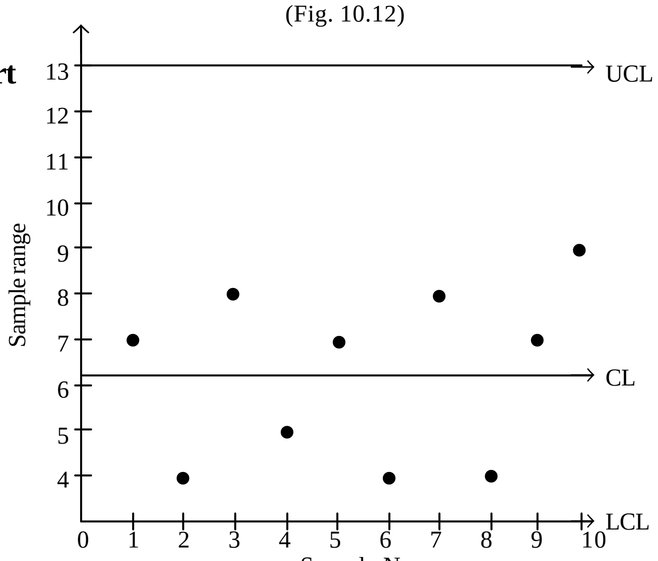
$$\text{C.L.} = \overline{R} = 6.3$$

$\overline{X}$ Chart



(Fig. 10.12)

R Chart



(Fig. 10.13)

Conclusion :

Since all the points of sample mean and range are within the control limits, the process is in control.

EXERCISE 10.5

1) The following are the $\overline{X}$ and R values for 20 samples of 5 readings. Draw $\overline{X}$ chart and R chart and write your conclusion.

Samples	1	2	3	4	5	6	7	8	9	10
$\overline{X}$	34	31.6	30.8	33	35	33.2	33	32.6	33.8	37.8
R	4	4	2	3	5	2	5	13	19	6

Samples	11	12	13	14	15	16	17	18	19	20
$\overline{X}$	35.8	38.4	34	35	38.8	31.6	33	28.2	31.8	35.6
R	4	4	14	4	7	5	5	3	9	6

(Given for $n = 5$, $A_2 = 0.58$, $D_3 = 0$ $D_4 = 2.12$)

2) From the following, draw $\overline{X}$ and R chart and write your conclusion.

Sample no.	1	2	3	4	5	6	7	8	9	10
	140	138	139	143	142	136	142	143	141	142
	143	143	133	141	142	144	147	137	142	137
	137	143	147	137	145	143	137	145	147	145
	134	145	148	138	135	136	142	137	140	140
	135	146	139	140	136	137	138	138	140	132

Sample no.	11	12	13	14	15	16	17	18	19	20
	137	137	142	137	144	140	137	137	142	136
	147	146	142	145	142	132	137	142	142	142
	142	142	139	144	143	144	142	142	143	140
	137	142	141	137	135	145	143	145	140	139
	135	140	142	140	144	141	141	143	135	137

(Given for $n = 5$, $A_2 = 0.58$, $D_3 = 0$, $D_4 = 2.12$)

3) From the following data construct $\overline{X}$ and R chart and write your conclusion

Sample no.	1	2	3	4	5	6	7	8	9
	46	41	43	37	37	37	44	35	37
	40	42	40	40	40	38	39	39	44
	48	49	46	47	46	49	43	48	48

Sample no.	10	11	12	13	14	15	16	17	18
	45	48	36	40	42	38	47	42	47
	43	44	42	39	40	40	44	45	42
	49	48	48	48	48	48	49	37	49

(Given for $n = 3$, $A_2 = 1.02$, $D_3 = 0$, $D_4 = 2.58$)

EXERCISE 10.6

Choose the correct answer

1) A time series is a set of data recorded
   (a) periodically (b) at equal time intervals
   (c) at successive points of time (d) all the above

2) A time series consists of
   (a) two components (b) three components
   (c) four components (d) none of these

3) The component of a time series attached to long term variation is termed as
   (a) cyclic variations (b) secular trend
   (c) irregular variation (d) all the above

4) The component of a time series which is attached to short term fluctuations is
   (a) seasonal variation (b) cyclic variation
   (c) irregular variation (d) all the above

5) Cyclic variations in a time series are casued by
   (a) lock out in a factory (b) war in a country
   (c) floods in the states (d) none of these

6) The terms prosperity, recession depression and recovery are in particular attached to

(a) Secular trend (b) seasonal fluctuation

(c) cyclic movements (d) irregular variation

7) An additive model of time series with the components T, S, C and I is

(a) $Y = T + S + C - I$ (b) $Y = T + S \times C + I$

(c) $Y = T + S + C + I$ (d) $Y = T + S + C \times I$

8) A decline in the sales of ice cream during November to March is associated with

(a) Seasonal variation (b) cyclical variation

(c) random variation (d) secular trend

9) Index number is a

(a) measure of relative changes

(b) a special type of an average

(c) a percentage relative

(d) all the above.

10) Index numbers are expressed

(a) in percentages (b) in ratios

(c) in terms of absolute value (d) all the above

11) Most commonly used index numbers are

(a) Diffusion index number (b) price index number

(c) value index number (d) none of these

12) Most frequently used index number formulae are

(a) weighted formulae (b) Unweighted formulae

(c) fixed weighted formulae (d) none of these

13) Laspeyre's index formula uses the weights of the

(a) base year quantities (b) current year prices

(c) average of the weights of number of years

(d) none of these

14) The weights used in Paasche's formula belong to

(a) the base period (b) the current period

(c) to any arbitrary chosen period (d) none of these

15) Variation in the items produced in a factory may be due to

(a) chance causes (b) assignable causes

(c) both (a) and (b) (d) neither (a) or (b)

16) Chance variation in the manufactured product is

(a) controlable (b) not controlable

(c) both (a) and (b) (d) none of these

17) The causes leading to vast variation in the specification of a product are usually due to

(a) random process (b) assignable causes

(c) non-traceable causes (d) all the above

18) Variation due to assignable causes in the product occur due to

(a) faulty process (b) carelessness of operators

(c) poor quality of raw material (d) all the above

19) Control charts in statistical quality consist of

(a) three control lines

(b) upper and lower control limits

(c) the level of process

(d) all the above

20) The range of correlation co-efficient is

(a) 0 to ∞ (b) $-\infty$ 10 ∞

(c) -1 to 1 (d) none of these

21) If X and Y are two variates, there can be atmost

(a) one regression line (b) two regression lines

(c) three regression lines (d) none of these

22) In a regression line of Y on X, the variable X is known as

(a) independent variable (b) dependent variable

(c) both (a) and (b) (d) none of these

23) Scatter diagram of the variate values (X, Y) give the idea about

(a) functional relationship (b) regression model

(c) distribution of errors (d) none of these

24) The lines of regression intersect at the point

(a) (X, Y) (b) $(\overline{X}, \overline{Y})$

(c) (0, 0) (d) none of these

25) The term regression was introduced by

(a) R.A.Fisher (b) Sir Francis Galton

(c) Karl pearson (d) none of these.

Discipline Course-I
Semester -I
Paper: Mathematical PhysicsI IA
Lesson: The D Operator & the Non-Homogeneous Equation
Lesson Developer: Savinder Kaur
College/Department: SGTB Khalsa College, University of Delhi

Table of Contents

Learning Objective

The student evolves further to calculate the solution of the Non-Homogeneous DE by finding the

- **Particular Integral for Special Forms of the Function $f(x)$ in the Non-Homogeneous DE**
- **rules to determine PI in shorter steps and learns the D-Operator**
  - **When function $f(x)$ is of the form e^{ax}**
  - **When function $f(x)$ is of the form $\sin ax$ or $\cos ax$**
  - **When function $f(x)$ is of the form x^m, m being a positive integer**
  - **When function $f(x)$ is of the form $e^{ax}V(x)$**

The D Operator & the Non-Homogeneous Equation

9.1 Particular Integral of Special Forms of the Function f(x)

As the previous two examples may have suggested finding PI could be very difficult involving tedious integrations. However, there are certain special forms of the function $f(x)$ which admits rules for finding PI in shorter steps. We would explore such functions and show our confidence in the rules developed;

9.2 *When function $f(x)$ is of the form e^{ax}*

If $f(x) = e^{ax}$ then we can see that

$$De^{ax} = ae^{ax}$$
$$D^2e^{ax} = D(De^{ax}) = D(ae^{ax}) = a^2e^{ax}$$

and so on

$$D^ne^{ax} = a^ne^{ax}$$

So if

$$L(D) = a_nD^n + a_{n-1}D^{n-1} + \cdots + a_1D + a_0$$

then

$$L(D)e^{ax} = (a_nD^n + a_{n-1}D^{n-1} + \cdots + a_1D + a_0)e^{ax}$$
$$L(D)e^{ax} = (a_na^n + a_{n-1}a^{n-1} + \cdots + a_1a + a_0)e^{ax}$$
$$L(D)e^{ax} = L(a)e^{ax}$$

Thus, operating on both sides by the "*inverse"* operator $\frac{1}{L(D)}$ we find that

$$\frac{1}{L(D)}L(D)e^{ax} = \frac{1}{L(D)}L(a)e^{ax}$$
$$e^{ax} = L(a)\frac{1}{L(D)}e^{ax}$$

Now if $L(a) \neq 0$ this can be interpreted as

$$\frac{1}{L(D)}e^{ax} = \frac{e^{ax}}{L(a)}$$

This beautiful result then states a rule that an n^{th} order Non-Homogeneous Linear DE with Constant coefficients

$$L(D)y = Ae^{ax}$$

has the PI

$$y = A\frac{e^{ax}}{L(a)}$$

which needs no integration to be performed.

There may arise a situation where $L(a) = 0$. This would then imply "a" to be an r^{th} *order* *root* of the n^{th} *order Non-Homogeneous Linear DE with Constant coefficients so that*

$$L(D) = (D-a)^r \varphi(D).$$

The DE can then be written as

$$\varphi(D)(D-a)^r y = Ae^{ax}$$

Operating on both sides by the "*inverse*" operator $\frac{1}{\varphi(D)}$ we find

$$\frac{1}{\varphi(D)}\varphi(D)(D-a)^r y = \frac{1}{\varphi(D)}Ae^{ax}$$
$$(D-a)^r y = A\frac{e^{ax}}{\varphi(a)}$$

From our previously learnt technique this yields

$$y = \frac{A}{\varphi(a)}\frac{x^r}{r!}e^{ax}$$

Example 9.2.1 Solve the equation

$$\boldsymbol{y'' + y' + y = e^{-x}}$$

Solution:
Step 1 The *DE* will be written with the D operator by replacing $y'' \to D^2y$ & $y' \to Dy$

$$(D^2+D+1)y = e^{-x}$$
$$L(D)y = e^{-x}$$

Step 2 The *Auxiliary Equation* for the corresponding homogeneous *DE*

$$L(D)y = 0$$

will be obtained by writing

$$L(\lambda) = 0$$
$$\lambda^2+\lambda+1 = 0$$
$$\lambda_1 = \frac{-1+\sqrt{1^2-4}}{2} \ \& \ \lambda_2 = \frac{-1-\sqrt{1^2-4}}{2}$$
$$\lambda_1 = \frac{-1+\sqrt{-3}}{2} \ \& \ \lambda_2 = \frac{-1-\sqrt{-3}}{2}$$

The roots are then found as

$$\lambda_1 = -\frac{1}{2}+i\frac{\sqrt{3}}{2} \ \& \ \lambda_2 = -\frac{1}{2}-i\frac{\sqrt{3}}{2}$$

The CF would be $C_1e^{\left(-\frac{1}{2}+i\frac{\sqrt{3}}{2}\right)x} + C_2e^{\left(-\frac{1}{2}-i\frac{\sqrt{3}}{2}\right)x}$ which can be represented as

$$CF = e^{-\frac{x}{2}}\left\{C_1\cos\left(\frac{\sqrt{3}}{2}x\right) + C_2\sin\left(\frac{\sqrt{3}}{2}x\right)\right\}$$

Step 3 The PI would now be obtained as

$$PI = \frac{1}{L(D)} e^{-x}$$

Since $f(x) = e^{-x}$ is an exponential function we will use the rule $\frac{1}{L(D)} e^{ax} = \frac{e^{ax}}{L(a)}$ to find the PI

$$PI = \frac{1}{L(D)} e^{-x} = \frac{e^{-x}}{L(-1)}$$

$$PI = \frac{e^{-x}}{\{(-1)^2 + (-1) + 1\}}$$

$$PI = \frac{e^{-x}}{\{1 + (-1) + 1\}} = e^{-x}$$

Step 4 The *General Solution* would therefore be

$$y = CF + PI = e^{-\frac{x}{2}} \left\{ C_1 \cos\left(\frac{\sqrt{3}}{2} x\right) + C_2 \sin\left(\frac{\sqrt{3}}{2} x\right) \right\} + e^{-x}$$

Example 9.2.2 Solve the equation

$$\boldsymbol{y'' - 4y' + 4y = e^x}$$

Solution:

Step 1 The *DE* will be written with the D operator by replacing $y'' \to D^2 y$ & $y' \to Dy$

$$(D^2 - 4D + 4)y = e^x$$

$$L(D)y = e^x$$

Step 2 The *Auxiliary Equation* for the corresponding homogeneous *DE*

$$L(D)y = 0$$

will be obtained by writing

$$L(\lambda) = 0$$

$$\lambda^2 - 4\lambda + 4 = 0$$

$$\lambda_1 = \frac{-(-4) + \sqrt{(-4)^2 - 4 \times 4}}{2} \ \& \ \lambda_2 = \frac{-(-4) - \sqrt{(-4)^2 - 4 \times 4}}{2}$$

$$\lambda_1 = 2 \ \& \ \lambda_2 = 2$$

The roots are then found to a double root

$$\lambda_1 = \lambda_2 = 2$$

The CF would be

$$CF = (C_1 x + C_2) e^{2x}$$

Step 3 The PI would now be obtained as

$$PI = \frac{1}{L(D)} e^x$$

Since $f(x) = e^{-x}$ is an exponential function we will use the rule $\frac{1}{L(D)}e^{ax} = \frac{e^{ax}}{L(a)}$ to find the PI

$$PI = \frac{1}{L(D)}e^{x} = \frac{e^{x}}{L(1)}$$
$$PI = \frac{e^{-x}}{\{1^2 - 4(1) + 4\}}$$

$$PI = e^{x}$$

Step 4 The *General Solution* would therefore be

$$y = CF + PI = (C_1x + C_2)e^{2x} + e^{x}$$

9.3 *When function $f(x)$ is of the form $\sin ax$ or $\cos ax$*

If $f(x) = \sin(ax + \theta)$ then we can see that

$$D\sin(ax+\theta) = a\cos(ax+\theta)$$
$$D^2\sin(ax+\theta) = D(D\sin(ax+\theta)) = D(a\cos(ax+\theta)) = -a^2\sin(ax+\theta)$$
$$D^3\sin(ax+\theta) = -a^3\cos(ax+\theta)$$
$$D^4\sin(ax+\theta) = a^4\cos(ax+\theta) = (-a^2)^2\cos(ax+\theta)$$

and so on
$$(D^2)^n\sin(ax+\theta) = (-a^2)^n\sin(ax+\theta)$$

So if
$$L(D) = a_nD^{2n} + \cdots + a_2D^4 + a_1D^2 + a_0$$
contains only even powers of the operator D then it can be seen as polynomial φ in D^2 of power n so that

$$\varphi(D^2)\sin(ax+\theta) = \{a_n(D^2)^n + \cdots + a_2(D^2)^2 + a_1(D^2) + a_0\}\sin(ax+\theta)$$
$$\varphi(D^2)\sin(ax+\theta) = \{a_n(-a^2)^n + \cdots + a_2(-a^2)^2 + a_1(-a^2) + a_0\}\sin(ax+\theta)$$
$$\varphi(D^2)\sin(ax+\theta) = \varphi(-a^2)\sin(ax+\theta)$$

Thus, operating on both sides by the "*inverse*" operator $\frac{1}{\varphi(D^2)}$ we find that

$$\frac{1}{\varphi(D^2)}\varphi(D^2)\sin(ax+\theta) = \frac{1}{\varphi(D^2)}\varphi(-a^2)\sin(ax+\theta)$$
$$\sin(ax+\theta) = \varphi(-a^2)\frac{1}{\varphi(D^2)}\sin(ax+\theta)$$

Now if $\varphi(-a^2) \neq 0$ this can be interpreted as

$$\frac{1}{\varphi(D^2)}\sin(ax+\theta) = \frac{1}{\varphi(-a^2)}\sin(ax+\theta)$$

This beautiful result then states a rule that an n^{th} order Non-Homogeneous Linear DE *with Constant coefficients*

$$L(D^2)y = A\sin(ax+\theta)$$

has the PI

$$y = A\frac{\sin(ax+\theta)}{L(-a^2)}$$

which needs no integration to be performed.

There may arise a situation where $L(-a^2)=0$. This would then imply "$-a^2$" to be an r^{th} *order root* of the *DE so that*

$$L(D^2) = (D^2+a^2)^r\varphi(D^2).$$

The DE can then be written as

$$\varphi(D^2)(D^2+a^2)^r y = A\sin(ax+\theta)$$

Operating on both sides by the "*inverse*" operator $\frac{1}{\varphi(D^2)}$ we find

$$\frac{1}{\varphi(D^2)}\varphi(D^2)(D^2+a^2)^r y = \frac{1}{\varphi(D^2)}A\sin(ax+\theta)$$

$$(D^2+a^2)^r y = A\frac{\sin(ax+\theta)}{\varphi(-a^2)}$$

From our previously learnt technique this yields

$$y = \frac{A}{\varphi(-a^2)}\frac{1}{(D^2+a^2)^r}\sin(ax+\theta)$$

There may also arise a situation wherein the DE contains odd powers of D too. This would then imply

$$\begin{aligned}(a_nD^n + a_{n-1}D^{n-1} + \cdots + a_4D^4 + a_3D^3 + a_2D^2 + a_1D + a_0)\sin(ax+\theta)\\ = [a_nD^n + \cdots + a_4(-a^2)^2 + a_3D^3 + a_2(-a^2) + a_1D + a_0]\sin(ax+\theta)\end{aligned}$$

$$L(D)\sin(ax+\theta) = \varphi(D)\sin(ax+\theta)$$

Operating on both sides by the "*inverse*" operator $\frac{1}{\varphi(D^2)}$ we find

$$\frac{1}{\varphi(D^2)}\varphi(D^2)(D^2+a^2)^r y = \frac{1}{\varphi(D^2)}A\sin(ax+\theta)$$

$$(D^2+a^2)^r y = A\frac{\sin(ax+\theta)}{\varphi(-a^2)}$$

From our previously learnt technique this yields

$$y = \frac{A}{\varphi(-a^2)}\frac{1}{(D^2+a^2)^r}\sin(ax+\theta)$$

Example 9.3.1 Solve the equation change

$$\boldsymbol{y'' + 4y = \cos 3x}$$

Solution:
Step 1 The *DE* will be written with the D operator by replacing $y'' \to D^2y$ & $y' \to Dy$

$$(D^2 + 4)y = \cos 3x$$
$$L(D)y = \cos 3x$$

Step 2 The *Auxiliary Equation* for the corresponding homogeneous *DE*

$$L(D)y = 0$$

will be obtained by writing

$$L(\lambda) = 0$$
$$\lambda^2 + 4 = 0$$
$$\lambda^2 = -4$$

The roots are then found as

$$\lambda_1 = +i2 \;\&\; \lambda_2 = -i2$$

The CF would be $C_1e^{i2x} + C_2e^{-i2x}$ which can be represented as

$$CF = C_1 \cos 2x + C_2 \sin 2x$$

Step 3 The PI would now be obtained as

$$PI = \frac{1}{L(D)} \cos 3x$$

Since $f(x) = \cos(ax)$ is an exponential function we will use the rule $\frac{1}{L(D^2)}\cos(ax) = \frac{1}{L(-a^2)}\cos(ax)$ to find the PI

$$PI = \frac{1}{(D^2 + 4)} \cos 3x$$
$$PI = \frac{\cos 3x}{\{(-3^2) + 4\}}$$
$$PI = \frac{\cos 3x}{\{-9 + 4\}}$$
$$PI = -\frac{1}{5} \cos 3x$$

Step 4 The *General Solution* would therefore be

$$y = CF + PI = C_1 \cos 2x + C_2 \sin 2x - \frac{1}{5} \cos 3x$$

Example 9.3.2 Solve the equation

$$\boldsymbol{y'' + 2n \cos \alpha\, y' + n^2 y = \sin nx}$$

Solution:
Step 1 The *DE* will be written with the D operator by replacing $y'' \to D^2y$ & $y' \to Dy$

$$(D^2 + 2n\cos\alpha\, D + n^2)y = \sin nx$$
$$L(D)y = \sin nx$$

Step 2 The *Auxiliary Equation* for the corresponding homogeneous *DE*

$$L(D)y = 0$$

will be obtained by writing

$$L(\lambda) = 0$$
$$\lambda^2 + 2n\cos\alpha\,\lambda + n^2 = 0$$

The roots are then found as

$$\lambda_1 = \frac{-(2n\cos\alpha) + \sqrt{(2n\cos\alpha)^2 - 4(n^2)}}{2} \;\&\; \lambda_2 = \frac{-(2n\cos\alpha) - \sqrt{(2n\cos\alpha)^2 - 4(n^2)}}{2}$$
$$\lambda_1 = \frac{-(2n\cos\alpha) + 2n\sqrt{\cos^2\alpha - 1}}{2} \;\&\; \lambda_2 = \frac{-(2n\cos\alpha) - 2n\sqrt{\cos^2\alpha - 1}}{2}$$
$$\lambda_1 = -n\cos\alpha + in\sin\alpha \;\&\; \lambda_2 = -n\cos\alpha - in\sin\alpha$$

The CF would be $C_1e^{(-n\cos\alpha + in\sin\alpha)x} + C_2e^{(-n\cos\alpha - in\sin\alpha)x}$ which can be represented as

$$CF = e^{-(n\cos\alpha)x}\{C_1\cos[(n\sin\alpha)x] + C_2\sin[(n\sin\alpha)x]\}$$

Step 3 The PI would now be obtained as

$$PI = \frac{1}{L(D)}\sin nx = \frac{1}{(D^2 + 2n\cos\alpha\, D + n^2)}\sin nx$$
$$PI = \frac{1}{\left((-n^2) + 2n\cos\alpha\, D + n^2\right)}\sin nx$$
$$PI = \frac{1}{(2n\cos\alpha\, D)}\sin nx$$
$$PI = \frac{1}{(2n\cos\alpha)}\int \sin nx\, dx$$
$$PI = \frac{1}{(2n^2\cos\alpha)}\int \sin nx\, d(nx)$$
$$PI = -\frac{\cos nx}{(2n^2\cos\alpha)}$$

Step 4 The *General Solution* would therefore be

$$y = CF + PI = e^{-(n\cos\alpha)x}\{C_1\cos[(n\sin\alpha)x] + C_2\sin[(n\sin\alpha)x]\} - \frac{\cos nx}{(2n^2\cos\alpha)}$$

9.4 *When function $f(x)$ is of the form x^m, m being a positive integer*

If $f(x) = x^m$ then we can see that

$$Dx^m = mx^{m-1}$$
$$D^2x^m = D(Dx^m) = D(mx^{m-1}) = m(m-1)x^{m-2}$$

and so on

$$D^n x^m = m(m-1)(m-2)\dots(m-n+1)x^{m-n}$$

So if $n = m+1$ then $D^n x^m = 0$ and $D^n x^m = 0 \,\forall\, n > m+1$. With this in mind, to evaluate $\frac{1}{L(D)}x^m$ we do the following

- Expand $\frac{1}{L(D)}$ in ascending powers of D as far as the term D^m as we would do for any polynomial expression
- Then operate on x^m by the different powers of D in the expression

Example 9.4.1 Solve the equation

$$\boldsymbol{y'' - 5y' + 6y = x}$$

Solution:

Step 1 The *DE* will be written with the D operator by replacing $y'' \to D^2y$ & $y' \to Dy$

$$(D^2 - 5D + 6)y = x$$
$$L(D)y = x$$

Step 2 The *Auxiliary Equation* for the corresponding homogeneous *DE* $L(D)y = 0$ will be obtained by writing $L(\lambda) = 0$

$$\lambda^2 - 5\lambda + 6 = 0$$

The roots are then found as

$$\lambda_1 = \frac{-(-5) + \sqrt{(-5)^2 - 4(6)}}{2} \;\&\; \lambda_2 = \frac{-(-5) - \sqrt{(-5)^2 - 4(6)}}{2}$$

$$\lambda_1 = \frac{5+1}{2} = 3 \;\&\; \lambda_2 = \frac{5-1}{2} = 2$$

The CF would be

$$CF = C_1e^{3x} + C_2e^{2x}$$

Step 3 The PI would now be obtained as

$$PI = \frac{1}{L(D)}x = \frac{1}{(D^2 - 5D + 6)}x$$

$$PI = \frac{1}{6\left(1 + \frac{(D^2 - 5D)}{6}\right)}x = \frac{1}{6}\left(1 + \frac{(D^2 - 5D)}{6}\right)^{-1} x$$

Since $f(x) = x$ is of power 1 we will expand only upto 1 power of D (any higher power term will vanish as shown earlier)

$$PI = \frac{1}{6}\left(1 + (-1)\frac{(D^2 - 5D)}{6} + \cdots\right)x = \frac{1}{6}\left(1 + \frac{5D}{6}\right)x$$

$$PI = \frac{1}{6}\left(x + \frac{5}{6}\right) = \frac{x}{6} + \frac{5}{36}$$

Step 4 The *General Solution* would therefore be

$$y = CF + PI = C_1e^{3x} + C_2e^{2x} + \frac{x}{6} + \frac{5}{36}$$

Example 9.4.2 Solve the equation

$$\boldsymbol{y'' + y' = x^3 + 2x^2}$$

Solution:
Step 1 The *DE* will be written with the D operator by replacing $y'' \to D^2y$ & $y' \to Dy$

$$(D^2 + D)y = x^3 + 2x^2$$
$$L(D)y = x^3 + 2x^2$$

Step 2 The *Auxiliary Equation* for the corresponding homogeneous *DE* $L(D)y = 0$ will be obtained by writing $L(\lambda) = 0$

$$\lambda^2 + \lambda = 0$$
$$\lambda(\lambda + 1) = 0$$

The roots are then found as

$$\lambda_1 = 0 \,\&\, \lambda_2 = -1$$

The CF would be

$$CF = C_1e^{0x} + C_2e^{-x} = C_1 + C_2e^{-x}$$

Step 3 The PI would now be obtained as

$$PI = \frac{1}{L(D)}(x^3 + 2x^2) = \frac{1}{(D^2 + D)}(x^3 + 2x^2) = \frac{1}{D(D + 1)}(x^3 + 2x^2) = \frac{1}{D}(1 + D)^{-1}(x^3 + 2x^2)$$

Since $f(x) = x^3 + 2x^2$ is of power 3 we will expand only upto 3 power of D (any higher power term will vanish as shown earlier)

$$PI = \frac{1}{D}(1 - D + D^2 - D^3 + \cdots)(x^3 + 2x^2) = \frac{1}{D}(1 - D + D^2 - D^3)(x^3 + 2x^2)$$
$$PI = \frac{1}{D}\left((x^3 + 2x^2) - D(x^3 + 2x^2) + D^2(x^3 + 2x^2) - D^3(x^3 + 2x^2)\right)$$
$$PI = \frac{1}{D}\left((x^3 + 2x^2) - (3x^2 + 4x) + (6x + 4) - (6 + 0)\right)$$
$$PI = \frac{1}{D}(x^3 + 2x^2 - 3x^2 + 6x - 4x + 4 - 6) = \frac{1}{D}(x^3 - x^2 + 2x - 2)$$
$$PI = \frac{x^4}{4} - \frac{x^3}{3} + x^2 - 2x$$

Step 4 The *General Solution* would therefore be

$$y = CF + PI = C_1 + C_2e^{-x} + \frac{x^4}{4} - \frac{x^3}{3} + x^2 - 2x$$

Example 9.4.3 Solve the equation

$$\boldsymbol{y'' + y' - 2y = x + \sin x}$$

Solution:

Step 1 The *DE* will be written with the D operator by replacing $y'' \to D^2y$ & $y' \to Dy$

$$(D^2 + D - 2)y = x + \sin x$$
$$L(D)y = x + \sin x$$

Step 2 The *Auxiliary Equation* for the corresponding homogeneous *DE* $L(D)y = 0$ will be obtained by writing $L(\lambda) = 0$

$$\lambda^2 + \lambda - 2 = 0$$

The roots are then found as

$$\lambda_1 = \frac{-(1) + \sqrt{(1)^2 - 4(-2)}}{2} = \frac{-1 + \sqrt{9}}{2} \ \& \ \lambda_2 = \frac{-(1) - \sqrt{(1)^2 - 4(-2)}}{2} = \frac{-1 - \sqrt{9}}{2}$$
$$\lambda_1 = 1 \ \& \ \lambda_2 = -2$$

The CF would be

$$CF = C_1e^x + C_2e^{-2x}$$

Step 3 The PI would now be obtained as

$$PI = \frac{1}{L(D)}(x + \sin x) = \frac{1}{L(D)}x + \frac{1}{L(D)}\sin x$$
$$PI = \frac{1}{(D^2 + D - 2)}x + \frac{1}{(D^2 + D - 2)}\sin x$$

Let's first solve for

$$PI_1 = \frac{1}{(D^2 + D - 2)}x = -\frac{1}{2\left(1 - \frac{(D^2 + D)}{2}\right)}x = -\frac{1}{2}\left(1 - \frac{(D^2 + D)}{2}\right)^{-1} x$$

Since $f(x) = x$ is of power 1 we will expand only upto 1 power of D (any higher power term will vanish as shown earlier)

$$PI_1 = -\frac{1}{2}\left(1 + \frac{(D^2 + D)}{2}\right)x = -\frac{1}{2}\left(1 + \frac{(D^2 + D)}{2}\right)x = -\frac{1}{2}\left(x + \frac{1}{2}\right)$$

Now let's solve for

$$PI_2 = \frac{1}{(D^2 + D - 2)}\sin x = \frac{1}{\left((-1^2) + D - 2\right)}\sin x = \frac{1}{(D - 3)}\sin x$$

$$PI_2 = \frac{(D + 3)}{(D + 3)(D - 3)}\sin x = \frac{(D + 3)}{(D^2 - 9)}\sin x = \frac{(D + 3)}{\left((-1^2) - 9\right)}\sin x = -\frac{(D + 3)}{10}\sin x$$

$$PI_2 = -\frac{1}{10}(\cos x + 3\sin x)$$

Thus,

$$PI = PI_1 + PI_2 = -\frac{1}{2}\left(x + \frac{1}{2}\right) - \frac{1}{10}(\cos x + 3\sin x)$$

Step 4 The *General Solution* would therefore be

$$y = CF + PI = C_1 e^x + C_2 e^{-2x} - \frac{1}{2}\left(x + \frac{1}{2}\right) - \frac{1}{10}(\cos x + 3\sin x)$$

9.5 *When function $f(x)$ is of the form $e^{ax}V(x)$*

If $f(x) = e^{ax}V(x)$ then we can see that

$$D\{e^{ax}V(x)\} = \{De^{ax}\}\,V(x) + e^{ax}\,\{DV(x)\} = \{ae^{ax}\}\,V(x) + e^{ax}\,\{DV(x)\}$$
$$D\{e^{ax}V(x)\} = e^{ax}\{(D+a)V(x)\}$$

Writing $V_1(x) = (D+a)V(x)$ we find that

$$D\{e^{ax}V(x)\} = e^{ax}V_1(x)$$

Therefore,

$$D^2\{e^{ax}V(x)\} = D\big\{D\{e^{ax}V(x)\}\big\} = D\{e^{ax}V_1(x)\} = e^{ax}\{(D+a)V_1(x)\} = e^{ax}\{(D+a)(D+a)V(x)\}$$
$$D^2\{e^{ax}V(x)\} = e^{ax}\{(D+a)^2V(x)\}$$

This suggests that in general,

$$D^n\{e^{ax}V(x)\} = e^{ax}\{(D+a)^nV(x)\}$$

So if $L(D) = a_nD^n + \dots + a_2D^2 + a_1D + a_0$ then

$$L(D)\{e^{ax}V(x)\} = (a_nD^n + \dots + a_2D^2 + a_1D + a_0)\{e^{ax}V(x)\}$$
$$L(D)\{e^{ax}V(x)\} = a_nD^n\{e^{ax}V(x)\} + \dots + a_2D^2\{e^{ax}V(x)\} + a_1D\{e^{ax}V(x)\} + a_0\{e^{ax}V(x)\}$$
$$L(D)\{e^{ax}V(x)\} = a_ne^{ax}\{(D+a)^nV(x)\} + \dots + a_2e^{ax}\{(D+a)^2V(x)\} + a_1e^{ax}\{(D+a)V(x)\} + a_0\{e^{ax}V(x)\}$$
$$L(D)\{e^{ax}V(x)\} = e^{ax}[a_n(D+a)^n + \dots + a_2(D+a)^2 + a_1(D+a) + a_0]V(x)$$

$$L(D)\{e^{ax}V(x)\} = e^{ax}L(D+a)V(x)$$

Thus, operating on both sides by the "*inverse*" operator $\frac{1}{L(D)}$ we find that

$$\frac{1}{L(D)}L(D)\{e^{ax}V(x)\} = \frac{1}{L(D)}\{e^{ax}L(D+a)V(x)\}$$
$$e^{ax}V(x) = \frac{1}{L(D)}\{e^{ax}L(D+a)V(x)\}$$

Now if we write $U(x) = L(D+a)V(x)$ then this can be interpreted as

$$e^{ax}\left\{\frac{1}{L(D+a)}U(x)\right\} = \frac{1}{L(D)}\{e^{ax}U(x)\}$$

This beautiful result then states a rule that an n^{th} order Non-Homogeneous Linear DE with Constant coefficients $L(D)y = e^{ax}V(x)$ has the PI

$$y = \frac{1}{L(D)}\{e^{ax}V(x)\} = e^{ax}\left\{\frac{1}{L(D+a)}V(x)\right\}$$

which simplifies the procedure by taking out the exponential term and displacing the D operatorin $L(D)$ by 'a'.

Example 9.5.1 Solve the equation

$$\boldsymbol{y'' - 2y' + 5y = e^{2x}\sin x}$$

Solution:
Step 1 The *DE* will be written with the D operator by replacing $y'' \to D^2y$ & $y' \to Dy$

$$(D^2 - 2D + 5)y = e^{2x}\sin x$$
$$L(D)y = e^{2x}\sin x$$

Step 2 The *Auxiliary Equation* for the corresponding homogeneous *DE* $L(D)y = 0$ will be obtained by writing $L(\lambda) = 0$

$$\lambda^2 - 2\lambda + 5 = 0$$

The roots are then found as

$$\lambda_1 = \frac{-(-2) + \sqrt{(-2)^2 - 4(5)}}{2} = \frac{2 + \sqrt{-16}}{2} \;\&\; \lambda_2 = \frac{-(-2) + \sqrt{(-2)^2 - 4(5)}}{2} = \frac{2 - \sqrt{-16}}{2}$$
$$\lambda_1 = 1 + i2 \;\&\; \lambda_2 = 1 - i2$$

The CF would be $C_1e^{(1+i2)x} + C_2e^{(1+i2)x}$ which can be represented as

$$CF = e^x\{C_1\cos 2x + C_2\sin 2x\}$$

Step 3 The PI would now be obtained as

$$PI = \frac{1}{L(D)}e^{2x}\sin x = e^{2x}\frac{1}{L(D+2)}\sin x$$
$$PI = e^{2x}\frac{1}{\{(D+2)^2 - 2(D+2) + 5\}}\sin x = e^{2x}\frac{1}{\{D^2 + 4 + 4D - 2D - 4 + 5\}}\sin x$$
$$PI = e^{2x}\frac{1}{\{D^2 + 2D + 5\}}\sin x$$

Now using the rule $\frac{1}{L(D^2)}\{\sin(ax)\} = \frac{1}{L(-a^2)}\{\sin(ax)\}$ we get

$$PI = e^{2x}\frac{1}{\{(-1^2) + 2D + 5\}}\sin x = e^{2x}\frac{1}{\{2D + 4\}}\sin x = \frac{e^{2x}}{2}\frac{1}{(D+2)}\sin x$$
$$PI = \frac{e^{2x}}{2}\frac{(D-2)}{(D-2)(D+2)}\sin x = \frac{e^{2x}}{2}\frac{(D-2)}{(D^2-4)}\sin x = \frac{e^{2x}}{2}\frac{(D-2)}{\left((-1^2) - 4\right)}\sin x$$
$$PI = -\frac{e^{2x}}{10}(D-2)\sin x = -\frac{e^{2x}}{10}(\cos x - 2\sin x)$$
$$PI = \frac{e^{2x}}{10}(2\sin x - \cos x)$$

Step 4 The *General Solution* would therefore be

$$y = CF + PI = e^{x}\{C_1 \cos 2x + C_2 \sin 2x\} + \frac{e^{2x}}{10}(2\sin x - \cos x)$$

Example 9.5.2 Solve the equation

$$\boldsymbol{y'' + \beta^2 y = Ae^{i\alpha x}x}$$

where $\alpha\,\&\,\beta$ are constant real numbers.

Solution:
Step 1 The *DE* will be written with the D operator by replacing $y'' \to D^2y$ & $y' \to Dy$

$$(D^2 + \beta^2)y = Ae^{i\alpha x}x$$
$$L(D)y = Ae^{i\alpha x}x$$

Step 2 The *Auxiliary Equation* for the corresponding homogeneous *DE* $L(D)y = 0$ will be obtained by writing $L(\lambda) = 0$

$$\lambda^2 + \beta^2 = 0$$
$$\lambda = \sqrt{-\beta^2}$$

The roots are then found as

$$\lambda_1 = i\beta \,\&\, \lambda_2 = -i\beta$$

The CF would be $C_1e^{i\beta x} + C_2e^{-i\beta x}$ which can be represented as

$$CF = C_1 \cos\beta x + C_2 \sin\beta x$$

Step 3 The PI would now be obtained as

$$PI = \frac{1}{L(D)}Ae^{i\alpha x}x = Ae^{i\alpha x}\frac{1}{L(D + i\alpha)}x$$
$$PI = Ae^{i\alpha x}\frac{1}{\{(D + i\alpha)^2 + \beta^2\}}x = Ae^{i\alpha x}\frac{1}{\{D^2 + 2i\alpha D - \alpha^2 + \beta^2\}}x$$
$$PI = Ae^{i\alpha x}\frac{1}{\{D^2 + 2i\alpha D + (\beta^2 - \alpha^2)\}}x$$

Since $f(x) = x$ is of power 1 we will expand only upto 1 power of D (any higher power term will vanish as shown earlier)

$$PI = \frac{A}{(\beta^2 - \alpha^2)}e^{i\alpha x}\frac{1}{\left\{1 + \frac{2i\alpha D + D^2}{(\beta^2 - \alpha^2)}\right\}}x = \frac{A}{(\beta^2 - \alpha^2)}e^{i\alpha x}\left\{1 + \frac{2i\alpha D + D^2}{(\beta^2 - \alpha^2)}\right\}^{-1}x$$
$$PI = \frac{A}{(\beta^2 - \alpha^2)}e^{i\alpha x}\left\{1 - \frac{2i\alpha D}{(\beta^2 - \alpha^2)}\right\}x$$
$$PI = \frac{A}{(\beta^2 - \alpha^2)}e^{i\alpha x}\left\{x - \frac{2i\alpha}{(\beta^2 - \alpha^2)}\right\}$$

Step 4 The *General Solution* would therefore be

$$y = CF + PI = C_1 \cos\beta x + C_2 \sin\beta x + \frac{A}{(\beta^2 - \alpha^2)} e^{i\alpha x}\left\{x - \frac{2i\alpha}{(\beta^2 - \alpha^2)}\right\}$$

However if $\alpha = \beta$ then

$$CF = C_1 \cos\alpha x + C_2 \sin\alpha x$$

and form step 3 above

$$PI = Ae^{i\alpha x}\frac{1}{\{D^2 + 2i\alpha D\}}x$$

$$PI = \frac{A}{2i\alpha}e^{i\alpha x}\frac{1}{D}\frac{1}{\left\{1 + \frac{D^2}{2i\alpha D}\right\}}x = \frac{A}{2i\alpha}e^{i\alpha x}\frac{1}{D}\frac{1}{\left\{1 + \frac{D}{2i\alpha}\right\}}x$$

$$PI = \frac{A}{2i\alpha}e^{i\alpha x}\frac{1}{D}\left\{1 + \frac{D}{2i\alpha}\right\}^{-1}x = \frac{A}{2i\alpha}e^{i\alpha x}\frac{1}{D}\left\{1 - \frac{D}{2i\alpha}\right\}x$$

$$PI = \frac{A}{2i\alpha}e^{i\alpha x}\frac{1}{D}\left\{x - \frac{1}{2i\alpha}\right\}$$

$$PI = \frac{A}{2i\alpha}e^{i\alpha x}\int\left\{x - \frac{1}{2i\alpha}\right\}dx$$

$$PI = \frac{A}{2i\alpha}e^{i\alpha x}\left\{\frac{x^2}{2} - \frac{x}{2i\alpha}\right\}$$

$$PI = \frac{A}{4\alpha^2}e^{i\alpha x}\left\{\frac{\alpha x^2}{i} + x\right\}$$

The *General Solution* would therefore be

$$y = CF + PI = C_1 \cos\alpha x + C_2 \sin\alpha x + \frac{A}{4\alpha^2}e^{i\alpha x}\left\{\frac{\alpha x^2}{i} + x\right\}$$

Summary

Particular Integral of Special Forms of the Function $f(x)$

- There are certain special forms of the function $f(x)$ which admits rules for finding PI of the Linear DE with constant coefficients in shorter steps.
- When function $f(x)$ is of the form e^{ax} then the PI $y = A\frac{e^{ax}}{L(a)}$

  There may arise a situation where $L(a) = 0$. This would then imply "a" to be an r^{th} *order root* of the n^{th} *order Non-Homogeneous Linear DE with Constant coefficients so that* $L(D) = (D - a)^r \varphi(D)$ *then the PI*

  $$y = \frac{A}{\varphi(a)}\frac{x^r}{r!}e^{ax}$$
- When function $f(x)$ is of the form $sin\,ax$ *or* $cos\,ax$ then the PI $y = A\frac{\sin(ax+\theta)}{L(-a^2)}$

  There may arise a situation where $L(-a^2) = 0$. This would then imply "$-a^2$" to be an r^{th} *order root* of the *DE so that* $L(D^2) = (D^2 + a^2)^r \varphi(D^2)$ *then the PI*

  $$y = \frac{A}{\varphi(-a^2)}\frac{1}{(D^2 + a^2)^r}\sin(ax + \theta)$$
- When function $f(x)$ is of the form x^m, m being a positive integer then the *PI* can be found by expanding $\frac{\boldsymbol{1}}{\boldsymbol{L(D)}}$ in ascending powers of $\boldsymbol{D}$ as far as the term $\boldsymbol{D^m}$ as we

would do for any polynomial expression and operating on $\boldsymbol{x^m}$ by the different powers of $\boldsymbol{D}$ in the expression

- When function $\boldsymbol{f}(\boldsymbol{x})$ is of the form $\boldsymbol{e^{ax}V}(\boldsymbol{x})$ then the $PI\ y = e^{ax}\left\{\frac{1}{L(D+a)}V(x)\right\}$

Bibliography/ References / Glossary

1. Advanced Engineering Mathematics by Erwin Kreysig
2. Advanced Engineering Mathematics by Michael D. Greenberg
3. Schaum's Outline: Theory and Problems of Advanced Calculus by Murray R. Spiegel
4. Mathematical Methods in Physical Sciences by Mary L. Boas
5. Calculus & Analytic Geometry by Fobes & Smyth
6. Essential Mathematical Methods by K.F. Riley & M.P. Hobson
7. Schaum's Outline: Theory and Problems of Differential Equations by Richard Bronson
8. Schaum's Outline: Theory and Problems of Differential Equations by Frank Ayres
9. Introductory Course in Differential Equations by Daniel A. Murray
10. Differential Equations by N.M. Kapoor
11. Higher Engineering Mathematics by B S Grewal
12. A Treatise on Differential Equations by A. R. Forsyth

The C Book — Table of Contents

This is a PDF version of a page on the GBdirect web site. The original page may be found at http://publications.gbdirect.co.uk/c_book/.

This is the PDF version of *The C Book*, second edition by Mike Banahan, Declan Brady and
Doran, originally published by Addison Wesley in 1991. This version is made freely availabl
[*http://publications.gbdirect.co.uk/c_book/copyright.html*].

While this book is no longer in print, it's content is still very relevant today. The C language i
popular, particularly for open source software [*http://ebusiness.gbdirect.co.uk/OpenSource*
and embedded programming [*http://training.gbdirect.co.uk/courses/c/embedded_c_training.*
hope this book will be useful, or at least interesting, to people who use C.

If you have any comments about this book, or if you find any bugs in its presentation, please
message to consulting@gbdirect.co.uk.

This PDF version made by Carlos José de Almeida Pereira - carlao2005(at)gmail(dot)com, Bahia, Brasil, to all happy C programmers over the world!

WARNING! The links inside this document will jump to the original page on the Web, not to specific place on the book. So, don't use them to offline reading. Sorry!

Preface

Next chapter [*http://publications.gbdirect.co.uk/c_book/chapter1/*]

About This Book

This book was written with two groups of readers in mind. Whether you are new to C and want to learn it, or already know the older version of the language but want to find out more about the new standard, we hope that you will find what follows both instructive and at times entertaining too.

This is not a tutorial introduction to programming. The book is designed for programmers who already have some experience of using a modern high-level procedural programming language. As we explain later, C isn't really appropriate for complete beginners—though many have managed to use it—so the book will assume that its readers have already done battle with the notions of statements, variables, conditional execution, arrays, procedures (or subroutines) and so on. Instead of wasting your time by ploughing through tedious descriptions of how to add two numbers together and explaining that the symbol for multiplication is `*`, the book concentrates on the things that are special to C. In particular, it's the *way* that C is used which is emphasized.

Those who already know C will be interested in the new Standard and how it affects existing C programs. The effect on existing programs might not at first seem to be important to newcomers, but in fact the ‘old’ and new versions of the language *are* an issue for the beginner too. For some years after the approval of the Standard, programmers will have to live in a world where they can easily encounter a mixture of both the new and the old language, depending on the age of the programs that they are working with. For that reason, the book highlights where the old and new features differ significantly. Some of the old features are no ornament to the language and are well worth avoiding; the Standard goes so far as to consider them obsolescent and recommends that they should not be used. For that reason they are not described in detail, but only far enough to allow a reader to understand what they mean. Anybody who intends to *write* programs using these old-style features should be reading a different book.

This is the second edition of the book, which has been revised to refer to the final, approved version of the Standard. The first edition of the book was based on a draft of the Standard which did contain some differences from the draft that was eventually approved. During the revision we have taken the opportunity to include more summary material and an extra chapter illustrating the use of C and the Standard Library to solve a number of small problems.

Chapter contents [*http://publications.gbdirect.co.uk/c_book/preface/*] | Next section [*http://publications.gbdirect.co.uk/c_book/preface/the_success_of_c.html*]

The Success of C

C is a remarkable language. Designed originally by one man, Dennis Ritchie, working at AT&T Bell Laboratories in New Jersey, it has increased in use until now it may well be one of the most widely-written computer languages in the world. The success of C is due to a number of factors, none of them key, but all of them important. Perhaps the most significant of all is that C was developed by real practioners of programming and was designed for practical day-to-day use, not for show or for demonstration. Like any well-designed tool, it falls easily to the hand and feels good to use. Instead of providing constraints, checks and rigorous boundaries, it concentrates on providing you with power and on not getting in your way.

Because of this, it's better for professionals than beginners. In the early stages of learning to program you need a protective environment that gives feedback on mistakes and helps you to get results quickly—programs that run, even if they don't do what you meant. C is not like that! A professional forester would use a chain-saw to cut down trees quickly, aware of the dangers of touching the blade when the machine is running; C programmers work in a similar way. Although modern C compilers do provide a limited amount of feedback when they notice something that is out of the ordinary, you almost always have the option of forcing the compiler to do what you said you wanted and to stop it from complaining. Provided that what you said you wanted was what you really did want, then you'll get the result you expected. Programming in C is like eating red meat and drinking strong rum except your arteries and liver are more likely to survive it.

Not only is C popular and a powerful asset in the armoury of the serious day-to-day programmer, there are other reasons for the success of this language. It has always been associated with the UNIX operating system and has benefited from the increasing popularity of that system. Although it is not the obvious first choice for writing large commercial data processing applications, C has the great advantage of always being available on commercial UNIX implementations. UNIX is written in C, so whenever UNIX is implemented on a new type of hardware, getting a C compiler to work for that system is the first task. As a result it is almost impossible to find a UNIX system without support for C, so the software vendors who want to target the UNIX marketplace find that C is the best bet if they want to get wide coverage of the systems available. Realistically, C is the first choice for portability of software in the UNIX environment.

C has also gained substantially in use and availability from the explosive expansion of the Personal Computer market. C could almost have been designed specifically for the development of software for the PC—developers get not only the readability and productivity of a high-level language, but also the power to get the most out of the PC architecture *without* having to resort to the use of assembly code. C is practically unique in its ability to span two levels of programming; as well as providing high-level control of flow, data structures and procedures—all of the stuff expected in a modern high-level language—it also allows systems programmers to address machine words, manipulate bits and get close to the underlying hardware if they want to. That combination of features is very desirable in the competitive PC software markeplace and an increasing number of software developers have made C their primary language as a result.

Finally, the extensibility of C has contributed in no small way to its popularity. Many other languages have failed to provide the file access and general input-output features that are needed for industrial-strength applications. Traditionally, in these languages I/O is built-in and is actually understood by the compiler. A master-stroke in the design of C (and interestingly, one of the strengths of the UNIX system too) has been to take the view that if you don't know how to provide a complete solution to a generic requirement, instead of providing half a solution (which invariably pleases nobody), you should allow the users to build their own. Software designers the world over have something to learn from this! It's the approach that has been taken by C, and not only for I/O. Through the use of *library functions* you can extend the language in many ways to provide features that the designers didn't think of. There's proof of this in the so-called Standard I/O Library (stdio), which matured more slowly than the language, but had become a sort of standard all of its own before the Standard Committee give it official blessing. It proved that it is possible to develop a model of file I/O and associated features that is portable to many more systems than UNIX, which is where it was first wrought. Despite the ability of C to provide access to low-level hardware features, judicious style and the use of the stdio package results in highly portable programs; many of which are to be found running on top of operating systems that look very different from one another. The nice thing about this library is that if you don't like what it does, but you have the appropriate technical skills, you can usually extend it to do what you do want, or bypass it altogether.

Previous section [*http://publications.gbdirect.co.uk/c_book/preface/about.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/preface/*] | Next section [*http://publications.gbdirect.co.uk/c_book/preface/standards.html*]

Standards

Remarkably, C achieved its success in the absence of a formal standard. Even more remarkable is that during this period of increasingly widespread use, there has never been any serious divergence of C into the number of dialects that has been the bane of, for example, BASIC. In fact, this is not so surprising. There has always been a “language reference manual”, the widely-known book written by Brian Kernighan and Dennis Ritchie, usually referred to as simply “K&R”.

> The C Programming Language,
> B.W. Kernighan and D. M. Ritchie,
> Prentice-Hall
> Englewood Cliffs,
> New Jersey,
> 1978

Further acting as a rigorous check on the expansion into numerous dialects, on UNIX systems there was only ever really one compiler for C; the so-called “Portable C Compiler”, originally written by Steve Johnson. This acted as a reference implementation for C—if the K&R reference was a bit obscure then the behaviour of the UNIX compiler was taken as the definition of the language.

Despite this almost ideal situation (a reference manual and a reference implementation are extremely good ways of achieving stability at a very low cost), the increasing number of alternative implementations of C to be found in the PC world did begin to threaten the stability of the language.

The X3J11 committee of the American National Standards Institute started work in the early 1980's to produce a formal standard for C. The committee took as its reference the K&R definition and began its lengthy and painstaking work. The job was to try to eliminate ambiguities, to define the undefined, to fix the most annoying deficiencies of the language and to preserve the spirit of C—all this as well as providing as much compatibility with existing practice as was possible. Fortunately, nearly all of the developers of the competing versions of C were represented on the committee, which in itself acted as a strong force for convergence right from the beginning.

Development of the Standard took a long time, as standards often do. Much of the work is not just technical, although that is a very time-consuming part of the job, but also procedural. It's easy to underrate the procedural aspects of standards work, as if it somehow dilutes the purity of the technical work, but in fact it is equally important. A standard that has no agreement or consensus in the industry is unlikely to be widely adopted and could be useless or even damaging. The painstaking work of obtaining consensus among committee members is critical to the success of a practical standard, even if at times it means compromising on technical “perfection”, whatever that might be. It is a democratic process, open to all, which occasionally results in aberrations just as much as can excessive indulgence by technical purists, and unfortunately the delivery date of the Standard was affected at the last moment by procedural, rather than technical issues. The technical work was completed by December 1988, but it took a further year to resolve procedural objections. Finally, approval to release the document as a formal American National Standard was given on December 7th, 1989.

Previous section [*http://publications.gbdirect.co.uk/c_book/preface/the_success_of_c.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/preface/*] | Next section [*http://publications.gbdirect.co.uk/c_book/preface/hosted_and_free_standing.html*]

Hosted and Free-Standing Environments

The dependency on the use of libraries to extend the language has an important effect on the practical use of C. Not only are the Standard I/O Library functions important to applications programmers, but there are a number of other functions that are widely taken almost for granted as being part of the language. String handling, sorting and comparison, character manipulation and similar services are invariably expected in all but the most specialized of applications areas.

Because of this unusually heavy dependency on libraries to do real work, it was most important that the Standard provided comprehensive definitions for the supporting functions too. The situation with the library functions was much more complicated than the relatively simple job of providing a tight definition for the language itself, because the library can be extended or modified by a knowledgeable user and was only partially defined in K&R. In practice, this led to numerous similar but different implementations of supporting libraries in common use. By far the hardest part of the work of the Committee was to reach a good definition of the library support that should be provided. In terms of benefit to the final user of C, it is this work that will prove to be by far and away the most valuable part of the Standard.

However, not all C programs are used for the same type of applications. The Standard Library is useful for ‘data processing’ types of applications, where file I/O and numeric and string oriented data are widely used. There is an equally important application area for C—the ‘embedded system’ area—which includes such things as process control, real-time and similar applications.

The Standard knows this and provides for it. A large part of the Standard is the definition of the library functions that must be supplied for *hosted environments*. A hosted environment is one that provides the standard libraries. The standard permits both hosted and *freestanding environments*. and goes to some length to differentiate between them. Who would want to go without libraries? Well, anybody writing ‘stand alone’ programs. Operating systems, embedded systems like machine controllers and firmware for instrumentation are all examples of the case where a hosted environment might be inappropriate. Programs written for a hosted environment have to be aware of the fact that the names of all the library functions are reserved for use by the implementation. There is no such restriction on the programmer working in a freestanding environment, although it isn't a good idea to go using names that are used in the standard library, simply because it will mislead readers of the program. Chapter 9 [*http://publications.gbdirect.co.uk/c_book/chapter9/*] describes the names and uses of the library functions.

Previous section [*http://publications.gbdirect.co.uk/c_book/preface/standards.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/preface/*] | Next section

[*http://publications.gbdirect.co.uk/c_book/preface/typographical_conventions.html*]

Typographical conventions

The book tries to keep a consistent style in its use of special or technical terms. Words with a special meaning to C, such as *reserved words* or the names of *library functions*, are printed in a different typeface. Examples are `int` and `printf`. Terms used by the book that have a meaning not to C but in the Standard or the text of the book, are *bold* if they have not been introduced recently. They are *not* bold everywhere, because that rapidly annoys the reader. As you have noticed, italics are also used for emphasis from time to time, and to introduce loosely defined terms. Whether or not the name of a function, keyword or so on starts with a capital letter, it is nonetheless capitalized when it appears at the start of a sentence; this is one problem where either solution (capitalize or not) is unsatisfactory. Occasionally quote marks are used around ‘special terms’ if there is a danger of them being understood in their normal English meaning because of surrounding context. Anything else is at the whim of the authors, or simply by accident.

Previous section [*http://publications.gbdirect.co.uk/c_book/preface/hosted_and_free_standing.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/preface/*] | Next section [*http://publications.gbdirect.co.uk/c_book/preface/order_of_topics.html*]

Order of topics

The order of presentation of topics in this book loosely follows the order that is taught in The Instruction Set's introductory course. It starts with an overview of the essential parts of the language that will let you start to write useful programs quite quickly. The introduction is followed by a detailed coverage of the material that was ignored before, then it goes on to discuss the standard libraries in depth. This means that in principle, if you felt so inclined, you could read the book as far as you like and stop, yet still have learnt a reasonably coherent subset of the language. Previous experience of C will render Chapter 1 [*http://publications.gbdirect.co.uk/c_book/chapter1/*] a bit slow, but it is still worth persevering with it, if only once.

Previous section [*http://publications.gbdirect.co.uk/c_book/preface/typographical_conventions.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/preface/*] | Next section [*http://publications.gbdirect.co.uk/c_book/preface/example_programs.html*]

Example programs

<gbdirect>

All but the smallest of the examples shown in the text have been tested using a compiler that claims to conform to the Standard. As a result, most of them stand a good chance of being correct, unless our interpretation of the Standard was wrong and the compiler developer made the same mistake. None the less, experience warns that despite careful checking, *some* errors are bound to creep in. Please be understanding with any errors that you may find.

Previous section [*http://publications.gbdirect.co.uk/c_book/preface/order_of_topics.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/preface/*] | Next section [*http://publications.gbdirect.co.uk/c_book/preface/higher_authority.html*]

Deference to Higher Authority

<gbdirect>

This book is an attempt to produce a readable and enlightening description of the language defined by the Standard. It sets out to to make interpretations of what the Standard actually means but to express them in ‘simpler’ English. We've done our best to get it right, but you must never forget that the only place that the language is fully defined is in the Standard itself. It is entirely possible that what we interpret the Standard to mean is at times not what the Standard Committee sought to specify, or that the way we explain it is looser and less precise than it is in the Standard. If you are in any doubt: READ THE STANDARD! It's not meant to be read for pleasure, but it is meant to be accurate and unambiguous; look nowhere else for the authoritative last word.

Previous section [*http://publications.gbdirect.co.uk/c_book/preface/example_programs.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/preface/*] | Next section [*http://publications.gbdirect.co.uk/c_book/preface/c_standard.html*]

Address for the Standard

<gbdirect>

Copies of the Standard can be obtained from:

X3 Secretariat,
CBEMA,
311 First Street, NW,
Suite 500,
Washington DC 20001-2178,
USA.
Phone (+1) (202) 737 8888

Mike Banahan
Declan Brady
Mark Doran

January 1991

Previous section [*http://publications.gbdirect.co.uk/c_book/preface/higher_authority.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/preface/*]

Chapter 1

An Introduction to C

Previous chapter [*http://publications.gbdirect.co.uk/c_book/preface/*] | Next chapter [*http://publications.gbdirect.co.uk/c_book/chapter2/*]

1.1. The form of a C program

<gbdirect>

If you're used to the block-structured form of, say, Pascal, then at the outer level the layout of a C program may surprise you. If your experience lies in the FORTRAN camp you will find it closer to what you already know, but the inner level will look quite different. C has borrowed shamelessly from both kinds of language, and from a lot of other places too. The input from so many varied sources has spawned a language a bit like a cross-bred terrier: inelegant in places, but a tenacious brute that the family is fond of. Biologists refer to this phenomenon as ‘hybrid vigour’. They might also draw your attention to the ‘chimera’, an artificial crossbreed of creatures such as a sheep and a goat. If it gives wool and milk, fine, but it might equally well just bleat and stink!

At the coarsest level, an obvious feature is the multi-file structure of a program. The language permits *separate compilation*, where the parts of a complete program can be kept in one or more *source files* and compiled independently of each other. The idea is that the compilation process will produce files which can then be *linked* together using whatever link editor or loader that your system provides. The block structure of the Algol-like languages makes this harder by insisting that the whole program comes in one chunk, although there are usually ways of getting around it.

The reason for C's approach is historical and rather interesting. It is supposed to speed things up: the idea is that compiling a program into relocatable *object code* is slow and expensive in terms of resources; compiling is hard work. Using the loader to bind together a number of object code modules should simply be a matter of sorting out the absolute addresses of each item in the modules when combined into a complete program. This should be relatively inexpensive. The expansion of the idea to arrange for the loader to scan *libraries* of object modules, and select the ones that are needed, is an obvious one. The benefit is that if you change one small part of a program then the expense of recompiling all of it may be avoided; only the module that was affected has to be recompiled.

All, the same, it's true that the more work put on to the loader, the slower it becomes, in fact sometimes it can be the slowest and most resource consuming part of the whole procedure. It is possible that, for some systems, it would be quicker to recompile everything in one go than to have to use the loader: Ada has sometimes been quoted as an example of this effect occurring. For C, the work that has to be done by the loader is not large and the approach is a sensible one. Figure 1.1 shows the way that this works.

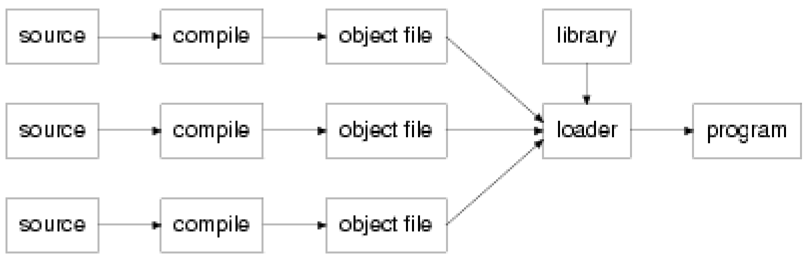


Figure 1.1. Separate compilation

This technique is important in C, where it is common to find all but the smallest of programs constructed from a number of separate source files. Furthermore, the extensive use that C makes of libraries means that even trivial programs pass through the loader, although that might not be obvious at the first glance or to the newcomer.

Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter1/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter1/functions.html*]

1.2. Functions

A C program is built up from a collection of items such as *functions* and what we could loosely call *global variables*. All of these things are given names at the point where they are defined in the program; the way that the names are used to access those items from a given place in the program is governed by rules. The rules are described in the Standard using the term *linkage*. For the moment we only need to concern ourselves with *external linkage* and *no linkage*. Items with external linkage are those that are accessible throughout the program (library functions are a good example); items with no linkage are also widely used but their accessibility is much more restricted. Variables used inside functions are usually ‘local’ to the function; they have no linkage. Although this book avoids the use of complicated terms like those where it can, sometimes there isn't a plainer way of saying things. Linkage is a term that you are going to become familiar with later. The only external linkage that we will see for a while will be when we are using functions.

Functions are C's equivalents of the functions and subroutines in FORTRAN, functions and procedures in Pascal and ALGOL. Neither BASIC in most of its simple mutations, nor COBOL has much like C's functions.

The idea of a function is, of course, to allow you to encapsulate one idea or operation, give it a name, then to call that operation from various parts of the rest of your program simply by using the name. The detail of what is going on is not immediately visible at the point of use, nor should it be. In well designed, properly structured programs, it should be possible to change the way that a function does its job (as long as the job itself doesn't change) with no effect on the rest of the program.

In a *hosted environment* there is one function whose name is special; it's the one called `main`. This function is the first one entered when your program starts running. In a *freestanding environment* the way that a program starts up is *implementation defined*; a term which means that although the Standard doesn't specify what must happen, the actual behaviour must be consistent and documented. When the program leaves the main function, the whole program comes to an end. Here's a simple program containing two functions:

```
#include <stdio.h>

/*
* Tell the compiler that we intend
* to use a function called show_message.
* It has no arguments and returns no value
* This is the "declaration".
*
*/

void show_message(void);
/*
* Another function, but this includes the body of
* the function. This is a "definition".
*/
main(){
     int count;

     count = 0;
     while(count < 10){
             show_message();
             count = count + 1;
     }

     exit(0);
}

/*
* The body of the simple function.
* This is now a "definition".
*/
void show_message(void){
     printf("hello\n");
}
```

Example 1.1

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter1/form_of_a_c_program.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter1/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter1/description_of_example.html*]

1.3. A description of Example 1.1

1.3.1. What was in it

Even such a small example has introduced a lot of C. Among other things, it contained two functions, a `#include` ‘statement’, and some *comment*. Since comment is the easiest bit to handle, let's look at that first.

1.3.2. Layout and comment

The layout of a C program is not very important to the compiler, although for readability it is important to use this freedom to carry extra information for the human reader. C allows you to put space, tab or newline characters practically anywhere in the program without any special effect on the meaning of the program. All of those three characters are the same as far as the compiler is concerned and are called collectively *white space*, because they just move the printing position without causing any ‘visible’ printing on an output device. White space can occur practically anywhere in a program except in the middle of *identifiers*, *strings*, or *character constants*. An identifier is simply the name of a function or some other object; strings and character constants will be discussed later—don't worry about them for the moment.

Apart from the special cases, the only place that white space must be used is to separate things that would otherwise run together and become confused. In the example above, the fragment `void show_message` needs space to separate the two words, whereas `show_message(` could have space in front of the `(` or not, it would be purely a matter of taste.

Comment is introduced to a C program by the pair of characters `/*`, which must not have a space between them. From then on, everything found up to and including the pair of characters `*/` is gobbled up and the whole lot is replaced by a single space. In Old C, this was not the case. The rule used to be that comment could occur anywhere that space could occur: the rule is now that comment is space. The significance of the change is minor and eventually becomes apparent in Chapter 7 [*http://publications.gbdirect.co.uk/c_book/chapter7/*] where we discuss the *preprocessor*. A consequence of the rule for the end of comment is that you can't put a piece of comment inside another piece, because the *first* `*/` pair will finish all of it. This is a minor nuisance, but you learn to live with it.

It is common practice to make a comment stand out by making each line of multi-line comment always start with a `*`, as the example illustrates.

1.3.3. Preprocessor statements

The first statement in the example is a *preprocessor directive*. In days gone by, the

C compiler used to have two phases: the *preprocessor*, followed by the real compiler. The preprocessor was a *macro processor*, whose job was to perform simple textual manipulation of the program before passing the modified text on to be compiled. The preprocessor rapidly became seen as an essential aspect of the compiler and so has now been defined as part of the language and cannot be bypassed.

The preprocessor only knows about *lines* of text; unlike the rest of the language it is sensitive to the end of a line and though it is possible to write multi-line preprocessor directives, they are uncommon and a source of some wonder when they are found. Any line whose first visible character is a `#` is a preprocessor directive.

In Example 1.1 the preprocessor directive `#include` causes the line containing it to be replaced completely by the contents of another file. In this case the filename is found between the < and > brackets. This is a widely used technique to incorporate the text of standard *header files* into your program without having to go through the effort of typing it all yourself. The `<stdio.h>` file is an important one, containing the necessary information that allows you to use the standard library for input and output. If you want to use the I/O library you *must* include `<stdio.h>`. Old C was more relaxed on this point.

1.3.3.1. Define statements

Another of the preprocessor's talents which is widely exploited is the `#define` statement. It is used like this:

```
#define IDENTIFIER      replacement
```

which says that the name represented by `IDENTIFIER` will be replaced by the text of replacement whenever `IDENTIFIER` occurs in the program text. Invariably, the identifier is a name in upper-case; this is a stylistic convention that helps the reader to understand what is going on. The replacement part can be any text at all—remember the preprocessor doesn't know C, it just works on text. The most common use of the statement is to declare names for constant numbers:

```
#define PI             3.141592
#define SECS_PER_MIN   60
#define MINS_PER_HOUR  60
#define HOURS_PER_DAY  24
```

and to use them like this

```
circumf = 2*PI*radius;
if(timer >= SECS_PER_MIN){
mins = mins+1;
        timer = timer - SECS_PER_MIN;
}
```

the output from the preprocessor will be as if you had written this:

```
circumf = 2*3.141592*radius;
if(timer >= 60){
        mins = mins+1;
       timer = timer - 60;
}
```

Summary

Preprocessor statements work on a line-by-line basis, the rest of C does not.

`#include` statements are used to read the contents of a specified file, typically to facilitate the use of library functions.

`#define` statements are typically used to give names for constants. By convention, the names are in upper case (capitalized).

1.3.4. Function declaration and definition

1.3.4.1. Declaration

After the `<stdio.h>` file is included comes a *function declaration*; it tells the compiler that `show_message` is a function which takes no arguments and returns no values. This demonstrates one of the changes made by the Standard: it is an example of a *function prototype*, a subject which Chapter 4 [*http://publications.gbdirect.co.uk/c_book/chapter4/*] discusses in detail. It isn't always necessary to declare functions in advance—C will use some (old) default rules in such cases—but it is now strongly recommended that you *do* declare them in advance. The distinction between a *declaration* and a *definition* is that the former simply describes the type of the function and any arguments that it might take, the latter is where the body of a function is provided. These terms become more important later.

By declaring `show_message` before it is used, the compiler is able to check that it is used correctly. The declaration describes three important things about the function: its name, its type, and the number and type of its arguments. The `void show_message(` part indicates that it is a function and that it returns a value of type `void`, which is discussed in a moment. The second use of `void` is in the declaration of the function's argument list, `(void)`, which indicates that there are *no* arguments to this function.

1.3.4.2. Definition

Right at the end of the program is the function definition itself; although it is only three lines long, it usefully illustrates a complete function.

In C, functions perform the tasks that some other languages split into two parts. Most languages use a function to return a value of some sort, typical examples being perhaps trigonometric functions like sin, cos, or maybe a square root function; C is the same in this respect. Other similar jobs are done by what look very much like functions but which don't return a value: FORTRAN uses *subroutines*, Pascal and Algol call them *procedures*. C simply uses functions for all of those jobs, with the *type* of the function's return value specified when the function is defined. In the example, the function `show_message` doesn't return a value so we specify that its type is `void`.

The use of `void` in that way is either crashingly obvious or enormously subtle, depending on your viewpoint. We could easily get involved here in an entertaining (though fruitless) philosophical side-track on whether `void` really is a value or not, but we won't. Whichever side of the question you favour, it's clear that you can't do anything with a `void` and that's what it means here—“I don't want to do anything with any value this function might or might not return”.

The type of the function is `void`, its name is `show_message`. The parentheses `()` following the function name are needed to let the compiler know that at this point we are talking about a function and not something else. If the function did take any arguments, then their names would be put between the parentheses. This one doesn't take any, which is made explicit by putting `void` between the parentheses.

For something whose essence is emptiness, abnegation and rejection, `void` turns out to be pretty useful.

The body of the function is a *compound statement*, which is a sequence of other statements surrounded by curly brackets `{}`. There is only one statement in there, but the brackets are still needed. In general, C allows you to put a compound statement anywhere that the language allows the use of a single simple statement; the job of the brackets being to turn several statements in a row into what is effectively a single statement.

It is reasonable to ask whether or not the brackets are strictly needed, if their only job is to bind multiple statements into one, yet all that we have in the example is a single statement. Oddly, the answer is yes—they *are* strictly needed. The only place in C where you can't put a single statement but *must* have a compound statement is when you are defining a function. The simplest function of all is therefore the empty function, which does nothing at all:

```
void do_nothing(void){}
```

The statement inside show_message is a call of the library function `printf`. `printf` is used to format and print things, this example being one of the simplest of its uses. `printf` takes one or more arguments, whose values are passed forward from the point of the call into the function itself. In this case the argument is a *string*. The contents of the string are interpreted by `printf` and used to control the way the values of the other arguments are printed. It bears a little resemblance to the FORMAT statement in FORTRAN; but not enough to predict how to use it.

Summary

Declarations are used to introduce the name of a function, its return type and the type (if any) of its arguments.

A function *definition* is a declaration with the body of the function given too.

A function returning no value should have its type declared as `void`. For example, `void func(/* list of arguments */);`

A function taking no arguments should be declared with `void` as its argument list. For example, `void func(void);`

1.3.5. Strings

In C, strings are a sequence of characters surrounded by quote marks:

```
"like this"
```

Because a string is a single element, a bit like an identifier, it is not allowed to continue across a line—although space or tab characters are permitted inside a string.

```
"This is a valid string"
"This has a newline in it
and is NOT a valid string"
```

To get a very long string there are two things that you can do. You could take advantage of the fact that absolutely everywhere in a C program, the sequence 'backslash end-of-line' disappears totally.

```
"This would not be valid but doesn't have \
a newline in it as far as the compiler is concerned"
```

The other thing you could do is to to use the string joining feature, which says that two adjacent strings are considered to be just one.

```
"All this " "comes out as "
"just one string"
```

Back to the example. The sequence '`\n`' in the string is an example of an *escape* sequence which in this case represents 'newline'. `Printf` simply prints the contents of the string on the program's output file, so the output will read 'hello', followed by a new line.

To support people working in environments that use character sets which are 'wider' than U.S. ASCII, such as the shift-JIS representation used in Japan, the Standard now allows *multibyte characters* to be present in strings and comments. The Standard defines the 96 characters that are the alphabet of C (see Chapter 2 [*http://publications.gbdirect.co.uk/c_book/chapter2/*]). If your system supports an extended character set, the only place that you may use these extended characters is in strings, character constants, comment and the names of *header files*. Support for extended character sets is an implementation defined feature, so you will have to look it up in your system's documentation.

1.3.6. The main function

In Example 1.1 there are actually two functions, `show_message` and `main`. Although main is a bit longer than `show_message` it is obviously built in the same shape: it has a name, the parentheses () are there, followed by the opening bracket `{` of the compound statement that must follow in a function definition. True, there's a lot more stuff too, but right at the end of the example you'll find the matching closing bracket `}` that goes with the first one to balance the numbers.

This is a much more realistic function now, because there are several statements inside the function body, not just one. You might also have noticed that the function is *not* declared to be `void`. There is a good reason for this: it returns a proper value. Don't worry about its arguments yet; they are discussed in Chapter 10 [*http://publications.gbdirect.co.uk/c_book/chapter10/*].

The most important thing about `main` is that it is the first function to be called. In a hosted environment your C language system arranges, magically, for a call on the `main` function (hence its name) when the program is first started. When the function is over, so is the program. It's obviously an important function. Equally important is the stuff *inside* `main`'s compound statement. As mentioned before, there can be several statements inside a compound statement, so let's look at them in turn.

1.3.7. Declarations

The first statement is this:

```
int count;
```

which is not an instruction to do anything, but simply introduces a variable to the program. It declares something whose name is `count`, and whose type is ‘integer’; in C the keyword that declares integers is unaccountably shortened to `int`. C has an idiosyncratic approach to these keywords with some having their names spelled in full and some being shortened like `int`. At least `int` has a meaning that is more or less intuitive; just wait until we get on to `static`.

As a result of that declaration the compiler now knows that there is something that will be used to store integral quantities, and that its name is `count`. In C, all variables must be declared before they are used; there is none of FORTRAN's implicit declarations. In a compound statement, all the declarations must come first; they must precede any ‘ordinary’ statements and are therefore somewhat special.

(Note for pedants: unless you specifically ask, the declaration of a variable like `count` is also a *definition*. The distinction will later be seen to matter.)

1.3.8. Assignment statement

Moving down the example we find a familiar thing, an *assignment statement*. This is where the first value is assigned to the variable `count`, in this case the value assigned is a constant whose value is zero. Prior to the assignment, the value of `count` was undefined and unsafe to use. You might be a little surprised to find that the assignment symbol (strictly speaking an *assignment operator*) is a single `=` sign. This is not fashionable in modern languages, but hardly a major blemish.

So far then, we have declared a variable and assigned the value of zero to it. What next?

1.3.9. The while statement

Next is one of C's loop control statements, the while statement. Look carefully at its form. The formal description of the `while` statement is this:

```
while(expression)
        statement
```

Is that what we have got? Yes it is. The bit that reads

```
count < 10
```

is a *relational expression*, which is an example of a valid expression, and the expression is followed by a compound statement, which is a form of valid statement. As a result, it fits the rules for a properly constructed `while` statement.

What it does must be obvious to anyone who has written programs before. For as long as the relationship `count < 10` holds true, the body of the loop is executed and the comparison repeated. If the program is ever to end, then the body of the loop must do something that will eventually cause the comparison to be false: of course it does.

There are just two statements in the body of the loop. The first one is a function call, where the function `show_message` is invoked. A function call is indicated by the name of the function followed by the parentheses `()` which contain its argument list—if it takes no arguments, then you provide none. If there were any arguments, they would be put between the parentheses like this:

```
/* call a function with several arguments */
function_name(first_arg, second_arg, third_arg);
```

and so on. The call of `printf` is another example. More is explained in Chapter 4 [*http://publications.gbdirect.co.uk/c_book/chapter4/*].

The last statement in the loop is another assignment statement. It adds one to the variable `count`, so that the requirement for program to stop will eventually be met.

1.3.10. The return statement

The last statement that is left to discuss is the `return` statement. As it is written, it looks like another function call, but in fact the rule is that the statement is written

```
return expression;
```

where the *expression* is optional. The example uses a common stylistic convention and puts the *expression* into parentheses, which has no effect whatsoever.

The return causes a value to be returned from the current function to its caller. If the expression is missing, then an unknown value is passed back to the caller—this is almost certainly a mistake unless the function returns `void`. `Main` wasn't declared with any type at all, unlike `show_message`, so what type of value does it return? The answer is `int`. There are a number of places where the language allows you to declare things by default: the default type of functions is `int`, so it is common to see them used in this way. An equivalent declaration for `main` would have been

```
int main(){
```

and exactly the same results would have occurred.

You can't use the same feature to get a default type for variables because their types must be provided explicitly.

What does the value returned from `main` mean, and where does it go? In Old C, the value was passed back to the operating system or whatever else was used to start the program running. In a UNIX-like environment, the value of `0` meant ‘success’ in some way, any other value (often `-1`) meant ‘failure’. The Standard has enshrined this, stating that `0` stands for correct termination of the program. This *does not* mean that 0 is to be passed back to the host environment, but whatever is the appropriate ‘success’ value for that system. Because there is sometimes confusion around this, you may prefer to use the defined values `EXIT_SUCCESS` and `EXIT_FAILURE` instead, which are defined in the header file `<stdlib.h>`. Returning from the `main` function is the same as calling the library function `exit` with the return value as an argument. The difference is that exit may be called from *anywhere* in the program, and terminates it at that point, after doing some tidying up activities. If you intend to use `exit`, you *must* include the header file `<stdlib.h>`. From now on, we shall use `exit` rather than returning from `main`.

Summary

The `main` function returns an `int` value.

Returning from `main` is the same as calling the `exit` function, but `exit` can be called from anywhere in a program.

Returning `0` or `EXIT_SUCCESS` is the way of indicating success, anything else indicates failure.

1.3.11. Progress so far

This example program, although short, has allowed us to introduce several important language features, amongst them:

- Program structure
- Comment
- File inclusion
- Function definition
- Compound statements
- Function calling
- Variable declaration
- Arithmetic
- Looping

although of course none of this has been covered rigorously.

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter1/functions.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter1/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter1/some_more_programs.html*]

1.4. Some more programs

While we're still in the informal phase, let's look at two more examples. You will have to work out for yourself what some of the code does, but as new or interesting features appear, they will be explained.

1.4.1. A program to find prime numbers

```
/*
*
* Dumb program that generates prime numbers.
*/
#include <stdio.h>
#include <stdlib.h>

main(){
    int this_number, divisor, not_prime;

    this_number = 3;

    while(this_number < 10000){
            divisor = this_number / 2;
            not_prime = 0;
            while(divisor > 1){
                    if(this_number % divisor == 0){
                            not_prime = 1;
                            divisor = 0;
                    }
                    else
                            divisor = divisor-1;
            }

            if(not_prime == 0)
                    printf("%d is a prime number\n", this_number);
            this_number = this_number + 1;
    }
    exit(EXIT_SUCCESS);
}
```

Example 1.2

What was interesting in there? A few new points, perhaps. The program works in a really stupid way: to see if a number is prime, it divides that number by all the numbers between half its value and two—if any divide without remainder, then the number isn't prime. The two operators that you haven't seen before are the remainder operator `%`, and the equality operator, which is a double equal sign `==`. That last one is without doubt the cause of more bugs in C programs than any other single factor.

The problem with the equality test is that wherever it can appear it is also legal to put the single `=` sign. The first, `==`, compares two things to see if they are equal, and is generally what you need in fragments like these:

```
if(a == b)
while (c == d)
```

The assignment operator `=` is, perhaps surprisingly, also legal in places like those, but of course it assigns the value of the right-hand expression to whatever is on the left. The problem is particularly bad if you are used to the languages where comparison for equality is done with what C uses for assignment. There's nothing that you can do to help, so start getting used to it now. (Modern compilers do tend to produce warnings when they think they have detected ‘questionable’ uses of assignment operators, but that is a mixed blessing when your choice was deliberate.)

There is also the introduction for the first time of the `if` statement. Like the `while` statement, it tests an expression to see if the expression is true. You might have noticed that also like the `while` statement, the expression that controls the `if` statement is in parentheses. That is always the case: all of the conditional control of flow statements require a parenthesized expression after the keyword that introduces them. The formal description of the `if` statement goes like this:

```
if(expression)
        statement

if(expression)
        statement
else
        statement
```

showing that it comes in two forms. Of course, the effect is that if the expression part is evaluated to be true, then the following statement is executed. If the evaluation is false, then the following statement is not executed. When there is an `else` part, the statement associated with it is executed only if the evaluation gives a false result.

`If` statements have a famous problem. In the following piece of code, is the *statement-2* executed or not?

```
if(1 > 0)
        if(1 < 0)
                statement-1
else
        statement-2
```

The answer is that it *is*. Ignore the indentation (which is misleading). The `else` could belong to either the first or second `if`, according to the description of the `if` statement that has just been given, so an extra rule is needed to make it unambiguous. The rule is simply that an `else` is associated with the nearest `else`-less `if` above it. To make the example work the way that the indentation implied, we have to invoke a compound statement:

```
if(1 > 0){
        if(1 < 0)
                statement-1
}
else
```

```
        statement-2
```

Here, at least, C adheres to the practice used by most other languages. In fact a lot of programmers who are used to languages where the problem exists have never even realized that it is there—they just thought that the disambiguating rule was ‘obvious’. Let's hope that everyone feels that way.

1.4.2. The division operators

The division operators are the division operator `/`, and the remainder operator `%`. Division does what you would expect, except that when it is applied to integer operands it gives a result that is truncated towards zero. For example, `5/2` gives `2`, `5/3` gives `1`. The remainder operator is the way to get the truncated remainder. `5%2` gives `1`, `5%3` gives `2`. The signs of the remainder and quotient depend on the divisor and dividend in a way that is defined in the Standard and shown in Chapter 2 [*http://publications.gbdirect.co.uk/c_book/chapter2/*].

1.4.3. An example performing input

It's useful to be able to perform input as well as to write programs that print out more or less interesting lists and tables. The simplest of the library routines (and the only one that we'll look at just now) is called `getchar`. It reads single characters from the program's input and returns an integer value. The value returned is a coded representation for that character and can be used to print the same character on the program output. It can also be compared against character constants or other characters that have been read, although the only test that makes sense is to see if both characters are the same. Comparing for greater or less than each other is not portable in general; there is no guarantee that `'a'` is less than `'b'`, although on most common systems that would be the case. The only guarantee that the Standard makes is that the codes for `'0'` through to `'9'` will always be consecutive. Here is one example.

```
#include <stdio>
#include <stdlib.h>
main(){
        int ch;

        ch = getchar();
        while(ch != 'a'){
                if(ch != '\n')
                        printf("ch was %c, value %d\n", ch, ch);
                ch = getchar();
        }
        exit(EXIT_SUCCESS);
}
```

Example 1.3

There are two interesting points in there. The first is to notice that at the end of each line of input read, the character represented by

```
'\n'
```

(a character constant) will be seen. This just like the way that the same symbol results in a new line when `printf` prints it. The model of I/O used by C is not based on a line by line view of the world, but character by character instead; if you choose to think in a line-oriented way, then `'\n'` allows you to mark the end of each 'line'. Second is the way that `%c` is used to output a character by `printf`, when it appears on the output as a character. Printing it with `%d` prints the same variable, but displays the integer value used by your program to represent the character.

If you try that program out, you may find that some systems do not pass characters one by one to a program, but make you type a whole line of input first. Then the whole line is made available as input, one character at a time. Beginners have been known to be confused: the program is started, they type some input, and nothing comes back. This behaviour is nothing to do with C; it depends on the computer and operating system in use.

1.4.4. Simple arrays

The use of *arrays* in C is often a problem for the beginner. The declaration of arrays isn't too difficult, especially the one-dimensional ones, but a constant source of confusion is the fact that their indices always count from 0. To declare an array of 5 `int`s, the declaration would look like this:

```
int something[5];
```

In array declarations C uses square brackets, as you can see. There is no support for arrays with indices whose ranges do not start at 0 and go up; in the example, the valid array elements are `something[0]` to `something[4]`. Notice very carefully that `something[5]` is *not* a valid array element.

This program reads some characters from its input, sorts them into the order suggested by their representation, then writes them back out. Work out what it does for yourself; the algorithm won't be given much attention in the explanation which follows.

```
#include <stdio>
#include <stdlib.h>
#define ARSIZE  10
main(){
        int ch_arr[ARSIZE],count1;
        int count2, stop, lastchar;

        lastchar = 0;
        stop = 0;
        /*
         * Read characters into array.
         * Stop if end of line, or array full.
         */
        while(stop != 1){
                ch_arr[lastchar] = getchar();
                if(ch_arr[lastchar] == '\n')
                        stop = 1;
                else
                        lastchar = lastchar + 1;
                if(lastchar == ARSIZE)
                        stop = 1;
        }
        lastchar = lastchar-1;
        /*
         * Now the traditional bubble sort.
         */
        count1 = 0;
        while(count1 < lastchar){
                count2 = count1 + 1;
                while(count2 <= lastchar){
                        if(ch_arr[count1] > ch_arr[count2]){
                                /* swap */
                                int temp;
                                temp = ch_arr[count1];
                                ch_arr[count1] = ch_arr[count2];
                                ch_arr[count2] = temp;
                        }
                        count2 = count2 + 1;
                }
                count1 = count1 + 1;
        }

        count1 = 0;
        while(count1 <= lastchar){
                printf("%c\n", ch_arr[count1]);
                count1 = count1 + 1;
        }
        exit(EXIT_SUCCESS);
}
```

Example 1.4

You might note that the defined constant `ARSIZE` is used everywhere instead of the actual array size. Because of that, to change the maximum number of characters that can be sorted by this program simply involves a change to one line and then re-compiling. Not so obvious but critical to the safety of the program is the detection of the array becoming full. Look carefully; you'll find that the program stops when element `ARSIZE-1` has been filled. That is because in an `N` element array, only elements `0` through to `N-1` are available (giving `N` in total).

Unlike some other languages it is unlikely that you will be told if you ‘run off’ the end of an array in C. It results in what is known as *undefined behaviour* on the part of your program, this generally being to produce obscure errors in the future. Most skilled programmers avoid this happening by rigorous testing to make sure either that it can't happen given the particular algorithm in use, or by putting in an explicit test before accessing a particular member of an array. This is a common source of run-time errors in C; you have been warned.

Summary

Arrays always number from `0`; you have no choice.

A `n`-element array has members which number from `0` to `n-1` only. Element `n` does not exist and to access it is a big mistake.

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter1/description_of_example.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter1/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter1/terminology.html*]

1.5. Terminology

In C programs there are two distinct types of things: things used to hold values and things that are functions. Instead of having to refer to them jointly with a clumsy phrase that maintains the distinction, we think that it's useful to call them both loosely ‘objects’. We do quite a lot of that later, because it's often the case that they follow more or less the same rules. Beware though, that this isn't quite what the Standard uses the term to mean. In the Standard, an ‘object’ is explicitly a region of allocated storage that is used to represent a value and a function is something different; this leads to the Standard often having to say ‘… functions and objects …’. Because we don't think that it leads to too much confusion and does improve the readability of the text in most cases, we will continue to use our looser interpretation of object to include functions and we will explicitly use the terms ‘data objects’ and ‘functions’ when the distinction is appropriate.

Be prepared to find this slight difference in meaning if you do read the Standard.

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter1/some_more_programs.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter1/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter1/summary.html*]

1.6. Summary

This chapter has introduced many of the basics of the language although informally. Functions, in particular, form the basic building block for C. Chapter 4 [*http://publications.gbdirect.co.uk/c_book/chapter4/*] provides a full description of these fundamental objects, but you should by now understand enough about them to follow their informal use in the intervening material.

Although the idea of library functions has been introduced, it has not been possible to illustrate the extent of their importance to the C application programmer. The Standard Library, described in Chapter 9 [*http://publications.gbdirect.co.uk/c_book/chapter9/*], is *extremely* important, both in the way that it helps to improve the portability of programs intended for general use and also in the aid to productivity that these useful functions can provide.

The use of variables, expressions and arithmetic are soon to be described in great detail. As this chapter has shown, at a simple level, C differs little from most other modern programming languages.

Only the use of structured data types still remains to be introduced, although arrays have had a very brief airing.

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter1/terminology.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter1/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter1/exercises.html*]

1.7. Exercises

Exercise 1.1. Type in and test Example 1.1 on your system.

Exercise 1.2. Using Example 1.2 as a pattern, write a program that prints prime pairs — a pair of prime numbers that differ by 2, for example 11 and 13, 29 and 31. (If you can detect a pattern between such pairs, congratulations! You are either a genius or just wrong.)

Exercise 1.3. Write a function that returns an integer: the decimal value of a string of digits that it reads using `getchar`. For example, if it reads 1 followed by 4 followed by 6, it will return the number 146. You may make the assumption that the digits 0–9 are consecutive in the computer's representation (the Standard says so) and that the function will only have to deal with valid digits and newline, so error checking is not needed.

Exercise 1.4. Use the function that you just wrote to read a sequence of numbers. Put them into an array declared in main, by repeatedly calling the function. Sort them into ascending numerical order, then print the sorted list.

Exercise 1.5. Again using the function from Exercise 1.3, write a program that will read numbers from its input, then print them out in binary, decimal and hexadecimal form. You should not use any features of `printf` apart from those mentioned in this chapter (especially the hexadecimal output format!). You are expected to work out what digits to print by calculating each one in turn and making sure that they are printed in the right order. This is not particularly difficult, but it is not trivial either.

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter1/summary.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter1/*]

Chapter 2

<gbdirect>

Variables and Arithmetic

Previous chapter [*http://publications.gbdirect.co.uk/c_book/chapter1/*] | Next chapter [*http://publications.gbdirect.co.uk/c_book/chapter3/*]

2.1. Some fundamentals

Here is where we start to look in detail at the bits that the last chapter chose to sweep under the carpet while it did its ‘Instant C’ introduction. The problem is, of course, the usual one of trying to introduce enough of the language to let you get a feel for what it's all about, without drowning beginners in a froth of detail that isn't essential at the time.

Because this is a lengthy chapter, and because it deliberately chooses to cover some subtle problems that are often missed out in introductory texts, you should make sure that you are in the right mood and proper frame of mind to read it.

The weary brain may find that the breaks for exercises are useful. We strongly recommend that you do actually attempt the exercises on the way through. They help to balance the weight of information, which otherwise turns into an indigestible lump.

It's time to introduce some of the fundamentals.

Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter2/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter2/alphabet_of_c.html*]

2.2. The alphabet of C

<gbdirect>

This is an interesting area; alphabets are important. All the same, this is the one part of this chapter that you can read superficially first time round without missing too much. Read it to make sure that you've seen the contents once, and make a mental note to come back to it later on.

2.2.1. Basic Alphabet

Few computer languages bother to define their alphabet rigorously. There's usually an assumption that the English alphabet augmented by a sprinkling of more or less arbitrary punctuation symbols will be available in every environment that is trying to support the language. The assumption is not always borne out by experience. Older languages suffer less from this sort of problem, but try sending C programs by Telex or restrictive e-mail links and you'll understand the difficulty.

The Standard talks about two different character sets: the one that programs are written in and the one that programs execute with. This is basically to allow for different systems for compiling and execution, which might use different ways of encoding their characters. It doesn't actually matter a lot except when you are using character constants in the preprocessor, where they may not have the same value as they do at execution time. This behaviour is implementation-defined, so it must be documented. Don't worry about it yet.

The Standard requires that an alphabet of 96 symbols is available for C as follows:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9
! " # % & ' ( ) * + , - . /
: ; < = > ? [ \ ] ^ _ { | } ~
```

space, horizontal and vertical tab

form feed, newline

Table 2.1. The Alphabet of C

It turns out that most of the commonly used computer alphabets contain all the symbols that are needed for C with a few notorious exceptions. The C alphabetic characters shown below are missing from the International Standards Organization ISO 646 standard 7-bit character set, which is as a subset of all the widely used computer alphabets.

```
# [ \ ] ^ { | } ~
```

To cater for systems that can't provide the full 96 characters needed by C, the Standard specifies a method of using the ISO 646 characters to represent the missing few; the technique is the use of *trigraphs*.

2.2.2. Trigraphs

Trigraphs are a sequence of three ISO 646 characters that get treated as if they were one character in the C alphabet; all of the trigraphs start with two question marks `??` which helps to indicate that ‘something funny’ is going on. Table 2.1 below shows the trigraphs defined in the Standard.

C character	Trigraph
`#`	`??=`
`[`	`??(`
`]`	`??)`
`{`	`??<`
`}`	`??>`
`\`	`??/`
`|`	`??!`
`~`	`??-`
`^`	`??'`

Table 2.2. Trigraphs

As an example, let's assume that your terminal doesn't have the `#` symbol. To write the preprocessor line

```
#define MAX     32767
```

isn't possible; you must use trigraph notation instead:

```
??=define MAX   32767
```

Of course trigraphs will work even if you do have a `#` symbol; they are there to help in difficult circumstances more than to be used for routine programming.

The `?` ‘binds to the right’, so in any sequence of repeated `?`s, only the two at the right could possibly be part of a trigraph, depending on what comes next—this disposes of any ambiguity.

It would be a mistake to assume that programs written to be highly portable would use trigraphs ‘in case they had to be moved to systems that only support ISO 646’. If your system can handle all 96 characters in the C alphabet, then that is what you should be using. Trigraphs will only be seen in restricted environments, and it is extremely simple to write a character-by-character translator between the two representations. However, all compilers that conform to the Standard will recognize trigraphs when they are seen.

Trigraph substitution is the very first operation that a compiler performs on its input text.

2.2.3. Multibyte Characters

Support for multibyte characters is new in the Standard. Why?

A very large proportion of day-to-day computing involves data that represents text of one form or another. Until recently, the rather chauvinist computing idustry has assumed that it is adequate to provide support for about a hundred or so printable characters (hence the 96 character alphabet of C), based on the requirements of the English language—not suprising, since the bulk of the development of commercial computing has been in the US market. This alphabet (technically called the *repertoire*) fits conveniently into 7 or 8 bits of storage, which is why the US-ASCII character set standard and the architecture of mini and microcomputers both give very heavy emphasis to the use of 8-bit bytes as the basic unit of storage.

C also has a byte-oriented approach to data storage. The smallest individual item of storage that can be directly used in C is the byte, which is defined to be at least 8 bits in size. Older systems or architectures that are not designed explicitly to support this may incur a performance penalty when running C as a result, although there are not many that find this a big problem.

Perhaps there was a time when the English alphabet was acceptable for data processing applications worldwide—when computers were used in environments where the users could be expected to adapt—but those days are gone. Nowadays it is absolutely essential to provide for the storage and processing of textual material in the native alphabet of whoever wants to use the system. Most of the US and Western European language requirements can be squeezed together into a character set that still fits in 8 bits per character, but Asian and other languages simply cannot.

There are two general ways of extending character sets. One is to use a fixed number of bytes (often two) for every character. This is what the wide character support in C is designed to do. The other method is to use a shift-in shift-out coding scheme; this is popular over 8-bit communication links. Imagine a stream of characters that looks like:

```
a b c <SI> a b g <SO> x y
```

where `<SI>` and `<SO>` mean 'switch to Greek' and 'switch back to English' respectively. A display device that agreed to use that method might well then display a, b, c, alpha, beta, gamma, x and y. This is roughly the scheme used by the shift-JIS Japanese standard, except that once the shift-in has been seen, *pairs* of characters together are used as the code for a single Japanese character. Alternative schemes exist which use more than one shift-in character, but they are less common.

The Standard now allows explicitly for the use of extended character sets. Only the 96 characters defined earlier are used for the C part of a program, but in comments, strings, character constants and header names (these are really data, not part of the program as such) extended characters are permitted if your environment supports them. The Standard lays down a number of pretty obvious rules about how you are allowed to use them which we will not repeat here. The most significant one is that a byte whose value is zero is interpreted as a *null character* irrespective of any shift state. That is important, because C uses a null character to indicate the end of strings and many library functions rely on it. An additional requirement is that multibyte sequences must start and end in the initial shift state.

The `char` type is specified by the Standard as suitable to hold the value of all of the characters in the 'execution character set', which will be defined in your system's documentation. This means that (in the example above) it could hold the value of '`a`' or '`b`' or even the "switch to Greek" character itself. Because of the shift-in shift-out mechanism, there would be no difference between the value stored in a char that was intended to represent '`a`' or the Greek 'alpha' character. To do that would mean using a different representation - probably needing more than 8 bits, which on many systems would be too big for a `char`. That is why the Standard introduces the `wchar_t`type. To use this, you must include the <stddef.h> header, because `wchar_t` is simply defined as an alternative name for one of C's other types. We discuss it further in Section 2.8 [*http://publications.gbdirect.co.uk/c_book/chapter2/expressions_and_arithmetic.html*].

Summary

- C requires at least 96 characters in the source program character set.
- Not all character sets in common use can stretch to 96 characters, trigraphs allow the basic ISO 646 character set to be used (at a pinch).
- Multibyte character support has been added by the Standard, with support for
  - Shift-encoded multibyte characters, which can be squeezed into 'ordinary'

character arrays, so still have `char` type.
  - Wide characters, each of which may use more storage than a regular character. These usually have a different type from `char`.

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter2/fundamentals.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter2/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter2/textual_program_structure.html*]

2.3. The Textual Structure of Programs

<gbdirect>

2.3.1. Program Layout

The examples so far have used the sort of indentation and line layout that is common in languages belonging to the same family as C. They are ‘free format’ languages and you are expected to use that freedom to lay the program out in a way that enhances its readability and highlights its logical structure. Space (including horizontal tab) characters can be used for indentation anywhere except in identifiers or keywords without any effect on the meaning of the program. New lines work in the same way as space and tab *except* on preprocessor command lines, which have a line-by-line structure.

If a line is getting too long for comfort there are two things you can do. Generally it will be possible to replace one of the spaces by a newline and use simply two lines instead, as this example shows.

```
/* a long line */
a = fred + bill * ((this / that) * sqrt(3.14159));
/* the same line */
a = fred + bill *
        ((this / that) *
        sqrt(3.14159));
```

If you're unlucky it may not be possible to break the lines like that. The preprocessor suffers most from the problem, because of its reliance on single-line ‘statements’. To help, it's useful to know that the sequence ‘backslash newline’ becomes invisible to the C translation system. As a result, the sequence is valid even in unusual places such as the middle of identifiers, keywords, strings and so on. Only trigraphs are processed before this step.

```
/*
 * Example of the use of line joining
 */
#define IMPORTANT_BUT_LONG_PREPROCESSOR_TEXT \
printf("this is effectively all ");\
printf("on a single line ");\
printf("because of line-joining\n");
```

The only time that you might want to use this way of breaking lines (outside of preprocessor control lines) is to prevent long strings from disappearing off the right-hand side of a program listing. New lines are *not* permitted inside strings and character constants, so you might think that the following is a good idea.

```
/* not a good way of folding a string */
printf("This is a very very very\
long string\n");
```

That will certainly work, but for strings it is preferable to make use of the string-joining feature introduced by the Standard:

```
/* This string joining will not work in Old C */
printf("This is a very very very"
       "long string\n");
```

The second example allows you to indent the continuation portion of the string without changing its meaning; adding indentation in the first example would have put the indentation into the string.

Incidentally, both examples contain what is probably a mistake. There is no space in front of the ‘long’ in the continuation string, which will contain the sequence ‘verylong’ as a result. Did you notice?

2.3.2. Comment

Comment, as has been said already, is introduced by the character pair `/*` and terminated by `*/`. It is translated into a single space wherever it occurs and so it follows exactly the same rules that spaces do. It's important to realize that it doesn't simply disappear, which it used to do in Old C, and that it is not possible to put comment into strings or character constants. Comment in such a place becomes part of the string or constant:

```
/*"This is comment"*/
"/*The quotes mean that this is a string*/"
```

Old C was a bit hazy about what the deletion of comment implied. You could argue that

```
int/**/egral();
```

should have the comment deleted and so be taken by the compiler to be a call of a function named `integral`. The Standard C rule is that comment is to be read as if were a space, so the example must be equivalent to

```
int egral();
```

which declares a function `egral` that returns type `int`.

2.3.3. Translation phases

The various character translation, line joining, comment recognition and other early phases of translation must be specified to occur in a certain order. The Standard says that the translation is to proceed as if the phases occurred in this order (there are more phases, but these are the important ones):

1. Trigraph translation.
2. Line joining.
3. Translate comment to space (but not in strings or character constants). At this stage, multiple white spaces may optionally be condensed into one.
4. Translate the program.

Each stage is completed before the next is started.

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter2/alphabet_of_c.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter2/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter2/keywords_and_identifiers.html*]

2.4. Keywords and identifiers

<gbdirect>

After covering the underlying alphabet, we can look at more interesting elements of C. The most obvious of the language elements are *keywords* and *identifiers*; their forms are identical (although their meanings are different).

2.4.1. Keywords

C keeps a small set of *keywords* for its own use. These keywords cannot be used as identifiers in the program — a common restriction with modern languages. Where users of Old C may be surprised is in the introduction of some new keywords; if those names were used as identifiers in previous programs, then the programs will have to be changed. It will be easy to spot, because it will provoke your compiler into telling you about invalid names for things. Here is the list of keywords used in Standard C; you will notice that none of them use upper-case letters.

`auto`	`double`	`int`	`struct`
`break`	`else`	`long`	`switch`
`case`	`enum`	`register`	`typedef`
`char`	`extern`	`return`	`union`
`const`	`float`	`short`	`unsigned`
`continue`	`for`	`signed`	`void`
`default`	`goto`	`sizeof`	`volatile`
`do`	`if`	`static`	`while`

Table 2.3. Keywords

The new keywords that are likely to surprise old programmers are: `const`, `signed`, `void` and `volatile` (although `void` has been around for a while). Eagle eyed readers may have noticed that some implementations of C used to use the keywords `entry`, `asm`, and `fortran`. These are not part of the Standard, and few will mourn them.

2.4.2. Identifiers

Identifier is the fancy term used to mean ‘name’. In C, identifiers are used to refer to a number of things: we've already seen them used to name variables and functions. They are also used to give names to some things we haven't seen yet, amongst which are *labels* and the ‘tags’ of *structures*, *unions*, and *enums*.

The rules for the construction of identifiers are simple: you may use the 52 upper and lower case alphabetic characters, the 10 digits and finally the underscore ‘_’, which is considered to be an alphabetic character for this purpose. The only restriction is the usual one; identifiers *must* start with an alphabetic character.

Although there is no restriction on the length of identifiers in the Standard, this is a point that needs a bit of explanation. In Old C, as in Standard C, there has *never* been any restriction on the length of identifiers. The problem is that there was never any guarantee that more than a certain number of characters would be checked when names were compared for equality—in Old C this was eight characters, in Standard C this has changed to 31.

So, practically speaking, the new limit is 31 characters—although identifiers *may* be longer, they must differ in the first 31 characters if you want to be sure that your programs are portable. The Standard allows for implementations to support longer names if they wish to, so if you do use longer names, make sure that you don't rely on the checking stopping at 31.

One of the most controversial parts of the Standard is the length of *external identifiers*. External identifiers are the ones that have to be visible outside the current source code file. Typical examples of these would be library routines or functions which have to be called from several different source files.

The Standard chose to stay with the old restrictions on these external names: they are not guaranteed to be different unless they differ from each other in the first six characters. Worse than that, upper and lower case letters may be treated the same!

The reason for this is a pragmatic one: the way that most C compilation systems work is to use operating system specific tools to bind library functions into a C program. These tools are outside the control of the C compiler writer, so the Standard has to impose realistic limits that are likely to be possible to meet. There is nothing to prevent any specific implementation from giving better limits than these, but for maximum portability the six monocase characters must be all that you expect. The Standard warns that it views both the use of only one case and any restriction on the length of external names to less than 31 characters as obsolescent features. A later standard may insist that the restrictions are lifted; let's hope that it is soon.

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter2/textual_program_structure.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter2/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter2/variable_declaration.html*]

2.5. Declaration of variables

You may remember that in Chapter 1 [*http://publications.gbdirect.co.uk/c_book/chapter1/*] we said that you have to declare the names of things before you can use them (the only exceptions to this rule are the names of functions returning `int`, because they are declared by default, and the names of *labels*). You can do it either by using a *declaration*, which introduces just the name and type of something but allocates no storage, or go further by using a *definition*, which also allocates the space used by the thing being declared.

The distinction between declaration and definition is an important one, and it is a shame that the two words sound alike enough to cause confusion. From now on they will have to be used for their formal meaning, so if you are in doubt about the differences between them, refer back to this point.

The rules about what makes a declaration into a definition are rather complicated, so they will be deferred for a while. In the meantime, here are some examples and rule-of-thumb guidelines which will work for the examples that we have seen so far, and will do for a while to come.

```
/*
* A function is only defined if its body is given
* so this is a declaration but not a definition
*/

int func_dec(void);

/*
* Because this function has a body, it is also
* a definition.
* Any variables declared inside will be definitions,
* unless the keyword 'extern' is used.
* Don't use 'extern' until you understand it!
*/

int def_func(void){
     float f_var;            /* a definition */
     int counter;            /* another definition */
     int rand_num(void);     /* declare (but not define) another function */

     return(0);
}
```

Exercise 2.1. Why are trigraphs used?

Exercise 2.2. When would you expect to find them in use, and when not?

Exercise 2.3. When is a newline not equivalent to a space or tab?

Exercise 2.4. When would you see the sequence of ‘backslash newline’ in use?

Exercise 2.5. What happens when two strings are put side by side?

Exercise 2.6. Why can't you put one piece of comment inside another one? (This prevents the technique of ‘commenting out’ unused bits of program, unless you are careful.)

Exercise 2.7. What are the longest names that may safely be used for variables?

Exercise 2.8. What is a declaration?

Exercise 2.9. What is a definition?

Now we go on to look at the *type* of variables and expressions.

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter2/keywords_and_identifiers.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter2/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter2/real_types.html*]

2.6. Real types

It's easier to deal with the real types first because there's less to say about them and they don't get as complicated as the integer types. The Standard breaks new ground by laying down some basic guarantees on the precision and range of the real numbers; these are found in the header file *float.h* which is discussed in detail in Chapter 9 [*http://publications.gbdirect.co.uk/c_book/chapter9/*]. For some users this is extremely important information, but it is of a highly technical nature and is likely only to be fully understood by numerical analysts.

The varieties of real numbers are these:

```
float
double
long double
```

Each of the types gives access to a particular way of representing real numbers in the target computer. If it only has one way of doing things, they might all turn out to be the same; if it has more than three, then C has no way of specifying the extra ones. The type `float` is intended to be the small, fast representation corresponding to what FORTRAN would call `REAL`. You would use `double` for extra precision, and `long double` for even more.

The main points of interest are that in the increasing 'lengths' of `float`, `double` and `long double`, each type must give at least the same range and precision as the previous type. For example, taking the value in a `double` and putting it into a `long double` must result in the same value.

There is no requirement for the three types of 'real' variables to differ in their properties, so if a machine only has one type of real arithmetic, all of C's three types could be implemented in the same way. None the less, the three types would be considered to be different from the point of view of type checking; it would be 'as if' they really were different. That helps when you move the program to a system where the three types really are different—there won't suddenly be a set of warnings coming out of your compiler about type mismatches that you didn't get on the first system.

In contrast to more 'strongly typed' languages, C permits expressions to mix all of the scalar types: the various flavours of integers, the real numbers and also the pointer types. When an expression contains a mixture of arithmetic (integer and real) types there are implicit conversions invoked which can be used to work out what the overall type of the result will be. These rules are quite important and are known as the *usual arithmetic conversions*; it will be worth committing them to memory later. The full set of rules is described in Section 2.8 [*http://publications.gbdirect.co.uk/c_book/chapter2/expressions_and_arithmetic.html*]; for the moment, we will investigate only the ones that involve mixing `float`, `double` and `long double` to see if they make sense.

The only time that the conversions are needed is when two different types are mixed in an expression, as in the example below:

```
int f(void){
        float f_var;
        double d_var;
        long double l_d_var;

        f_var = 1; d_var = 1; l_d_var = 1;
        d_var = d_var + f_var;
        l_d_var = d_var + f_var;
        return(l_d_var);
}
```

Example 2.1

There are a lot of forced conversions in that example. Getting the easiest of them out of the way first, let's look at the assignments of the constant value `1` to each of the variables. As the section on constants will point out, that `1` has type `int`, i.e. it is an integer, not a real constant. The assignment converts the integer value to the appropriate real type, which is easy to cope with.

The interesting conversions come next. The first of them is on the line

```
d_var = d_var + f_var;
```

What is the type of the expression involving the + operator? The answer is easy when you know the rules. Whenever two different real types are involved in an expression, the lower precision type is first implicitly converted to the higher precision type and then the arithmetic is performed at that precision. The example involves both a `double` and a `float`, so the value of `f_var` is converted to type `double` and is then added to the value of the double `d_var`. The result of the expression is naturally of type `double` too, so it is clearly of the correct type to assign to `d_var`.

The second of the additions is a little bit more complicated, but still perfectly O.K. Again, the value of `f_var` is converted and the arithmetic performed with the precision of `double`, forming the sum of the two variables. Now there's a problem. The result (the sum) is `double`, but the assignment is to a `long double`. Once again the obvious procedure is to convert the lower precision value to the higher one, which is done, and then make the assignment.

So we've taken the easy ones. The difficult thing to see is what to do when forced to assign a higher precision result to a lower precision destination. In those cases it may be necessary to lose precision, in a way specified by the implementation. Basically, the implementation must specify whether and in what way it rounds or truncates. Even worse, the destination may be unable to hold the value at all. The Standard says that in these cases loss of precision may occur; if the destination is unable to hold the necessary value—say by attempting to add the largest representable number to itself—then the behaviour is undefined, your program is faulty and you can make no predictions whatsoever about any subsequent behaviour.

It is no mistake to re-emphasize that last statement. What the Standard means by *undefined behaviour* is exactly what it says. Once a program's behaviour has entered the undefined region, absolutely anything can happen. The program might be stopped by the operating system with an appropriate message, or just as likely nothing observable would happen and the program be allowed to continue with an erroneous value stored in the variable in question. It is your responsibility to prevent your program from exhibiting undefined behaviour. Beware!

Summary of real arithmatic

- Arithmetic with any two real types is done at the highest precision of the members involved.
- Assignment involves loss of precision if the receiving type has a lower precision than the value being assigned to it.

- Further conversions are often implied when expressions mix other types, but they have not been described yet.

2.6.1. Printing real numbers

The usual output function, `printf`, can be used to format real numbers and print them. There are a number of ways to format these numbers, but we'll stick to just one for now. Table 2.4 below shows the appropriate format description for each of the real types.

Type	Format
`float`	`%f`
`double`	`%f`
`long double`	`%lf`

Table 2.4. Format codes for real numbers

Here's an example to try:

```
#include <stdio.h>
#include <stdlib.h>

#define BOILING 212     /* degrees Fahrenheit */

main(){
      float f_var; double d_var; long double l_d_var;
      int i;

      i = 0;
      printf("Fahrenheit to Centigrade\n");
      while(i <= BOILING){
              l_d_var = 5*(i-32);
              l_d_var = l_d_var/9;
              d_var = l_d_var;
              f_var = l_d_var;
              printf("%d %f %f %lf\n", i,
                      f_var, d_var, l_d_var);
              i = i+1;
      }
      exit(EXIT_SUCCESS);
}
```

Example 2.2

Try that example on your own computer to see what results you get.

Exercise 2.10. Which type of variable can hold the largest range of values?

Exercise 2.11. Which type of variable can store values to the greatest precision?

Exercise 2.12. Are there any problems possible when assigning a `float` or `double` to a `double` or `long double`?

Exercise 2.13. What could go wrong when assigning, say, a `long double` to a `double`?

Exercise 2.14. What predictions can you make about a program showing ‘undefined behaviour’?

Previous section

[*http://publications.gbdirect.co.uk/c_book/chapter2/variable_declaration.html*] |
Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter2/*] | Next section
[*http://publications.gbdirect.co.uk/c_book/chapter2/integral_types.html*]

2.7. Integral types

The real types were the easy ones. The rules for the integral types are more complicated, but still tolerable, and these rules really should be learnt. Fortunately, the only types used in C for routine data storage are the real and integer types, or *structures* and *arrays* built up from them. C doesn't have special types for character manipulation or the handling of logical (boolean) quantities, but uses the integral types instead. Once you know the rules for the reals and the integers you know them all.

We will start by looking at the various types and then the conversion rules.

2.7.1. Plain integers

There are two types (often called ‘flavours’) of integer variables. Other types can be built from these, as we'll see, but the plain undecorated `ints` are the base. The most obvious of the pair is the ‘signed’ `int`, the less obvious is its close relative, the `unsigned int`. These variables are supposed to be stored in whatever is the most convenient unit for the machine running your program. The `int` is the natural choice for undemanding requirements when you just need a simple integral variable, say as a counter in a short loop. There isn't any guarantee about the number of bits that an int can hold, except that it will *always* be 16 or more. The standard header file `<limits.h>` details the actual number of bits available in a given implementation.

Curiously, Old C had no guarantee whatsoever about the length of an `int`, but consensus and common practice has always assumed at least 16 bits.

Actually, `<limits.h>` doesn't quite specify a number of bits, but gives maximum and minimum values for an `int` instead. The values it gives are 32767 and -32767 which implies 16 bits or more, whether ones or twos complement arithmetic is used. Of course there is nothing to stop a given implementation from providing a greater range in either direction.

The range specified in the Standard for an `unsigned int` is 0 to at least 65535, meaning that it cannot be negative. More about these shortly.

If you aren't used to thinking about the number of bits in a given variable, and are beginning to get worried about the portability implications of this apparently machine-dependent concern for the number of bits, then you're doing the right thing. C takes portability seriously and actually bothers to tell you what values and ranges are guaranteed to be safe. The bitwise operators encourage you to think about the number of bits in a variable too, because they give direct access to the bits, which you manipulate one by one or in groups. Almost paradoxically, the overall result is that C programmers have a healthy awareness of portability issues which leads to more portable programs. This is *not* to say that you can't write C programs that are horribly non-portable!

2.7.2. Character variables

A bit less obvious than int is the other of the plain integral types, the `char`. It's basically just another sort of `int`, but has a different application. Because so many C programs do a lot of character handling, it's a good idea to provide a special type to help, especially if the range provided by an `int` uses up much more storage than is needed by characters. The limits file tells us that three things are guaranteed about `char` variables: they have at least 8 bits, they can store a value of at least +127, and the minimum value of a `char` is zero or lower. This means that the only guaranteed range is 0–127. Whether or not `char` variables behave as `signed` or `unsigned` types is implementation defined.

In short, a character variable will probably take less storage than an `int` and will most likely be used for character manipulation. It's still an integer type though, and can be used for arithmetic, as this example shows.

```
include <limits.h>
include <stdio.h>
include <stdlib.h>

main(){
      char c;

      c = CHAR_MIN;
      while(c != CHAR_MAX){
              printf("%d\n", c);
              c = c+1;
      }

      exit(EXIT_SUCCESS);
}
```

Example 2.3

Running that program is left as an exercise for the easily amused. If you are bothered about where `CHAR_MIN` and `CHAR_MAX` come from, find `limits.h` and read it.

Here's a more enlightening example. It uses character constants, which are formed by placing a character in single quotes:

```
'x'
```

Because of the rules of arithmetic, the type of this sort of constant turns out to be `int`, but that doesn't matter since their value is always small enough to assign them to `char` variables without any loss of precision. (Unfortunately, there is a related version where that guarantee does not hold. Ignore it for the moment.) When a character variable is printed using the `%c` format with `printf`, the appropriate character is output. You can use `%d`, if you like, to see what integer value is used to represent the character. Why `%d`? Because a `char` is just another integral type.

It's also useful to be able to read characters into a program. The library function `getchar` is used for the job. It reads characters from the program's *standard input* and returns an `int` value suitable for storing into a `char`. The int value is for one reason only: not only does getchar return all possible character values, but it also returns an *extra* value to indicate that end-of-input has been seen. The range of a `char` might not be enough to hold this extra value, so the int has to be used.

The following program reads its input and counts the number of commas and full stops that it sees. On end-of-input, it prints the totals.

```
#include <stdio.h>
#include <stdlib.h>
main(){
      int this_char, comma_count, stop_count;

      comma_count = stop_count = 0;
      this_char = getchar();
      while(this_char != EOF){
              if(this_char == '.')
                      stop_count = stop_count+1;
              if(this_char == ',')
                      comma_count = comma_count+1;
              this_char = getchar();
      }
      printf("%d commas, %d stops\n", comma_count,
                      stop_count);
      exit(EXIT_SUCCESS);
}
```

Example 2.4

The two features of note in that example were the multiple assignment to the two counters and the use of the defined constant `EOF`. `EOF` is the value returned by `getchar` on end of input (it stands for End Of File), and is defined in `<stdio.h>`. The multiple assignment is a fairly common feature of C programs.

Another example, perhaps. This will either print out the whole lower case alphabet, if your implementation has its characters stored consecutively, or something even more interesting if they aren't. C doesn't make many guarantees about the ordering of characters in internal form, so this program produces *non-portable* results!

```
#include <stdio.h>
#include <stdlib.h>
main(){
        char c;

        c = 'a';
        while(c <= 'z'){
                printf("value %d char %c\n", c, c);
                c = c+1;
        }

        exit(EXIT_SUCCESS);
}
```

Example 2.5

Yet again this example emphasizes that a `char` is only another form of integer variable and can be used just like any other form of variable. It is *not* a ‘special’ type with its own rules.

The space saving that a `char` offers when compared to an `int` only becomes worthwhile if a lot of them are being used. Most character-processing operations involve the use of not just one or two character variables, but large arrays of them. That's when the saving can become noticeable: imagine an array of 1024 `int`s. On a lot of common machines that would eat up 4096 8-bit bytes of storage, assuming the common length of 4 bytes per `int`. If the computer architecture allows it to be done in a reasonably efficient way, the C implementor will probably have arranged for `char` variables to be packed one per byte, so the array would only use 1024 bytes and the space saving would be 3072 bytes.

Sometimes it doesn't matter whether or not a program tries to save space; sometimes it does. At least C gives you the option of choosing an appropriate type.

2.7.3. More complicated types

The last two types were simple, in both their declaration and subsequent use. For serious systems programming they just aren't adequate in the precision of control over storage that they provide and the behaviour that they follow. To correct this problem, C provides extra forms of integral types, split into the categories of `signed` and `unsigned`. (Although both these terms are reserved words, they will also be used as adjectives.) The difference between the two types is obvious. Signed types are those that are capable of being negative, the unsigned types cannot be negative at any time. Unsigned types are usually used for one of two reasons: to get an extra bit of precision, or when the concept of being negative is simply not present in the data that is being represented. The latter is by far the better reason for choosing them.

Unsigned types also have the special property of never overflowing in arithmetic. Adding 1 to a signed variable that already contains the maximum possible positive number for its type will result in overflow, and the program's behaviour becomes undefined. That can never happen with unsigned types, because they are defined to work ‘modulo one greater than the maximum number that they can hold’. What this means is best illustrated by example:

```
#include <stdio.h>
#include <stdlib.h>
main(){
      unsigned int x;
      x = 0;
      while(x >= 0){
              printf("%u\n", x);
              x = x+1;
      }

      exit(EXIT_SUCCESS);
}
```

Example 2.6

Assuming that the variable `x` is stored in 16 bits, then its range of values will be 0–65535 and that sequence will be printed endlessly. The program can't terminate: the test

```
x >= 0
```

must always be true for an unsigned variable.

For both the signed and unsigned integral types there are three subtypes: `short`, ordinary and `long`. Taking those into account, here is a list of all of the possible integral types in C, except for the character types:

```
unsigned short int
unsigned int
unsigned long int
signed short int
signed int
signed long int
```

In the last three, the `signed` keyword is unnecessary because the `int` types are signed types anyway: you *have* to say `unsigned` to get anything different. It's also permissible, but not recommended, to drop the int keyword from any of those declarations provided that there is at least one other keyword present—the `int` will be ‘understood’ to be present. For example `long` is equivalent to `signed long int`. The long and short kinds give you more control over the amount of space used to store variables. Each has its own minimum range specified in `<limits.h>` which in practice means at least 16 bits in a `short` and an `int`, and at least 32 bits in a `long`, whether signed or unsigned. As always, an implementation can choose to give you more bits than the minimum if it wants to. The only restriction is that the limits must be equalled or bettered, and that you don't get more bits in a shorter type than a longer one (not an unreasonable rule).

The only character types are the `signed char` and the `unsigned char`. The difference between `char` and `int` variables is that, unless otherwise stated, all `int`s are signed. The same is not true for `char`s, which are signed or unsigned depending on the implementor's choice; the choice is presumably taken on efficiency grounds. You can of course explicitly force signed or unsignedness with the right keyword. The only time that it is likely to matter is if you are using character variables as extra short `short`s to save more space.

Summary of integral types

- The integral types are the `short`, `long`, `signed`, `unsigned` and plain `int`s.
- The commonest is the ordinary `int`, which is signed unless declared not to be.
- The `char` variables *can* be made signed or unsigned, as you prefer, but in the absence of indications to the contrary, they will be allocated the most efficient type.

2.7.4. Printing the integral types

Once again you can use `printf` to print these various types. Character variables work exactly the same way that the other integral variables do, so you can use the standard format letters to print their contents—although the actual numbers stored in them are not likely to be very interesting. To see their contents interpreted as characters, use `%c` as was done earlier. All of the integral types can be printed as if they were signed decimal numbers by using the `%d` format, or `%ld` for long types. Other useful formats are shown in Table 2.5; notice that in every case a letter ‘l’ is put in front of the normal format letter if a `long` is to be printed. That's not just there to get the right result printed: the behaviour of `printf` is undefined if the wrong format is given.

Format	Use with
`%c`	`char` (in character form)
`%d`	decimal `signed int`, `short`, `char`
`%u`	decimal `unsigned int`, `unsigned short`, `unsigned char`
`%x`	hexadecimal `int`, `short`, `char`
`%o`	octal `int`, `short`, `char`
`%ld`	decimal `signed long`
`%lu %lx %lo`	as above, but for `long`s

Table 2.5. More format codes

A full description of the format codes that you can use with printf is given in Chapter 9 [*http://publications.gbdirect.co.uk/c_book/chapter9/*].

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter2/real_types.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter2/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter2/expressions_and_arithmetic.html*]

2.8. Expressions and arithmetic

Expressions in C can get rather complicated because of the number of different types and operators that can be mixed together. This section explains what happens, but can get deep at times. You may need to re-read it once or twice to make sure that you have understood all of the points.

First, a bit of terminology. Expressions in C are built from combinations of *operators* and *operands*, so for example in this expression

```
x = a+b*(-c)
```

we have the operators `=`, `+` `*` and `-`. The operands are the variables `x`, `a`, `b` and `c`. You will also have noticed that parentheses can be used for grouping sub-expressions such as the `-c`. Most of C's unusually rich set of operators are either *binary operators*, which take two operands, or *unary operators*, which take only one. In the example, the `-` was being used as a unary operator, and is performing a different task from the binary subtraction operator which uses the same `-` symbol. It may seem like hair-splitting to argue that they are different operators when the job that they do seems conceptually the same, or at least similar. It's worth doing though, because, as you will find later, some of the operators have both a binary and a unary form where the two meanings bear no relation to each other; a good example would be the binary multiplication operator `*`, which in its unary form means indirection via a pointer variable!

A peculiarity of C is that operators may appear consecutively in expressions without the need for parentheses to separate them. The previous example could have been written as

```
x = a+b*-c;
```

and still have been a valid expression. Because of the number of operators that C has, and because of the strange way that assignment works, the *precedence* of the operators (and their *associativity*) is of much greater importance to the C programmer than in most other languages. It will be discussed fully after the introduction of the important arithmetic operators.

Before that, we must investigate the type conversions that may occur.

2.8.1. Conversions

C allows types to be mixed in expressions, and permits operations that result in type conversions happening implicitly. This section describes the way that the conversions must occur. Old C programmers should read this carefully, because the rules have changed — in particular, the promotion of `float` to `double`, the promotions of short integral types and the introduction of *value preserving* rules are genuinely different in Standard C.

Although it isn't directly relevant at the moment, we must note that the integral and the floating types are jointly known as *arithmetic types* and that C also supports other types (notably pointer types). The rules that we discuss here are appropriate only in expressions that have arithmetic types throughout - additional rules come into play when expressions mix pointer types with arithmetic types and these are discussed much later.

There are various types of conversion in arithmetic expressions:

- The *integral promotions*
- Conversions between integral types
- Conversions between floating types
- Conversions between floating and integral types

Conversions between floating (real) types were discussed in Section 2.8 [*http://publications.gbdirect.co.uk/c_book/chapter2/expressions_and_arithmetic.html*]; what we do next is to specify how the other conversions are to be performed, then look at *when* they are required. You will need to learn them by heart if you ever intend to program seriously in C.

The Standard has, among some controversy, introduced what are known as *value preserving* rules, where a knowledge of the target computer is required to work out what the type of an expression will be. Previously, whenever an unsigned type occurred in an expression, you knew that the result had to be `unsigned` too. Now, the result will only be `unsigned` if the conversions demand it; in many cases the result will be an ordinary `signed` type.

The reason for the change was to reduce some of the surprises possible when you mix signed and unsigned quantities together; it isn't always obvious when this has happened and the intention is to produce the 'more commonly required' result.

2.8.1.1. Integral promotions

No arithmetic is done by C at a precision shorter than `int`, so these conversions are implied almost whenever you use one of the objects listed below in an expression. The conversion is defined as follows:

- Whenever a `short` or a `char` (or a *bitfield* or *enumeration type* which we haven't met yet) has the integral promotions applied
  - if an `int` can hold all of the values of the original type then the value is converted to `int`
  - otherwise, the conversion will be to `unsigned int`

This preserves both the value and the sign of the original type. Note that whether a plain `char` is treated as signed or unsigned is implementation dependent.

These promotions are applied very often—they are applied as part of the *usual arithmetic conversions*, and to the operands of the shift, unary `+`, `-`, and `~` operators. They are also applied when the expression in question is an argument to a function but no type information has been provided as part of a function prototype, as explained in Chapter 4 [*http://publications.gbdirect.co.uk/c_book/chapter4/*].

2.8.1.2. Signed and unsigned integers

A lot of conversions between different types of integers are caused by mixing the various flavours of integers in expressions. Whenever these happen, the integral promotions will already have been done. For all of them, if the new type can hold all of the values of the old type, then the value remains unchanged.

When converting from a signed integer to an unsigned integer whose length is equal to or longer than the original type, then if the signed value was nonnegative, its value is unchanged. If the value was negative, then it is converted to the signed form of the longer type and then made unsigned by conceptually adding it to one greater than the maximum that can be held in the unsigned type. In a twos complement system, this preserves the original bit-pattern for positive numbers and guarantees 'sign-extension' of negative numbers.

Whenever an integer is converted into a shorter unsigned type, there can be no ‘overflow’, so the result is defined to be ‘the non-negative remainder on division by the number one greater than the largest unsigned number that can be represented in the shorter type’. That simply means that in a two's complement environment the low-order bits are copied into the destination and the high-order ones discarded.

Converting an integer to a shorter signed type runs into trouble if there is not enough room to hold the value. In that case, the result is implementation defined (although most old-timers would expect that simply the low-order bit pattern is copied).

That last item could be a bit worrying if you remember the integral promotions, because you might interpret it as follows—if I assign a `char` to another `char`, then the one on the right is first promoted to one of the kinds of `int`; could doing the assignment result in converting (say) an `int` to a char and provoking the ‘implementation defined’ clause? The answer is no, because assignment is specified not to involve the integral promotions, so you are safe.

2.8.1.3. Floating and integral

Converting a floating to an integral type simply throws away any fractional part. If the integral type can't hold the value that is left, then the behaviour is undefined—this is a sort of overflow.

As has already been said, going up the scale from `float` to `double` to `long double`, there is no problem with conversions—each higher one in the list can hold all the values of the lower ones, so the conversion occurs with no loss of information.

Converting in the opposite direction, if the value is outside the range that can be held, the behaviour is undefined. If the value *is* in range, but can't be held exactly, then the result is one of the two nearest values that *can* be held, chosen in a way that the implementation defines. This means that there will be a loss of precision.

2.8.1.4. The usual arithmetic conversions

A lot of expressions involve the use of subexpressions of mixed types together with operators such as `+`, `*` and so on. If the operands in an expression have different types, then there will have to be a conversion applied so that a common resulting type can be established; these are the conversions:

- If either operand is a `long double`, then the other one is converted to `long double` and that is the type of the result.
- Otherwise, if either operand is a `double`, then the other one is converted to `double`, and that is the type of the result.
- Otherwise, if either operand is a `float`, then the other one is converted to `float`, and that is the type of the result.
- Otherwise the integral promotions are applied to both operands and the following conversions are applied:
    - If either operand is an `unsigned long int`, then the other one is converted to `unsigned long int`, and that is the type of the result.
    - Otherwise, if either operand is a `long int`, then the other one is converted to `long int`, and that is the type of the result.
    - Otherwise, if either operand is an `unsigned int`, then the other one is converted to `unsigned int`, and that is the type of the result.
    - Otherwise, both operands must be of type `int`, so that is the type of the result.

The Standard contains a strange sentence: ‘The values of floating operands and of the results of floating expressions may be represented in greater precision and range than that required by the type; the types are not changed thereby’. This is in fact to allow the

Old C treatment of `floats`. In Old C, `float` variables were automatically promoted to `double`, the way that the integral promotions promote `char` to `int`. So, an expression involving purely `float` variables may be done as if they were `double`, but the type of the result must appear to be `float`. The only effect is likely to be on performance and is not particularly important to most users.

Whether or not conversions need to be applied, and if so which ones, is discussed at the point where each operator is introduced.

In general, the type conversions and type mixing rules don't cause a lot of trouble, but there is one pitfall to watch out for. Mixing signed and unsigned quantities is fine until the signed number is negative; then its value can't be represented in an unsigned variable and something has to happen. The standard says that to convert a negative number to unsigned, the largest possible number that can be held in the unsigned plus one is added to the negative number; that is the result. Because there can be no overflow in an unsigned type, the result always has a defined value. Taking a 16-bit `int` for an example, the unsigned version has a range of 0–65535. Converting a signed value of -7 to this type involves adding 65536, resulting in 65529. What is happening is that the Standard is enshrining previous practice, where the bit pattern in the signed number is simply assigned to the unsigned number; the description in the standard is exactly what would happen if you did perform the bit pattern assignment on a two's complement computer. The one's complement implementations are going to have to do some real work to get the same result.

Putting it plainly, a small magnitude negative number will result in a large positive number when converted to unsigned. If you don't like it, suggest a better solution—it is plainly a mistake to try to assign a negative number to an unsigned variable, so it's your own fault.

Well, it's easy to say ‘don't do it’, but it can happen by accident and the results can be *very* surprising. Look at this example.

```
#include <stdio.h>
#include <stdlib.h>
main(){
      int i;
      unsigned int stop_val;

      stop_val = 0;
      i = -10;

      while(i <= stop_val){
              printf("%d\n", i);
              i = i + 1;
      }
      exit(EXIT_SUCCESS);
}
```

Example 2.7

You might expect that to print out the list of values from `-10` to `0`, but it won't. The problem is in the comparison. The variable `i`, with a value of `-10`, is being compared against an unsigned `0`. By the rules of arithmetic (check them) we must convert both types to `unsigned int` first, then make the comparison. The `-10` becomes at least `65526` (see `<limits.h>`) when it's converted, and is plainly somewhat larger than `0`, so the loop is never executed. The moral is to steer clear of unsigned numbers unless you really have to use them, and to be perpetually on guard when they are mixed with signed numbers.

2.8.1.5. Wide characters

The Standard, as we've already said, now makes allowances for extended character sets. You can either use the shift-in shift-out encoding method which allows the multibyte charactes to be stored in ordinary C strings (which are really arrays of `chars`, as we explore later), or you can use a representation that uses more than one byte of storage per character for every character. The use of shift sequences only works if you process the characters in strict order; it is next to useless if you want to create an array of characters and access them in non-sequential order, since the actual index of each `char` in the array and the logical index of each of the encoded characters are not easily determined. Here's the illustration we used before, annotated with the actual and the logical array indexes:

```
0 1 2  3   4 5 6  7   8 9 (actual array index)
a b c <SI> a b g <SO> x y
0 1 2      3 4 5      6 7 (logical index)
```

We're still in trouble even if we do manage to use the index of `5` to access the ‘correct’ array entry, since the value retrieved is indistinguishable from the value that encodes the letter ‘`g`’ anyhow. Clearly, a better approach for this sort of thing is to come up with a distinct value for all of the characters in the character set we are using, which may involve more bits than will fit into a char, and to be able to store each one as a separate item without the use of shifts or other position-dependent techniques. That is what the `wchar_t` type is for.

Although it is always a synonym for one of the other integral types, `wchar_t` (whose definition is found in `<stddef.h>`) is defined to be the implementation-dependent type that should be used to hold extended characters when you need an array of them. The Standard makes the following guarantees about the values in a wide character:

- A `wchar_t` can hold distinct values for each member of the largest character set supported by the implementation.
- The null character has the value of zero.
- Each member of the basic character set (see Section 2.2.1 [*http://publications.gbdirect.co.uk/c_book/chapter2/alphabet_of_c.html#section-1*]) is encoded in a `wchar_t` with the same value as it has in a `char`.

There is further support for this method of encoding characters. Strings, which we have already seen, are implemented as arrays of `char`, even though they look like this:

```
"a string"
```

To get strings whose type is `wchar_t`, simply prefix a string with the letter `L`. For example:

```
L"a string"
```

In the two examples, it is very important to understand the differences. Strings are implemented as arrays and although it might look odd, it is entirely permissible to use array indexing on them:

```
"a string"[4]
L"a string"[4]
```

are both valid expressions. The first results in an expression whose type is `char` and whose value is the internal representation of the letter ‘`r`’ (remember arrays index from zero, not one). The second has the type `wchar_t` and also has the value of the internal representation of the letter ‘`r`’.

It gets more interesting if we are using extended characters. If we use the notation `<a>`, `<b>`, and so on to indicate ‘additional’ characters beyond the normal character set which are encoded using some form of shift technique, then these examples show the problems.

```
"abc<a><b>"[3]
L"abc<a><b>"[3]
```

The second one is easiest: it has a type of `wchar_t` and the appropriate internal encoding for whatever `<a>` is supposed to be—say the Greek letter alpha. The first one is unpredictable. Its type is unquestionably `char`, but its value is probably the value of the ‘shift-in’ marker.

As with strings, there are also wide character constants.

```
'a'
```

has type `char` and the value of the encoding for the letter ‘`a`’.

```
L'a'
```

is a constant of type `wchar_t`. If you use a multibyte character in the first one, then you have the same sort of thing as if you had written

```
'xy'
```

—multiple characters in a character constant (actually, this is valid but means something funny). A single multibyte character in the second example will simply be converted into the appropriate `wchar_t` value.

If you don't understand all the wide character stuff, then all we can say is that we've done our best to explain it. Come back and read it again later, when it might suddenly click. In practice it does manage to address the support of extended character sets in C and once you're used to it, it makes a lot of sense.

Exercise 2.15. Assuming that `chars`, `ints` and `longs` are respectively 8, 16 and 32 bits long, and that `char` defaults to `unsigned char` on a given system, what is the resulting type of expressions involving the following combinations of variables, after the usual arithmetic conversions have been applied?

a. Simply `signed char`.
b. Simply `unsigned char`.
c. `int`, `unsigned int`.
d. `unsigned int`, `long`.
e. `int`, `unsigned long`.
f. `char`, `long`.
g. `char`, `float`.
h. `float`, `float`.
i. `float`, `long double`.

2.8.1.6. Casts

From time to time you will find that an expression turns out not to have the type that you wanted it to have and you would like to force it to have a different type. That is what *casts* are for. By putting a type name in parentheses, for example

```
(int)
```

you create a unary operator known as a *cast*. A cast turns the value of the expression on its right into the indicated type. If, for example, you were dividing two integers `a/b` then the expression would use integer division and discard any remainder. To force the fractional part to be retained, you could either use some intermediate float variables, or a cast. This example does it both ways.

```
#include <stdio.h>
#include <stdlib.h>

/*
* Illustrates casts.
* For each of the numbers between 2 and 20,
* print the percentage difference between it and the one
* before
*/
main(){
      int curr_val;
      float temp, pcnt_diff;

      curr_val = 2;
      while(curr_val <= 20){
              /*
               * % difference is
               * 1/(curr_val)*100
               */
              temp = curr_val;
              pcnt_diff = 100/temp;
              printf("Percent difference at %d is %f\n",
                      curr_val, pcnt_diff);
              /*
               * Or, using a cast:
               */
              pcnt_diff = 100/(float)curr_val;
              printf("Percent difference at %d is %f\n",
                      curr_val, pcnt_diff);
              curr_val = curr_val + 1;
      }
      exit(EXIT_SUCCESS);
}
```

Example 2.8

The easiest way to remember how to write a cast is to write down exactly what you would use to declare a variable of the type that you want. Put parentheses around the entire declaration, then delete the variable name; that gives you the cast. Table 2.6 shows a few simple examples—some of the types shown will be new to you, but it's the complicated ones that illustrate best how casts are written. Ignore the ones that you don't understand yet, because you will be able to use the table as a reference later.

Declaration	Cast	Type
`int x;`	`(int)`	`int`
`float f;`	`(float)`	`float`
`char x[30];`	`(char [30])`	array of `char`
`int *ip;`	`(int *)`	pointer to `int`
`int (*f)();`	`(int (*)())`	pointer to function returning `int`

Table 2.6. Casts

2.8.2. Operators

2.8.2.1. The multiplicative operators

Or, put another way, multiplication `*`, division `/` and the remainder operator `%`.

Multiplication and division do what is expected of them for both real and integral types, with integral division producing a truncated result. The truncation is towards zero. The remainder operator is only defined to work with integral types, because the division of real numbers supposedly doesn't produce a remainder.

If the division is not exact and neither operand is negative, the result of `/` is positive and rounded toward zero—to get the remainder, use `%`. For example,

```
9/2 == 4
9%2 == 1
```

If either operand is negative, the result of `/` may be the nearest integer to the true result on either side, and the sign of the result of `%` may be positive or negative. Both of these features are implementation defined.

It is always true that the following expression is equal to zero:

```
(a/b)*b + a%b - a
```

unless `b` is zero.

The usual arithmetic conversions are applied to both of the operands.

2.8.2.2. Additive operators

Addition `+` and subtraction `-` also follow the rules that you expect. The binary operators and the unary operators both have the same symbols, but rather different meanings. For example, the expressions `a+b` and `a-b` both use a binary operator (the `+` or `-` operators), and result in addition or subtraction. The unary operators with the same symbols would be written `+b` or `-b`.

The unary minus has an obvious function—it takes the negative value of its operand; what does the unary plus do? In fact the answer is almost nothing. The unary plus is a new addition to the language, which balances the presence of the unary minus, but doesn't have any effect on the value of the expression. Very few Old C users even noticed that it was missing.

The usual arithmetic conversions are applied to both of the operands of the binary forms of the operators. Only the integral promotions are performed on the operands of the unary forms of the operators.

2.8.2.3. The bitwise operators

One of the great strengths of C is the way that it allows systems programmers to do what had, before the advent of C, always been regarded as the province of the assembly code programmer. That sort of code was by definition highly non-portable. As C demonstrates, there isn't any magic about that sort of thing, and into the bargain it turns out to be surprisingly portable. What is it? It's what is often referred to as ‘bit-twiddling’—the manipulation of individual bits in integer variables. None of the bitwise operators may be used on real operands because they aren't considered to have individual or accessible bits.

There are six bitwise operators, listed in Table 2.7, which also shows the arithmetic conversions that are applied.

Operator	Effect	Conversions
`&`	bitwise AND	usual arithmetic conversions
`|`	bitwise OR	usual arithmetic conversions
`^`	Bitwise XOR	usual arithmetic conversions

Operator	Effect	Conversions
`<<`	left shift	integral promotions
`>>`	right shift	integral promotions
`~`	one's complement	integral promotions

Table 2.7. Bitwise operators

Only the last, the one's complement, is a unary operator. It inverts the state of every bit in its operand and has the same effect as the unary minus on a one's complement computer. Most modern computers work with two's complement, so it isn't a waste of time having it there.

Illustrating the use of these operators is easier if we can use hexadecimal notation rather than decimal, so now is the time to see hexadecimal constants. Any number written with `0x` at its beginning is interpreted as hexadecimal; both `15` and `0xf` (or `0XF`) mean the same thing. Try running this or, better still, try to predict what it does first and then try running it.

```
#include <stdio.h>
#include <stdlib.h>

main(){
      int x,y;
      x = 0; y = ~0;

      while(x != y){
              printf("%x & %x = %x\n", x, 0xff, x&0xff);
              printf("%x | %x = %x\n", x, 0x10f, x|0x10f);
              printf("%x ^ %x = %x\n", x, 0xf00f, x^0xf00f);
              printf("%x >> 2 = %x\n", x, x >> 2);
              printf("%x << 2 = %x\n", x, x << 2);
              x = (x << 1) | 1;
      }
      exit(EXIT_SUCCESS);
}
```

Example 2.9

The way that the loop works in that example is the first thing to study. The controlling variable is `x`, which is initialized to zero. Every time round the loop it is compared against `y`, which has been set to a word-length independent pattern of all `1`s by taking the one's complement of zero. At the bottom of the loop, `x` is shifted left once and has 1 ORed into it, giving rise to a sequence that starts `0`, `1`, `11`, `111`, … in binary.

For each of the AND, OR, and XOR (exclusive OR) operators, `x` is operated on by the operator and some other interesting operand, then the result printed.

The left and right shift operators are in there too, giving a result which has the type and value of their left-hand operand shifted in the required direction a number of places specified by their right-hand operand; the type of both of the operands must be integral. Bits shifted off either end of the left operand simply disappear. Shifting by more bits than there are in a word gives an implementation dependent result.

Shifting left guarantees to shift zeros into the low-order bits.

Right shift is fussier. Your implementation is allowed to choose whether, when shifting signed operands, it performs a logical or arithmetic right shift. This means that a logical shift shifts zeros into the most significant bit positions; an arithmetic shift copies the current contents of the most significant bit back into itself. The position is clearer if an unsigned operand is right shifted, because there is no choice: it must be a logical shift. For that reason, whenever right shift is being used, you would expect to find that the thing being shifted had been declared to be unsigned, or cast to unsigned for the shift, as in the example:

```
int i,j;
i = (unsigned)j >> 4;
```

The second (right-hand) operand of a shift operator does not have to be a constant; any integral expression is legal. Importantly, the rules involving mixed types of operands do not apply to the shift operators. The result of the shift has the same type as the thing that got shifted (after the integral promotions), and depends on nothing else.

Now something different; one of those little tricks that C programmers find helps to write better programs. If for any reason you want to form a value that has `1`s in all but its least significant so-many bits, which are to have some other pattern in them, you don't have to know the word length of the machine. For example, to set the low order bits of an `int` to `0x0f0` and all the other bits to `1`, this is the way to do it:

```
int some_variable;
some_variable = ~0xf0f;
```

The one's complement of the desired low-order bit pattern has been one's complemented. That gives exactly the required result and is completely independent of word length; it is a very common sight in C code.

There isn't a lot more to say about the bit-twiddling operators, and our experience of teaching C has been that most people find them easy to learn. Let's move on.

2.8.2.4. The assignment operators

No, that isn't a mistake, ‘operators’ was meant to be plural. C has several assignment operators, even though we have only seen the plain `=` so far. An interesting thing about them is that they are all like the other binary operators; they take two operands and produce a result, the result being usable as part of an expression. In this statement

```
x = 4;
```

the value `4` is assigned to `x`. The result has the type of `x` and the value that was assigned. It can be used like this

```
a = (x = 4);
```

where `a` will now have the value `4` assigned to it, after `x` has been assigned to. All of the simpler assignments that we have seen until now (except for one example) have simply discarded the resulting value of the assignment, even though it is produced.

It's because assignment has a result that an expression like

```
a = b = c = d;
```

works. The value of `d` is assigned to `c`, the result of that is assigned to `b` and so on. It makes use of the fact that expressions involving only assignment operators are evaluated from right to left, but is otherwise like any other expression. (The rules explaining what groups right to left and vice versa are given in Table 2.9.)

If you look back to the section describing ‘conversions’, there is a description of what happens if you convert longer types to shorter types: that is what happens when the left-hand operand of an assignment is shorter than the right-hand one. No conversions are applied to the right-hand operand of the simple assignment operator.

The remaining assignment operators are the compound assignment operators. They allow a useful shorthand, where an assignment containing the same left- and right-hand sides can be compressed; for example

```
x = x + 1;
```

can be written as

```
x += 1;
```

using one of the compound assignment operators. The result is the same in each case. It is a useful thing to do when the left-hand side of the operator is a complicated expression, not just a variable; such things occur when you start to use arrays and pointers. Most experienced C programmers tend to use the form given in the second example because somehow it 'feels better', a sentiment that no beginner has ever been known to agree with. Table 2.8 lists the compound assignment operators; you will see them used a lot from now on.

`*=`	`/=`	`%=`
`+=`	`-=`	
`&=`	`|=`	`^=`
`>>=`	`<<=`	

Table 2.8. Compound assignment operators

In each case, arithmetic conversions are applied as if the expression had been written out in full, for example as if `a+=b` had been written `a=a+b`.

Reiterating: the result of an assignment operator has both the value and the type of the object that was assigned to.

2.8.2.5. Increment and decrement operators

It is so common to simply add or subtract 1 in an expression that C has two special unary operators to do the job. The increment operator `++` adds `1`, the decrement `--` subtracts `1`. They are used like this:

```
x++;
++x;
x--;
--x;
```

where the operator can come either before or after its operand. In the cases shown it doesn't matter where the operator comes, but in more complicated cases the difference has a definite meaning and must be used properly.

Here is the difference being used.

```
#include <stdio.h>
#include <stdlib.h>
main(){
      int a,b;
      a = b = 5;
      printf("%d\n", ++a+5);
      printf("%d\n", a);
      printf("%d\n", b++ +5);
      printf("%d\n", b);
      exit(EXIT_SUCCESS);
}
```

Example 2.10

The results printed were

```
11
6
10
6
```

The difference is caused by the different positions of the operators. If the inc/decrement operator appears in front of the variable, then its value is changed by one and the *new* value is used in the expression. If the operator comes after the variable, then the *old* value is used in the expression and the variable's value is changed afterwards.

C programmers never add or subtract one with statements like this

```
x += 1;
```

they invariably use one of

```
x++; /* or */ ++x;
```

as a matter of course. A warning is in order though: it is not safe to use a variable more than once in an expression if it has one of these operators attached to it. There is no guarantee of when, within an expression, the affected variable will actually change value. The compiler might choose to 'save up' all of the changes and apply them at once, so an expression like this

```
y = x++ + --x;
```

does not guarantee to assign twice the original value of `x` to `y`. It might be evaluated as if it expanded to this instead:

```
y = x + (x-1);
```

because the compiler notices that the overall effect on the value of `x` is zero.

The arithmetic is done exactly as if the full addition expression had been used, for example `x=x+1`, and the usual arithmetic conversions apply.

Exercise 2.16. Given the following variable definitions

```
int i1, i2;
float f1, f2;
```

a. How would you find the remainder when `i1` is divided by `i2`?
b. How would you find the remainder when `i1` is divided by the value of `f1`, treating `f1` as an integer?
c. What can you predict about the sign of the remainders calculated in the previous two questions?
d. What meanings can the `-` operator have?
e. How would you turn off all but the low-order four bits in `i1`?
f. How would you turn on all the low-order four bits in `i1`?
g. How would you turn off only the low-order four bits in `i1`?
h. How would you put into `i1` the low-order 8 bits in `i2`, but swapping the significance of the lowest four with the next
i. What is wrong with the following expression?

   ```
   f2 = ++f1 + ++f1;
   ```

2.8.3. Precedence and grouping

After looking at the operators we have to consider the way that they work together. For things like addition it may not seem important; it hardly matters whether

```
a + b + c
```

is done as

```
(a + b) + c
```

or

```
a + (b + c)
```

does it? Well, yes in fact it does. If `a+b` would overflow and `c` held a value very close to `-b`, then the second grouping might give the correct answer where the first would cause undefined behaviour. The problem is much more obvious with integer division:

```
a/b/c
```

gives very different results when grouped as

```
a/(b/c)
```

or

```
(a/b)/c
```

If you don't believe that, try it with `a=10`, `b=2`, `c=3`. The first gives `10/(2/3)`; `2/3` in integer division gives `0`, so we get `10/0` which immediately overflows. The second grouping gives `(10/2)`, obviously `5`, which divided by `3` gives `1`.

The grouping of operators like that is known as *associativity*. The other question is one of *precedence*, where some operators have a higher priority than others and force evaluation of sub-expressions involving them to be performed before those with lower precedence operators. This is almost universal practice in high-level languages, so we ‘know’ that

```
a + b * c + d
```

groups as

```
a + (b * c) + d
```

indicating that multiplication has higher precedence than addition.

The large set of operators in C gives rise to 15 levels of precedence! Only very boring people bother to remember them all. The complete list is given in Table 2.9, which indicates both precedence and associativity. Not all of the operators have been mentioned yet. Beware of the use of the same symbol for both unary and binary operators: the table indicates which are which.

Operator	Direction	Notes
`() [] -> .`	left to right	1
`! ~ ++ -- - + (cast) * & sizeof`	right to left	all unary
`* / %`	left to right	binary
`+ -`	left to right	binary
`<< >>`	left to right	binary
`< <= > >=`	left to right	binary

Operator	Direction	Notes
`==` `!=`	left to right	binary
`&`	left to right	binary
`^`	left to right	binary
`|`	left to right	binary
`&&`	left to right	binary
`||`	left to right	binary
`?:`	right to left	2
`=` `+=` and all combined assignment	right to left	binary
`,`	left to right	binary

1. Parentheses are for expression grouping, *not* function call.

2. This is unusual. See Section 3.4.1 [*http://publications.gbdirect.co.uk/c_book/chapter3/strange_operators.html#section-1*].

Table 2.9. Operator precedence and associativity

The question is, what can you do with that information, now that it's there? Obviously it's important to be able to work out both how to write expressions that evaluate in the proper order, and also how to read other people's. The technique is this: first, identify the unary operators and the operands that they refer to. This isn't such a difficult task but it takes some practice, especially when you discover that operators such as unary `*` can be applied an arbitrary number of times to their operands; this expression

```
a*****b
```

means `a` multiplied by *something*, where the something is an expression involving `b` and several unary `*` operators.

It's not too difficult to work out which are the unary operators; here are the rules.

1. `++` and `-` are always unary operators.
2. The operator immediately to the right of an operand is a binary operator unless (1) applies, when the operator to its right is binary.
3. All operators to the left of an operand are unary unless (2) applies.

Because the unary operators have very high precedence, you can work out what they do before worrying about the other operators. One thing to watch out for is the way that `++` and `--` can be before or after their operands; the expression

```
a + -b++ + c
```

has two unary operators applied to `b`. The unary operators all associate right to left, so although the `-` comes first when you read the expression, it really parenthesizes (for clarity) like this:

```
a + -(b++) + c
```

The case is a little clearer if the prefix, rather than the postfix, form of the increment/decrement operators is being used. Again the order is right to left, but at least the operators come all in a row.

After sorting out what to do with the unary operators, it's easy to read the expression from left to right. Every time you see a binary operator, remember it. Look to the right: if the next binary operator is of a lower precedence, then the operator you just remembered is part of a subexpression to evaluate before anything else is seen. If the next operator is of the same precedence, keep repeating the procedure as long as equal precedence operators are seen. When you eventually find a lower precedence operator, evaluate the subexpression on the left according to the associativity rules. If a higher precedence operator is found on the right, forget the previous stuff: the operand to the left of the higher precedence operator is part of a subexpression separate from anything on the left so far. It belongs to the new operator instead.

If that lot isn't clear don't worry. A lot of C programmers have trouble with this area and eventually learn to parenthesize these expressions ‘by eye’, without ever using formal rules.

What *does* matter is what happens when you have fully parenthesized these expressions. Remember the ‘usual arithmetic conversions’? They explained how you could predict the type of an expression from the operands involved. Now, even if you mix all sorts of types in a complicated expression, the types of the subexpressions are determined only from the the types of the operands in the subexpression. Look at this.

```
#include <stdio.h>
#include <stdlib.h>

main(){
      int i,j;
      float f;

      i = 5; j = 2;
      f = 3.0;

      f = f + j / i;
      printf("value of f is %f\n", f);
      exit(EXIT_SUCCESS);
}
```

Example 2.11

The value printed is `3.0000`, not `5.0000`—which might surprise some, who thought that because a `float` was involved the whole statement involving the division would be done in that real type.

Of course, the division operator had only int types on either side, so the arithmetic was done as integer division and resulted in zero. The addition had a `float` and an `int` on either side, so the conversions meant that the `int` was converted to `float` for the arithmetic, and that was the correct type for the assignment, so there were no further conversions.

The previous section on casts showed one way of changing the type of an expression from its natural one to the one that you want. Be careful though:

```
(float)(j/i)
```

would still use integer division, then convert the result to `float`. To keep the remainder, you should use

```
(float)j/i
```

which would force real division to be used.

2.8.4. Parentheses

C allows you to override the normal effects of precedence and associativity by the use of parentheses as the examples have illustrated. In Old C, the parentheses had no further meaning, and in particular did *not* guarantee anything about the order of evaluation in expressions like these:

```
int a, b, c;
a+b+c;
```

```
(a+b)+c;
a+(b+c);
```

You used to need to use explicit temporary variables to get a particular order of evaluation—something that matters if you know that there are risks of overflow in a particular expression, but by forcing the evaluation to be in a certain order you can avoid it.

Standard C says that evaluation *must* be done in the order indicated by the precedence and grouping of the expression, unless the compiler can tell that the result will not be affected by any regrouping it might do for optimization reasons.

So, the expression `a = 10+a+b+5;` cannot be rewritten by the compiler as `a = 15+a+b;` unless it can be guaranteed that the resulting value of a will be the same for all combinations of initial values of `a` and `b`. That would be true if the variables were both unsigned integral types, or if they were signed integral types but in that particular implementation overflow did not cause a run-time exception and overflow was reversible.

2.8.5. Side Effects

To repeat and expand the warning given for the increment operators: it is unsafe to use the same variable more than once in an expression if evaluating the expression changes the variable and the new value could affect the result of the expression. This is because the change(s) may be 'saved up' and only applied at the end of the statement. So `f = f+1;` is safe even though `f` appears twice in a value-changing expression, `f++;` is also safe, but `f = f++;` is unsafe.

The problem can be caused by using an assignment, use of the increment or decrement operators, or by calling a function that changes the value of an external variable that is also used in the expression. These are generally known as 'side effects'. C makes almost no promise that side effects will occur in a predictable order within a single expression. (The discussion of 'sequence points' in Chapter 8 [*http://publications.gbdirect.co.uk/c_book/chapter8/*] will be of interest if you care about this.)

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter2/integral_types.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter2/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter2/constants.html*]

2.9. Constants

2.9.1. Integer constants

The normal integral constants are obvious: things like `1`, `1034` and so on. You can put `l` or `L` at the end of an integer constant to force it to be long. To make the constant `unsigned`, one of `u` or `U` can be used to do the job.

Integer constants can be written in hexadecimal by preceding the constant with `0x` or `0X` and using the upper or lower case letters `a`, `b`, `c`, `d`, `e`, `f` in the usual way.

Be careful about octal constants. They are indicated by starting the number with `0` and only using the digits `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`. It is easy to write `015` by accident, or out of habit, and not to realize that it is not in decimal. The mistake is most common with beginners, because experienced C programmers already carry the scars.

The Standard has now invented a new way of working out what type an integer constant is. In the old days, if the constant was too big for an `int`, it got promoted to a `long` (without warning). Now, the rule is that a plain decimal constant will be fitted into the first in this list

```
int   long   unsigned long
```

that can hold the value.

Plain octal or hexadecimal constants will use this list

```
int   unsigned int   long   unsigned long
```

If the constant is suffixed by `u` or `U`:

```
unsigned int   unsigned long
```

If it is suffixed by `l` or `L`:

```
long   unsigned long
```

and finally, if it suffixed by both `u` or `U` and `l` or `L`, it can only be an `unsigned long`.

All that was done to try to give you ‘what you meant’; what it does mean is that it is hard to work out exactly what the type of a constant expression is if you don't know something about the hardware. Hopefully, good compilers will warn when a constant is promoted up to another length and the `U` or `L` etc. is not specified.

A nasty bug hides here:

```
printf("value of 32768 is %d\n", 32768);
```

On a 16-bit two's complement machine, `32768` will be a `long` by the rules given above. But `printf` is only expecting an `int` as an argument (the `%d` indicates that). The type of the argument is just wrong. For the ultimate in safety-conscious programming, you should cast such cases to the right type:

```
printf("value of 32768 is %d\n", (int)32768);
```

It might interest you to note that there are no negative constants; writing `-23` is an expression involving a positive constant and an operator.

Character constants actually have type `int` (for historical reasons) and are written by placing a sequence of characters between single quote marks:

```
'a'
'b'
'like this'
```

Wide character constants are written just as above, but prefixed with `L`:

```
L'a'
L'b'
L'like this'
```

Regrettably it *is* valid to have more than one character in the sequence, giving a machine-dependent result. Single characters are the best from the portability point of view, resulting in an ordinary integer constant whose value is the machine representation of the single character. The introduction of extended characters may cause you to stumble over this by accident; if `'<a>'` is a multibyte character (encoded with a shift-in shift-out around it) then `'<a>'` will be a plain character constant, but containing several characters, just like the more obvious `'abcde'`. This is bound to lead to trouble in the future; let's hope that compilers will warn about it.

To ease the way of representing some special characters that would otherwise be hard to get into a character constant (or hard to read; does `' '` contain a space or a tab?), there is what is called an escape sequence which can be used instead. Table 2.10 shows the *escape sequences* defined in the Standard.

Sequence	Represents
`\a`	audible alarm
`\b`	backspace
`\f`	form feed
`\n`	newline
`\r`	carriage return
`\t`	tab
`\v`	vertical tab
`\\`	backslash
`\'`	quote
`\"`	double quote
`\?`	question mark

Table 2.10. C escape sequences

It is also possible to use numeric escape sequences to specify a character in terms of the internal value used to represent it. A sequence of either `\ooo` or `\xhhhh`, where the `ooo` is up to three octal digits and `hhhh` is any number of hexadecimal digits respectively. A common version of it is `'\033'`, which is used by those who know that on an ASCII based machine, octal `33` is the ESC (escape) code. Beware that the hexadecimal version will absorb any number of valid following hexadecimal digits; if you want a string containing the character whose value is hexadecimal `ff` followed by a *letter* `f`, then the safe way to do it is to use the string joining feature:

```
"\xff" "f"
```

The string

```
"\xfff"
```

only contains one character, with all three of the `f`s eaten up in the hexadecimal sequence.

Some of the escape sequences aren't too obvious, so a brief explanation is needed. To get a single quote as a character constant you type `'\''`, to get a question mark you may have to use `'\?'`; not that it matters in that example, but to get two of them in there you can't use `'??'`, because the sequence `??'` is a trigraph! You would have to use `'\?\?'`. The escape `\"` is only necessary in strings, which will come later.

There are two distinct purposes behind the escape sequences. It's obviously necessary to be able to represent characters such as single quote and backslash unambiguously: that is one purpose. The second purpose applies to the following sequences which control the motions of a printing device when they are sent to it, as follows:

`\a`	Ring the bell if there is one. Do not move.
`\b`	Backspace.
`\f`	Go to the first position on the 'next page', whatever that may mean for the output device.
`\n`	Go to the start of the next line.
`\r`	Go back to the start of the current line.
`\t`	Go to the next horizontal tab position.
`\v`	Go to the start of the line at the next vertical tab position.

For `\b`, `\t`, `\v`, if there is no such position, the behaviour is unspecified. The Standard carefully avoids mentioning the physical directions of movement of the output device which are not necessarily the top to bottom, left to right movements common in Western cultural environments.

It is guaranteed that each escape sequence has a unique integral value which can be stored in a `char`.

2.9.2. Real constants

These follow the usual format:

```
1.0
2.
.1
2.634
.125
2.e5
2.e+5
```

```
.125e-3
2.5e5
3.1E-6
```

and so on. For readability, even if part of the number is zero, it is a good idea to show it:

```
1.0
0.1
```

The exponent part shows the number of powers of ten that the rest of the number should be raised to, so

```
3.0e3
```

is equivalent in value to the integer constant

```
3000
```

As you can see, the `e` can also be `E`. These constants all have type `double` unless they are suffixed with `f` or `F` to mean `float` or `l` or `L` to mean long double.

For completeness, here is the formal description of a real constant:

A real constant is one of:

- A *fractional constant* followed by an optional *exponent*.
- A *digit sequence* followed by an *exponent*.

In either case followed by an optional one of `f`, `l`, `F`, `L`, where:

- A *fractional constant* is one of:
  - An optional *digit sequence* followed by a decimal point followed by a *digit sequence*.
  - A *digit sequence* followed by a decimal point.
- An exponent is one of
  - `e` or `E` followed by an optional `+` or `-` followed by a *digit sequence*.
- A *digit sequence* is an arbitrary combination of one or more digits.

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter2/expressions_and_arithmetic.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter2/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter2/summary.html*]

2.10. Summary

This has been a lengthy, and perhaps disconcerting chapter.

The alphabet of C, although of relevance, is not normally a day-to-day consideration of practising programmers, so it has been discussed but can now be largely ignored.

Much the same can be said regarding keywords and identifiers, since the topic is not complicated and simply becomes committed to memory.

The declaration of variables is rarely a problem, although it is worth re-emphasizing the distinction between a declaration and a definition. If that still remains unclear, you might find it of benefit to go back and re-read the description.

Beyond any question, the real complexity lies in what happens when the integral promotions and the arithmetic conversions occur. For beginners, it is often worthwhile to remember that here is a difficult and arduous piece of terrain. Nothing else in the language requires so much attention or is so important to the production of correct, reliable programs. Beginners should *not* try to remember it all, but to go on now and to gain confidence with the rest of the language. After two or three months' practice at using the easier parts of the language, the time really does come when you can no longer afford to ignore Section 2.8 [*http://publications.gbdirect.co.uk/c_book/chapter2/expressions_and_arithmetic.html*].

Many highly experienced C programmers never bother to learn the different precedences of operators, except for a few important cases. A precedence table pinned above your desk, for easy reference, is a valuable tool.

The Standard has substantially affected parts of the language described in this chapter. In particular, the changes to the conversions and the change from ‘unsignedness preserving’ to ‘value preserving’ rules of arithmetic may cause some surprises to experienced C programmers. Even they have some real re-learning to do.

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter2/constants.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter2/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter2/exercises.html*]

2.11. Exercises

Exercise 2.17. First, fully parenthesize the following expressions according to the precedence and associativity rules. Then, replacing the variables and constants with the appropriate type names, show how the type of the expression is derived by replacing the highest precedence expressions with its resulting type.

The variables are:

```
char c;
int i;
unsigned u;
float f;
```

For example: `i = u+1;` parenthesizes as `(i = (u + 1));`

The types are

```
(int = (unsigned + int));
```

then

```
(int = (unsigned)); /* usual arithmetic conversions */
```

then

```
(int); /* assignment */
```

a. `c = u * f + 2.6L;`
b. `u += --f / u % 3;`
c. `i <<= u * - ++f;`
d. `u = i + 3 + 4 + 3.1;`
e. `u = 3.1 + i + 3 + 4;`
f. `c = (i << - --f) & 0xf;`

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter2/summary.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter2/*]

Chapter 3

Control of Flow and Logical Expressions

Previous chapter [*http://publications.gbdirect.co.uk/c_book/chapter2/*] | Next chapter [*http://publications.gbdirect.co.uk/c_book/chapter4/*]

3.1. The Task ahead

In this chapter we look at the various ways that the control of flow statements can be used in a C program, including some statements that haven't been introduced so far. They are almost always used in conjunction with logical expressions to select the next action. Examples of *logical expressions* that have been seen already are some simple ones used in `if` or `while` statements. As you might have expected, you can use expressions more complicated than simple comparison (>, <=, == etc.); what may surprise you is the type of the result.

3.1.1. Logical expressions and Relational Operators

All of the examples we have used so far have deliberately avoided using complicated logical expressions in the control of flow statements. We have seen expressions like this

```
if(a != 100){...
```

and presumably you have formed the idea that C supports the concept of 'true' and 'false' for these relationships. In a way, it does, but in a way that differs from what is often expected.

All of the relational operators shown in Table 3.1 are used to compare two operands in the way indicated. When the operands are arithmetic types, the usual arithmetic conversions are applied to them.

Operator	Operation
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal to
!=	not equal to

Table 3.1. Relational operators

Be extra careful of the test for equality, ==. As we have already pointed out, it is often valid to use assignment = where you might have meant == and C can't tell you about your mistake. The results are normally different and it takes a long time for beginners to get used to using == and =.

Now, that usefully introduces the question 'why?'. Why are both valid? The answer is simple. C's concept of 'true' and 'false' boils down to simply 'non-zero' and 'zero', respectively. Where we have seen expressions involving relational operators used to control `do` and `if` statements, we have just been using expressions with numeric results. If the expression evaluates to non-zero, then the result is effectively true. If the reverse is the case, then of course the result is false. Anywhere that the relational operators appear, so may any other expression.

The relational operators work by comparing their operands and giving zero for false and (remember this) one for true. The result is of type `int`. This example shows how they work.

```
#include <stdio.h>
#include <stdlib.h>

main(){
    int i;

    i = -10;
    while(i <= 5){
            printf("value of i is %d, ", i);
            printf("i == 0 = %d, ", i==0 );
            printf("i > -5 = %d\n", i > -5);
            i++;
    }
    exit(EXIT_SUCCESS);
}
```

Example 3.1

Which produces this on its standard output:

```
value of i is -10, i == 0 = 0, i > -5 = 0
value of i is -9, i == 0 = 0, i > -5 = 0
value of i is -8, i == 0 = 0, i > -5 = 0
value of i is -7, i == 0 = 0, i > -5 = 0
value of i is -6, i == 0 = 0, i > -5 = 0
value of i is -5, i == 0 = 0, i > -5 = 0
value of i is -4, i == 0 = 0, i > -5 = 1
value of i is -3, i == 0 = 0, i > -5 = 1
value of i is -2, i == 0 = 0, i > -5 = 1
value of i is -1, i == 0 = 0, i > -5 = 1
value of i is 0, i == 0 = 1, i > -5 = 1
value of i is 1, i == 0 = 0, i > -5 = 1
value of i is 2, i == 0 = 0, i > -5 = 1
value of i is 3, i == 0 = 0, i > -5 = 1
value of i is 4, i == 0 = 0, i > -5 = 1
value of i is 5, i == 0 = 0, i > -5 = 1
```

In this probably mistaken piece of code, what do you think happens?

```
if(a = b)...
```

The value of `b` is assigned to `a`. As you know, the result has the type of a and whatever value was assigned to `a`. The if will execute the next statement if the value assigned is not zero. If zero is assigned, the next statement is ignored. So now you understand what happens if you confuse the assignment with the equality operator!

In all of the statements that test the value of an expression, the `if`, `while`, `do`, and for statements, the expression is simply tested to see if its value is zero or not.

We will look at each one in turn.

Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter3/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter3/flow_control.html*]

3.2. Control of flow

3.2.1. The if statement

The `if` statement has two forms:

```
if(expression) statement

if(expression) statement1
else statement2
```

In the first form, if (and only if) the *expression* is non-zero, the *statement* is executed. If the *expression* is zero, the *statement* is ignored. Remember that the *statement* can be compound; that is the way to put several statements under the control of a single `if`.

The second form is like the first except that if the statement shown as *statement1* is selected then *statement2* will not be, and vice versa.

Either form is considered to be a single statement in the syntax of C, so the following is completely legal.

```
if(expression)
    if(expression) statement
```

The first `if (expression)` is followed by a properly formed, complete `if` statement. Since that is legally a statement, the first `if` can be considered to read

```
if(expression) statement
```

and is therefore itself properly formed. The argument can be extended as far as you like, but it's a bad habit to get into. It is better style to make the statement compound even if it isn't necessary. That makes it a lot easier to add extra statements if they are needed and generally improves readability.

The form involving `else` works the same way, so we can also write this.

```
if(expression)
  if(expression)
    statement
  else
    statement
```

As Chapter 1 [*http://publications.gbdirect.co.uk/c_book/chapter1/*] has said already, this is now ambiguous. It is not clear, except as indicated by the indentation, which of the `if`s is responsible for the `else`. If we follow the rules that the previous example suggests, then the second `if` is followed by a statement, and is therefore itself a statement, so the `else` belongs to the first `if`.

That is *not* the way that C views it. The rule is that an `else` belongs to the first if above that hasn't already got an `else`. In the example we're discussing, the else goes with the second if.

To prevent any unwanted association between an `else` and an `if` just above it, the if can be hidden away by using a compound statement. To repeat the example in Chapter 1 [*http://publications.gbdirect.co.uk/c_book/chapter1/*], here it is.

```
if(expression){
    if(expression)
            statement
}else
    statement
```

Putting in all the compound statement brackets, it becomes this:

```
if(expression){
    if(expression){
        statement
    }
}else{
    statement
}
```

If you happen not to like the placing of the brackets, it is up to you to put them where you think they look better; just be consistent about it. You probably need to know that this a subject on which feelings run deep.

3.2.2. The while and do statements

The `while` is simple:

```
while(expression)
    statement
```

The *statement* is only executed if the *expression* is non-zero. After every execution of the *statement*, the *expression* is evaluated again and the process repeats if it is non-zero. What could be plainer than that? The only point to watch out for is that the *statement* may never be executed, and that if nothing in the statement affects the value of the *expression* then the `while` will either do nothing or loop for ever, depending on the initial value of the expression.

It is occasionally desirable to guarantee at least one execution of the statement following the while, so an alternative form exists known as the do statement. It looks like this:

```
do
    statement
while(expression);
```

and you should pay close attention to that semicolon—it is not optional! The effect is that the statement part is executed before the controlling expression is evaluated, so this guarantees at least one trip around the loop. It was an unfortunate decision to use the keyword `while` for both purposes, but it doesn't seem to cause too many problems in practice.

If you feel the urge to use a `do`, think carefully. It is undoubtedly essential in certain cases, but experience has shown that the use of `do` statements is often associated with poorly constructed code. Not every time, obviously, but as a general rule you should stop and ask yourself if you have made the right choice. Their use often indicates a hangover of thinking methods learnt with other languages, or just sloppy design. When you *do* convince yourself that nothing else will give you just what is wanted, then go ahead - be daring—use it.

3.2.2.1. Handy hints

A *very* common trick in C programs is to use the result of an assignment to control `while` and `do` loops. It is so commonplace that, even if you look at it the first time and blench, you've got no alternative but to learn it. It falls into the category of ‘idiomatic’ C and eventually becomes second nature to anybody who really uses the language. Here is the most common example of all:

```
#include <stdio.h>
#include <stdlib.h>

main(){
    int input_c;

    /* The Classic Bit */
    while( (input_c = getchar()) != EOF){
            printf("%c was read\n", input_c);
    }
    exit(EXIT_SUCCESS);
}
```

Example 3.2

The clever bit is the expression assigning to `input_c`. It is assigned to, compared with `EOF` (End Of File), and used to control the loop all in one go. Embedding the assignment like that is a handy embellishment. Admittedly it only saves one line of code, but the benefit in terms of readability (once you have got used to seeing it) is quite large. Learn where the parentheses are, too. They're necessary for precedence reasons—work out why!

Note that `input_c` is an `int`. This is because `getchar` has to be able to return not only every possible value of a `char`, but also an extra value, `EOF`. To do that, a type longer than a `char` is necessary.

Both the `while` and the `do` statements are themselves syntactically a single statement, just like an `if` statement. They occur anywhere that any other single statement is permitted. If you want them to control *several* statements, then you will have to use a compound statement, as the examples of `if` illustrated.

3.2.3. The for statement

A very common feature in programs is loops that are controlled by variables used as a counter. The counter doesn't always have to count consecutive values, but the usual arrangement is for it to be initialized outside the loop, checked every time around the loop to see when to finish and updated each time around the loop. There are three important places, then, where the loop control is concentrated: initialize, check and update. This example shows them.

```
#include <stdio.h>
#include <stdlib.h>
main(){
      int i;

      /* initialise */
      i = 0;
      /* check */
      while(i <= 10){
```

```
                printf("%d\n", i);
                /* update */
                i++;
        }
        exit(EXIT_SUCCESS);
}
```

Example 3.3

As you will have noticed, the initialization and check parts of the loop are close together and their location is obvious because of the presence of the `while` keyword. What is harder to spot is the place where the update occurs, especially if the value of the controlling variable is used within the loop. In that case, which is by far the most common, the update has to be at the very end of the loop: far away from the initialize and check. Readability suffers because it is hard to work out how the loop is going to perform unless you read the whole body of the loop carefully. What is needed is some way of bringing the initialize, check and update parts into one place so that they can be read quickly and conveniently. That is exactly what the for statement is designed to do. Here it is.

`for (`*`initialize`*`; `*`check`*`; `*`update`*`) `*`statement`*

The *initialize* part is an expression; nearly always an assignment expression which is used to initialize the control variable. After the initialization, the *check* expression is evaluated: if it is non-zero, the statement is executed, followed by evaluation of the update expression which generally increments the control variable, then the sequence restarts at the check. The loop terminates as soon as the check evaluates to zero.

There are two important things to realize about that last description: one, that each of the three parts of the for statement between the parentheses are just expressions; two, that the description has carefully explained what they are intended to be used for without proscribing alternative uses—that was done deliberately. You can use the expressions to do whatever you like, but at the expense of readability if they aren't used for their intended purpose.

Here is a program that does the same thing twice, the first time using a `while` loop, the second time with a `for`. The use of the increment operator is exactly the sort of use that you will see in everyday practice.

```
#include <stdio.h>
#include <stdlib.h>
main(){
        int i;

        i = 0;
        while(i <= 10){
                printf("%d\n", i);
                i++;
        }

        /* the same done using ``for'' */
        for(i = 0; i <= 10; i++){
                printf("%d\n", i);
        }
        exit(EXIT_SUCCESS);
}
```

Example 3.4

There isn't any difference betweeen the two, except that in this case the `for` loop is more convenient and maintainable than the `while` statement. You should always use the `for` when it's appropriate; when a loop is being controlled by some sort of counter. The `while` is more at home when an indeterminate number of cycles of the loop are part of the problem. As always, it needs a degree of judgement on behalf of the author of the program; an understanding of form, style, elegance and the poetry of a well written program. There is no evidence that the software business suffers from a surfeit of those qualities, so feel free to exercise them if you are able.

Any of the initialize, check and update expressions in the for statement can be omitted, although the semicolons must stay. This can happen if the counter is already initialized, or gets updated in the body of the loop. If the check expression is omitted, it is assumed to result in a ‘true’ value and the loop never terminates. A common way of writing never-ending loops is either

```
for(;;)
```

or

```
while(1)
```

and both can be seen in existing programs.

3.2.4. A brief pause

The control of flow statements that we've just seen are quite adequate to write programs of any degree of complexity. They lie at the core of C and even a quick reading of everyday C programs will illustrate their importance, both in the provision of essential functionality and in the structure that they emphasize. The remaining statements are used to give programmers finer control or to make it easier to deal with exceptional conditions. Only the `switch` statement is enough of a heavyweight to need no justification for its use; yes, it can be replaced with lots of `if`s, but it adds a lot of readability. The others, `break`, `continue` and `goto`, should be treated like the spices in a delicate sauce. Used carefully they can turn something commonplace into a treat, but a heavy hand will drown the flavour of everything else.

3.2.5. The switch statement

This is not an essential part of C. You could do without it, but the language would have become significantly less expressive and pleasant to use.

It is used to select one of a number of alternative actions depending on the value of an expression, and nearly always makes use of another of the lesser statements: the `break`. It looks like this.

```
switch (expression){
case const1:    statements
case const2:    statements
default:        statements
}
```

The *expression* is evaluated and its value is compared with all of the *const1* etc. expressions, which must all evaluate to different constant values (strictly they are *integral constant expressions*, see Chapter 6 [*http://publications.gbdirect.co.uk/c_book/chapter6/*] and below). If any of them has the same value as the *expression* then the statement following the `case` label is selected for execution. If the `default` is present, it will be selected when there is no matching value found. If there is no `default` and no matching value, the entire `switch` statement will do nothing and execution will continue at the following statement.

One curious feature is that the cases are *not* exclusive, as this example shows.

```
#include <stdio.h>
#include <stdlib.h>

main(){
      int i;
      for(i = 0; i <= 10; i++){
              switch(i){
                      case 1:
                      case 2:
                              printf("1 or 2\n");
                      case 7:
                              printf("7\n");
                      default:
                              printf("default\n");
              }
      }
      exit(EXIT_SUCCESS);
}
```

Example 3.5

The loop cycles with `i` having values 0–10. A value of `1` or `2` will cause the printing of the message `1 or 2` by selecting the first of the `printf` statements. What you might not expect is the way that the remaining messages would also appear! It's because the `switch` only selects one entry point to the body of the statement; after starting at a given point all of the following statements are also executed. The `case` and `default` labels simply allow you to indicate *which* of the statements is to be selected. When `i` has the value of `7`, only the last two messages will be printed. Any value other than `1`, `2`, or `7` will find only the last message.

The labels can occur in any order, but no two values may be the same and you are allowed either one or no `default` (which doesn't have to be the last label). Several labels can be put in front of one statement and several statements can be put after one label.

The expression controlling the `switch` can be of any of the integral types. Old C used to insist on *only* `int` here, and some compilers would forcibly truncate longer types, giving rise on rare occasions to some very obscure bugs.

3.2.5.1. The major restriction

The biggest problem with the `switch` statement is that it doesn't allow you to select mutually exclusive courses of action; once the body of the statement has been entered any subsequent statements within the body will all be executed. What is needed is the `break` statement. Here is the previous example, but amended to make sure that the messages printed come out in a more sensible order. The `break` statements cause execution to leave the `switch` statement immediately and prevent any further statements in the body of the `switch` from being executed.

```
#include <stdio.h>
#include <stdlib.h>
main(){
      int i;
      for(i = 0; i <= 10; i++){
              switch(i){
                      case 1:
                      case 2:
                              printf("1 or 2\n");
                              break;
                      case 7:
                              printf("7\n");
                              break;
                      default:
                              printf("default\n");
              }
      }
      exit(EXIT_SUCCESS);
}
```

Example 3.6

The `break` has further uses. Its own section follows soon.

3.2.5.2. Integral Constant Expression

Although Chapter 6 [*http://publications.gbdirect.co.uk/c_book/chapter6/*] deals with constant expressions, it is worth looking briefly at what an integral constant expression is, since that is what must follow the `case` labels in a `switch` statement. Loosely speaking, it is any expression that does not involve any value-changing operation (like increment or assignment), function calls or comma operators. The operands in the expression must all be integer constants, character constants, enumeration constants, `sizeof` epressions and floating-point constants that are the immediate operands of casts. Any cast operators must result in integral types.

Much what you would expect, really.

3.2.6. The break statement

This is a simple statement. It only makes sense if it occurs in the body of a `switch`, `do`, `while` or `for` statement. When it is executed the control of flow jumps to the statement immediately following the body of the statement containing the `break`. Its use is widespread in `switch` statements, where it is more or less essential to get the control that most people want.

The use of the `break` within loops is of dubious legitimacy. It has its moments, but is really only justifiable when exceptional circumstances have happened and the loop has to be abandoned. It would be nice if more than one loop could be abandoned with a single break but that isn't how it works. Here is an example.

```
#include <stdio.h>
#include <stdlib.h>
main(){
        int i;

        for(i = 0; i < 10000; i++){
                if(getchar() == 's')
                        break;
                printf("%d\n", i);
        }
        exit(EXIT_SUCCESS);
}
```

Example 3.7

It reads a single character from the program's input before printing the next in a sequence of numbers. If an ‘s’ is typed, the break causes an exit from the loop.

If you want to exit from more than one level of loop, the `break` is the wrong thing to use. The `goto` is the only easy way, but since it can't be mentioned in polite company, we'll leave it till last.

3.2.7. The continue statement

This statement has only a limited number of uses. The rules for its use are the same as for `break`, with the exception that it doesn't apply to `switch` statements. Executing a `continue` starts the next iteration of the smallest enclosing `do`, `while` or `for` statement immediately. The use of `continue` is largely restricted to the top of loops, where a decision has to be made whether or not to execute the rest of the body of the loop. In this example it ensures that division by zero (which gives undefined behaviour) doesn't happen.

```
#include <stdio.h>
#include <stdlib.h>
main(){
      int i;

      for(i = -10; i < 10; i++){
              if(i == 0)
                      continue;
              printf("%f\n", 15.0/i);
              /*
               * Lots of other statements .....
               */
      }
      exit(EXIT_SUCCESS);
}
```

Example 3.7

You could take a puritanical stance and argue that, instead of a conditional `continue`,, the body of the loop should be made conditional instead—but you wouldn't have many supporters. Most C programmers would rather have the `continue` than the extra level of indentation, particularly if the body of the loop is large.

Of course the `continue` can be used in other parts of a loop, too, where it may occasionally help to simplify the logic of the code and improve readability. It deserves to be used sparingly.

Do remember that `continue` has no special meaning to a `switch` statement, where `break` does have. Inside a `switch`, `continue` is only valid if there is a loop that encloses the `switch`, in which case the next iteration of the loop will be started.

There is an important difference between loops written with `while` and `for`. In a `while`, a continue will go immediately to the test of the controlling expression. The same thing in a `for` will do two things: first the update expression is evaluated, then the controlling expresion is evaluated.

3.2.8. goto and labels

Everybody knows that the `goto` statement is a ‘bad thing’. Used without care it is a great way of making programs hard to follow and of obscuring any structure in their flow. Dijkstra wrote a famous paper in 1968 called ‘Goto Statement Considered Harmful’, which everybody refers to and almost nobody has read.

What's especially annoying is that there are times when it is the most appropriate thing to use in the circumstances! In C, it is used to escape from multiple nested loops, or to go to an error handling exit at the end of a function. You will need a *label* when you use a `goto`; this example shows both.

```
goto L1;
/* whatever you like here */
L1: /* anything else */
```

A label is an identifier followed by a colon. Labels have their own ‘name space’ so they can't clash with the names of variables or functions. The name space only exists for the function containing the label, so label names can be re-used in different functions. The label can be used before it is declared, too, simply by mentioning it in a `goto` statement.

Labels *must* be part of a full statement, even if it's an empty one. This usually only matters when you're trying to put a label at the end of a compound statement—like this.

```
label_at_end: ; /* empty statement */
}
```

The `goto` works in an obvious way, jumping to the labelled statements. Because the name of the label is only visible inside its own function, you can't jump from one function to another one.

It's hard to give rigid rules about the use of `goto`s but, as with the `do`, `continue` and the `break` (except in `switch` statements), over-use should be avoided. Think carefully every time you feel like using one, and convince yourself that the structure of the program demands it. More than one `goto` every 3–5 functions is a symptom that should be viewed with deep suspicion.

Summary

Now we've seen all of the control of flow statements and examples of their use. Some should be used whenever possible, some are not for use line by line but for special purposes where their particular job is called for. It is possible to write elegant and beautiful programs in C if you are prepared to take the extra bit of care necessary; the specialized control of flow statements give you the chance to add the extra polish that some other languages lack.

All that remains to be done to complete the picture of flow of control in C is to finish off the logical operators.

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter3/task_ahead.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter3/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter3/logical_expressions.html*]

3.3. More logical expressions

This chapter has already shown how C makes no distinction between 'logical' and other values. The relational operators all give a result of `0` or `1` for false and true, respectively. Whenever the control of flow statements demand it, an expression is evaluated to determine what to do next. A `0` means 'don't do it'; anything else means 'do'. It means that the fragments below are all quite reasonable.

```
while (a<b)...
while (a)....
if ( (c=getchar()) != EOF )...
```

No experienced C programmer would be surprised by any of them. The second of them, `while (a)`, is a common abbreviation for `while (a != 0)`, as you should be able to work out.

What we need now is a way of writing more complicated expressions involving these logical true and false values. So far, it has to be done like this, when we wanted to say `if(a<b AND c<d)`

```
if (a < b){
      if (c < d)...
}
```

It will not be a source of great amazement to find that there is a way of expressing such a statement.

There are three operators involved in this sort of operation: the logical `AND &&`, the logical `OR ||` and the `NOT !`. The last is unary, the other two are binary. All of them take expressions as their operands and give as results either `1` or `0`. The `&&` gives `1` only when both of its operands are non-zero. The `||` gives `0` only when both operands are zero. The `!` gives `0` if its operand is non-zero and vice versa. Easy really. The results are of type `int` for all three.

Do not confuse `&` and `|` (the bitwise operators) with their logical counterparts. They are not the same.

One special feature of the logical operators, found in very few of the other operators, is their effect on the sequence of evaluation of an expression. They evaluate left to right (after precedence is taken into account) and every logical expression ceases evaluation as soon as the overall result can be determined. For example, a sequence of `||`s can stop as soon as one operand is found to be non-zero. This next fragment guarantees never to divide by zero.

```
if (a!=0 && b/a > 5)...
/* alternative */
if (a && b/a > 5)
```

In either version `b/a` will only be evaluated if a is non-zero. If a were zero, the overall result would already have been decided, so the evaluation must stop to conform with C's rules for the logical operators.

The unary NOT is simple. It isn't all that common to see it in use largely because most expresssions can be rearranged to do without it. The examples show how.

```
if (!a)...
/* alternative */
if (a==0)...

if(!(a>b))
/* alternative */
if(a <= b)

if (!(a>b && c<d))...
/* alternative */
if (a<=b || c>=d)...
```

Each of the examples and the alternatives serve to show ways of avoiding (or at least doing without) the `!` operator. In fact, it's most useful as an aid to readability. If the problem that you are solving has a natural logical relationship inherent in it—say the `(b*b-4*a*c) > 0` found in quadratic equation solving—then it probably reads better if you write `if( !((b*b-4*a*c) > 0))` than `if( (b*b-4*a*c) <= 0)`—but it's up to you. Pick the one that feels right.

Most expressions using these logical operators work out just about right in terms of the precedence rules, but you can get a few nasty surprises. If you look back to the precedence tables, you will find that there are some operators with lower precedence than the logical ones. In particular, this is a very common mistake:

```
if(a&b == c){...
```

What happens is that `b` is compared for equality with `c`, then the `1` or `0` result is anded with `a`! Some distinctly unexpected behaviour has been caused by that sort of error.

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter3/flow_control.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter3/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter3/strange_operators.html*]

3.4. Strange operators

There are two operators left to mention which look decidedly odd. They aren't ‘essential’, but from time to time do have their uses. Don't ignore them completely. This is the only place where we describe them, so our description includes what happens when they are mixed with pointer types, which makes them look more complicated than they really are.

3.4.1. The ?: operator

Like playing the accordion, this is easier to demonstrate than to describe.

`expression1``?`*`expression2`*`:`*`expression3`*

If *expression1* is true, then the result of the whole expression is *expression2*, otherwise it is *expression3*; depending on the value of *expression1*, only one of them will be evaluated when the result is calculated.

The various combinations of types that are permitted for *expression2* and *expression3* and, based on those, the resulting type of the whole expression, are complicated. A lot of the complexity is due to types and notions that we haven't seen so far. For completeness they are described in detail below, but you'll have to put up with a number of forward references.

The easiest case is when both expressions have arithmetic type (i.e. integral or real). The usual arithmetic conversions are applied to find a common type for both expressions and then that is the type of the result. For example

```
a>b?1:3.5
```

contains a constant (1) of type `int` and another (3.5) of type `double`. Applying the arithmetic conversions gives a result of type `double`.

Other combinations are also permitted.

- If both operands are of compatible structure or union types, then that is the type of the result.
- If both operands have `void` type, then that is the type of the result.

Various pointer types can be mixed.

- Both operands may be pointers to (possibly *qualified*) compatible types.
- One operand may be a pointer and the other a *null pointer constant*.
- One operand may be a *pointer to an object or incomplete type* and the other a pointer to (possibly qualified) `void`.

The type of the result when pointers are involved is derived in two separate steps.

1. If either of the operands is a pointer to a qualified type, the result is a pointer to a type that is qualified by all the qualifiers of both operands.
2. If one operand is a null pointer constant, then the result has the type of the other operand. If one operand is a pointer to void, the other operand is

converted to pointer to `void` and that is the type of the result. If both operands are pointers to compatible types (ignoring any qualifiers) the the result has the *composite type*.

Qualifiers, composite types and compatible types are all subjects discussed later.

The shortest useful example that we can think of is this one, where the string to be printed by `printf` is selected using this magical operator.

```
#include <stdio.h>
#include <stdlib.h>

main(){
      int i;

      for(i=0; i <= 10; i++){
              printf((i&1) ? "odd\n" : "even\n");
      }
      exit(EXIT_SUCCESS);
}
```

Example 3.9

It's cute when you need it, but the first time that they see it most people look very uncomfortable for a while, then recollect an urgent appointment somewhere else.

After evaluating the first operand there is one of the *sequence points* described in Chapter 8 [*http://publications.gbdirect.co.uk/c_book/chapter8/*].

3.4.2. The comma operator

This wins the prize for ‘most obscure operator’. It allows a list of expressions to be separated by commas:

`expression-1,expression-2,expression-3,...,expression-n`

and it goes on as long as you like. The *expressions* are evaluated strictly left to right and their values discarded, except for the last one, whose type and value determine the result of the overall expression. Don't confuse this version of the comma with any of the other uses C finds for it, especially the one that separates function arguments. Here are a couple of examples of it in use.

```
#include <stdio.h>
#include <stdlib.h>

main(){
      int i, j;

      /* comma used - this loop has two counters */
      for(i=0, j=0; i <= 10; i++, j = i*i){
              printf("i %d j %d\n", i, j);
      }

      /*
       * In this futile example, all but the last
       * constant value is discarded.
       * Note use of parentheses to force a comma
       * expression in a function call.
       */
      printf("Overall: %d\n", ("abc", 1.2e6, 4*3+2));
```

```
        exit(EXIT_SUCCESS);
}
```

Example 3.10

Unless you are feeling very adventurous, the comma operator is just as well ignored. Be prepared to see it only on special occasions.

After evaluating each operand there is one of the *sequence points* described in Chapter 8 [*http://publications.gbdirect.co.uk/c_book/chapter8/*].

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter3/logical_expressions.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter3/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter3/summary.html*]

3.5. Summary

<gbdirect>

This chapter has described the entire range of control of flow available in C. The only areas that cause even moderate surprise are the way in which cases in a switch statement are not mutually exclusive, and the fact that goto cannot transfer control to any function except the one that is currently active. None of this is intellectually deep and it has never been known to cause problems either to beginners or programmers experienced in other languages.

The logical expressions all give integral results. This is perhaps slightly unusual, but once again takes very little time to learn.

Probably the most surprising part about the whole chapter will have been to learn of the conditional and comma operators. A strong case could be made for the abolition of the conditional operator, were it not for compatibility with existing code, but the comma operator does have important uses, especially for automatic generators of C programs.

The Standard has not had much effect on the contents of this chapter. Prospective users of C should ensure that they are completely familiar with all of the topics discussed here (except the conditional and comma operators). They are essential to the practical use of the language, and none of the material is hard.

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter3/strange_operators.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter3/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter3/exercises.html*]

3.6. Exercises

Exercise 3.1. What is the type and value of the result of the relational operators?

Exercise 3.2. What is the type and value of the result of the logical operators (`&&`, `||`, and `!`)?

Exercise 3.3. What is unusual about the logical operators?

Exercise 3.4. Why is `break` useful in `switch` statements?

Exercise 3.5. Why is `continue` not very useful in `switch` statements?

Exercise 3.6. What is a possible problem using `continue` in `while` statements?

Exercise 3.7. How can you jump from one function to another?

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter3/summary.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter3/*]

Chapter 4

Functions

Previous chapter [*http://publications.gbdirect.co.uk/c_book/chapter3/*] | Next chapter [*http://publications.gbdirect.co.uk/c_book/chapter5/*]

4.1. Changes

<gbdirect>

The single worst feature of Old C was that there was no way to declare the number and types of a function's arguments and to have the compiler check that the use of the function was consistent with its declaration. Although it didn't do a lot of damage to the success of C, it did result in portability and maintainability problems that we all could have done without.

The Standard has changed that state of affairs. You can now declare functions in a way that allows their use to be checked, and which is also largely compatible with the old style (so old programs still work, provided they had no errors before). Another useful feature is a portable way of using functions with a variable number of arguments, like `printf`, which used to be non-portable; the only way to implement it relied upon intimate knowledge of the hardware involved.

The Standard's way of fixing this problem was in large measure to plagiarize from C++, which had already tried out the new ideas in practice. This model has been so successful that lots of 'Old' C compilers adopted it on their way to conforming to the Standard.

The Standard still retains compatibility with Old C function declarations, but that is purely for the benefit of existing programs. Any new programs should make full use of the much tighter checking that the Standard permits and strenuously avoid the old syntax (which may disappear one day).

Footnotes

1. Stroustrup B. (1991). *The C++ Programming Language* 2nd edn. Reading, MA: Addison-Wesley

Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter4/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter4/function_types.html*]

4.2. The type of functions

All functions have a type: they return a value of that type whenever they are used. The reason that C doesn't have ‘procedures’, which in most other languages are simply functions without a value, is that in C it is permissible (in fact well-nigh mandatory) to discard the eventual value of most expressions. If that surprises you, think of an assignment

```
a = 1;
```

That's a perfectly valid assignment, but don't forget that it has a value too. The value is discarded. If you want a bigger surprise, try this one:

```
1;
```

That is an expression followed by a semicolon. It is a well formed statement according to the rules of the language; nothing wrong with it, it is just useless. A function used as a procedure is used in the same way—a value is *always* returned, but you don't use it:

```
f(argument);
```

is also an expression with a discarded value.

It's all very well saying that the value returned by a function can be ignored, but the fact remains that if the function really *does* return a value then it's probably a programming error not to do something with it. Conversely, if no useful value is returned then it's a good idea to be able to spot anywhere that it is used by mistake. For both of those reasons, functions that don't return a useful value should be declared to be `void`.

Functions can return any type supported by C (except for arrays and functions), including the pointers, structures and unions which are described in later chapters. For the types that can't be returned from functions, the restrictions can often be sidestepped by using pointers instead.

All functions can be called recursively.

4.2.1. Declaring functions

Unfortunately, we are going to have to use some jargon now. This is one of the times that the use of an appropriate technical term really does reduce the amount of repetitive descriptive text that would be needed. With a bit of luck, the result is a shorter, more accurate and less confusing explanation. Here are the terms.

declaration
    The point at which a name has a type associated with it.
definition
    Also a declaration, but at this point some storage is reserved for the named object. The rules for what makes a declaration into a definition can be complicated, but are easy for functions: You turn a function declaration into a definition by providing a body for the function in the form of a compound statement.
formal parameters
parameters

These are the names used inside a function to refer to its arguments.

actual arguments

arguments

These are the values used as arguments when the function is actually called. In other words, the values that the *formal parameters* will have on entry to the function.

The terms ‘parameter’ and ‘argument’ do tend to get used as if they were interchangeable, so don't read too much into it if you see one or the other in the text below.

If you use a function before you declare it, it is implicitly declared to be ‘function returning `int`’. Although this will work, and was widely used in Old C, in Standard C it is bad practice—the use of undeclared functions leads to nasty problems to do with the number and type of arguments that are expected for them. All functions should be fully declared before they are used. For example, you might be intending to use a function in a private library called, say, `aax1`. You know that it takes no arguments and returns a `double`. Here is how it should be declared:

```
  double aax1(void);
```

and here is how it might be used:

```
main(){
      double return_v, aax1(void);
      return_v = aax1();
      exit(EXIT_SUCCESS);
}
```

Example 4.1

The declaration was an interesting one. It defined `return_v`, actually causing a variable to come into existence. It also declared `aax1` without defining it; as we know, functions only become *defined* when a body is provided for them. Without a declaration in force, the default rules mean that `aax1` would have been assumed to be `int`, even though it really does return a `double`—which means that your program will have undefined behaviour. Undefined behaviour is disastrous!

The presence of `void` in the argument list in the declaration shows that the function really takes no arguments. If it had been missing, the declaration would have been taken to give no information about the function's arguments. That way, compatibility with Old C is maintained at the price of the ability of the compiler to check.

To *define* a function you also have to provide a body for it, in the form of a compound statement. Since no function can itself contain the definition of a function, functions are all separate from each other and are only found at the outermost level of the program's structure. Here is a possible definition for the function `aax1`.

```
  double
  aax1(void) {
        /* code for function body */
        return (1.0);
  }
```

It is unusual for a block-structured language to prohibit you from defining functions inside other functions, but this is one of the characteristics of C. Although it isn't obvious, this helps to improve the run-time performance of C by reducing the housekeeping associated with function calls.

4.2.2. The return statement

The `return` statement is very important. Every function except those returning `void` should have at least one, each `return` showing what value is supposed to be returned at that point. Although it is possible to return from a function by falling through the last `}`, unless the function returns `void` an unknown value will be returned, resulting in undefined behaviour.

Here is another example function. It uses `getchar` to read characters from the program input and returns whatever it sees except for space, tab or newline, which it throws away.

```
#include <stdio.h>

int
non_space(void){
      int c;
      while ( (c=getchar ())=='\t' || c== '\n' || c==' ')
              ; /* empty statement */
      return (c);
}
```

Look at the way that all of the work is done by the test in the `while` statement, whose body was an empty statement. It is not an uncommon sight to see the semicolon of the empty statement sitting there alone and forlorn, with only a piece of comment for company and readability. Please, please, never write it like this:

```
  while (something);
```

with the semicolon hidden away at the end like that. It's too easy to miss it when you read the code, and to assume that the following statement is under the control of the `while`.

The type of expression returned must match the type of the function, or be capable of being converted to it as if an assignment statement were in use. For example, a function declared to return `double` could contain

```
  return (1);
```

and the integral value will be converted to `double`. It is also possible to have just `return` without any expression—but this is probably a programming error unless the function returns `void`. Following the `return` with an expression is *not* permitted if the function returns `void`.

4.2.3. Arguments to functions

Before the Standard, it was not possible to give any information about a function's arguments except in the definition of the function itself. The information was only used in the body of the function and was forgotten at the end. In those bad old days, it was quite possible to define a function that had three `double` arguments and only to pass it one `int`, when it was called. The program would compile normally, but simply not work properly. It was considered to be the programmer's job to check that the number and the type of arguments to a function matched correctly. As you would expect, this turned out to be a first-rate source of bugs and portability problems. Here is an example of the definition and use of a function with arguments, but omitting for the moment to declare the function fully.

```
#include <stdio.h>
#include <stdlib.h>
main(){
      void pmax();                      /* declaration */
      int i,j;
      for(i = -10; i <= 10; i++){
              for(j = -10; j <= 10; j++){
                      pmax(i,j);
              }
```

```
        }
        exit(EXIT_SUCCESS);
}
/*
* Function pmax.
* Returns:      void
* Prints larger of its two arguments.
*/
void
pmax(int a1, int a2){                   /* definition */
        int biggest;

        if(a1 > a2){
                biggest = a1;
        }else{
                biggest = a2;
        }

        printf("larger of %d and %d is %d\n",
                a1, a2, biggest);
}
```

Example 4.2

What can we learn from this? To start with, notice the careful declaration that `pmax` returns `void`. In the function definition, the matching `void` occurs on the line before the function name. The reason for writing it like that is purely one of style; it makes it easier to find function definitions if their names are always at the beginning of a line.

The function declaration (in `main`) gave no indication of any arguments to the function, yet the use of the function a couple of lines later involved two arguments. That is permitted by both the old and Standard versions of C, but *must nowadays be considered to be bad practice*. It is much better to include information about the arguments in the declaration too, as we will see. The old style is now an ‘obsolescent feature’ and may disappear in a later version of the Standard.

Now on to the function definition, where the body is supplied. The definition shows that the function takes two arguments, which will be known as `a1` and `a2` throughout the body of the function. The types of the arguments are specified too, as can be seen.

In the function definition you don't *have* to specify the type of each argument because they will default to `int`, but this is bad style. If you adopt the practice of always declaring arguments, even if they do happen to be `int`, it adds to a reader's confidence. It indicates that you meant to use that type, instead of getting it by accident: it wasn't simply forgotten. The definition of `pmax` *could* have been this:

```
/* BAD STYLE OF FUNCTION DEFINITION */

void
pmax(a1, a2){
        /* and so on */
```

The proper way to declare and define functions is through the use of *prototypes*.

4.2.4. Function prototypes

The introduction of *function prototypes* is the biggest change of all in the Standard.

A function prototype is a function declaration or definition which includes information about the number and types of the arguments that the function takes.

Although you are allowed not to specify any information about a function's arguments in a declaration, it is purely because of backwards compatibility with Old C and should be avoided.

A declaration without any information about the arguments is *not* a prototype.

Here's the previous example 'done right'.

```
#include <stdio.h>
#include <stdlib.h>

main(){
      void pmax(int first, int second);       /*declaration*/
      int i,j;
      for(i = -10; i <= 10; i++){
              for(j = -10; j <= 10; j++){
                      pmax(i,j);
              }
      }
      exit(EXIT_SUCCESS);
}

void
pmax(int a1, int a2){                            /*definition*/
      int biggest;

      if(a1 > a2){
              biggest = a1;
      }
      else{
              biggest = a2;
      }

      printf("largest of %d and %d is %d\n",
              a1, a2, biggest);
}
```

Example 4.3

This time, the declaration provides information about the function arguments, so it's a prototype. The names `first` and `second` are not an essential part of the declaration, but they are allowed to be there because it makes it easier to refer to named arguments when you're documenting the use of the function. Using them, we can describe the function simply by giving its declaration

```
  void pmax (int xx, int yy );
```

and then say that `pmax` prints whichever of the arguments `xx` or `yy` is the larger. Referring to arguments by their position, which is the alternative (e.g. the fifth argument), is tedious and prone to miscounting.

All the same, you can miss out the names if you want to. This declaration is entirely equivalent to the one above.

```
  void pmax (int,int);
```

All that is needed is the type names.

For a function that has no arguments the declaration is

```
  void f_name (void);
```

and a function that has one `int`, one `double` and an unspecified number of other arguments is declared this way:

```
  void f_name (int,double,...);
```

The ellipsis (...) shows that other arguments follow. That's useful because it allows functions like `printf` to be written. Its declaration is this:

```
  int printf (const char *format_string,...)
```

where the type of the first argument is ‘pointer to `const char`’; we'll discuss what that means later.

Once the compiler knows the types of a function's arguments, having seen them in a prototype, it's able to check that the use of the function conforms to the declaration.

If a function is called with arguments of the wrong type, the presence of a prototype means that the actual argument is converted to the type of the formal argument ‘as if by assignment’. Here's an example: a function is used to evaluate a square root using Newton's method of successive approximations.

```
#include <stdio.h>
#include <stdlib.h>
#define DELTA 0.0001
main(){
      double sq_root(double); /* prototype */
      int i;

      for(i = 1; i < 100; i++){
              printf("root of %d is %f\n", i, sq_root(i));
      }
      exit(EXIT_SUCCESS);
}

double
sq_root(double x){      /* definition */
      double curr_appx, last_appx, diff;

      last_appx = x;
      diff = DELTA+1;

      while(diff > DELTA){
              curr_appx = 0.5*(last_appx
                      + x/last_appx);
              diff = curr_appx - last_appx;
              if(diff < 0)
                      diff = -diff;
              last_appx = curr_appx;
      }
      return(curr_appx);
}
```

Example 4.4

The prototype tells everyone that `sq_root` takes a single argument of type `double`. The argument actually passed in the main function is an `int`, so it has to be converted to `double` first. The critical point is that if no prototype had been seen, C would assume that the programmer had meant to pass an `int` and an `int` is what would be passed. The Standard simply notes that this results in undefined behaviour, which is as understated as saying that catching rabies is unfortunate. This is a *very serious error* and has led to many, many problems in Old C programs.

The conversion of `int` to `double` could be done because the compiler had seen a protoytpe for the function and knew what to do about it. As you would expect, there are various rules used to decide which conversions are appropriate, so we need to look at them next.

4.2.5. Argument Conversions

When a function is called, there are a number of possible conversions that will be applied to the values supplied as arguments depending on the presence or absence of a prototype. Let's get one thing clear: although you *can* use these rules to work out what to do if you haven't used prototypes, it is a recipe for pain and misery in the long run. It's so easy to use prototypes that there really is no excuse for not having them, so the only time you will need to use these rules is if you are being adventurous and using functions with a variable number of arguments, using the ellipsis notation in the prototype that is explained in Chapter 9 [*http://publications.gbdirect.co.uk/c_book/chapter9/*].

The rules mention the *default argument promotions and compatible type*. Where they are used, the default argument promotions are:

- Apply the integral promotions (see Chapter 2 [*http://publications.gbdirect.co.uk/c_book/chapter2/*]) to the value of each argument
- If the type of the argument is `float` it is converted to `double`

The introduction of prototypes (amongst other things) has increased the need for precision about ‘compatible types’, which was not much of an issue in Old C. The full list of rules for type compatibility is deferred until Chapter 8 [*http://publications.gbdirect.co.uk/c_book/chapter8/*], because we suspect that most C programmers will never need to learn them. For the moment, we will simply work on the basis that if two types are the same, they are indisputably compatible.

The conversions are applied according to these rules (which are intended to be guidance on how to apply the Standard, not a direct quote):

1. At the point of calling a function, if no prototype is in scope, the arguments all undergo the default argument promotions. Furthermore:
   - If the number of arguments does not agree with the number of formal parameters to the function, the behaviour is undefined.
   - If the function definition was not a definition containing a prototype, then the type of the actual arguments after promotion must be *compatible* with the types of the formal parameters in the definition after they too have had the promotions applied. Otherwise the behaviour is undefined.
   - If the function definition was a definition containing a prototype, and the types of the actual arguments after promotion are not compatible with the formal parameters in the prototype, then the behaviour is undefined. The behaviour is also undefined it the prototype included ellipsis (`, ...`).

2. At the point of calling a function, if a prototype *is* in scope, the arguments are converted, as if by assignment, to the types specified in the prototype. Any arguments which fall under the variable argument list category (specified by the `...` in the prototype) still undergo the default argument conversions.

   It *is* possible to write a program so badly that you have a prototype in scope when you call the function, but for the function definition itself not to have a prototype. Why anyone should do this is a mystery, but in this case, the function that is called must have a type that is *compatible* with the apparent type at the point of the call.

The order of evaluation of the arguments in the function call is explicitly not defined by the

Standard.

4.2.6. Function definitions

Function prototypes allow the same text to be used for both the declaration and definition of a function. To turn a declaration:

```
double
some_func(int a1, float a2, long double a3);
```

into a definition, we provide a body for the function:

```
double
some_func(int a1, float a2, long double a3){
      /* body of function */
      return(1.0);
}
```

by replacing the semicolon at the end of the declaration with a compound statement.

In either a definition or a declaration of a function, it serves as a prototype if the parameter types are specified; both of the examples above are prototypes.

The Old C syntax for the declaration of a function's formal arguments is still supported by the Standard, although it should not be used by new programs. It looks like this, for the example above:

```
double
some_func(a1, a2, a3)
      int a1;
      float a2;
      long double a3;
{

      /* body of function */
      return(1.0);
}
```

Because no type information is provided for the parameters at the point where they are named, this form of definition does *not* act as a prototype. It declares only the return type of the function; nothing is remembered by the compiler about the types of the arguments at the end of the definition.

The Standard warns that support for this syntax may disappear in a later version. It will not be discussed further.

Summary

1. Functions can be called recursively.
2. Functions can return any type that you can declare, except for arrays and functions (you can get around that restriction to some extent by using pointers). Functions returning no value should return `void`.
3. Always use function prototypes.
4. Undefined behaviour results if you call or define a function anywhere in a program unless either
   - a prototype is *always* in scope for *every call* or definition, or
   - you are very, very careful.
5. Assuming that you *are* using prototypes, the values of the arguments to a function call are converted to the types of the formal parameters exactly as if they had been assigned using the = operator.

6. Functions taking no arguments should have a prototype with (`void`) as the argument specification.

7. Functions taking a variable number of arguments must take at least one named argument; the variable arguments are indicated by `...` as shown:

   ```
   int
   vfunc(int x, float y, ...);
   ```

   Chapter 9 [*http://publications.gbdirect.co.uk/c_book/chapter9/*] describes how to write this sort of function.

4.2.7. Compound statements and declarations

As we have seen, functions always have a compound statement as their body. It is possible to declare new variables inside any compound statement; if any variables of the same name already exist, then the old ones are hidden by the new ones within the new compound statement. This is the same as in every other block-structured language. C restricts the declarations to the head of the compound statement (or ‘block’); once any other kind of statement has been seen in the block, declarations are no longer permitted within that block.

How can it be possible for names to be hidden? The following example shows it happening:

```
int a;                  /* visible from here onwards */

void func(void){
      float a;        /* a different 'a' */
      {
              char a; /* yet another 'a' */
      }
                      /* the float 'a' reappears */
}
                      /* the int 'a' reappears */
```

Example 4.5

A name declared inside a block hides any outer versions of the same name until the end of the block where it is declared. Inner blocks can also re-declare that name—you can do this for ever.

The *scope* of a name is the range in which it has meaning. Scope starts from the point at which the name is mentioned and continues from there onwards to the end of the block in which it is declared. If it is external (outside of any function) then it continues to the end of the file. If it is internal (inside a function), then it disappears at the end of the block containing it. The scope of any name can be suspended by redeclaring the name inside a block.

Using knowledge of the scope rules, you can play silly tricks like this one:

```
main () {}
int i;
f () {}
f2 () {}
```

Now `f` and `f2` can use `i`, but `main` can't, because the declaration of the variable comes later than that of `main`. This is not an aspect that is used very much, but it is implicit in the way that C processes declarations. It is a source of confusion for anyone reading the file (external declarations are generally expected to precede any function definitions in a file) and should be avoided.

The Standard has changed things slightly with respect to a function's formal parameters. They are now considered to have been declared inside the first compound statement, even though textually they aren't: this goes for both the new and old ways of function definition. So, if a function has a formal parameter with the same name as something declared in the outermost compound statement, this causes an error which will be detected by the compiler.

In Old C, accidental redefinition of a function's formal parameter was a horrible and particularly difficult mistake to track down. Here is what it would look like:

```
/* erroneous redeclaration of arguments */

func(a, b, c){
        int a;  /* AAAAgh! */
}
```

The pernicious bit is the new declaration of a in the body of the function, which hides the parameter called `a`. Since the problem has now been eliminated we won't investigate it any further.

Footnotes

1. Stroustrup B. (1991). *The C++ Programming Language* 2nd edn. Reading, MA: Addison-Wesley

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter4/changes.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter4/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter4/recursion_and_argument_passing.html*]

4.3. Recursion and argument passing

<gbdirect>

So far, we've seen how to give functions a type (how to declare the return value and the type of any arguments the function takes), and how the definition is used to give the body of the function. Next we need to see what the arguments can be used for.

4.3.1. Call by value

The way that C treats arguments to functions is both simple and consistent, with no exceptions to the single rule.

When a function is called, any arguments that are provided by the caller are simply treated as expressions. The value of each expression has the appropriate conversions applied and is then used to initialize the corresponding formal parameter in the called function, which behaves in exactly the same way as any other local variables in the function. It's illustrated here:

```
void called_func(int, float);

main(){
      called_func(1, 2*3.5);
      exit(EXIT_SUCCESS);
}

void
called_func(int iarg, float farg){
      float tmp;

      tmp = iarg * farg;
}
```

Example 4.6

The arguments to `called_func` in `main` are two expressions, which are evaluated. The value of each expression is used to initialize the parameters `iarg` and `farg` in `called_func`, and the parameters are indistinguishable from the other local variable declared in `called_func`, which is `tmp`.

The initialization of the formal parameters is the last time that any communication occurs between the caller and the called function, except for the return value.

For those who are used to FORTRAN and `var` arguments in Pascal, where a function *can* change the values of its arguments: forget it. You cannot affect the values of a function's actual arguments by anything that you try. Here is an example to show what we mean.

```
#include <stdio.h>
#include <stdlib.h>
main(){
      void changer(int);
```

```
        int i;

        i = 5;
        printf("before i=%d\n", i);
        changer(i);
        printf("after i=%d\n", i);
        exit(EXIT_SUCCESS);
}

void
changer(int x){
        while(x){
                printf("changer: x=%d\n", x);
                x--;
        }
}
```

Example 4.7

The result of running that is:

```
before i=5
changer: x=5
changer: x=4
changer: x=3
changer: x=2
changer: x=1
after i=5
```

The function `changer` uses its formal parameter `x` as an ordinary variable—which is exactly what it is. Although the value of `x` is changed, the variable `i` (in main) is unaffected. That is the whole point—the arguments in C are passed into a function by their value only, no changes made by the function are passed back.

4.3.2. Call by reference

It is possible to write functions that take *pointers* as their arguments, giving a form of call by reference. This is described in Chapter 5 [*http://publications.gbdirect.co.uk/c_book/chapter5/*] and *does* allow functions to change values in their callers.

4.3.3. Recursion

With argument passing safely out of the way we can look at recursion. Recursion is a topic that often provokes lengthy and unenlightening arguments from opposing camps. Some think it is wonderful, and use it at every opportunity; some others take exactly the opposite view. Let's just say that when you need it, you really *do* need it, and since it doesn't cost much to put into a language, as you would expect, C supports recursion.

Every function in C may be called from any other or itself. Each invocation of a function causes a new allocation of the variables declared inside it. In fact, the declarations that we have been using until now have had something missing: the keyword `auto`, meaning ‘automatically allocated’.

```
/* Example of auto */
main(){
        auto int var_name;
        .
```

```
          .
          .
}
```

The storage for auto variables is automatically allocated and freed on function entry and return. If two functions both declare large automatic arrays, the program will only have to find room for both arrays if both functions are active at the same time. Although auto is a keyword, it is never used in practice because it's the default for internal declarations and is invalid for external ones. If an explicit initial value (see ‘ initialization’) isn't given for an automatic variable, then its value will be unknown when it is declared. In that state, any use of its value will cause undefined behaviour.

The real problem with illustrating recursion is in the selection of examples. Too often, simple examples are used which don't really get much out of recursion. The problems where it really helps are almost always well out of the grasp of a beginner who is having enough trouble trying to sort out the difference between, say, definition and declaration without wanting the extra burden of having to wrap his or her mind around a new concept as well. The chapter on data structures will show examples of recursion where it is a genuinely useful technique.

The following example uses recursive functions to evaluate expressions involving single digit numbers, the operators `*`, `%`, `/`, `+`, `-` and parentheses in the same way that C does. (Stroustrup[1] [*http://publications.gbdirect.co.uk/c_book/chapter4/recursion_and_argument_passing.html#foot1*], in his book about C++, uses almost an identical example to illustrate recursion. This happened purely by chance.) The whole expression is evaluated and its value printed when a character not in the ‘language’ is read. For simplicity no error checking is performed. Extensive use is made of the `ungetc` library function, which allows the last character read by `getchar` to be ‘unread’ and become once again the next character to be read. Its second argument is one of the things declared in `stdio.h`.

Those of you who understand BNF notation might like to know that the expressions it will understand are described as follows:

```
<primary> ::= digit | (<exp>)
<unary>   ::= <primary> | -<unary> | +<unary>
<mult>    ::= <unary> | <mult> * <unary> |
              <mult> / <unary> | <mult> % <unary>
<exp>     ::= <exp> + <mult> | <exp> - <mult> | <mult>
```

The main places where recursion occurs are in the function `unary_exp`, which calls itself, and at the bottom level where `primary` calls the top level all over again to evaluate parenthesized expressions.

If you don't understand what it does, try running it. Trace its actions by hand on inputs such as

```
1
1+2
1+2 * 3+4
1+--4
1+(2*3)+4
```

That should keep you busy for a while!

```
/*
* Recursive descent parser for simple C expressions.
* Very little error checking.
*/
```

```
#include <stdio.h>
#include <stdlib.h>

int expr(void);
int mul_exp(void);
int unary_exp(void);
int primary(void);

main(){
      int val;

      for(;;){
              printf("expression: ");
              val = expr();
              if(getchar() != '\n'){
                      printf("error\n");
                      while(getchar() != '\n')
                              ; /* NULL */
              } else{
                      printf("result is %d\n", val);
              }
      }
      exit(EXIT_SUCCESS);
}

int
expr(void){
      int val, ch_in;

      val = mul_exp();
      for(;;){
              switch(ch_in = getchar()){
              default:
                      ungetc(ch_in,stdin);
                      return(val);
              case '+':
                      val = val + mul_exp();
                      break;
              case '-':
                      val = val - mul_exp();
                      break;
              }
      }
}

int
mul_exp(void){
      int val, ch_in;

      val = unary_exp();
      for(;;){
              switch(ch_in = getchar()){
              default:
                      ungetc(ch_in, stdin);
                      return(val);
              case '*':
                      val = val * unary_exp();
                      break;
```

```
                case '/':
                        val = val / unary_exp();
                        break;
                case '%':
                        val = val % unary_exp();
                        break;
                }
        }
}

int
unary_exp(void){
        int val, ch_in;

        switch(ch_in = getchar()){
        default:
                ungetc(ch_in, stdin);
                val = primary();
                break;
        case '+':
                val = unary_exp();
                break;
        case '-':
                val = -unary_exp();
                break;
        }
        return(val);
}

int
primary(void){
        int val, ch_in;

        ch_in = getchar();
        if(ch_in >= '0' && ch_in <= '9'){
                val = ch_in - '0';
                goto out;
        }
        if(ch_in == '('){
                val = expr();
                getchar();      /* skip closing ')' */
                goto out;
        }
        printf("error: primary read %d\n", ch_in);
        exit(EXIT_FAILURE);
out:
        return(val);
}
```

Example 4.8

Footnotes

1. Stroustrup B. (1991). *The C++ Programming Language* 2nd edn. Reading, MA: Addison-Wesley

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter4/function_types.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter4/*] | Next section

[*http://publications.gbdirect.co.uk/c_book/chapter4/linkage.html*]

4.4. Linkage

Although the simple examples have carefully avoided the topic, we now have to look into the effects of scope and linkage, terms used to describe the accessibility of various objects in a C program. Why bother? It's because realistic programs are built up out of multiple files and of course libraries. It is clearly crucial that that functions in one file should be able to refer to functions (or other objects) in other files and libraries; naturally there are a number of concepts and rules that apply to this mechanism.

If you are relatively new to C, there are more important subjects to cover first. Come back to this stuff later instead.

There are essentially two types of object in C: the internal and external objects. The distinction between external and internal is to do with functions: anything declared outside a function is external, anything inside one, including its formal parameters, is internal. Since no function can be defined inside another, functions themselves are always external. At the outermost level, a C program is a collection of *external objects*.

Only external objects participate in this cross-file and library communication.

The term used by the Standard to describe the accessibility of objects from one file to another, or even within the same file, is *linkage*. There are three types of linkage: *external linkage*, *internal linkage* and *no linkage*. Anything internal to a function—its arguments, variables and so on—*always* has no linkage and so can only be accessed from inside the function itself. (The way around this is to declare something inside a function but prefix it with the keyword `extern` which says ‘it isn't really internal’, but we needn't worry about that just yet.)

Objects that have external linkage are all considered to be located at the outermost level of the program; this is the default linkage for functions and anything declared outside of a function. *All instances of a particular name with external linkage refer to the same object in the program.* If two or more declarations of the same name have external linkage but incompatible types, then you've done something very silly and have undefined behaviour. The most obvious example of external linkage is the printf function, whose declaration in `<stdio.h>` is

```
int printf(const char *, ...);
```

From that we can tell that it's a function returning int and with a particular prototype—so we know everything about its type. We also know that it has external linkage, because that is the default for every external object. As a result, everywhere that the name `print` is used with external linkage, we are referring to this function.

Quite often, you want to be able to declare functions and other objects within a single file in a way that allows them to reference each other but *not* to be accessible from outside that file. This is often necessary in the modules that support library functions, where the additional framework that makes those functions work is not interesting to the user and would be a positive nuisance if the names of those things became visible outside the module. You do it through the use of *internal linkage*.

Names with internal linkage only refer to the same object within a single source file. You do this by prefixing their declarations with the keyword `static`, which changes the linkage of external objects from external linkage to internal linkage. It is also possible to declare internal objects to be `static`, but that has an entirely different meaning which we can defer for the moment.

It's confusing that the types of linkage and the types of object are both described by the terms ‘internal’ and ‘external’; this is to some extent historical. C archaeologists may know that at one time the two were equivalent and one implied the other—for us it's unfortunate that the terms remain but the meanings have diverged. To summarize:

Type of linkage	Type of object	Accessibility
external	external	throughout the program
internal	external	a single file
none	internal	local to a single function

Table 4.1. Linkage and accessibility

Finally, before we see an example, it is important to know that all objects with external linkage must have one and only one definition, although there can be as many compatible declarations as you like. Here's the example.

```
/* first file */

int i; /* definition */
main () {
  void f_in_other_place (void);   /* declaration */
  i = 0
}
/* end of first file */


/* start of second file */

extern int i; /* declaration */
void f_in_other_place (void){   /* definition */
  i++;
}
/* end of second file */
```

Example 4.9

Although the full set of rules is a bit more complex, the basic way of working out what constitutes a definition and a declaration is not hard:

- A function declaration without a body for the function is just a declaration.
- A function declaration with a body for the function is a definition.
- At the external level, a declaration of an object (like the variable`i`) is a definition unless it has the keyword `extern` in front of it, when it is a declaration only.

Chapter 8 [*http://publications.gbdirect.co.uk/c_book/chapter8/*] revisits the definition and declaration criteria to a depth that will cause decompression sickness when you surface.

In the example it's easy to see that each file is able to access the objects defined in the other by using their names. Just from that example alone you should be able to work out how to construct programs with multiple files and functions and variables declared or defined as appropriate in each of them.

Here's another example, using `static` to restrict the accessibility of functions and other things.

```
/* example library module */
/* only 'callable' is visible outside */
static buf [100];
static length;
static void fillup(void);

int
callable (){
      if (length ==0){
              fillup ();
      }
      return (buf [length--]);
}

static void
fillup (void){
      while (length <100){
              buf [length++] = 0;
      }
}
```

Example 4.10

A user of this module can safely re-use the names declared here, `length`, `buf`, and `fillup`, without any danger of surprising effects. Only the name `callable` is accessible outside this module.

A very useful thing to know is that any external object that has no other initalizer (and except for functions we haven't seen any initializers yet) is always set to the value of zero before the program starts. This is widely used and relied on—the previous example relies on it for the initial value of `length`.

4.4.1. Effect of scope

There's one additional complicating factor beyond simply linkage. Linkage allows you to couple names together on a per-program or a per-file basis, but scope determines the visibility of the names. Fortunately, the rules of scope are completely independent of anything to do with linkage, so you don't have to remember funny combinations of both.

What introduces the complexity is the dreaded extern keyword. The nice regular block structure gets blown to pieces with this, which although at a first glance is simple and obvious, does some very nasty things to the fabric of the language. We'll leave its nasty problems to Chapter 8 [*http://publications.gbdirect.co.uk/c_book/chapter8/*], since they only rear up if you deliberately start to do perverse things with it and then say ‘what does this mean’? We've already seen it used to ensure that the declaration of something at the outer block level (the external level) of the program is a declaration and not a definition (but beware: you can still override the `extern` by, for example, providing an *initializer* for the object).

Unless you prefix it with `extern`, the declaration of any data object (not a function) at the outer level is also a definition. Look back to Example 4.9 to see this in use.

All function declarations implicitly have the `extern` stuck in front of them, whether or not you put it there too. These two ways of declaring `some_function` are equivalent and are always declarations:

```
void some_function(void);

extern void some_function(void);
```

The thing that mysteriously turns those declarations into definitions is that when you also provide the body of the function, that is effectively the initializer for the function, so the comment about initializers comes into effect and the declaration becomes a definition. So far, no problem.

Now, what is going on here?

```
void some_function(void){
      int i_var;
      extern float e_f_var;
}

void another_func(void){
      int i;
      i = e_f_var;    /* scope problem */
}
```

What happened was that although the declaration of `e_f_var` declares that something called `e_f_var` is of type `float` and is accessible throughout the entire program, the *scope* of the name disappears at the end of the function that contains it. That's why it is meaningless inside `another_func`—the name of `e_f_var` is out of scope, just as much as `i_var` is.

So what use is that? It's sometimes handy if you only want to make use of an external object from within a single function. If you followed the usual practice and declared it at the head of the particular source file, then there is no easy way for the reader of that file to see which functions actually use it. By restricting the access and the scope of the name to the place where is needed, you do communicate to a later reader of the program that this is a very restricted use of the name and that there is no intention to make widespread use of it throughout the file. Of course, any half-way decent cross-reference listing would communicate that anyway, so the argument is a bit hard to maintain.

Chapter 8 [*http://publications.gbdirect.co.uk/c_book/chapter8/*] is the place to find out more. There's a set of guidelines for how to get the results that are most often wanted from multi-file construction, and a good deal more detail on what happens when you mix `extern`, `static` and internal and external declarations. It isn't the sort of reading that you're likely to do for pleasure, but it does answer the 'what if' questions.

4.4.2. Internal static

You are also allowed to declare internal objects as `static`. Internal variables with this attribute have some interesting properties: they are initialized to zero when the program starts, they retain their value between entry to and exit from the statement containing their declaration and there is only one copy of each one, which is shared between all recursive calls of the function containing it.

Internal statics can be used for a number of things. One is to count the number of times that a function has been called; unlike ordinary internal variables whose value is lost after leaving their function, statics are convenient for this. Here's a function that always returns a number between 0 and 15, but remembers how often it was called.

```
int
small_val (void) {
      static unsigned count;
      count ++;
      return (count % 16);
}
```

Example 4.11

They can help detect excessive recursion:

```
void
```

```
r_func (void){
      static int depth;
      depth++;
      if (depth > 200) {
              printf ("excessive recursion\n");
              exit (1);
      }
      else {
              /* do usual thing,
               * not shown here.
               * This last action
               * occasionally results in another
               * call on r_func()
               */
              x_func();
      }
      depth--;
}
```

Example 4.12

Footnotes

1. Stroustrup B. (1991). *The C++ Programming Language* 2nd edn. Reading, MA: Addison-Wesley

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter4/recursion_and_argument_passing.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter4/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter4/summary.html*]

4.5. Summary

With the appropriate declarations, you can have names that are visible throughout the program or limited to a single file or limited to a single function, as appropriate.

Here are the combinations of the use of the keywords, the types of declarations and the resulting linkage:

Declaration	Keyword	Resulting Linkage	Accessibility	Note
external	none	external	entire program	2
external	`extern`	external	entire program	2
external	`static`	internal	a single file	2
internal	none	none	a single function	
internal	`extern`	external	entire program	1
internal	`static`	none	a single function	2

Although the accessibility of internal declarations prefixed with `extern` is program-wide, watch out for the scope of the name.

External (or internal `static`) objects are initialized once only, at program start-up. The absence of explicit initialization is taken to be a default initialization of zero.

Table 4.2. Summary of Linkage

There are a few golden rules for the use of functions that are worth re-stating too.

- To use a function returning other than `int`, a declaration or definition must be in scope.
- Do not return from a function by falling out of its body unless its type is `void`.

A declaration of the types of arguments that a function takes is not mandatory, but it is extremely strongly recommended.

Functions taking a variable number of arguments can be written portably if you use the methods described in Section 9.9 [*http://publications.gbdirect.co.uk/c_book/chapter9/stdarg.html*].

Functions are the cornerstone of C. Of all the changes to the language, the Standard has had by far its most obvious effect by introducing function prototypes. This change has won widespread approval throughout the user community and should help to produce a substantial improvement in reliability of C programs, as well as opening the possibility of optimization by compilers in areas previously closed to them.

The use of call-by-value is sometimes surprising to people who have used languages that prefer a different mechanism, but at least the C approach is the ‘safest’ most of the time.

The attempts by the Standard to remove ambiguity in the scope and meaning of declarations are interesting, but frankly have explored an obscure region which rarely caused any difficulties in practice.

From the beginner's point of view, it is important to learn thoroughly everything discussed in this chapter, perhaps with the exception of the linkage rules. They can be deferred for a more leisurely inspection at some later time.

Footnotes

1. Stroustrup B. (1991). *The C++ Programming Language* 2nd edn. Reading, MA: Addison-Wesley

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter4/linkage.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter4/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter4/exercises.html*]

4.6. Exercises

If you skipped the section on Linkage, then Exercise 4.2, Exercise 4.3, and Exercise 4.4 will cause you problems; it's up to you whether or not you want to read it and then try them.

Write a function and the appropriate declaration for the following tasks:

Exercise 4.1. A function called `abs_val` that returns `int` and takes an `int` argument. It returns the absolute value of its argument, by negating it if it is negative.

Exercise 4.2. A function called `output` that takes a single character argument and sends it to the program output with `putchar`. It will remember the current line number and column number reached on the output device—the only values passed to the function are guaranteed to be alphanumeric, punctuation, space and newline characters.

Exercise 4.3. Construct a program to test `output`, where that function is in a separate file from the functions that are used to test it. In the same file as `output` will be two functions called `current_line` and `current_column` which return the values of the line and column counters. Ensure that those counters are made accessible only from the file that contains them.

Exercise 4.4. Write and test a recursive function that performs the admittedly dull task of printing a list of numbers from 100 down to 1. On entry to the function it increments a static variable. If the variable has a value below 100, it calls itself again. Then it prints the value of the variable, decrements it and returns. Check that it works.

Exercise 4.5. Write functions to calculate the sine and cosine of their input. Choose appropriate types for both argument and return value. The series (given below) can be used to approximate the answer. The function should return when the value of the final term is less than 0.000001 of the current value of the function.

```
sin x = x - pow(x,3)/fact(3) + pow(x,5)/fact(5)...
cos x = 1 - pow(x,2)/fact(2) + pow(x,5)/fact(5)...
```

Note the fact that the sign in front of each term alternates (`--+--+--+...`). `pow(x,n)` returns `x` to the *n*th power, `fact(n)` factorial of `n` (1 × 2 × 3 × ⋯ × *n*). You will have to write such functions. Check the results against published tables.

Footnotes

1. Stroustrup B. (1991). *The C++ Programming Language* 2nd edn. Reading, MA: Addison-Wesley

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter4/summary.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter4/*]

Chapter 5

<gbdirect>

Arrays and Pointers

Previous chapter [*http://publications.gbdirect.co.uk/c_book/chapter4/*] | Next chapter [*http://publications.gbdirect.co.uk/c_book/chapter6/*]

5.1. Opening shots

<gbdirect>

5.1.1. So why is this important?

The arithmetic data types and operators of C are interesting but hardly rivetting. They show, collectively, a certain imagination and spirit that has stamped C with a special flavour, but they form the sauce, not the meat, of this particular dish. For most users, it's functions and the parts of the language covered in this chapter that provide the real feel of C.

For the new reader, this is the part of the language that causes the biggest problems. Most beginners with C are at least familiar with the use of arithmetic, functions and arrays; those are not the problem areas. The difficulties arise when we get on to the structured types (structures and unions), and the way that C just wouldn't be C without the use of pointers.

Pointers aren't a feature that you can choose to ignore. They're used everywhere; their influence affects the whole language and must be the single most noticeable feature of all but the simplest C programs. If you think that this is one of the bits you can skip because it's hard and doesn't look too important, you are wrong! Most of the examples used so far in this book have had pointers used in them (although not obviously), so you might as well accept the inevitable and learn how to use them properly.

The most natural way to introduce the use of pointers is by looking into arrays first. C intertwines arrays and pointers so closely that they are hard to separate. Since you are expected to be familiar with the use of arrays, their treatment will be brief and aimed at using them to illustrate the use of pointers when they are seen later.

5.1.2. Effect of the Standard

The new Standard has left very little mark on the contents of this chapter; a lot of it would be nearly word for word the same even if it only talked about Old C. The inference to be drawn is that nothing was wrong with the old version of the language, and that there was nothing to be gained by fixing what wasn't broken. This may be received with some relief by those readers who already knew this part of the old language and who, like the Committee, felt that it was good enough to leave alone.

Even so, the introduction of *qualified types* by the Standard does add some complexity to this chapter. The rules about exactly how the various arithmetic and relational operators work when they are applied to pointers have been clarified, which adds bulk to the text but has not changed things substantially. In the early examples we do not pay a lot of attention to them, but after that they are introduced gradually and where appropriate.

Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter5/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter5/arrays.html*]

5.2. Arrays

<gbdirect>

Like other languages, C uses arrays as a way of describing a collection of variables with identical properties. The group has a single name for all of the members, with the individual members being selected by an *index*. Here's an array being declared:

```
double ar[100];
```

The name of the array is `ar` and its members are accessed as `ar[0]` through to `ar[99]` inclusive, as Figure 5.1 shows.

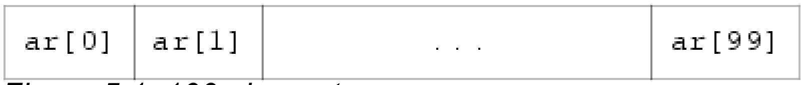


Figure 5.1. 100 element array

Each of the hundred members is a separate variable whose type is `double`. Without exception, all arrays in C are numbered from `0` up to one less than the bound given in the declaration. This is a prime cause of surprise to beginners—watch out for it. For simple examples of the use of arrays, look back at earlier chapters where several problems are solved with their help.

One important point about array declarations is that they don't permit the use of varying subscripts. The numbers given must be constant expressions which can be evaluated at compile time, not run time. For example, this function incorrectly tries to use its argument in the size of an array declaration:

```
f(int x){
      char var_sized_array[x];        /* FORBIDDEN */
}
```

It's forbidden because the value of x is unknown when the program is compiled; it's a run-time, not a compile-time, value.

To tell the truth, it would be easy to support arrays whose *first* dimension is variable, but neither Old C nor the Standard permits it, although we do know of one Very Old C compiler that used to do it.

5.2.1. Multidimensional arrays

Multidimensional arrays can be declared like this:

```
int three_dee[5][4][2];
int t_d[2][3]
```

The use of the brackets gives a clue to what is going on. If you refer to the precedence table given in Section 2.8.3 [*http://publications.gbdirect.co.uk/c_book/chapter2/expressions_and_arithmetic.html#section-3*] (Table 2.9), you'll see that `[]` associates left to right and that, as a result, the first declaration gives us a five-element array called three_dee. The members of that array are each a four element array whose members are an array of two ints. We have declared arrays of arrays, as Figure 5.2 shows for two dimensions.

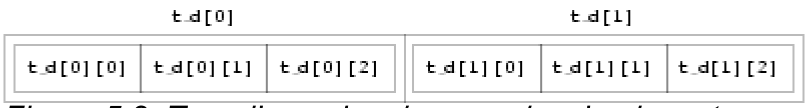


Figure 5.2. Two-dimensional array, showing layout

In the diagram, you will notice that `t_d[0]` is one element, immediately followed by `t_d[1]` (there is no break). It so happens that both of those elements are themselves arrays of three integers. Because of C's storage layout rules, `t_d[1][0]` is immediately after `t_d[0][2]`. It would be possible (but very poor practice) to access `t_d[1][0]` by making use of the lack of array-bound checking in C, and to use the expression `t_d[0][3]`. That is not recommended—apart from anything else, if the declaration of `t_d` ever changes, then the results will be likely to surprise you.

That's all very well, but does it really matter in practice? Not much it's true; but it is interesting to note that in terms of actual machine storage layout the rightmost subscript 'varies fastest'. This has an impact when arrays are accessed via pointers. Otherwise, they can be used just as would be expected; expressions like these are quite in order:

```
three_dee[1][3][1] = 0;
three_dee[4][3][1] += 2;
```

The second of those is interesting for two reasons. First, it accesses the very last member of the entire array—although the subscripts were declared to be `[5][4][2]`, the highest usable subscript is always one less than the one used in the declaration. Second, it shows where the combined assignment operators are a real blessing. For the experienced C programmer it is much easier to tell that only one array member is being accessed, and that it is being incremented by two. Other languages would have to express it like this:

```
three_dee[4][3][1] = three_dee[4][3][1] + 2;
```

It takes a conscious effort to check that the same array member is being referenced on both sides of the assignment. It makes thing easier for the compiler too: there is only one array indexing calculation to do, and this is likely to result in shorter, faster code. (Of course a clever compiler would notice that the left- and right-hand sides look alike and would be able to generate equally efficient code—but not all compilers are clever and there are lots of special cases where even clever compilers are unable to make use of the information.)

It may be of interest to know that although C offers support for multidimensional arrays, they aren't particularly common to see in practice. One-dimensional arrays are present in most programs, if for no other reason than that's what strings are. Two dimensional arrays are seen occasionally, and arrays of higher order than that are most uncommon. One of the reasons is that the array is a rather inflexible data structure, and the ease of building and manipulating other types of data structures in C means that they tend to replace arrays in the more advanced programs. We will see more of this when we look at pointers.

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter5/opening_shots.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter5/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter5/pointers.html*]

5.3. Pointers

Using pointers is a bit like riding a bicycle. Just when you think that you'll never understand them—suddenly you do! Once learned the trick is hard to forget. There's no real magic to pointers, and a lot of readers will already be familiar with their use. The only peculiarity of C is how heavily it relies on the use of pointers, compared with other languages, and the relatively permissive view of what you can do with them.

5.3.1. Declaring pointers

Of course, just like other variables, you have to declare pointers before you can use them. Pointer declarations look much like other declarations: but don't be misled. When pointers are declared, the keyword at the beginning (c int, char and so on) declares the type of variable that the pointer will point to. The pointer itself is not of that type, it is of type pointer to that type. A given pointer only points to one particular type, not to all possible types. Here's the declaration of an array and a pointer:

```
int ar[5], *ip;
```

We now have an array and a pointer (see Figure 5.3):

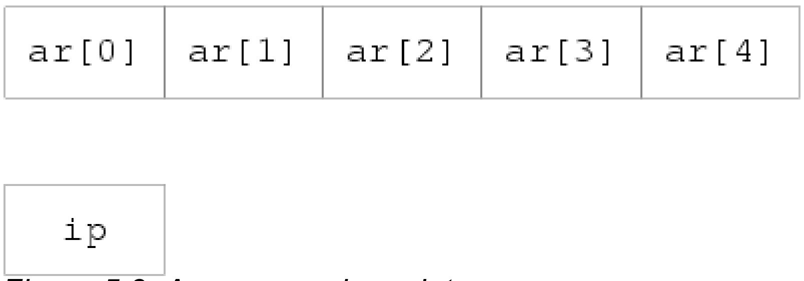


Figure 5.3. An array and a pointer

The `*` in front of `ip` in the declaration shows that it is a pointer, not an ordinary variable. It is of type `pointer to int`, and can only be used to refer to variables of type int. It's still uninitialized, so to do anything useful with it, it has to be made to point to something. You can't just stick some integer value into it, because integer values have the type `int`, not `pointer to int`, which is what we want. (In any case, what would it mean if this fragment were valid:

```
ip = 6;
```

What would `ip` be pointing to? In fact it could be construed to have a number of meanings, but the simple fact is that, in C, that sort of thing is just wrong.)

Here is the right way to initialize a pointer:

```
int ar[5], *ip;
ip = &ar[3];
```

In that example, the pointer is made to point to the member of the array `ar` whose index is `3`, i.e. the fourth member. This is important. You can assign values to pointers just like ordinary variables; the difference is simply in what the value means. The values of the variables that we have now are shown in Figure 5.4 (`??` means uninitialized).

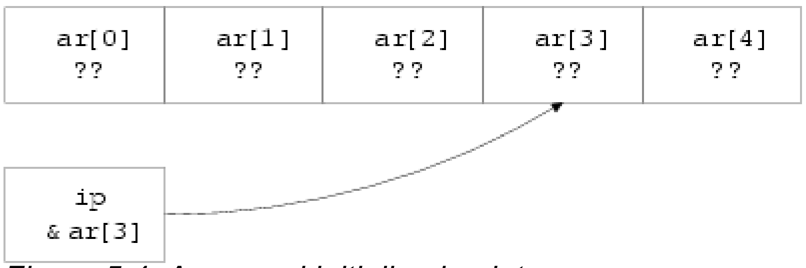


Figure 5.4. Array and initialized pointer

You can see that the variable `ip` has the value of the expression `&ar[3]`. The arrow indicates that, when used as a pointer, `ip` points to the variable `ar[3]`.

What is this new unary `&`? It is usually described as the ‘address-of’ operator, since on many systems the pointer will hold the store address of the thing that it points to. If you understand what addresses are, then you will probably have more trouble than those who don't: thinking about pointers as if they were addresses generally leads to grief. What seems a perfectly reasonable address manipulation on processor X can almost always be shown to be impossible on manufacturer Y's washing machine controller which uses 17-bit addressing when it's on the spin cycle, and reverses the order of odd and even bits when it's out of bleach. (Admittedly, it's unlikely that *anyone* could get C to work an an architecture like that. But you should see some of the ones it *does* work on; they aren't much better.)

We will continue to use the term ‘address of’ though, because to invent a different one would be even worse.

Applying the `&` operator to an operand returns a pointer to the operand:

```
int i;
float f;
      /* '&i' would be of type pointer to int */
      /* '&f' would be of type pointer to float */
```

In each case the pointer would point to the object named in the expression.

A pointer is only useful if there's some way of getting at the thing that it points to; C uses the unary `*` operator for this job. If `p` is of type ‘pointer to something’, then *p refers to the thing that is being pointed to. For example, to access the variable `x` via the pointer `p`, this would work:

```
#include <stdio.h>
#include <stdlib.h>
main(){
      int x, *p;

      p = &x;         /* initialise pointer */
      *p = 0;         /* set x to zero */
      printf("x is %d\n", x);
      printf("*p is %d\n", *p);

      *p += 1;        /* increment what p points to */
      printf("x is %d\n", x);

      (*p)++;         /* increment what p points to */
      printf("x is %d\n", x);

      exit(EXIT_SUCCESS);
}
```

Example 5.1

You might be interested to note that, since `&` takes the address of an object, returning a pointer to it, and since `*` means 'the thing pointed to by the pointer', the `&` and `*` in the combination `*&` effectively cancel each other out. (But be careful. Some things, constants for example, don't have addresses and the `&` operator cannot be applied to them; `&1.5` is not a pointer to anything, it's an error.) It's also interesting to see that C is one of the few languages that allows an expression on the left-hand side of an assignment operator. Look back at the example: the expression `*p` occurs twice in that position, and then the amazing `(*p)++;` statement. That last one is a great puzzle to most beginners—even if you've managed to wrap your mind around the concept that `*p = 0` writes zero into the thing pointed to by `p`, and that `*p += 1` adds one to where `p` points, it still seems a bit much to apply the `++` operator to `*p`.

The precedence of `(*p)++` deserves some thought. It will be given more later, but for the moment let's work out what happens. The brackets ensure that the * applies to p, so what we have is 'post-increment the thing pointed to by p'. Looking at Table 2.9, it turns out that ++ and * have equal precedence, but they associate right to left; in other words, without the brackets, the implied operation would have been *(p++), whatever that would mean. Later on you'll be more used to it—for the moment, we'll be careful with brackets to show the way that those expressions work.

So, provided that a pointer holds the address of something, the notation `*pointer` is equivalent to giving the name of the something directly. What benefit do we get from all this? Well, straight away it gets round the call-by-value restriction of functions. Imagine a function that has to return, say, two integers representing a month and a day within that month. The function has some (unspecified) way of determining these values; the hard thing to do is to return two separate values. Here's a skeleton of the way that it can be done:

```
#include <stdio.h>
#include <stdlib.h>
void
date(int *, int *);     /* declare the function */

main(){
      int month, day;
      date (&day, &month);
      printf("day is %d, month is %d\n", day, month);
      exit(EXIT_SUCCESS);
}

void
date(int *day_p, int *month_p){
      int day_ret, month_ret;
      /*
       * At this point, calculate the day and month
       * values in day_ret and month_ret respectively.
       */
      *day_p = day_ret;
      *month_p = month_ret;
}
```

Example 5.2

Notice carefully the advance declaration of `date` showing that it takes two arguments of type 'pointer to `int`'. It returns `void`, because the values are passed back via the pointers, not the usual return value. The `main` function passes pointers as arguments to date, which first uses the internal variables `day_ret` and `month_ret` for its calculations, then takes those values and assigns them to the places pointed to by its arguments.

When `date` is called, the situation looks like Figure 5.5.

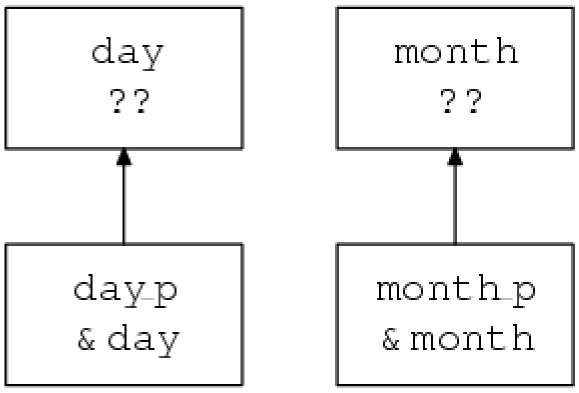


Figure 5.5. Just as `date` is called

The arguments have been passed to `date`, but in `main`, day and month are uninitialized. When date reaches the return statement, the situation is as shown in Figure 5.6 (assuming that the values for day and month are 12 and 5 respectively).

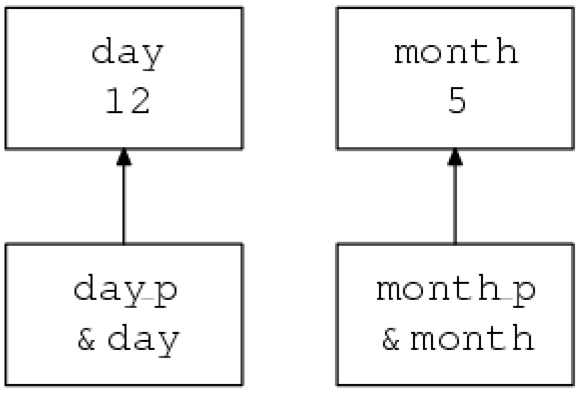


Figure 5.6. Just as `date` is about to return

One of the great benefits introduced by the new Standard is that it allows the types of the arguments to date to be declared in advance. A great favourite (and disastrous) mistake in C is to forget that a function expects pointers as its arguments, and to pass something else instead. Imagine what would have happened if the call of date above had read

```
date(day, month);
```

and no previous declaration of date had been visible. The compiler would not have known that date expects pointers as arguments, so it would pass the `int` values of `day` and `month` as the arguments. On a large number of computers, pointers and integers can be passed in the same way, so the function would execute, then pass back its return values by putting them into wherever `day` and `month` would point if their contents were pointers. This is very unlikely to give any sensible results, and in general causes unexpected corruption of data elsewhere in the computer's store. It can be extremely hard to track down!

Fortunately, by declaring `date` in advance, the compiler has enough information to warn that a mistake has almost certainly been made.

Perhaps surprisingly, it isn't all that common to see pointers used to give this call-by-reference functionality. In the majority of cases, call-by-value and a single return value are adequate. What is *much* more common is to use pointers to ‘walk’ along arrays.

5.3.2. Arrays and pointers

Array elements are just like other variables: they have addresses.

```
int ar[20], *ip;
```

```
ip = &ar[5];
*ip = 0;        /* equivalent to ar[5] = 0; */
```

The address of `ar[5]` is put into `ip`, then the place pointed to has zero assigned to it. By itself, this isn't particularly exciting. What *is* interesting is the way that pointer arithmetic works. Although it's simple, it's one of the cornerstones of C.

Adding an integral value to a pointer results in another pointer of the same type. Adding *n* gives a pointer which points n elements further along an array than the original pointer did. (Since *n* can be negative, subtraction is obviously possible too.) In the example above, a statement of the form

```
*(ip+1) = 0;
```

would set `ar[6]` to zero, and so on. Again, this is not obviously any improvement on ‘ordinary’ ways of accessing an array, but the following is.

```
int ar[20], *ip;

for(ip = &ar[0]; ip &lt; &ar[20]; ip++)
      *ip = 0;
```

That example is a classic fragment of C. A pointer is set to point to the start of an array, then, while it still points inside the array, array elements are accessed one by one, the pointer incrementing between each one. The Standard endorses existing practice by guaranteeing that it's permissible to use the *address* of `ar[20]` even though no such element exists. This allows you to use it for checks in loops like the one above. The guarantee only extends to one element beyond the end of an array and no further.

Why is the example better than indexing? Well, most arrays are accessed sequentially. Very few programming examples actually make use of the ‘random access’ feature of arrays. If you do just want sequential access, using a pointer can give a worthwhile improvement in speed. In terms of the underlying address arithmetic, on most architectures it takes one multiplication and one addition to access a one-dimensional array through a subscript. Pointers require no arithmetic at all—they nearly always hold the store address of the object that they refer to. In the example above, the only arithmetic that has to be done is in the `for` loop, where one comparison and one addition are done each time round the loop. The equivalent, using indexes, would be this:

```
int ar[20], i;
for(i = 0; i < 20; i++)
      ar[i] = 0;
```

The same amount of arithmetic occurs in the loop statement, but an extra address calculation has to be performed for every array access.

Efficiency is not normally an important issue, but here it can be. Loops often get traversed a substantial number of times, and every microsecond saved in a big loop can matter. It isn't always easy for even a smart compiler to recognize that this is the sort of code that could be ‘pointerized’ behind the scenes, and to convert from indexing (what the programmer wrote) to actually use a pointer in the generated code.

If you have found things easy so far, read on. If not, it's a good idea to skip to Section 5.3.3 [*http://publications.gbdirect.co.uk/c_book/chapter5/pointers.html#section-3*]. What follows, while interesting, isn't essential. It has been known to frighten even experienced C programmers.

To be honest, C doesn't really ‘understand’ array indexing, except in declarations. As far as the compiler is concerned, an expression like `x[n]` is translated into `*(x+n)` and use made of the fact that an array name is converted into a pointer to the array's first element whenever the name occurs in an expression. That's why, amongst other things, array elements count from zero: if `x` is an array name, then in an expression, `x` is equivalent to `&x[0]`, i.e. a pointer to the first element of the array. So, since `*(&x[0])` uses the pointer to get to `x[0]`, `*(&x[0] + 5)` is the same as `*(x + 5)` which is the same as `x[5]`. A curiosity springs out of all this. If `x[5]` is translated into `*(x + 5)`, and the expression `x + 5` gives the same result as `5 + x` (it does), then `5[x]` should give the identical result to `x[5]`! If you don't believe that, here is a program that compiles and runs successfully:

```
#include <stdio.h>
#include <stdlib.h>
#define ARSZ 20
main(){
      int ar[ARSZ], i;
      for(i = 0; i < ARSZ; i++){
              ar[i] = i;
              i[ar]++;
              printf("ar[%d] now = %d\n", i, ar[i]);
      }

      printf("15[ar] = %d\n", 15[ar]);
      exit(EXIT_SUCCESS);
}
```

Example 5.3

Summary

- Arrays always index from zero—end of story.
- There are no multidimensional arrays; you use arrays of arrays instead.
- Pointers point to things; pointers to different types are themselves different types. They have nothing in common with each other or any other types in C; there are no automatic conversions between pointers and other types.
- Pointers can be used to simulate ‘call by reference’ to functions, but it takes a little work to do it.
- Incrementing or adding something to a pointer can be used to step along arrays.
- To facilitate array access by incrementing pointers, the Standard guarantees that in an *n* element array, although element *n* does not exist, use of its address is not an error—the valid range of addresses for an array declared as `int ar[N]` is `&ar[0]` through to `&ar[N]`. You must not try to access this last pseudo-element.

5.3.3. Qualified types

If you are confident that you have got a good grasp of the basic declaration and use of pointers we can continue. If not, it's important to go back over the previous material and make sure that there is nothing in it that you still find obscure; although what comes next looks more complicated than it really is, there's no need to make it worse by starting unprepared.

The Standard introduces two things called *type qualifiers*, neither of which were in Old C. They can be applied to any declared type to modify its behaviour—hence the term ‘qualifier’—and although one of them can be ignored for the moment (the one named `volatile`), the other, `const`, cannot.

If a declaration is prefixed with the keyword `const`, then the thing that is declared is announced to the world as being constant. You must not attempt to modify (change the value of) `const` objects, or you get undefined behaviour. Unless you have used some very dirty tricks, the compiler will know that the thing you are trying to modify is constant, so it can warn you.

There are two benefits in being able to declare things to be `const`.

1. It documents the fact that the thing is unmodifiable and the compiler helps to check. This is especially reassuring in the case of functions which take pointers as arguments. If the declaration of a function shows that the arguments are pointers to constant objects, then you know that the function is not allowed to change them through the pointers.
2. If the compiler knows that things are constant, it can often do increased amounts of optimization or generate better code.

Of course, constants are not much use unless you can assign an initial value to them. We won't go into the rules about initialization here (they are in Chapter 6 [*http://publications.gbdirect.co.uk/c_book/chapter6/*]), but for the moment just note that any declaration can also assign the value of a constant expression to the thing being declared. Here are some example declarations involving const:

```
const int x = 1;         /* x is constant */
const float f = 3.5;     /* f is constant */
const char y[10];        /* y is an array of 10 const ints */
                         /* don't think about initializing it yet! */
```

What is more interesting is that pointers can have this qualifier applied in two ways: either to the thing that it points to (pointer to const), or to the pointer itself (constant pointer). Here are examples of *that*:

```
int i;                   /* i is an ordinary int */
const int ci = 1;        /* ci is a constant int */
int *pi;                 /* pi is a pointer to an int */
const int *pci;          /* pc is a pointer to a constant int */
      /* and now the more complicated stuff */

/* cpi is a constant pointer to an int */
int *const cpi = &i;

/* cpci is a constant pointer to an constant int */
const int *const cpci = &ci;
```

The first declaration (of `i`) is unsurprising. Next, the declaration of `ci` shows that it is a constant integer, and therefore may not be modified. If we didn't initialize it, it would be pretty well useless.

It isn't hard to understand what a pointer to an integer and a pointer to a constant integer do—but note that they are different types of pointer now and can't be freely intermixed. You can change the values of both `pi` and `pci` (so that they point to other things); you can change the value of the thing that `pi` points to (it's not a constant integer), but you are only allowed to inspect the value of the thing that `pci` points to because that is a constant.

The last two declarations are the most complicated. If the pointers themselves are constant, then you are not allowed to make them point somewhere else—so they need to be initialized, just like `ci`. Independent of the `const` or other status of the pointer itself, naturally the thing that it points to can also be `const` or non-`const`, with the appropriate constraints on what you can do with it.

A final piece of clarification: what constitutes a qualified type? In the example, `ci` was clearly of a qualified type, but pci was not, since the pointer was not qualified, only the thing that it points to. The only things that had qualified type in that list were: `ci`, `cpi`, and `cpci`.

Although the declarations do take some mental gymnastics to understand, it just takes a little time to get used to seeing them, after which you will find that they seem quite natural. The complications come later when we have to explain whether or not you are allowed to (say) compare an ordinary pointer with a constant pointer, and if so, what does it mean? Most of those rules are ‘obvious’ but they do have to be stated.

Type qualifiers are given a further airing in Chapter 8 [*http://publications.gbdirect.co.uk/c_book/chapter8/*].

5.3.4. Pointer arithmetic

Although a more rigorous description of pointer arithmetic is given later, we'll start with an approximate version that will do for the moment.

Not only can you add an integral value to a pointer, but you can also compare or subtract two pointers of the same type. They must both point into the same array, or the result is undefined. The difference between two pointers is defined to be the number of array elements separating them; the type of this difference is implementation defined and will be one of `short`, `int`, or `long`. This next example shows how the difference can be calculated and used, but before you read it, you need to know an important point.

In an expression the name of an array is converted to a pointer to the first element of the array. The only places where that is not true are when an array name is used in conjunction with `sizeof`, when a string is used to initialize an array or when the array name is the subject of the address-of operator (unary `&`). We haven't seen any of those cases yet, they will be discussed later. Here's the example.

```
#include <stdio.h>
#include <stdlib.h>
#define ARSZ 10

main(){
      float fa[ARSZ], *fp1, *fp2;

      fp1 = fp2 = fa; /* address of first element */
      while(fp2 != &fa[ARSZ]){
              printf("Difference: %d\n", (int)(fp2-fp1));
              fp2++;
      }
      exit(EXIT_SUCCESS);
}
```

Example 5.4

The pointer `fp2` is stepped along the array, and the difference between its current and original values is printed. To make sure that `printf` isn't handed the wrong type of argument, the difference between the two pointers is forced to be of type `int` by using the cast `(int)`. That allows for machines where the difference between two pointers is specified to be long.

Unfortunately, if the difference does happen to be `long` and the array is enormous, the last example may give the wrong answers. This is a safe version, using a cast to force a long value to be passed:

```
#include <stdio.h>
#define ARSZ 10

main(){
      float fa[ARSZ], *fp1, *fp2;

      fp1 = fp2 = fa; /* address of first element */
      while(fp2 != &fa[ARSZ]){
              printf("Difference: %ld\n", (long)(fp2-fp1));
              fp2++;
      }
```

```
        return(0);
}
```

Example 5.5

5.3.5. void, null and dubious pointers

C is careful to keep track of the type of each pointer and will not in general allow you to use pointers of different types in the same expression. A pointer to char is a different type of pointer from a pointer to `int` (say) and you cannot assign one to the other, compare them, substitute one for the other as an argument to a function .... in fact they may even be stored differently in memory and even be of different lengths.

Pointers of different types are not the same. There are no implicit conversions from one to the other (unlike the arithmetic types).

There are a few occasions when you *do* want to be able to sidestep some of those restrictions, so what can you do?

The solution is to use the special type, introduced for this purpose, of 'pointer to `void`'. This is one of the Standard's invented features: before, it was tacitly assumed that 'pointer to `char`' was adequate for the task. This has been a reasonably successful assumption, but was a rather untidy thing to do; the new solution is both safer and less misleading. There isn't any other use for a pointer of that type—`void *` can't actually point to anything—so it improves readability. A pointer of type `void *` can have the value of any other pointer assigned to and can, conversely, be assigned to any other pointer. This must be used with great care, because you can end up in some heinous situations. We'll see it being used safely later with the malloc library function.

You may also on occasion want a pointer that is guaranteed not to point to any object—the so-called *null pointer*. It's common practice in C to write routines that return pointers. If, for some reason, they can't return a valid pointer (perhaps in case of an error), then they will indicate failure by returning a null pointer instead. An example could be a table lookup routine, which returns a pointer to the object searched for if it is in the table, or a null pointer if it is not.

How do you write a null pointer? There are two ways of doing it and both of them are equivalent: either an integral constant with the value of `0` or that value converted to type `void *` by using a cast. Both versions are called the *null pointer constant*. If you assign a null pointer constant to any other pointer, or compare it for equality with any other pointer, then it is first converted the type of that other pointer (neatly solving any problems about type compatibility) and will not appear to have a value that is equal to a pointer to any object in the program.

The only values that can be assigned to pointers apart from 0 are the values of other pointers of the same type. However, one of the things that makes C a useful replacement for assembly language is that it allows you to do the sort of things that most other languages prevent. Try this:

```
int *ip;
ip = (int *)6;
*ip = 0xFF;
```

What does that do? The pointer has been initialized to the value of 6 (notice the cast to turn an integer 6 into a pointer). This is a highly machine-specific operation, and the bit pattern that ends up in the pointer is quite possibly nothing like the machine representation of 6. After the initialization, hexadecimal `FF` is written into wherever the pointer is pointing. The int at location 6 has had `0xFF` written into it—subject to whatever 'location 6' means on this particular machine.

It may or may not make sense to do that sort of thing; C gives you the power to express it, it's up to you to get it right. As always, it's possible to do things like this by accident, too, and to be *very* surprised by the results.

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter5/arrays.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter5/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter5/character_handling.html*]

5.4. Character handling

<gbdirect>

C is widely used for character and string handling applications. This is odd, in some ways, because the language doesn't really have any built-in string handling features. If you're used to languages that know about string handling, you will almost certainly find C tedious to begin with.

The standard library contains lots of functions to help with string processing but the fact remains that it still feels like hard work. To compare two strings you have to call a function instead of using an equality operator. There is a bright side to this, though. It means that the language isn't burdened by having to support string processing directly, which helps to keep it small and less cluttered. What's more, once you get your string handling programs working in C, they do tend to run very quickly.

Character handling in C is done by declaring arrays (or allocating them dynamically) and moving characters in and out of them ‘by hand’. Here is an example of a program which reads text a line at a time from its standard input. If the line consists of the string of characters `stop`, it stops; otherwise it prints the length of the line. It uses a technique which is invariably used in C programs; it reads the characters into an array and indicates the end of them with an extra character whose value is explicitly 0 (zero). It uses the library `strcmp` function to compare two strings.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define LINELNG 100     /* max. length of input line */

main(){
      char in_line[LINELNG];
      char *cp;
      int c;

      cp = in_line;
      while((c = getc(stdin)) != EOF){
              if(cp == &in_line[LINELNG-1] || c == '\n'){
                      /*
                       * Insert end-of-line marker
                       */
                      *cp = 0;
                      if(strcmp(in_line, "stop") == 0 )
                              exit(EXIT_SUCCESS);
                      else
                              printf("line was %d characters long\n",
                                      (int)cp-in_line);
                      cp = in_line;
              }
              else
                      *cp++ = c;
      }
      exit(EXIT_SUCCESS);
}
```

Example 5.6

Once more, the example illustrates some interesting methods used widely in C programs. By far the most important is the way that strings are represented and manipulated.

Here is a possible implementation of `strcmp`, which compares two strings for equality and returns zero if they are the same. The library function actually does a bit more than that, but the added complication can be ignored for the moment. Notice the use of `const` in the argument declarations. This shows that the function will not modify the contents of the strings, but just inspects them. The definitions of the standard library functions make extensive use of this technique.

```
/*
* Compare two strings for equality.
* Return 'false' if they are.
*/
int
str_eq(const char *s1, const char *s2){
      while(*s1 == *s2){
              /*
               * At end of string return 0.
               */
              if(*s1 == 0)
                      return(0);
              s1++; s2++;
      }
      /* Difference detected! */
      return(1);
}
```

Example 5.7

5.4.1. Strings

Every C programmer ‘knows’ what a string is. It is an array of `char` variables, with the last character in the string followed by a null. ‘But I thought a string was something in double quote marks’, you cry. You are right, too. In C, a sequence like this

```
"a string"
```

is really a character array. It's the only example in C where you can declare something at the point of its use.

Be warned: in Old C, strings were stored just like any other character array, and were modifiable. Now, the Standard states that although they are are arrays of `char`, (not `const char`), attempting to modify them results in undefined behaviour.

Whenever a string in quotes is seen, it has two effects: it provides a declaration and a substitute for a name. It makes a hidden declaration of a char array, whose contents are initialized to the character values in the string, followed by a character whose integer value is zero. The array has no name. So, apart from the name being present, we have a situation like this:

```
char secret[9];
secret[0] = 'a';
secret[1] = ' ';
secret[2] = 's';
secret[3] = 't';
```

```
secret[4] = 'r';
secret[5] = 'i';
secret[6] = 'n';
secret[7] = 'g';
secret[8] = 0;
```

an array of characters, terminated by zero, with character values in it. But when it's declared using the string notation, it hasn't got a name. How can we use it?

Whenever C sees a quoted string, the presence of the string itself serves as the name of the hidden array—not only is the string an implicit sort of declaration, it is as if an array name had been given. Now, we all remember that the name of an array is equivalent to giving the address of its first element, so what is the type of this?

```
"a string"
```

It's a pointer of course: a pointer to the first element of the hidden unnamed array, which is of type `char`, so the pointer is of type ‘pointer to `char`’. The situation is shown in Figure 5.7.

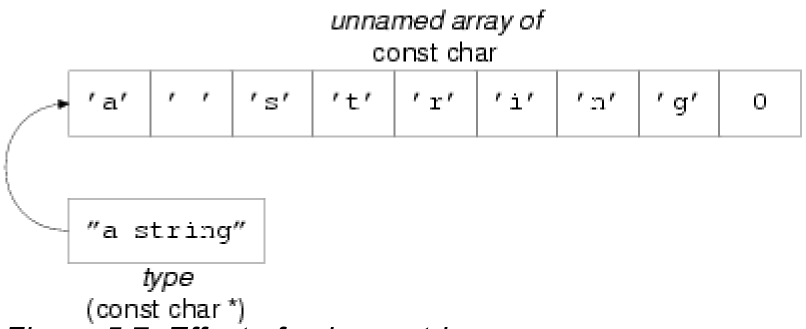


Figure 5.7. Effect of using a string

For proof of that, look at the following program:

```
#include <stdio.h>
#include <stdlib.h>
main(){
      int i;
      char *cp;

      cp = "a string";
      while(*cp != 0){
              putchar(*cp);
              cp++;
      }
      putchar('\n');

      for(i = 0; i < 8; i++)
              putchar("a string"[i]);
      putchar('\n');
      exit(EXIT_SUCCESS);
}
```

Example 5.8

The first loop sets a pointer to the start of the array, then walks along until it finds the zero at the end. The second one ‘knows’ about the length of the string and is less useful as a result. Notice how the first one is independent of the length—that is a most important point to remember. It's the way that strings are handled in C almost without exception; it's certainly the format that all of the library string manipulation functions expect. The zero at the end allows string processing routines to find out that they have reached the end of the string—look back now to the example function `str_eq`. The function takes two character pointers as arguments (so a string would be acceptable as one or both arguments). It compares them for equality by checking that the strings are character-for-character the same. If they are the same at any point, then it checks to make sure it hasn't reached the end of them both with `if(*s1 == 0)`: if it has, then it returns 0 to show that they were equal. The test could just as easily have been on `*s2`, it wouldn't have made any difference. Otherwise a difference has been detected, so it returns 1 to indicate failure.

In the example, `strcmp` is called with two arguments which look quite different. One is a character array, the other is a string. In fact they're the same thing—a character array terminated by zero (the program is careful to put a zero in the first ‘empty’ element of `in_line`), and a string in quotes—which is a character array terminated by a zero. Their use as arguments to strcmp results in character pointers being passed, for the reasons explained to the point of tedium above.

5.4.2. Pointers and increment operators

We said that we'd eventually revisit expressions like

```
(*p)++;
```

and now it's time. Pointers are used so often to walk down arrays that it just seems natural to use the `++` and `--` operators on them. Here we write zeros into an array:

```
#define ARLEN 10

int ar[ARLEN], *ip;

ip = ar;
while(ip < &ar[ARLEN])
        *(ip++) = 0;
```

Example 5.9

The pointer `ip` is set to the start of the array. While it remains inside the array, the place that it points to has zero written into it, then the increment takes effect and the pointer is stepped one element along the array. The postfix form of `++` is particularly useful here.

This is very common stuff indeed. In most programs you'll find pointers and increment operators used together like that, not just once or twice, but on almost every line (or so it seems while you find them difficult). What is happening, and what combinations can we get? Well, the `*` means indirection, and `++` or `--` mean increment; either pre- or post-increment. The combinations can be pre- or post-increment of either the pointer or the thing it points to, depending on where the brackets are put. Table 5.1 gives a list.

`++(*p)`	pre-increment thing pointed to
`(*p)++`	post-increment thing pointed to
`*(p++)`	access via pointer, post-increment pointer
`*(++p)`	access via pointer which has already been incremented

Table 5.1. Pointer notation

Read it carefully; make sure that you understand the combinations.

The expressions in the list above can usually be understood after a bit of head-scratching. Now, given that the precedence of `*`, `++` and `--` is the same in all three cases and that they associate right to left, can you work out what happens if the brackets are removed? Nasty, isn't it? Table 5.2 shows that there's only one case where the brackets have to be there.

With parentheses	Without, if possible
`++(*p)`	`++*p`

With parentheses	Without, if possible
`(*p)++`	`(*p)++`
`*(p++)`	`*p++`
`*(++p)`	`*++p`

Table 5.2. More pointer notation

The usual reaction to that horrible sight is to decide that you don't care that the parentheses can be removed; you will *always* use them in your code. That's all very well but the problem is that most C programmers have learnt the important precedence rules (or at least learnt the table above) and *they* very rarely put the parentheses in. Like them, we don't—so if you want to be able to read the rest of the examples, you had better learn to read those expressions with or without parentheses. It'll be worth the effort in the end.

5.4.3. Untyped pointers

In certain cases it's essential to be able to convert pointers from one type to another. This is always done with the aid of casts, in expressions like the one below:

```
(type *) expression
```

The *expression* is converted into 'pointer to *type*', regardless of the expression's previous type. This is only supposed to be done if you're sure that you know what you're trying to do. It is not a good idea to do much of it until you have got plenty of experience. Furthermore, do *not* assume that the cast simply suppresses diagnostics of the 'mismatched pointer' sort from your compiler. On several architectures it is necessary to calculate new values when pointer types are changed.

There are also some occasions when you will want to use a 'generic' pointer. The most common example is the `malloc` library function, which is used to allocate storage for objects that haven't been declared. It is used by telling it how much storage is wanted—enough for a `float`, or an array of `int`, or whatever. It passes back a pointer to enough storage, which it allocates in its own mysterious way from a pool of free storage (the way that it does this is its own business). That pointer is then cast into the right type—for example if a `float` needs 4 bytes of free store, this is the flavour of what you would write:

```
float *fp;

fp = (float *)malloc(4);
```

`Malloc` finds 4 bytes of store, then the address of that piece of storage is cast into pointer-to-float and assigned to the pointer.

What type should `malloc` be declared to have? The type must be able to represent every known value of every type of pointer; there is no guarantee that any of the basic types in C can hold such a value.

The solution is to use the `void *` type that we've already talked about. Here is the last example with a declaration of `malloc`:

```
void *malloc();
float *fp;

fp = (float *)malloc(4);
```

The rules for assignment of pointers show that there is no need to use a cast on the return value from `malloc`, but it is often done in practice.

Obviously there needs to be a way to find out what value the argument to `malloc` should be: it will be different on different machines, so you can't just use a constant like 4. That is what the `sizeof` operator is for.

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter5/pointers.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter5/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter5/sizeof_and_malloc.html*]

5.5. Sizeof and storage allocation

The `sizeof` operator returns the size in bytes of its operand. Whether the result of `sizeof` is `unsigned int` or `unsigned long` is implementation defined—which is why the declaration of malloc above ducked the issue by omitting any parameter information; normally you would use the `stdlib.h` header file to declare `malloc` correctly. Here is the last example done portably:

```
#include <stdlib.h>     /* declares malloc() */
float *fp;

fp = (float *)malloc(sizeof(float));
```

The operand of `sizeof` only has to be parenthesized if it's a type name, as it was in the example. If you are using the name of a data object instead, then the parentheses can be omitted, but they rarely are.

```
#include <stdlib.h>

int *ip, ar[100];
ip = (int *)malloc(sizeof ar);
```

In the last example, the array `ar` is an array of `100 ints`; after the call to `malloc` (assuming that it was successful), `ip` will point to a region of store that can also be treated as an array of `100 ints`.

The fundamental unit of storage in C is the `char`, and by definition

```
sizeof(char)
```

is equal to 1, so you could allocate space for an array of ten `char`s with

```
malloc(10)
```

while to allocate room for an array of ten `int`s, you would have to use

```
malloc(sizeof(int[10]))
```

If `malloc` can't find enough free space to satisfy a request it returns a null pointer to indicate failure. For historical reasons, the `stdio.h` header file contains a defined constant called NULL which is traditionally used to check the return value from `malloc` and some other library functions. An explicit `0` or `(void *)0` could equally well be used.

As a first illustration of the use of `malloc`, here's a program which reads up to `MAXSTRING` strings from its input and sort them into alphabetical order using the library `strcmp` routine. The strings are terminated by a '`\n`' character. The sort is done by keeping an array of pointers to the strings and simply exchanging the pointers until the order is correct. This saves having to copy the strings themselves, which improves the efficency somewhat.

The example is done first using fixed size arrays, then another version uses malloc and allocates space for the strings at run time. Unfortunately, the array of pointers is still fixed in size: a better solution would use a linked list or similar data structure to store the pointers and would have no fixed arrays at all. At the moment, we haven't seen how to do that.

The overall structure is this:

```
while(number of strings read < MAXSTRING
      && input still remains){

              read next string;
}
sort array of pointers;
print array of pointers;
exit;
```

A number of functions are used to implement this program:

`char *next_string(char *destination)`

> Read a line of characters terminated by '`\n`' from the program's input. The first `MAXLEN-1` characters are written into the array pointed to by destination.
>
> If the first character read is `EOF`, return a null pointer, otherwise return the address of the start of the string (destination). On return, destination always points to a null-terminated string.

`void sort_arr(const char *p_array[])`

> `P_array[]` is an array of pointers to characters. The array can be arbitrarily long; its end is indicated by the first element containing a null pointer.
>
> `Sort_arr` sorts the pointers so that the pointers point to strings which are in alphabetical order when the array is traversed in index order.

`void print_arr(const char *p_array[])`

> Like `sort_arr`, but prints the strings in index order.

It will help to understand the examples if you remember that in an expression, an array's name is converted to the address of its first element. Similarly, for a two-dimensional array (such as `strings` below), then the expression `strings[1][2]` has type char, but `strings[1]` has type 'array of `char`' which is therefore converted to the address of the first element: it is equivalent to `&strings[1][0]`.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAXSTRING       50      /* max no. of strings */
#define MAXLEN          80      /* max length. of strings */

void print_arr(const char *p_array[]);
void sort_arr(const char *p_array[]);
char *next_string(char *destination);

main(){
      /* leave room for null at end */
      char *p_array[MAXSTRING+1];

      /* storage for strings */
      char strings[MAXSTRING][MAXLEN];
```

```
        /* count of strings read */
        int nstrings;

        nstrings = 0;
        while(nstrings < MAXSTRING &&
                next_string(strings[nstrings]) != 0){

                p_array[nstrings] = strings[nstrings];
                nstrings++;
        }
        /* terminate p_array */
        p_array[nstrings] = 0;

        sort_arr(p_array);
        print_arr(p_array);
        exit(EXIT_SUCCESS);
}

void print_arr(const char *p_array[]){
        int index;
        for(index = 0; p_array[index] != 0; index++)
                printf("%s\n", p_array[index]);
}


void sort_arr(const char *p_array[]){
        int comp_val, low_index, hi_index;
        const char *tmp;

        for(low_index = 0;
                p_array[low_index] != 0 &&
                        p_array[low_index+1] != 0;
                        low_index++){

                for(hi_index = low_index+1;
                        p_array[hi_index] != 0;
                                hi_index++){

                        comp_val=strcmp(p_array[hi_index],
                                p_array[low_index]);
                        if(comp_val >= 0)
                                continue;
                        /* swap strings */
                        tmp = p_array[hi_index];
                        p_array[hi_index] = p_array[low_index];
                        p_array[low_index] = tmp;
                }
        }
}



char *next_string(char *destination){
        char *cp;
        int c;

        cp = destination;
        while((c = getchar()) != '\n' && c != EOF){
                if(cp-destination < MAXLEN-1)
```

```
                        *cp++ = c;
        }
        *cp = 0;
        if(c == EOF && cp == destination)
                return(0);
        return(destination);
}
```

Example 5.10

It is no accident that `next_string` returns a pointer. We can now dispense with the strings array by getting `next_string` to allocate its own storage.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAXSTRING       50      /* max no. of strings */
#define MAXLEN          80      /* max length. of strings */

void print_arr(const char *p_array[]);
void sort_arr(const char *p_array[]);
char *next_string(void);

main(){
      char *p_array[MAXSTRING+1];
      int nstrings;

      nstrings = 0;
      while(nstrings < MAXSTRING &&
              (p_array[nstrings] = next_string()) != 0){

              nstrings++;
      }
      /* terminate p_array */
      p_array[nstrings] = 0;

      sort_arr(p_array);
      print_arr(p_array);
      exit(EXIT_SUCCESS);
}

void print_arr(const char *p_array[]){
      int index;
      for(index = 0; p_array[index] != 0; index++)
              printf("%s\n", p_array[index]);
}


void sort_arr(const char *p_array[]){
      int comp_val, low_index, hi_index;
      const char *tmp;

      for(low_index = 0;
              p_array[low_index] != 0 &&
                      p_array[low_index+1] != 0;
                      low_index++){

              for(hi_index = low_index+1;
                      p_array[hi_index] != 0;
```

```
                                hi_index++){

                        comp_val=strcmp(p_array[hi_index],
                                p_array[low_index]);
                        if(comp_val >= 0)
                                continue;
                        /* swap strings */
                        tmp = p_array[hi_index];
                        p_array[hi_index] = p_array[low_index];
                        p_array[low_index] = tmp;
                }
        }
}

char *next_string(void){
        char *cp, *destination;
        int c;

        destination = (char *)malloc(MAXLEN);
        if(destination != 0){
                cp = destination;
                while((c = getchar()) != '\n' && c != EOF){
                        if(cp-destination < MAXLEN-1)
                                *cp++ = c;
                }
                *cp = 0;
                if(c == EOF && cp == destination)
                        return(0);
        }
        return(destination);
}
```

Example 5.11

Finally, for the extremely brave, here is the whole thing with even `p_array` allocated using `malloc`. Further, most of the array indexing is rewritten to use pointer notation. If you are feeling queasy, skip this example. It is hard. One word of explanation: `char **p` means a pointer to a pointer to a character. Many C programmers find this hard to deal with.

```
#include <stdio.h>
#include <stdlib.hi>
#include <string.h>

#define MAXSTRING       50      /* max no. of strings */
#define MAXLEN          80      /* max length. of strings */

void print_arr(const char **p_array);
void sort_arr(const char **p_array);
char *next_string(void);

main(){
        char **p_array;
        int nstrings;   /* count of strings read */

        p_array = (char **)malloc(
                        sizeof(char *[MAXSTRING+1]));
        if(p_array == 0){
                printf("No memory\n");
                exit(EXIT_FAILURE);
        }

        nstrings = 0;
        while(nstrings < MAXSTRING &&
                (p_array[nstrings] = next_string()) != 0){

                nstrings++;
        }
        /* terminate p_array */
        p_array[nstrings] = 0;

        sort_arr(p_array);
        print_arr(p_array);
        exit(EXIT_SUCCESS);
}

void print_arr(const char **p_array){
        while(*p_array)
                printf("%s\n", *p_array++);
}


void sort_arr(const char **p_array){
        const char **lo_p, **hi_p, *tmp;

        for(lo_p = p_array;
                *lo_p != 0 && *(lo_p+1) != 0;
                                        lo_p++){
                for(hi_p = lo_p+1; *hi_p != 0; hi_p++){

                        if(strcmp(*hi_p, *lo_p) >= 0)
                                continue;
                        /* swap strings */
                        tmp = *hi_p;
                        *hi_p = *lo_p;
                        *lo_p = tmp;
                }
        }
}



char *next_string(void){
        char *cp, *destination;
        int c;

        destination = (char *)malloc(MAXLEN);
        if(destination != 0){
                cp = destination;
                while((c = getchar()) != '\n' && c != EOF){
                        if(cp-destination < MAXLEN-1)
                                *cp++ = c;
                }
                *cp = 0;
                if(c == EOF && cp == destination)
                        return(0);
        }
        return(destination);
}
```

Example 5.12

To further illustrate the use of `malloc`, another example program follows which can cope with arbitrarily long strings. It simply reads strings from its standard input, looking for a newline character to mark the end of the string, then prints the string on its standard output. It stops when it detects end-of-file. The characters are put into an array, the end of the string being indicated (as always) by a zero. The newline is not stored, but used to detect when a full line of input should be printed on the output. The program doesn't know how long the string will be, so it starts by allocating ten characters—enough for a short string.

If the string is more than ten characters long, `malloc` is called to allocate room for the current string plus ten more characters. The current characters are copied into the new space, the old storage previously allocated is released and the program continues using the new storage.

To release storage allocated by `malloc`, the library function `free` is used. If you don't release storage when it isn't needed any more, it just hangs around taking up space. Using `free` allows it to be ‘given away’, or at least re-used later.

The program reports errors by using `fprintf`, a close cousin of `printf`. The only difference between them is that fprintf takes an additional first argument which indicates where its output should go. There are two constants of the right type for this purpose defined in `stdio.h`. Using `stdout` indicates that the program's standard output is to be used; `stderr` refers to the program's standard error stream. On some systems both may be the same, but other systems do make the distinction.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define GROW_BY 10      /* string grows by 10 chars */

main(){
      char *str_p, *next_p, *tmp_p;
      int ch, need, chars_read;

      if(GROW_BY < 2){
              fprintf(stderr,
                      "Growth constant too small\n");
              exit(EXIT_FAILURE);
      }

      str_p = (char *)malloc(GROW_BY);
      if(str_p == NULL){
              fprintf(stderr,"No initial store\n");
              exit(EXIT_FAILURE);
      }

      next_p = str_p;
      chars_read = 0;
      while((ch = getchar()) != EOF){
              /*
               * Completely restart at each new line.
               * There will always be room for the
               * terminating zero in the string,
               * because of the check further down,
               * unless GROW_BY is less than 2,
               * and that has already been checked.
               */
```

```
                if(ch == '\n'){
                        /* indicate end of line */
                        *next_p = 0;
                        printf("%s\n", str_p);
                        free(str_p);
                        chars_read = 0;
                        str_p = (char *)malloc(GROW_BY);
                        if(str_p == NULL){
                                fprintf(stderr,"No initial store\n");
                                exit(EXIT_FAILURE);
                        }
                        next_p = str_p;
                        continue;
                }
                /*
                 * Have we reached the end of the current
                 * allocation ?
                 */
                if(chars_read == GROW_BY-1){
                        *next_p = 0;    /* mark end of string */
                        /*
                         * use pointer subtraction
                         * to find length of
                         * current string.
                         */
                        need = next_p - str_p +1;
                        tmp_p = (char *)malloc(need+GROW_BY);
                        if(tmp_p == NULL){
                                fprintf(stderr,"No more store\n");
                                exit(EXIT_FAILURE);
                        }
                        /*
                         * Copy the string using library.
                         */
                        strcpy(tmp_p, str_p);
                        free(str_p);
                        str_p = tmp_p;
                        /*
                         * and reset next_p, character count
                         */
                        next_p = str_p + need-1;
                        chars_read = 0;
                }
                /*
                 * Put character at end of current string.
                 */
                *next_p++ = ch;
                chars_read++;
        }
        /*
         * EOF - but do unprinted characters exist?
         */
        if(str_p - next_p){
                *next_p = 0;
                fprintf(stderr,"Incomplete last line\n");
                printf("%s\n", str_p);
        }
        exit(EXIT_SUCCESS);
```

```
}
```

Example 5.13

That may not be a particularly realistic example of how to handle arbitrarily long strings—for one thing, the maximum storage demand is *twice* the amount needed for the longest string—but it does actually work. It also costs rather a lot in terms of copying around. Both problems could be reduced by using the library `realloc` function instead.

A more sophisticated method might use a linked list, implemented with the use of *structures*, as described in the next chapter. That would have its drawbacks too though, because then the standard library routines wouldn't work for a different method of storing strings.

5.5.1. What sizeof can't do

One common mistake made by beginners is shown below:

```
#include <stdio.h>
#include <stdlib.h>

const char arr[] = "hello";
const char *cp = arr;
main(){

      printf("Size of arr %lu\n", (unsigned long)
                      sizeof(arr));
      printf("Size of *cp %lu\n", (unsigned long)
                      sizeof(*cp));
      exit(EXIT_SUCCESS);
}
```

Example 5.14

The numbers printed will *not* be the same. The first will, correctly, identify the size of `arr` as `6`; five characters followed by a null. The second one will always, on every system, print `1`. That's because the type of `*cp` is `const char`, which can only have a size of `1`, whereas the type of `arr` is different: array of `const char`. The confusion arises because this is the one place that the use of an array is not converted into a pointer first. It is never possible, using `sizeof`, to find out how long an array a pointer points to; you *must* have a genuine array name instead.

5.5.2. The type of sizeof

Now comes the question of just what this does:

```
sizeof ( sizeof (anything legal) )
```

That is to say, what type does the result of `sizeof` have? The answer is that it is implementation defined, and will be either `unsigned long` or `unsigned int`, depending on your implementation. There are two safe things to do: either always cast the return value to unsigned long, as the examples have done, or to use the defined type `size_t` provided in the `<stddef.h>` header file. For example:

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>

main(){
      size_t sz;
      sz = sizeof(sz);
      printf("size of sizeof is %lu\n",
              (unsigned long)sz);
      exit(EXIT_SUCCESS);
}
```

Example 5.15

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter5/character_handling.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter5/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter5/function_pointers.html*]

5.6. Pointers to functions

A useful technique is the ability to have pointers to functions. Their declaration is easy: write the declaration as it would be for the function, say

```
int func(int a, float b);
```

and simply put brackets around the name and a `*` in front of it: that declares the pointer. Because of precedence, if you don't parenthesize the name, you declare a function returning a pointer:

```
/* function returning pointer to int */
int *func(int a, float b);

/* pointer to function returning int */
int (*func)(int a, float b);
```

Once you've got the pointer, you can assign the address of the right sort of function just by using its name: like an array, a function name is turned into an address when it's used in an expression. You can call the function using one of two forms:

```
(*func)(1,2);
/* or */
func(1,2);
```

The second form has been newly blessed by the Standard. Here's a simple example.

```
#include <stdio.h>
#include <stdlib.h>

void func(int);

main(){
      void (*fp)(int);

      fp = func;

      (*fp)(1);
      fp(2);

      exit(EXIT_SUCCESS);
}

void
func(int arg){
      printf("%d\n", arg);
}
```

Example 5.16

If you like writing finite state machines, you might like to know that you can have an array of pointers to functions, with declaration and use like this:

```
void (*fparr[])(int, float) = {
                              /* initializers */
                      };
/* then call one */

fparr[5](1, 3.4);
```

Example 5.17

But we'll draw a veil over it at this point!

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter5/sizeof_and_malloc.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter5/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter5/pointer_expressions.html*]

5.7. Expressions involving pointers

<gbdirect>

Because of the introduction of qualified types and of the notion of incomplete types, together with the use of `void *`, there are now some complicated rules about how you can mix pointers and what arithmetic with pointers really permits you to do. Most people will survive quite well without ever learning this explicitly, because a lot of it is ‘obvious’, but we will include it here in case you do want to know. For the final word in accuracy, obviously you will want to see what the Standard says. What follows is our interpretation in (hopefully) plainer English.

You don't yet know the Standard means when it talks about *objects* or *incomplete types*. So far we have tended to use the term loosely, but properly speaking an object is a piece of data storage whose contents is to be interpreted as a value. A function is not an object. An incomplete type is one whose name and type are mostly known, but whose size hasn't yet been determined. You can get these in two ways:

1. By declaring an array but omitting information about its size: `int x[];`. In that case, there must be additional information given later in a definition for the array. The type remains incomplete until the later definition.
2. By declaring a *structure* or *union* but not defining its contents. The contents must be defined in a later declaration. The type remains incomplete until the later declaration.

There will be some more discussion of incomplete types in later chapters.

Now for what you are allowed to do with pointers. Note that wherever we talk about qualified types they can be qualified with `const`, `volatile`, or both; the examples are illustrated with const only.

5.7.1. Conversions

Pointers to `void` can be freely converted backwards and forwards with pointers to any object or incomplete type. Converting a pointer to an object or an incomplete type to `void *` and then back gives a value which is equal to the original one:

```
int i;
int *ip;
void *vp;

ip = &i;
vp = ip;
ip = vp;
if(ip != &i)
      printf("Compiler error\n");
```

An unqualified pointer type may be converted to a qualified pointer type, but the reverse is not true. The two values will be equal:

```
int i;
int *ip, *const cpi;
```

```
ip = &i;
cpi = ip;       /* permitted */
if(cpi != ip)
      printf("Compiler error\n");
ip = cpi;       /* not permitted */
```

A null pointer constant (see earlier) will not be equal to a pointer to any object or function.

5.7.2. Arithmetic

Expressions can add (or subtract, which is equivalent to adding negative values) integral values to the value of a pointer to any object type. The result has the type of the pointer and if *n* is added, then the result points *n* array elements away from the pointer. The most common use is repeatedly to add `1` to a pointer to step it from the start to the end of an array, but addition or subtraction of values other than one is possible.

It the pointer resulting from the addition points in front of the array or past the non-existent element just after the last element of the array, then you have had overflow or underflow and the result is undefined.

The last-plus-one element of an array has always been assumed to be a valid address for a pointer and the Standard confirms this. You mustn't actually access that element, but the address is guaranteed to exist rather than being an overflow condition.

We've been careful to use the term ‘expression’ rather than saying that you actually add something to the pointer itself. You can do that, but only if the pointer is not qualified with `const` (of course). The increment and decrement operators are equivalent to adding or subtracting 1.

Two pointers to *compatible types* whether or not qualified may be subtracted. The result has the type `ptrdiff_t`, which is defined in the header file `<stddef.h>`. Both pointers must point into the same array, or one past the end of the array, otherwise the behaviour is undefined. The value of the result is the number of array elements that separate the two pointers. E.g.:

```
int x[100];
int *pi, *cpi = &x[99]; /* cpi points to the last element of x */

pi = x;
if((cpi - pi) != 99)
      printf("Error\n");

pi = cpi;
pi++;                   /* increment past end of x */
if((pi - cpi) != 1)
      printf("Error\n");
```

5.7.3. Relational expressions

These allow us to compare pointers with each other. You can only compare

- Pointers to compatible object types with each other
- Pointers to compatible incomplete types with each other

It does not matter if the types that are pointed to are qualified or unqualified.

If two pointers compare equal to each other then they point to the same thing, whether it is an object or the non-existent element off the end of an array (see arithmetic, above). If two pointers point to the same thing, then they compare equal to each other. The relational operators >, <= and so on all give the result that you would expect if the pointers point into the same array: if one pointer compares less than another, then it points nearer to the front of the array.

A null pointer constant can be assigned to a pointer; that pointer will then compare equal to the null pointer constant (which is pretty obvious). A null pointer constant or a null pointer will not compare equal to a pointer that points to anything which actually exists.

5.7.4. Assignment

You can use pointers with the assignment operators if the following conditions are met:

- The left-hand operand is a pointer and the right-hand operand is a null pointer constant.
- One operand is a pointer to an object or incomplete type; the other is a pointer to `void` (whether qualified or not).
- Both of the operands are pointers to compatible types (whether qualified or not).

In the last two cases, the type pointed to by the left-hand side must have at least the same qualifiers as the type pointed to by the right-hand side (possibly more).

So, you can assign a pointer to int to a pointer to `const int` (more qualifiers on the left than the right) but you cannot assign a pointer to `const int` to a pointer to `int`. If you think about it, it makes sense.

The `+=` and `-=` operators can involve pointers as long as the left-hand side is a pointer to an object and the right-hand side is an integral expression. The arithmetic rules above describe what happens.

5.7.5. Conditional operator

The description of the behaviour of this operator when it is used with pointers has already been given in Chapter 3 [*http://publications.gbdirect.co.uk/c_book/chapter3/*].

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter5/function_pointers.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter5/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter5/arrays_and_address_of.html*]

5.8. Arrays, the & operator and function

<gbdirect>

We have already emphasized that in most cases, the name of an array is converted into the address of its first element; one notable exception being when it is the operand of `sizeof`, which is essential if the stuff to do with `malloc` is to work. Another case is when an array name is the operand of the `&` address-of operator. Here, it is converted into the *address of the whole array*. What's the difference? Even if you think that addresses would be in some way 'the same', the critical difference is that they have different types. For an array of n elements of type T, then the address of the first element has type 'pointer to T'; the address of the whole array has type 'pointer to array of *n* elements of type T'; clearly very different. Here's an example of it:

```
int ar[10];
int *ip;
int (*ar10i)[10];       /* pointer to array of 10 ints */

ip = ar;                /* address of first element */
ip = &ar[0];            /* address of first element */
ar10i = &ar;            /* address of whole array */
```

Where do pointers to arrays matter? Not often, in truth, although of course we know that declarations that look like multidimensional arrays are really arrays of arrays. Here is an example which uses that fact, but you'll have to work out what it does for yourself. It is *not* common to do this sort of thing in practice:

```
int ar2d[5][4];
int (*ar4i)[4]; /* pointer to array of 4 ints */

for(ar4i= ar2d; ar4i < &(ar2d[5]); ar4i++)
        (*ar4i)[2] = 0; /* ar2d[n][2] = 0 */
```

More important than addresses of arrays is what happens when you declare a function that takes an array as an argument. Because of the 'conversion to the address of its first element' rule, even if you do try to pass an array to a function by giving its name as an argument, you actually end up passing a pointer to its first element. The usual rule really does apply in this case! But what if you declare that the function *does* have an argument whose type is 'array of something'—like this:

```
void f(int ar[10]);
```

What happens? The answer may suprise you slightly. The compiler looks at that and says to itsel 'Ho ho. That's going to be a pointer when the function is called' and then rewrites the parameter type to be a pointer. As a result, all three of these declarations are identical:

```
void f(int ar[10]);
void f(int *ar);
void f(int ar[]);       /* since the size of the array is irrelevant! *
```

Having seen that, your reaction might be to look for a solid object to bang your head against for a while, but we don't recommend it. Take a grip on yourself instead and put in the effort to work out

- Why that is isn't really such a shock
- Why, given a function declaration like that, then within the function, expressions of the form `ar[5]` and so on work as expected anyhow

Give that last one some thought. When you get to the bottom of it, you really will have grasped what arrays and pointers are about.

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter5/pointer_expressions.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter5/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter5/summary.html*]

5.9. Summary

You have been introduced to arrays, pointers and the storage allocater. The last of the topics will prove to be more useful in the next chapter, but the other two are are central to the language.

You *cannot* use C properly without understanding the use of pointers. Arrays are simple and unsurprising, except for the fact that when it's used in an expression, an array name usually converts into a pointer to its first element; that often takes time to sink in.

The C approach to support for strings often causes raised eyebrows. The null-terminated array of character model is both powerful and flexible. The fact that string manipulation is not built in to the language at first glance seems to rule C out of serious contention for character-oriented work, yet that is exactly where the language scores well compared with the alternatives, at least when speed is important. All the same, it's hard work for the programmer.

Pointer arithmetic is easy and extremely convenient. It's harder for ex-assembler programmers to learn, because of the tendency to try to translate it into what they ‘know’ the machine is doing. However, much harder for people with very low-level experience is the idea of the non-equivalence of pointers of different types. Try hard to throw away the idea that pointers contain addresses (in the hardware sense) and it will repay the effort.

The facility to obtain arbitrary pieces of storage using `malloc` and the associated stuff is extremely important. You might wish to defer it for a while, but don't leave it for too long. An obvious feature of C programs written by inexperienced users is their dependence on fixed size arrays. `Malloc` gives you considerably more flexibility and is worth the effort to learn about.

The examples of the use of `sizeof` should help to eliminate a few common misconceptions about what it does. You may not use it all that often, but when you do need it, there's no substitute.

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter5/arrays_and_address_of.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter5/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter5/exercises.html*]

5.10. Exercises

Exercise 5.1. What is the valid range of indices for an array of ten objects?

Exercise 5.2. What happens if you take the address of the 11th member of that array?

Exercise 5.3. When is it valid to compare the values of two pointers?

Exercise 5.4. What is the use of a pointer to `void`?

Exercise 5.5. Write functions which:

a. Compare two strings for equality. If they are equal, zero is returned, otherwise the difference in value between the first two non-matching characters.
b. Find the first occurrence of a specific character in a given string. Return a pointer to the occurrence in the string, or zero if it is not found.
c. Take two strings as arguments. If the first exists in the second as a substring, return a pointer to the first occurrence, otherwise zero.

Exercise 5.6. Explain the examples using malloc to somebody else.

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter5/summary.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter5/*]

Chapter 6

<gbdirect>

Structured Data Types

Previous chapter [*http://publications.gbdirect.co.uk/c_book/chapter5/*] | Next chapter [*http://publications.gbdirect.co.uk/c_book/chapter7/*]

6.1. History

<gbdirect>

The development of the early computer languages went either one way or the other. COBOL concentrated on the structure of data but not on arithmetic or algorithms, FORTRAN and Algol leant the other way. Scientific users wanted to do numeric work on relatively unstructured data (although arrays were soon found to be indispensable) and commercial users needed only basic arithmetic but knew that the key issue was the structure of the data.

The ideas that have influenced C are a mixture of the two schools; it has the structured control of flow expected in a language of its age, and has also made a start on data structures. So far we have concentrated on the algorithmic aspects of the language and haven't thought hard about data storage. Whilst it's true that arrays fall into the general category of data structuring, they are so simple, and so commonly in use, that they don't deserve a chapter to themselves. Until now we have been looking at a kind of block-structured FORTRAN.

The trend in the late 1980s and early '90s seems to be towards integrating both the data and the algorithms; it's then called Object-Oriented programming. There is no specific support for that in C. C++ is a language based on C that does offer support for Object-Oriented techniques, but it is out of our scope to discuss it further.

For a large class of problems in computing, it is the data and not the algorithms that are the most interesting. If the initial design gets its data structures right, the rest of the effort in putting a program together is often quite small. However, you need help from the language. If there is no support for structured data types other than arrays, writing programs becomes both less convenient and also more prone to errors. It is the job of a good language to do more than just allow you to do something; it must actively *help* as well.

C offers arrays, structures and unions as its contribution to data structuring. They have proved to be entirely adequate for most users' needs over the years and remain essentially unchanged by the Standard.

Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter6/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter6/structures.html*]

6.2. Structures

<gbdirect>

Arrays allow for a named collection of identical objects. This is suitable for a number of tasks, but isn't really very flexible. Most real data objects are complicated things with an inherent structure that does not fit well on to array style storage. Let's use a concrete example.

Imagine that the job is something to do with a typesetting package. In this system, the individual characters have not only their character values but also some additional attributes like font and point size. The font doesn't affect the character as such, but only the way that it is displayed: this is the normal font, *this is in italics* and this is ***in bold font***. Point size is similar. It describes the size of the characters when they are printed. For example, the point size of this text increases now. It goes back again now. If our characters have three independent attributes, how can they be represented in a single object?

With C it's easy. First work out how to represent the individual attributes in the basic types. Let's assume that we can still store the character itself in a char, that the font can be encoded into a `short` (1 for regular, 2 italic, 3 bold etc.) and that the point size will also fit a short. These are all quite reasonable assumptions. Most systems only support a few tens of fonts even if they are very sophisticated, and point sizes are normally in the range 6 to the small hundreds. Below 6 is almost invisible, above 50 is bigger than the biggest newspaper banner headlines. So we have a char and two shorts that are to be treated as a single entity. Here's how to declare it in C.

```
struct wp_char{
      char wp_cval;
      short wp_font;
      short wp_psize;
};
```

That effectively declares a new type of object which can be used in your program. The whole thing is introduced by the `struct` keyword, which is followed by an optional identifier known as the `tag`, `wp_char` in this case. The tag only serves the purpose of giving a name to this type of structure and allows us to refer to the type later on. After a declaration like the one just seen, the tag can be used like this:

```
struct wp_char x, y;
```

That defines two variables called `x` and `y` just as it would have done if the definition had been

```
int x, y;
```

but of course in the first example the variables are of type `struct wp_char`, and in the second their type is `int`. The tag is a name for the new type that we have introduced.

It's worth remembering that structure tags can safely be used as ordinary identifiers as well. They only mean something special when they are preceded by the keyword `struct`. It is quite common to see a structured object being defined with the same name as its structure tag.

```
struct wp_char wp_char;
```

That defines a variable called `wp_char` of type `struct wp_char`. This is described by saying that structure tags have their own ‘name space’ and cannot collide with other names. We'll investigate tags some more in the discussion of ‘incomplete types’.

Variables can also be defined immediately following a structure declaration.

```
struct wp_char{
      char wp_cval;
      short wp_font;
      short wp_psize;
}v1;

struct wp_char v2;
```

We now have two variables, `v1` and `v2`. If all the necessary objects are defined at the end of the structure declaration, the way that `v1` was, then the tag becomes unneccessary (except if it is needed later for use with `sizeof` and in casts) and is often not present.

The two variables are structured objects, each containing three separate *members* called `wp_cval`, `wp_font` and `wp_psize`. To access the individual members of the structures, the 'dot operator is used:

```
v1.wp_cval = 'x';
v1.wp_font = 1;
v1.wp_psize = 10;

v2 = v1;
```

The individual members of `v1` are initialized to suitable values, then the whole of `v1` is copied into `v2` in an assignment.

In fact the only operation permitted on whole structures is assignment: they can be assigned to each other, passed as arguments to functions and returned by functions. However, it is not a very efficient operation to copy structures and most programs avoid structure copying by manipulating pointers to structures instead. It is generally quicker to copy pointers around than structures. A surprising omission from the language is the facility to compare structures for equality, but there is a good reason for this which will be mentioned shortly.

Here is an example using an array of structures like the one before. A function is used to read characters from the program's standard input and return an appropriately initialized structure. When a newline has been read or the array is full, the structures are sorted into order depending on the character value, and then printed out.

```
#include <stdio.h>
#include <stdlib.h>

#define ARSIZE 10

struct wp_char{
      char wp_cval;
      short wp_font;
      short wp_psize;
}ar[ARSIZE];

/*
* type of the input function -
* could equally have been declared above;
* it returns a structure and takes no arguments.
*/
struct wp_char infun(void);

main(){
      int icount, lo_indx, hi_indx;
```

```
        for(icount = 0; icount < ARSIZE; icount++){
                ar[icount] = infun();
                if(ar[icount].wp_cval == '\n'){
                        /*
                         * Leave the loop.
                         * not incrementing icount means that the
                         * '\n' is ignored in the sort
                         */
                        break;
                }
        }

        /* now a simple exchange sort */

        for(lo_indx = 0; lo_indx <= icount-2; lo_indx++)
                for(hi_indx = lo_indx+1; hi_indx <= icount-1; hi_indx++){
                        if(ar[lo_indx].wp_cval > ar[hi_indx].wp_cval){
                                /*
                                 * Swap the two structures.
                                 */
                                struct wp_char wp_tmp = ar[lo_indx];
                                ar[lo_indx] = ar[hi_indx];
                                ar[hi_indx] = wp_tmp;
                        }
                }

        /* now print */
        for(lo_indx = 0; lo_indx < icount; lo_indx++){
                printf("%c %d %d\n", ar[lo_indx].wp_cval,
                                ar[lo_indx].wp_font,
                                ar[lo_indx].wp_psize);
        }
        exit(EXIT_SUCCESS);
}

struct wp_char
infun(void){
        struct wp_char wp_char;

        wp_char.wp_cval = getchar();
        wp_char.wp_font = 2;
        wp_char.wp_psize = 10;

        return(wp_char);
}
```

Example 6.1

Once it is possible to declare structures it seems pretty natural to declare arrays of them, use them as members of other structures and so on. In fact the only restriction is that a structure cannot contain an example of itself as a member—in which case its size would be an interesting concept for philosophers to debate, but hardly useful to a C programmer.

6.2.1. Pointers and structures

If what the last paragraph says is true—that it is more common to use pointers to structures than to use the structures directly—we need to know how to do it. Declaring pointers is easy of course:

```
struct wp_char *wp_p;
```

gives us one straight away. But how do we access the members of the structure? One way might be to look through the pointer to get the whole structure, then select the member:

```
/* get the structure, then select a member */
(*wp_p).wp_cval
```

that would certainly work (the parentheses are there because . has a higher precedence than `*`). It's not an easy notation to work with though, so C introduces a new operator to clean things up; it is usually known as the ‘pointing-to’ operator. Here it is being used:

```
/* the wp_cval in the structure wp_p points to */
wp_p->wp_cval = 'x';
```

and although it might not look a lot easier than its alternative, it pays off when the structure contains pointers, as in a linked list. The pointing-to syntax is much easier if you want to follow two or three stages down the links of a linked list. If you haven't come across linked lists before, you're going to learn a lot more than just the use of structures before this chapter finishes!

If the thing on the left of the `.` or `->` operator is qualified (with `const` or `volatile`) then the result is also has those qualifiers associated with it. Here it is, illustrated with pointers; when the pointer points to a qualified type the result that you get is also qualified:

```
#include <stdio.h>
#include <stdlib.h>

struct somestruct{
      int i;
};

main(){
      struct somestruct *ssp, s_item;
      const struct somestruct *cssp;

      s_item.i = 1;   /* fine */
      ssp = &s_item;
      ssp->i += 2;    /* fine */
      cssp = &s_item;
      cssp->i = 0;    /* not permitted - cssp points to const objects */

      exit(EXIT_SUCCESS);
}
```

Not all compiler writers seem to have noticed that requirement—the compiler that we used to test the last example failed to warn that the final assignment violated a constraint.

Here is the Example 6.1 rewritten using pointers, and with the input function infun changed to accept a pointer to a structure rather than returning one. This is much more likely to be what would be seen in practice.

(It is fair to say that, for a really efficient implementation, even the copying of structures would probably be dropped, especially if they were large. Instead, an array of pointers would be used, and the pointers exchanged until the sorted data could be found by traversing the pointer array in index order. That would complicate things too much for a simple example.)

```
#include <stdio.h>
#include <stdlib.h>

#define ARSIZE 10

struct wp_char{
      char wp_cval;
      short wp_font;
      short wp_psize;
}ar[ARSIZE];

void infun(struct wp_char *);

main(){
      struct wp_char wp_tmp, *lo_indx, *hi_indx, *in_p;

      for(in_p = ar; in_p < &ar[ARSIZE]; in_p++){
              infun(in_p);
              if(in_p->wp_cval == '\n'){
                      /*
                       * Leave the loop.
                       * not incrementing in_p means that the
                       * '\n' is ignored in the sort
                       */
                      break;
              }
      }

      /*
       * Now a simple exchange sort.
       * We must be careful to avoid the danger of pointer underflow,
       * so check that there are at least two entries to sort.
       */

      if(in_p-ar > 1) for(lo_indx = ar; lo_indx <= in_p-2; lo_indx++){
              for(hi_indx = lo_indx+1; hi_indx <= in_p-1; hi_indx++){
                      if(lo_indx->wp_cval > hi_indx->wp_cval){
                              /*
                               * Swap the structures.
                               */
                              struct wp_char wp_tmp = *lo_indx;
                              *lo_indx = *hi_indx;
                              *hi_indx = wp_tmp;
                      }
              }
      }

      /* now print */
      for(lo_indx = ar; lo_indx < in_p; lo_indx++){
              printf("%c %d %d\n", lo_indx->wp_cval,
                              lo_indx->wp_font,
                              lo_indx->wp_psize);
      }
      exit(EXIT_SUCCESS);
}

void
infun( struct wp_char *inp){

      inp->wp_cval = getchar();
      inp->wp_font = 2;
      inp->wp_psize = 10;

      return;
}
```

Example 6.2

The next issue is to consider what a structure looks like in terms of storage layout. It's best not to worry about this too much, but it is sometimes useful if you have to use C to access record-structured data written by other programs. The `wp_char` structure will be allocated storage as shown in Figure 6.1.

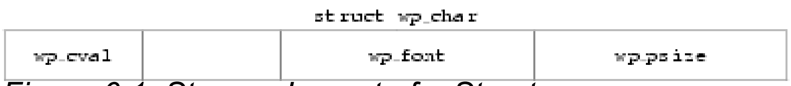


Figure 6.1. Storage Layout of a Structure

The diagram assumes a number of things: that a `char` takes 1 byte of storage; that a `short` needs 2 bytes; and that `short`s must be aligned on even byte addresses in this architecture. As a result the structure contains an unnamed 1-byte member inserted by the compiler for architectural reasons. Such addressing restrictions are quite common and can often result in structures containing 'holes'.

The Standard makes some guarantees about the layout of structures and unions:

- Members of a structure are allocated within the structure in the order of their appearance in the declaration and have ascending addresses.
- There must not be any padding in front of the first member.
- The address of a structure is the same as the address of its first member, provided that the appropriate cast is used. Given the previous declaration of `struct wp_char`, if item is of type `struct wp_char`, then `(char *)item == &item.wp_cval`.
- Bit fields (see Section 6.4 [*http://publications.gbdirect.co.uk/c_book/chapter6/bitfields.html*]) don't actually have addresses, but are conceptually packed into *units* which obey the rules above.

6.2.2. Linked lists and other structures

The combination of structures and pointers opens up a lot of interesting possibilities. This is not a textbook on complex linked data structures, but it will go on to describe two very common examples of the breed: linked lists and trees. Both have a feature in common: they consist of structures containing pointers to other structures, all the structures typically being of the same type. Figure 6.2 shows a picture of a linked list.

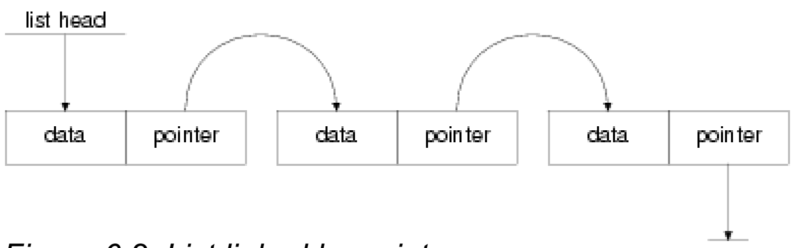


Figure 6.2. List linked by pointers

The sort of declaration needed for that is this:

```
struct list_ele{
        int data;       /* or whatever you like here */
        struct list_ele *ele_p;
};
```

Now, at first glance, it seems to contain itself—which is forbidden—but in fact it only contains a *pointer* to itself. How come the pointer declaration is allowed? Well, by the time the compiler reaches the pointer declaration it already knows that there is such a thing as a `struct list_ele` so the declaration is permitted. In fact, it is possible to make a incomplete declaration of a structure by saying

```
struct list_ele;
```

at some point before the full declaration. A declaration like that declares an *incomplete type*. This will allow the declaration of pointers before the full declaration is seen. It is also important in the case of cross-referencing structures where each must contain a pointer to the other, as shown in the following example.

```
struct s_1;     /* incomplete type */

struct s_2{
      int something;
      struct s_1 *sp;
};

struct s_1{     /* now the full declaration */
      float something;
      struct s_2 *sp;
};
```

Example 6.3

This illustrates the need for incomplete types. It also illustrates an important thing about the names of structure members: they inhabit a name-space per structure, so element names can be the same in different structures without causing any problems.

Incomplete types may only be used where the size of the structure isn't needed yet. A full declaration must have been given by the time that the size is used. The later full declaration mustn't be in an inner block because then it becomes a new declaration of a different structure.

```
struct x;       /* incomplete type */

/* valid uses of the tag */
struct x *p, func(void);

void f1(void){
      struct x{int i;};       /* redeclaration! */
}

/* full declaration now */
struct x{
      float f;
}s_x;

void f2(void){
      /* valid statements */
      p = &s_x;
      *p = func();
      s_x = func();
}

struct x
func(void){
      struct x tmp;
      tmp.f = 0;
      return (tmp);
}
```

Example 6.4

There's one thing to watch out for: you get a incomplete type of a structure *simply by mentioning its name!* That means that this works:

```
struct abc{ struct xyz *p;};
      /* the incomplete type 'struct xyz' now declared */
struct xyz{ struct abc *p;};
      /* the incomplete type is now completed */
```

There's a horrible danger in the last example, though, as this shows:

```
struct xyz{float x;} var1;

main(){
      struct abc{ struct xyz *p;} var2;

      /* AAAGH - struct xyz REDECLARED */
      struct xyz{ struct abc *p;} var3;
}
```

The result is that `var2.p` can hold the address of `var1`, but emphatically not the address of `var3` which is of a different type! It can be fixed (assuming that it's not what you wanted) like this:

```
struct xyz{float x;} var1;

main(){
      struct xyz;     /* new incomplete type 'struct xyz' */
      struct abc{ struct xyz *p;} var2;
      struct xyz{ struct abc *p;} var3;
}
```

The type of a structure or union is completed when the closing } of its declaration is seen; it must contain at least one member or the behaviour is undefined.

The other principal way to get incomplete types is to declare arrays without specifying their size—their type is incomplete until a later declaration provides the missing information:

```
int ar[];       /* incomplete type */
int ar[5];      /* completes the type */
```

If you try that out, it will only work if the declarations are outside any blocks (external declarations), but that's for other reasons.

Back to the linked list. There were three elements linked into the list, which could have been built like this:

```
struct list_ele{
      int data;
      struct list_ele *pointer;
}ar[3];

main(){

      ar[0].data = 5;
      ar[0].pointer = &ar[1];
      ar[1].data = 99;
      ar[1].pointer = &ar[2];
      ar[2].data = -7;
      ar[2].pointer = 0;      /* mark end of list */
      return(0);
}
```

Example 6.5

and the contents of the list can be printed in two ways. The array can be traversed in order of index, or the pointers can be used as in the following example.

```
#include <stdio.h>
#include <stdlib.h>

struct list_ele{
      int data;
      struct list_ele *pointer;
}ar[3];

main(){

      struct list_ele *lp;

      ar[0].data = 5;
      ar[0].pointer = &ar[1];
      ar[1].data = 99;
      ar[1].pointer = &ar[2];
      ar[2].data = -7;
      ar[2].pointer = 0;      /* mark end of list */

      /* follow pointers */
      lp = ar;
      while(lp){
              printf("contents %d\n", lp->data);
              lp = lp->pointer;
      }
      exit(EXIT_SUCCESS);
}
```

Example 6.6

It's the way that the pointers are followed which makes the example interesting. Notice how the pointer in each element is used to refer to the next one, until the pointer whose value is `0` is found That value causes the `while` loop to stop. Of course the pointers can be arranged in any order a all, which is what makes the list such a flexible structure. Here is a function which could be included as part of the last program to sort the linked list into numeric order of its data fields. It rearranges the pointers so that the list, when traversed in pointer sequence, is found to be in order. It is important to note that the data itself is not copied. The function must return a pointer to the head of the list, because that is not necessarily at `ar[0]` any more.

```
struct list_ele *
sortfun( struct list_ele *list )
{

      int exchange;
      struct list_ele *nextp, *thisp, dummy;

      /*
       * Algorithm is this:
       * Repeatedly scan list.
       * If two list items are out of order,
       * link them in the other way round.
       * Stop if a full pass is made and no
       * exchanges are required.
       * The whole business is confused by
       * working one element behind the
       * first one of interest.
```

```
         * This is because of the simple mechanics of
         * linking and unlinking elements.
         */

        dummy.pointer = list;
        do{
                exchange = 0;
                thisp = &dummy;
                while( (nextp = thisp->pointer)
                        && nextp->pointer){
                        if(nextp->data < nextp->pointer->data){
                                /* exchange */
                                exchange = 1;
                                thisp->pointer = nextp->pointer;
                                nextp->pointer =
                                        thisp->pointer->pointer;
                                thisp->pointer->pointer = nextp;
                        }
                        thisp = thisp->pointer;
                }
        }while(exchange);

        return(dummy.pointer);
}
```

Example 6.7

Expressions such as `thisp->pointer->pointer` are commonplace in list processing. It's worth making sure that you understand it; the notation emphasizes the way that links are followed.

6.2.3. Trees

Another very popular data structure is the tree. It's actually a linked list with branches; a common type is the *binary tree* which has elements (`nodes`) looking like this:

```
struct tree_node{
        int data;
        struct tree_node *left_p, *right_p;
};
```

For historical and essentially irrelevant reasons, trees in computer science work upside down. They have their *root* node at the top and their *branches* spread out downwards. In Figure 6.3, the ‘data’ members of the nodes are replaced by values which will be used in the discussion that follows.

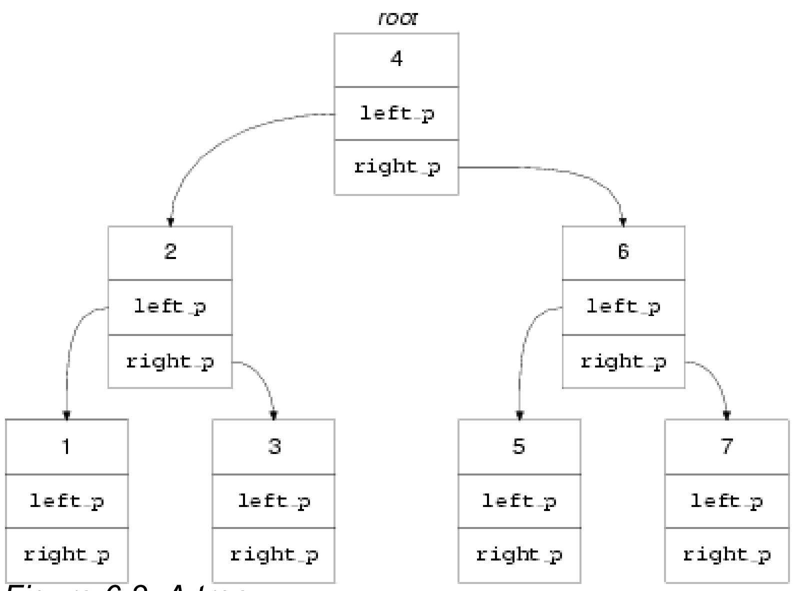


Figure 6.3. A tree

Trees may not seem very exciting if your main interest lies in routine character handling and processing, but they are extremely important to the designers of databases, compilers and other complex tools.

The advantage of a tree is that, if it is properly arranged, the layout of the data can support binary searching very simply. It is always possible to add new nodes to a tree at the appropriate place and a tree is basically a flexible and useful data structure.

Look at Figure 6.3. The tree is carefully constructed so that it can be searched to find whether a given value can be found in the data portions of the nodes. Let's say we want to find if a value *x* is already present in the tree. The algorithm is this:

```
Start at the root of the tree:
if the tree is empty (no nodes)
        then return ‘failure’.
else if the data in the current node is equal
        to the value being searched for
        then return ‘success’.
else if the data in the current node is greater than the
        value being searched for
        then search the tree indicated by the left pointer
else search the tree indicated by the right pointer.
```

Here it is in C:

```
#include <stdio.h>
#include <stdlib.h>
struct tree_node{
      int data;
      struct tree_node *left_p, *right_p;
}tree[7];
/*
* Tree search algorithm.
* Searches for value 'v' in tree,
* returns pointer to first node found containing
* the value otherwise 0.
*/
struct tree_node *
t_search(struct tree_node *root, int v){

      while(root){
```

```
                if(root->data == v)
                        return(root);
                if(root->data > v)
                        root = root->left_p;
                else
                        root = root->right_p;
        }
        /* value not found, no tree left */
        return(0);
}

main(){
        /* construct tree by hand */
        struct tree_node *tp, *root_p;
        int i;
        for(i = 0; i < 7; i++){
                int j;
                j = i+1;

                tree[i].data = j;
                if(j == 2 || j == 6){
                        tree[i].left_p = &tree[i-1];
                        tree[i].right_p = &tree[i+1];
                }
        }
        /* root */
        root_p = &tree[3];
        root_p->left_p = &tree[1];
        root_p->>right_p = &tree[5];

        /* try the search */
        tp = t_search(root_p, 9);
        if(tp)
                printf("found at position %d\n", tp-tree);
        else
                printf("value not found\n");
        exit(EXIT_SUCCESS);
}
```

Example 6.8

So that works fine. It is also interesting to note that, given a value, it can always be inserted at the appropriate point in the tree. The same search algorithm is used, but, instead of giving up when it finds that the value is not already in the tree, a new node is allocated by `malloc`, and is hung on the tree at the very place where the first null pointer was found. This is a mite more complicated t do because of the problem of handling the root pointer itself, and so a pointer to a pointer is used. Read the example carefully; it is not likely that you ever find anything more complicated than this in practice. If you can understand it, there is not much that should worry you about the vast majority of C language programs.

```
#include <stdio.h>
#include <stdlib.h>

struct tree_node{
        int data;
        struct tree_node *left_p, *right_p;
};

/*
* Tree search algorithm.
* Searches for value 'v' in tree,
* returns pointer to first node found containing
* the value otherwise 0.
*/
struct tree_node *
t_search(struct tree_node *root, int v){

      while(root){
              printf("looking for %d, looking at %d\n",
                      v, root->data);
              if(root->data == v)
                      return(root);
              if(root->data > v)
                      root = root->left_p;
              else
                      root = root->right_p;
      }
      /* value not found, no tree left */
      return(0);
}
/*
* Insert node into tree.
* Return 0 for success,
* 1 for value already in tree,
* 2 for malloc error
*/
int
t_insert(struct tree_node **root, int v){

      while(*root){
              if((*root)->data == v)
                      return(1);
              if((*root)->data > v)
                      root = &((*root)->left_p);
              else
                      root = &((*root)->right_p);
      }
      /* value not found, no tree left */
      if((*root = (struct tree_node *)
              malloc(sizeof (struct tree_node)))
                      == 0)
              return(2);
      (*root)-&data = v;
      (*root)-&left_p = 0;
      (*root)-&right_p = 0;
      return(0);
}

main(){
      /* construct tree by hand */
      struct tree_node *tp, *root_p = 0;
      int i;

      /* we ingore the return value of t_insert */
      t_insert(&root_p, 4);
      t_insert(&root_p, 2);
      t_insert(&root_p, 6);
      t_insert(&root_p, 1);
      t_insert(&root_p, 3);
```

```
        t_insert(&root_p, 5);
        t_insert(&root_p, 7);

        /* try the search */
        for(i = 1; i < 9; i++){
                tp = t_search(root_p, i);
                if(tp)
                        printf("%d found\n", i);
                else
                        printf("%d not found\n", i);
        }
        exit(EXIT_SUCCESS);
}
```

Example 6.9

Finally, the algorithm that allows you to walk along the tree visiting all the nodes in order is beautiful. It is the cleanest example of recursion that you are likely to see. Look at it and work out what it does.

```
void
t_walk(struct tree_node *root_p){

        if(root_p == 0)
                return;
        t_walk(root_p->left_p);
        printf("%d\n", root_p->data);
        t_walk(root_p->right_p);
}
```

Example 6.10

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter6/history.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter6/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter6/unions.html*]

6.3. Unions

Unions don't take long to explain. They are the same as structures, except that, where you would have written `struct` before, now you write `union`. Everything works the same way, but with one big exception. In a structure, the members are allocated separate consecutive chunks of storage. In a union, every member is allocated the same piece of storage. What would you use them for? Well, sometimes you want a structure to contain different values of different types at different times but to conserve space as much as possible. Using a union, it's up to you to keep track of whatever type you put into it and make sure that you retrieve the right type at the right time. Here's an example:

```
#include <stdio.h>
#include <stdlib.h>

main(){
        union {
                float u_f;
                int u_i;
        }var;

        var.u_f = 23.5;
        printf("value is %f\n", var.u_f);
        var.u_i = 5;
        printf("value is %d\n", var.u_i);
        exit(EXIT_SUCCESS);
}
```

Example 6.11

If the example had, say, put a `float` into the union and then extracted it as an int, a strange value would have resulted. The two types are almost certainly not only stored differently, but of different lengths. The `int` retrieved would probably be the low-order bits of the machine representation of a `float`, and might easily be made up of part of the mantissa of the `float` plus a piece of the exponent. The Standard says that if you do this, the behaviour is implementation defined (not undefined). The behaviour is defined by the Standard in one case: if some of the members of a union are structures with a 'common initial sequence' (the first members of each structure have compatible type and in the case of *bitfields* are the same length), and the union currently contains one of them, then the common initial part of each can be used interchangeably. Oh good.

The C compiler does no more than work out what the biggest member in a union can be and allocates enough storage (appropriately aligned if neccessary). In particular, no checking is done to make sure that the right sort of use is made of the members. That is your task, and you'll soon find out if you get it wrong. The members of a union all start at the same address—there is guaranteed to be no padding in front of any of them.

The most common way of remembering what is in a union is to embed it in a structure, with another member of the structure used to indicate the type of thing currently in the union. Here is how it might be used:

```
#include <stdio.h>
#include <stdlib.h>

/* code for types in union */
#define FLOAT_TYPE      1
#define CHAR_TYPE       2
#define INT_TYPE        3

struct var_type{
      int type_in_union;
      union{
              float   un_float;
              char    un_char;
              int     un_int;
      }vt_un;
}var_type;

void
print_vt(void){

      switch(var_type.type_in_union){
              default:
                      printf("Unknown type in union\n");
                      break;
              case FLOAT_TYPE:
                      printf("%f\n", var_type.vt_un.un_float);
                      break;
              case CHAR_TYPE:
                      printf("%c\n", var_type.vt_un.un_char);
                      break;
              case INT_TYPE:
                      printf("%d\n", var_type.vt_un.un_int);
                      break;
      }
}

main(){

      var_type.type_in_union = FLOAT_TYPE;
      var_type.vt_un.un_float = 3.5;

      print_vt();

      var_type.type_in_union = CHAR_TYPE;
      var_type.vt_un.un_char = 'a';

      print_vt();
      exit(EXIT_SUCCESS);
}
```

Example 6.12

That also demonstrates how the dot notation is used to access structures or unions inside other structures or unions. Some current C compilers allow you to miss bits out of the names of embedded objects provided that they are not ambiguous. In the example, such an unambiguous name would be `var_type.un_int` and the compiler would work out what you meant. None the less this is not permitted by the Standard.

It is because of unions that structures cannot be compared for equality. The possibility that a structure might contain a union makes it hard to compare such structures; the compiler can't tell what the union currently contains and so wouldn't know how to compare the structures. This sounds a bit hard to swallow and isn't 100% true—most structures don't contain unions—but there is also a philosophical issue at stake about just what is meant by ‘equality’ when applied to structures. Anyhow, the union business gives the Standard a good excuse to avoid the issue by not supporting structure comparison.

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter6/structures.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter6/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter6/bitfields.html*]

6.4. Bitfields

While we're on the subject of structures, we might as well look at bitfields. They can only be declared inside a structure or a union, and allow you to specify some very small objects of a given number of bits in length. Their usefulness is limited and they aren't seen in many programs, but we'll deal with them anyway. This example should help to make things clear:

```
struct {
      /* field 4 bits wide */
      unsigned field1 :4;
      /*
       * unnamed 3 bit field
       * unnamed fields allow for padding
       */
      unsigned        :3;
      /*
       * one-bit field
       * can only be 0 or -1 in two's complement!
       */
      signed field2   :1;
      /* align next field on a storage unit */
      unsigned        :0;
      unsigned field3 :6;
}full_of_fields;
```

Example 6.13

Each field is accessed and manipulated as if it were an ordinary member of a structure. The keywords `signed` and `unsigned` mean what you would expect, except that it is interesting to note that a 1-bit signed field on a two's complement machine can only take the values `0` or `-1`. The declarations are permitted to include the `const` and `volatile` qualifiers.

The main use of bitfields is either to allow tight packing of data or to be able to specify the fields within some externally produced data files. C gives no guarantee of the ordering of fields within machine words, so if you do use them for the latter reason, you program will not only be non-portable, it will be compiler-dependent too. The Standard says that fields are packed into ‘storage units’, which are typically machine words. The packing order, and whether or not a bitfield may cross a storage unit boundary, are implementation defined. To force alignment to a storage unit boundary, a zero width field is used before the one that you want to have aligned.

Be careful using them. It can require a surprising amount of run-time code to manipulate these things and you can end up using more space than they save.

Bit fields do not have addresses—you can't have pointers to them or arrays of them.

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter6/unions.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter6/*] | Next section

[*http://publications.gbdirect.co.uk/c_book/chapter6/enums.html*]

6.5. Enums

<gbdirect>

These fall into the category of ‘half baked’. They aren't proper enumerated types, as in Pascal, and only really serve to help you reduce the number of `#define` statements in your program. They look like this:

```
enum e_tag{
        a, b, c, d=20, e, f, g=20, h
}var;
```

Just as with structures and unions, the `e_tag` is the tag, and `var` is the definition of a variable.

The names declared inside the enumeration are constants with `int` type. Their values are these:

```
a == 0
b == 1
c == 2
d == 20
e == 21
f == 22
g == 20
h == 21
```

so you can see that, in the absence of anything to the contrary, the values assigned start at zero and increase. A specific value can be given if you want, when the increase will continue one at a time afterwards; the specific value must be an *integral constant* (see later) that is representable in an int. It is possible for more than one of the names to have the same value.

The only use for these things is to give a better-scoped version of this:

```
#define a 0
#define b 1
/* and so on */
```

It's better scoped because the declaration of enumerations follows the standard scope rules for C, whereas `#define` statements have file scope.

Not that you are likely to care, but the Standard states that enumeration types are of a type that is compatible with an implementation-defined one of the integral types. So what? For interest's sake here is an illustration:

```
enum ee{a,b,c}e_var, *ep;
```

The names `a`, `b`, and `c` all behave as if they were `int` constants when you use them; `e_var` has type `enum ee` and `ep` is a pointer to enum `ee`. The compatibility requirement means that (amongst other implications) there will be an integral type whose address can be assigned to `ep` without violating the type-compatibility requirements for pointers.

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter6/bitfields.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter6/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter6/qualifiers_and_derived_types.html*]

6.6. Qualifiers and derived types

Arrays, structures and unions are ‘derived from’ (contain) other types; none of them may be derived from incomplete types. This means that a structure or union cannot contain an example of itself, because its own type is incomplete until the declaration is complete. Since a pointer to an incomplete type is not itself an incomplete type, it *can* be used in the derivation of arrays, structures and unions.

If any of the types that these things are derived from are qualified with `const` or `volatile`, they do *not* inherit that qualification. This means that if a structure contains a `const` object, the structure itself is not qualified with const and any non-const members can still be modified. This is what you would expect. However, the Standard does says that if any derived type contains a type that is qualified with `const` (or recursively any inner type does) then it is not modifiable—so a structure that contains a const cannot be on the left-hand side of an assignment operator.

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter6/enums.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter6/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter6/initialization.html*]

6.7. Initialization

Now that we have seen all of the data types supported by C, we can look at the subject of initialization. C allows ordinary variables, structures, unions and arrays to be given initial values in their definitions. Old C had some strange rules about this, reflecting an unwillingness by compiler writers to work too hard. The Standard has rationalized this, and now it is possible to initialize things as and when you want.

There are basically two sorts of initialization: at compile time, and at run time. Which one you get depends on the *storage duration* of the thing being initialized.

Objects with *static duration* are declared either outside functions, or inside them with the keyword `extern` or `static` as part of the declaration. These can *only* be initialized at compile time.

Any other object has *automatic duration*, and can only be initialized at run time. The two categories are mutually exclusive.

Although they are related, storage duration and *linkage* (see Chapter 4 [*http://publications.gbdirect.co.uk/c_book/chapter4/*]) are different and should not be confused.

Compile-time initialization can only be done using *constant expressions*; run-time initialization can be done using *any* expression at all. The Old C restriction, that only simple variables (not arrays, structures or unions) could be initialized at run time, has been lifted.

6.7.1. Constant expressions

There are a number of places where constant expressions must be used. The definition of what constitutes a constant expression is relatively simple.

A *constant expression* is evaluated by the compiler, not at run-time. It may be used anywhere that a constant may be used. Unless it is part of the operand of `sizeof`, it may not contain any assignment, increment or decrement operations, function calls or comma operators; that may seem odd, but it's because `sizeof` only needs to evaluate the type of an expression, not its value.

If real numbers are evaluated at compile-time, then the Standard insists that they are evaluated with at least as much precision and range as will be used at run-time.

A more restricted form, called the *integral constant expression* exists. This has integral type and only involves operands that are integer constants, enumeration constants, character constants, `sizeof` expressions and real constants that are the immediate operands of casts. Any cast operators are only allowed to convert arithmetic types to integral types. As with the previous note on `sizeof` expressions, since they don't have to be evaluated, just their type determined, no restrictions apply to their contents.

The *arithmetic constant expression* is like the integral constant expression, but allows real constants to be used and restricts the use of casts to converting one arithmetic type to another.

The *address constant* is a pointer to an object that has static storage duration or a pointer to a function. You can get these by using the `&` operator or through the usual conversions of array and function names into pointers when they are used in expressions. The operators `[]`, `.`, `->`, `&` (address of) and `*` (pointer dereference) as well as casts of pointers can all be used in the expression as long as they don't involve accessing the value of any object.

6.7.2. More initialization

The various types of constants are permitted in various places; integral constant expressions are particularly important because they are the only type of expression that may be used to specify the size of arrays and the values in `case` statement prefixes. The types of constants that are permitted in initializer expressions are less restricted; you are allowed to use: arithmetic constant expressions; null pointer or address constants; an address constant for an object plus or minus an integral constant expression. Of course it depends on the type of thing being initialized whether or not a particular type of constant expression is appropriate.

Here is an example using several initialized variables:

```
#include <stdio.h>
#include <stdlib.h>

#define NMONTHS 12

int month = 0;

short month_days[] =
      {31,28,31,30,31,30,31,31,30,31,30,31};

char *mnames[] ={
      "January", "February",
      "March", "April",
      "May", "June",
      "July", "August",
      "September", "October",
      "November", "December"
};

main(){

      int day_count = month;

      for(day_count = month; day_count < NMONTHS;
              day_count++){
              printf("%d days in %s\n",
                      month_days[day_count],
                      mnames[day_count]);
      }
      exit(EXIT_SUCCESS);
}
```

Example 6.14

Initializing ordinary variables is easy: put `= expression` after the variable name in a declaration, and the variable is initialized to the value of the expression. As with all objects, whether you can use any expression, or just a constant expression, depends on its storage duration.

Initializing arrays is easy for one-dimensional arrays. Just put a list of the values you want, separated by commas, inside curly brackets. The example shows how to do it. If you don't give a size for the array, then the number of initializers will determine the size. If you do give a size, then there must be at most that many initializers in the list. Too many is an error, too few will just initialize the first elements of the array.

You could build up a string like this:

```
char str[] = {'h', 'e', 'l', 'l', 'o', 0};
```

but because it is so often necessary to do that, it is also permitted to use a quoted string literal to initialize an array of chars:

```
char str[] = "hello";
```

In that case, the null at the end of the string will also be included if there is room, or if no size was specified. Here are examples:

```
/* no room for the null */
char str[5] = "hello";

/* room for the null */
char str[6] = "hello";
```

The example program used string literals for a different purpose: *there* they were being used to initialize an array of character pointers; a very different prospect.

For structures that have automatic duration, an expression of the right type can be used to initialize them, or else a bracketed list of constant expressions must be used:

```
#include <stdio.h>
#include <stdlib.h>

struct s{
      int a;
      char b;
      char *cp;
}ex_s = {
      1, 'a', "hello"
      };

main(){
      struct s first = ex_s;
      struct s second = {
              2, 'b', "byebye"
              };

      exit(EXIT_SUCCESS);
}
```

Example 6.15

Only the first member of a union can be initialized.

If a structure or union contains unnamed members, whether unnamed bitfields or padding for alignment, they are ignored in the initialization process; they don't have to be counted when you provide the initializers for the real members of the structure.

For objects that contain sub-objects within them, there are two ways of writing the initializer. It can be written out with an initializer for each member:

```
struct s{
      int a;
      struct ss{
              int c;
              char d;
      }e;
}x[] = {
      1, 2, 'a',
      3, 4, 'b'
      };
```

Example 6.16

which will assign `1` to `x[0].a`, `2` to `x[0].e.c`, `a` to `x[0].e.d` and `3` to `x[1].a` and so on.

It is *much* safer to use internal braces to show what you mean, or one missed value will cause havoc.

```
struct s{
      int a;
      struct ss{
              int c;
              char d;
      }e;
}x[] = {
      {1, {2, 'a'}},
      {3, {4, 'b'}}
      };
```

Example 6.17

Always fully bracket initializers—that is much the safest thing to do.

It is the same for arrays as for structures:

```
float y[4][3] = {
      {1, 3, 5},       /* y[0][0], y[0][1], y[0][2] */
      {2, 4, 6},       /* y[1][0], y[1][1], y[1][2] */
      {3, 5, 7}        /* y[2][0], y[2][1], y[2][2] */
};
```

Example 6.18

that gives full initialization to the first three rows of `y`. The fourth row, `y[3]`, is uninitialized.

Unless they have an explicit initializer, all objects with static duration are given implicit initializers—the effect is as if the constant `0` had been assigned to their components. This is in fact widely used—it is an assumption made by most C programs that external objects and internal static objects start with the value zero.

Initialization of objects with automatic duration is only guaranteed if their compound statement is entered ‘at the top’. Jumping into the middle of one may result in the initialization not happening—this is often undesirable and should be avoided. It is explicitly noted by the Standard with regard to `switch` statements, where providing initializers in declarations *cannot* be of any use; this is because a declaration is not linguistically a ‘statement’ and only statements may be labelled. As a result it is not possible for initializers in `switch` statements ever to be executed, because the entry to the block containing them *must* be below the declarations!

A declaration inside a function (block scope) can, using various techniques outlined in Chapter 4 [*http://publications.gbdirect.co.uk/c_book/chapter4/*] and Chapter 8 [*http://publications.gbdirect.co.uk/c_book/chapter8/*], be made to refer to an object that has either *external* or *internal linkage*. If you've managed to do that, and it's not likely to happen by accident, then you can't initialize the object as part of that declaration. Here is one way of trying it:

```
int x;                          /* external linkage */
main(){
      extern int x = 5;         /* forbidden */
}
```

Our test compiler didn't notice that one, either.

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter6/qualifiers_and_derived_types.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter6/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter6/summary.html*]

6.8. Summary

You now understand structures and unions. Bitfields and enumeration types really are not very important and you could manage quite well without them.

It is hard to emphasize how important is the use of structures, pointers and malloc in serious programs. If you aren't familiar with the use of structured data in the form of lists, trees and so on, get a good book now. Better still, try to enrol on a good course. Except in very specialized applications, it is usually the ability to structure data well, not the ability to write complicated algorithms, that makes it possible to construct clean, small and maintainable programs. Experienced software designers often say that once the right structure of the data has been determined, the rest is ‘simple’.

Undoubtedly, one of the reasons for the popularity of C among experienced software specialists is the freedom that it gives in the structuring of data, without sacrificing speed.

Initialization should not be overlooked. Although simple in concept, it is surprising how inconvenient many other languages make this. The ludicrous extreme is to insist on the use of assignment statements; C has a practical and convenient approach. If the concept of ‘fully bracketed initializers’ seems a bit unpleasant, don't worry. It is rare that you have to do it in practice; all that you need is to know how to do simple initialization and to know a book that describes the more complex initialization. To get the full low-down read the Standard, which is uncharacteristically penetrable when it discusses the matter; verging at times on lucidity.

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter6/initialization.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter6/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter6/exercises.html*]

6.9. Exercises

Exercise 6.1. What is the declaration of an untagged structure containing two `ints` called `a` and `b`?

Exercise 6.2. Why is such a declaration of limited use?

Exercise 6.3. What would the structure look like with a tag of `int_struc` and two variables called `x` and `y` of the structure type being defined?

Exercise 6.4. How would you declare a third variable later, with the the same type as `x` and `y` but called `z`?

Exercise 6.5. Assuming that `p` is the right type of pointer, how would you make it point to `z` and then set `z.a` to zero, using the pointer?

Exercise 6.6. What are the two ways of declaring a structure with incomplete type?

Exercise 6.7. What is unusual about a string `"like this"` when it's used to initialize a character array?

Exercise 6.8. What if it initializes a `char *`?

Exercise 6.9. Find out what a doubly linked list is. Reimplement the linked list example using one. Is it any easier to insert and delete elements in a doubly linked list?

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter6/summary.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter6/*]

Chapter 7

<gbdirect>

The Preprocessor

Previous chapter [*http://publications.gbdirect.co.uk/c_book/chapter6/*] | Next chapter [*http://publications.gbdirect.co.uk/c_book/chapter8/*]

7.1. Effect of the Standard

There's a neither-fish-nor-fowl feel to the preprocessor. It leads an uncomfortable existence bolted on to the side of C without the benefit of either integrating properly with the rest of the language or, given one's natural reaction of revulsion at its ugly nature, being something that you could choose to do without. Back in the pre-history of C it actually was optional and people did write C without it; it's more or less an accident that it's come to be seen as being part of the bag and baggage of the C programming environment. It was used to make up for a couple of modest deficiencies in the language—the definition of constants and the inclusion of standard definitions—and slipped in through the back door as a result.

There has never been a widely accepted formal standard for a lot of what the preprocessor does and differing versions of it have been implemented in different systems. As a result, programs using anything other than the very basic features have proved to be a problem: it's hard to port them.

The primary job of the Standard was to define the behaviour of the preprocessor in line with common practice; this has been done and will not surprise anyone who was familiar with Old C. The Standard has gone further, amid an element of controversy, and specifies a number of additional features that were pioneered in some of the preprocessor's more popular dialects. The controversy results from the fact that although these features may be useful, there has never been much agreement on how to implement them. On the grounds that programs using these techniques were clearly non-portable already, the Standard has not worried too much about backwards compatibility in these areas. The fact that there is now a standard for these advanced features should improve the overall portability of C programs in the future.

At the simplest level the preprocessor is easy to use and can help a lot to make programs easy to read and maintain. Using the advanced features is best left to experts. In our experience, only the very simplest use of `#define` and the conditional compilation `#if` family are suitable for beginners. If this is your first encounter with C, read the chapter once to see what you can pick up and use the exercises to test your basic understanding. Otherwise, we would suggest that at least six months experience is the minimum prerequisite for a full attack. Because of that, we don't try too hard to give an easy introduction in this chapter, but concentrate on getting down to detail.

Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter7/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter7/how_the_preprocessor_works.html*]

7.2. How the preprocessor works

Although the preprocessor (Figure 7.1) is probably going to be implemented as an integral part of an Standard C compiler, it can equally well be though of as a separate program which transforms C source code containing preprocessor directives into source code with the directives removed.

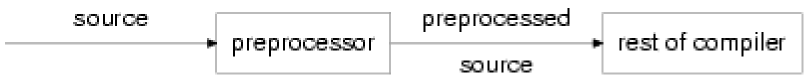


Figure 7.1. The preprocessor

It's important to remember that the preprocessor is not working to the same rules as the rest of C. It works on a line-by-line basis, so the end of a line means something special to it. The rest of C thinks that end-of-line is little different from a space or tab character.

The preprocessor doesn't know about the scope rules of C. Preprocessor directives like `#define` take effect as soon as they are seen and remain in effect until the end of the file that contains them; the program's block structure is irrelevant. This is one of the reasons why it's a good idea to make sparing use of these directives. The less you have in your program that doesn't obey the 'normal' scope rules, the less likely you are to make mistakes. This is mainly what gives rise to our comments about the poor level of integration between the preprocessor and the rest of C.

The Standard gives some complicated rules for the syntax of the preprocessor, especially with respect to *tokens*. To understand the operation of the preprocessor you need to know a little about them. The text that is being processed is not considered to be a uniform stream of characters, but is separated into tokens then processed piecemeal.

For a full definition of the process, it is best to refer to the Standard, but an informal description follows. Each of the terms used to head the list below is used later in descriptions of the rules.

1. *header-name*
   - '<' *almost any character* '>'
2. *preprocessing-token*
   - a *header-name* as above but only when the subject of `#include`,
   - or an *identifier* which is any C identifier or keyword,
   - or a *constant* which is any integral or floating constant,
   - or a *string-literal* which is a normal C string,
   - or an *operator* which is one of the C operators,
   - or one of *[ ] ( ) { } * , : = ; ... #* (punctuators)
   - or any non-white-space character not covered by the list above.

The '*almost any character*' above means any character except '>' or newline.

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter7/effect_of_the_standard.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter7/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter7/directives.html*]

7.3. Directives

Directives are always introduced by a line that starts with a # character, optionally preceded by white space characters (although it isn't common practice to indent the #). Table 7.1 below is a list of the directives defined in the Standard.

Directive	Meaning
`# include`	include a source file
`# define`	define a macro
`# undef`	undefine a macro
`# if`	conditional compilation
`# ifdef`	conditional compilation
`# ifndef`	conditional compilation
`# elif`	conditional compilation
`# else`	conditional compilation
`# endif`	conditional compilation
`# line`	control error reporting
`# error`	force an error message
`# pragma`	used for implementation-dependent control
`#`	null directive; no effect

Table 7.1. Preprocessor directives

The meanings and use of these features are described in the following sections. Make a note that the # and the following keyword, if any, are individual items. They may be separated by white space.

7.3.1. The null directive

This is simple: a plain # on a line by itself does nothing!

7.3.2. # define

There are two ways of defining *macros*, one of which looks like a function and one which does not. Here is an example of each:

```
#define FMAC(a,b) a here, then b

#define NONFMAC some text here
```

Both definitions define a macro and some *replacement text*, which will be used to replace later occurrences of the macro name in the rest of the program. After those definitions, they can be used as follows, with the effect of the macro replacement shown in comments:

```
NONFMAC
```

```
/* some text here */

FMAC(first text, some more)
/* first text here, then some more */
```

For the non-function macro, its name is simply replaced by its replacement text. The function macro is also replaced by its replacement text; wherever the replacement text contains an identifier which is the name of one of the macro's ‘formal parameters’, the actual text given as the argument is used in place of the identifier in the replacement text. The scope of the names of the formal parameters is limited to the body of the `#define` directive.

For both forms of macro, leading or trailing white space around the replacement text is discarded.

A curious ambiguity arises with macros: how do you define a non-function macro whose replacement text happens to start with the opening parenthesis character (? The answer is simple. If the definition of the macro has a space in front of the (, then it isn't the definition of a function macro, but a simple replacement macro instead. When you *use* function-like macros, there's no equivalent restriction.

The Standard allows either type of macro to be redefined at any time, using another `# define`, provided that there isn't any attempt to change the type of the macro and that the tokens making up both the original definition and the redefinition are identical in number, ordering, spelling and use of white space. In this context all white space is considered equal, so this would be correct:

```
# define XXX abc/*comment*/def hij
# define XXX abc def hij
```

because comment is a form of white space. The token sequence for both cases (*w-s* stands for a white-space token) is:

```
# w-s define w-s XXX w-s abc w-s def w-s hij w-s
```

7.3.2.1. Macro substitution

Where will occurrences of the macro name cause the replacement text to be substituted in its place? Practically anywhere in a program that the identifier is recognized as a separate token, except as the identifier following the # of a preprocessor directive. You can't do this:

```
#define define XXX

#define YYY ZZZ
```

and expect the second `#define` to be replaced by `#XXX`, causing an error.

When the identifier associated with a non-function macro is seen, it is replaced by the macro replacement tokens, then *rescanned* (see later) for further replacements to make.

Function macros can be used like real functions; white space around the macro name, the argument list and so on, may include the newline character:

```
#define FMAC(a, b) printf("%s %s\n", a, b)

FMAC ("hello",
      "sailor"
      );
/* results in */
```

```
printf("%s %s\n", "hello", "sailor")
```

The ‘arguments’ of a function macro can be almost any arbitrary token sequence. Commas are used to separate the arguments from each other but can be hidden by enclosing them within parentheses, `(` and `)`. Matched pairs of `(` and `)` inside the argument list balance each other out, so a `)` only ends the invocation of the macro if the corresponding (is the one that started the macro invocation.

```
#define CALL(a, b) a b

CALL(printf, ("%d %d %s\n",1, 24, "urgh"));
/* results in */
printf ("%d %d %s\n",1, 24, "urgh");
```

Note very carefully that the parentheses around the second argument to CALL were preserved in the replacement: they were not stripped from the text.

If you want to use macros like `printt`, taking a variable number of arguments, the Standard is no help to you. They are not supported.

If any argument contains no preprocessor tokens then the behaviour is undefined. The same is true if the sequence of preprocessor tokens that forms the argument would otherwise have been another preprocessor directive:

```
#define CALL(a, b) a b

/* undefined behaviour in each case.... */
CALL(,hello)
CALL(xyz,
#define abc def)
```

In our opinion, the second of the erroneous uses of `CALL` *should* result in defined behaviour—anyone capable of writing that would clearly benefit from the attentions of a champion weightlifter wielding a heavy leather bullwhip.

When a function macro is being processed, the steps are as follows:

1. All of its arguments are identified.
2. Except in the cases listed in item 3 below, if any of the tokens in an argument are themselves candidates for macro replacement, the replacement is done until no further replacement is possible. If this introduces commas into the argument list, there is no danger of the macro suddenly seeming to have a different number of arguments; the arguments are *only* determined in the step above.
3. In the macro replacement text, identifiers naming one of the macro formal arguments are replaced by the (by now expanded) token sequence supplied as the actual argument. The replacement is suppressed only if the identifier is preceded by one of `#` or `##`, or followed by `##`.

7.3.2.2. Stringizing

There is special treatment for places in the macro replacement text where one of the macro formal parameters is found preceded by `#`. The token list for the actual argument has any leading or trailing white space discarded, then the `#` and the token list are turned into a single string literal. Spaces between the tokens are treated as space characters in the string. To prevent ‘unexpected’ results, any `"` or `\` characters within the new string literal are preceded by `\`.

This example demonstrates the feature:

```
#define MESSAGE(x) printf("Message: %s\n", #x)
```

```
MESSAGE (Text with "quotes");
/*
* Result is
* printf("Message: %s\n", "Text with \"quotes\"");
*/
```

7.3.2.3. Token pasting

A `##` operator may occur anywhere in the replacement text for a macro except at the beginning or end. If a parameter name of a function macro occurs in the replacement text preceded or followed by one of these operators, the actual token sequence for the corresponding macro argument is used to replace it. Then, for both function and non-function macros, the tokens surrounding the `##` operator are joined together. If they don't form a valid token, the behaviour is undefined. Then rescanning occurs.

As an example of token pasting, here is a multi-stage operation, involving rescanning (which is described next).

```
#define REPLACE some replacement text
#define JOIN(a, b) a ## b

JOIN(REP, LACE)
becomes, after token pasting,
REPLACE
becomes, after rescanning
some replacement text
```

7.3.2.4. Rescanning

Once the processing described above has occurred, the replacement text plus the following tokens of the source file is rescanned, looking for more macro names to replace. The one exception is that, within a macro's replacement text, the name of the macro itself is not expanded. Because macro replacement can be nested, it is possible for several macros to be in the process of being replaced at any one point: none of their names is a candidate for further replacement in the ‘inner’ levels of this process. This allows redefinition of existing functions as macros:

```
#define exit(x) exit((x)+1)
```

These macro names which were not replaced now become tokens which are immune from future replacement, even if later processing might have meant that they had become available for replacement. This prevents the danger of infinite recursion occurring in the preprocessor. The suppression of replacement is only if the macro name results directly from *replacement* text, not the other source text of the program. Here is what we mean:

```
#define m(x) m((x)+1)
/* so */
m(abc);
/* expands to */
m((abc)+1);
/*
* even though the m((abc)+1) above looks like a macro,
* the rules say it is not to be re-replaced
*/

m(m(abc));
/*
* the outer m( starts a macro invocation,
* but the inner one is replaced first (as above)
* with m((abc)+1), which becomes the argument to the outer call,
* giving us effectively
*/
m(m((abc+1));
/*
* which expands to
*/
m((m((abc+1))+1);
```

If that doesn't make your brain hurt, then go and read what the Standard says about it, which will.

7.3.2.5. Notes

There is a subtle problem when using arguments to function macros.

```
/* warning - subtle problem in this example */
#define SQR(x)  ( x * x )
/*
* Wherever the formal parameters occur in
* the replacement text, they are replaced
* by the actual parameters to the macro.
*/
printf("sqr of %d is %d\n", 2, SQR(2));
```

The formal parameter of `SQR` is `x`; the actual argument is `2`. The replacement text results in

```
printf("sqr of %d is %d\n", 2, ( 2 * 2 ));
```

The use of the parentheses should be noticed. The following example is likely to give trouble:

```
/* bad example */
#define DOUBLE(y) y+y

printf("twice %d is %d\n", 2, DOUBLE(2));
printf("six times %d is %d\n", 2, 3*DOUBLE(2));
```

The problem is that the last expression in the second printf is replaced by

```
3*2+2
```

which results in `8`, not `12`! The rule is that when using macros to build expressions, careful parenthesizing is necessary. Here's another example:

```
SQR(3+4)

/* expands to */

( 3+4 * 3+4 )
/* oh dear, still wrong! */
```

so, when formal parameters occur in the replacement text, you should look carefully at them too. Correct versions of `SQR` and `DOUBLE` are these:

```
#define SQR(x) ((x)*(x))
#define DOUBLE(x) ((x)+(x))
```

Macros have a last little trick to surprise you with, as this shows.

```
#include <stdio.h>
#include <stdlib.h>
#define DOUBLE(x)  ((x)+(x))

main(){
      int a[20], *ip;

      ip = a;
      a[0] = 1;
      a[1] = 2;
      printf("%d\n", DOUBLE(*ip++));
      exit(EXIT_SUCCESS);
}
```

Example 7.1

Why is this going to cause problems? Because the replacement text of the macro refers to `*ip++` twice, so `ip` gets incremented twice. Macros should never be used with expressions that involve side effects, unless you check very carefully that they are safe.

Despite these warnings, they provide a very useful feature, and one which will be used a lot from now on.

7.3.3. # undef

The name of any `#define`d identifier can be forcibly forgotten by saying

```
#undef  NAME
```

It isn't an error to `#undef` a name which isn't currently defined.

This occasionally comes in handy. Chapter 9 [*http://publications.gbdirect.co.uk/c_book/chapter9/*] points out that some library functions may actually be macros, not functions, but by undefing their names you are guaranteed access to a real function.

7.3.4. # include

This comes in two flavours:

```
#include <filename>
#include "filename"
```

both of which cause a new file to be read at the point where they occur. It's as if the single line containing the directive is replaced by the contents of the specified file. If that file contains erroneous statements, you can reasonably expect that the errors will be reported with a correct file name and line number. It's the compiler writer's job to get that right. The Standard specifies that at least eight nested levels of `# include` must be supported.

The effect of using brackets <> or quotes `"  "` around the filename is to change the places searched to find the specified file. The brackets cause a search of a number of implementation defined places, the quotes cause a search of somewhere associated with the original source file. Your implementation notes must tell you the specific details of what is meant by 'place'. If the form using quotes can't find the file, it tries again as if you had used brackets.

In general, brackets are used when you specify standard library header files, quotes are used for private header files—often specific to one program only.

Although the Standard doesn't define what constitutes a valid file name, it does specify that there must be an implementation-defined unique way of translating file names of the form `xxx.x` (where `x` represents a 'letter'), into source file names. Distinctions of upper and lower case may be ignored and the implementation may choose only to use six significant characters before the '`.`' character.

You can also write this:

```
# define NAME <stdio.h>
# include NAME
```

to get the same effect as

```
# include <stdio.h>
```

but it's a rather roundabout way of doing it, and unfortunately it's subject to implementation defined rules about how the text between < and > is treated.

It's simpler if the replacement text for `NAME` comes out to be a string, for example

```
#define NAME "stdio.h"

#include NAME
```

There is no problem with implementation defined behaviour here, but the paths searched are different, as explained above.

For the first case, what happens is that the token sequence which replaces NAME is (by the rules already given)

```
<
stdio
.
h
>
```

and for the second case

```
"stdio.h"
```

The second case is easy, since it's just a *string-literal* which is a legal token for a `# include` directive. It is implementation defined how the first case is treated, and whether or not the sequence of tokens forms a legal *header-name*.

Finally, the last character of a file which is being `include`d must be a plain newline. Failure to include a file successfully is treated as an error.

7.3.5. Predefined names

The following names are predefined within the preprocessor:

`__LINE__`
    The current source file line number, a decimal integer constant.
`__FILE__`
    The 'name' of the current source code file, a string literal.
`__DATE__`

    The current date, a string literal. The form is

```
Apr 21 1990
```

where the month name is as defined in the library function `asctime` and the first digit of the date is a space if the date is less than 10.

`__TIME__`
: The time of the translation; again a string literal in the form produced by asctime, which has the form `"hh:mm:ss"`.

`__STDC__`
: The integer constant 1. This is used to test if the compiler is Standard-conforming, the intention being that it will have different values for different releases of the Standard.

A common way of using these predefined names is the following:

```
#define TEST(x) if(!(x))\
      printf("test failed, line %d file %s\n",\
              __LINE__, __FILE__)

/**/

TEST(a != 23);

/**/
```

Example 7.2

If the argument to `TEST` gives a false result, the message is printed, including the filename and line number in the message.

There's only one minor caveat: the use of the `if` statement can cause confusion in a case like this:

```
if(expression)
      TEST(expr2);
else
      statement_n;
```

The else will get associated with the hidden if generated by expanding the `TEST` macro. This is most unlikely to happen in practice, but will be a thorough pain to track down if it ever does sneak up on you. It's good style to make the bodies of every control of flow statement compound anyway; then the problem goes away.

None of the names `__LINE__`, `__FILE__`, `__DATE__`, `__TIME__`, `__STDC__` or defined may be used in `#define` or `#undef` directives.

The Standard specifies that any other reserved names will either start with an underscore followed by an upper case letter or another underscore, so you know that you are free to use any other names for your own purposes (but watch out for additional names reserved in Library header files that you may have included).

7.3.6. #line

This is used to set the value of the built in names `__LINE__` and `__FILE__`. Why do this? Because a lot of tools nowadays actually generate C as their output. This directive allows them to control the current line number. It is of very limited interest to the ‘ordinary’ C programmer.

Its form is

```
# line number optional-string-literal newline
```

The number sets the value of `__LINE__`, the string literal, if present, sets the value of `__FILE__`.

In fact, the sequence of tokens following `#line` will be macro expanded. After expansion, they are expected to provide a valid directive of the right form.

7.3.7. Conditional compilation

A number of the directives control conditional compilation, which allows certain portions of a program to be selectively compiled or ignored depending upon specified conditions. The directives concerned are: `#if`, `#ifdef`, `#ifndef`, `#elif`, `#else`, `#endif` together with the preprocessor unary operator defined.

The way that they are used is like this:

```
#ifdef  NAME
/* compile these lines if NAME is defined */
#endif
#ifndef NAME
/* compile these lines if NAME is not defined */
#else
/* compile these lines if NAME is defined */
#endif
```

So, `#ifdef` and `#endif` can be used to test the definition or otherwise of a given macro name. Of course the `#else` can be used with `#ifdef` (and `#if` or `#elif`) too. There is no ambiguity about what a given `#else` binds to, because the use of `#endif` to delimit the scope of these directives eliminates any possible ambiguity. The Standard specifies that at least eight levels of nesting of conditional directives must be supported, but in practice there is not likely to be any real limit.

These directives are most commonly used to select small fragments of C that are machine specific (when it is not possible to make the whole program completely machine independent), or sometimes to select different algorithms depending on the need to make trade-offs.

The `#if` and `#elif` constructs take a single integral constant expression as their arguments. Preprocessor integral constant expressions are the same as other integral constant expressions except that they must not contain cast operators. The token sequence that makes up the constant expression undergoes macro replacement, except that names prefixed by defined are not expanded. In this context, the expression `defined NAME` or `defined ( NAME )` evaluates to `1` if `NAME` is currently defined, `0` if it is not. Any other identifiers in the expression *including those that are C keywords* are replaced with the value `0`. Then the expression is evaluated. The replacement even of keywords means that `sizeof` can't be used in these expressions to get the result that you would normally expect.

As with the other conditional statements in C, a resulting value of zero is used to represent ‘false’, anything else is ‘true’.

The preprocessor always must use arithmetic with at least the ranges defined in the `<limits.h>` file and treats int expressions as long int and unsigned int as unsigned long int. Character constants do not necessarily have the same values as they do at execution time, so for highly portable programs, it's best to avoid using them in preprocessor expressions. Overall, the rules mean that it is possible to get arithmetic results from the preprocessor which are different from the results at run time; although presumably only if the translation and execution are done on different machines.

Here's an example.

```
#include <limits.h>

#if ULONG_MAX+1 != 0
      printf("Preprocessor: ULONG_MAX+1 != 0\n");
#endif

      if(ULONG_MAX+1 != 0)
              printf("Runtime: ULONG_MAX+1 != 0\n");
```

Example 7.3

It is conceivable that the preprocessor might perform arithmetic with a greater range than that used in the target environment. In that case, the preprocessor expression `ULONG_MAX+1` might not ‘overflow’ to give the result of `0`, whereas in the execution environment, it *must*.

The following skeleton example illustrates the use of such constants and also the ‘conditional else’, `#elif`.

```
#define NAME    100

#if     ((NAME > 50) && (defined __STDC__))
/* do something */
#elif   NAME > 25
/* do something else*/
#elif   NAME > 10
/* do something else */
#else
/* last possibility */
#endif
```

A word of warning. These conditional compilation directives do not obey the same scope rules as the rest of C. They should be used sparingly, unless your program is rapidly to become unreadable. It is impossible to read C when it is laced with these things every few lines. The urge to maim the author of a piece of code becomes very strong when you suddenly come across

```
#else
      }
#endif
```

with no `#if` or whatever immediately visible above. They should be treated like chilli sauce; essential at times, but more than a tiny sprinkle is too much.

7.3.8. #pragma

This was the Standard Committee's way of ‘opening the back door’. It allows implementation-defined things to take place. If the implementation was not expecting what you wrote (i.e. doesn't recognize it), it is ignored. Here is a possible example:

```
#pragma byte_align
```

which could be used to tell the implementation that all structure members should be aligned on byte addresses - some processor architectures are able to cope with word-sized structure members aligned on byte addresses, but with a penalty in access speed being incurred.

It could, of course, mean anything else that the implementation chooses it to mean.

If your implementation doesn't have any special meaning for this, then it will have no effect. It will *not* count as an error.

It will be interesting to see the sort of things that this gets used for.

7.3.9. #error

This directive is followed by one or more tokens at the end of the line. A diagnostic message is produced by the compiler, which includes those tokens, but no further detail is given in the Standard. It might be used like this to abort a compilation on unsuitable hardware:

```
#include <limits.h>
#if CHAR_MIN > -128
#error character range smaller than required
#endif
```

which would be expected to produce some sort of meaningful compilation error and message.

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter7/how_the_preprocessor_works.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter7/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter7/summary.html*]

7.4. Summary

To be honest, although many of the facilities provided by the preprocessor undoubtedly provide extra power and flexibility, it really is rather overcomplicated.

There are only a very few aspects that are really important.

The ability to define macros and function macros is very important, being widely used in almost every C program except the most trivial.

The conditional compilation has two important uses; one is the ability to compile with or without debugging statements included in a program, the other is to be able to select machine or application dependent statements.

Obviously, file inclusion is fundamentally important.

Having said the above, most of the rest of the features described in this chapter can be forgotten with very little loss of functionality. Perhaps each programming team should have just one preprocessor specialist who has the job of designing project-dependent macros using the arcane features such as stringizing and token pasting. Most users of C would benefit much more by putting that learning effort into other parts of the language, or, when they fully understand C, techniques of software quality control. The world would be a better place.

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter7/directives.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter7/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter7/exercises.html*]

7.5. Exercises

These exercises are intended to test only a basic understanding of the preprocessor, suitable for a beginner. Many users will never need a more detailed understanding.

Exercise 7.1. How would you arrange that the identifier `MAXLEN` is replaced by the value `100` throughout a program?

Exercise 7.2. What is likely to cause problems in a definition of the form `#define VALUE 100+MAXLEN`?

Exercise 7.3. Write a macro called `REM` which takes two integer arguments and 'returns' the remainder when the first is divided by the second.

Exercise 7.4. Repeat the last example, but use casts so that any arithmetic type of argument may be used, assuming that there are no overflow problems.

Exercise 7.5. What do the `<>` brackets around a filename in a `#include` directive signify?

Exercise 7.6. What would `""` mean in place of the `<>`?

Exercise 7.7. How would you use the preprocessor to select implementation-specific fragments of a program?

Exercise 7.8. What sort of arithmetic does the preprocessor use?

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter7/summary.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter7/*]

Chapter 8

Specialized Areas of C

Previous chapter [*http://publications.gbdirect.co.uk/c_book/chapter7/*] | Next chapter [*http://publications.gbdirect.co.uk/c_book/chapter9/*]

8.1. Government Health Warning

The previous chapters have introduced the fundamentals of the language and have covered nearly all of the language that the Standard defines. There are a number of murky and convoluted backwaters left unexplored on grounds of sympathy and compassion for the sufferer, and some without any better home. This chapter gathers them together—it's the toxic waste dump for the nasty bits of C.

Pull on your rubber gloves, read the following sections and make notes where you think the material is important to you; re-read them from time to time as well. What seemed uninteresting and painful the first time round may change as your experience grows, or your natural immunity improves.

What we cover here is *not* an exhumation of all the pathogenic elements—we leave that for another book—but it does serve to round up most of the commonly encountered difficult or extraordinary material.

Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter8/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter8/declarations_and_definitions.html*]

8.2. Declarations, Definitions and Accessibility

Chapter 4 [*http://publications.gbdirect.co.uk/c_book/chapter4/*] introduced the concepts of *scope* and *linkage*, showing how they can be combined to control the accessibility of things throughout a program. We deliberately gave a vague description of exactly what constitutes a *definition* on the grounds that it would give you more pain than gain at that stage. Eventually it has to be spelled out in detail, which we do in this chapter. Just to make things interesting, we need to throw in *storage class* too.

You'll probably find the interactions between these various elements to be both complex and confusing: that's because they are! We try to eliminate some of the confusion and give some useful rules of thumb in Section 8.2.6 [*http://publications.gbdirect.co.uk/c_book/chapter8/declarations_and_definitions.html#section-6*] below—but to understand them, you still need to read the stuff in between at least once.

For a full understanding, you need a good grasp of three distinct but related concepts. The Standard calls them:

- duration
- scope
- linkage

and describes what they mean in a fairly readable way (for a standard). Scope and linkage have already been described in Chapter 4 [*http://publications.gbdirect.co.uk/c_book/chapter4/*], although we do present a review of them below.

8.2.1. Storage class specifiers

There are five keywords under the category of *storage class specifiers*, although one of them, `typedef`, is there more out of convenience than utility; it has its own section later since it doesn't really belong here. The ones remaining are `auto`, `extern`, `register`, and `static`.

Storage class specifiers help you to specify the type of storage used for data objects. Only one storage class specifier is permitted in a declaration—this makes sense, as there is only one way of storing things—and if you omit the storage class specifier in a declaration, a default is chosen. The default depends on whether the declaration is made outside a function (external declarations) or inside a function (internal declarations). For external declarations the default storage class specifier will be `extern` and for `internal` declarations it will be `auto`. The only exception to this rule is the declaration of functions, whose *default* storage class specifier is always `extern`.

The positioning of a declaration, the storage class specifiers used (or their defaults) and, in some cases, preceding declarations of the same name, can all affect the linkage of a name, although fortunately not its scope or duration. We will investigate the easier items first.

8.2.1.1. Duration

The *duration* of an object describes whether its storage is allocated once only, at program start-up, or is more transient in its nature, being allocated and freed as necessary.

There are only two types of duration of objects: *static duration* and *automatic duration*. Static duration means that the object has its storage allocated permanently, automatic means that the storage is allocated and freed as necessary. It's easy to tell which is which: you only get automatic duration if

- the declaration is inside a function
- *and* the declaration does not contain the `static` or `extern` keywords
- *and* the declaration is not the declaration of a function

(if you work through the rules, you'll find that the formal parameters of a function always meet all three requirements—they are always 'automatic').

Although the presence of `static` in a declaration unambiguously ensures that it has static duration, it's interesting to see that it is by no means the only way. This is a notorious source of confusion, but we just have to accept it.

Data objects declared inside functions are given the default storage class specifier of `auto` unless some other storage class specifier is used. In the vast majority of cases, you don't want these objects to be accessible from outside the function, so you want them to have *no linkage*. Either the default, `auto`, or the explicit `register storage` class specifier results in an object with no linkage and *automatic duration*. Neither `auto` nor `register` can be applied to a declaration that occurs outside a function.

The `register` storage class is quite interesting, although it is tending to fall into disuse nowadays. It suggests to the compiler that it would be a good idea to store the object in one or more hardware registers in the interests of speed. The compiler does not have to take any notice of this, but to make things easy for it, `register` variables do not have an address (the `&` address-of operator is forbidden) because some computers don't support the idea of addressable registers. Declaring too many `register` objects may slow the program down, rather than speed it up, because the compiler may either have to save more registers on entrance to a function, often a slow process, or there won't be enough registers remaining to be used for intermediate calculations. Determining when to use registers will be a machine-specific choice and should only be taken when detailed measurements show that a particular function needs to be speeded up. Then you will have to experiment. In our opinion, you should never declare register variables during program development. Get the program working first, then measure it, then, maybe, judicious use of registers will give a useful increase in performance. But that work will have to be repeated for every type of processor you move the program to; even within one family of processors the characteristics are often different.

A final note on `register` variables: this is the only storage class specifier that may be used in a function prototype or function definition. In a function prototype, the storage class specifier is simply ignored, in a function definition it is a hint that the actual parameter should be stored in a register if possible. This example shows how it might be used:

```
#include <stdio.h>
#include <stdlib.h>

void func(register int arg1, double arg2);

main(){
      func(5, 2);
      exit(EXIT_SUCCESS);
}

/*
* Function illustrating that formal parameters
* may be declared to have register storage class.
*/
void func(register int arg1, double arg2){
      /*
       * Illustrative only - nobody would do this
       * in this context.
       * Cannot take address of arg1, even if you want to
       */
      double *fp = &arg2;

      while(arg1){
              printf("res = %f\n", arg1 * (*fp));
              arg1--;
      }
}
```

Example 8.1

So, the duration of an object depends on the storage class specifier used, whether it's a data object or function, and the position (block or file scope) of the declaration concerned. The linkage is also dependent on the storage class specifier, what kind of object it is and the scope of the declaration. Table 8.1 and Table 8.2 show the resulting storage duration and apparent linkage for the various combinations of storage class specifiers and location of the declaration. The actual linkage of objects with static duration is a bit more complicated, so use these tables only as a guide to the simple cases and take a look at what we say later about definitions.

Storage Class Specifier	Function or Data Object	Linkage	Duration
`static`	either	internal	static
`extern`	either	probably external	static
none	function	probably external	static
none	data object	external	static

Table 8.1. External declarations (outside a function)

The table above omits the `register` and `auto` storage class specifiers because they are not permitted in file-scope (external) declarations.

Storage Class Specifier	Function or Data Object	Linkage	Duration
`register`	data object only	none	automatic
`auto`	data object only	none	automatic
`static`	data object only	none	static
`extern`	either	probably external	static
none	data object	none	automatic
none	function	probably external	static

Table 8.2. Internal declarations

Internal `static` variables retain their values between calls of the function that contains them, which is useful in certain circumstances (see Chapter 4 [*http://publications.gbdirect.co.uk/c_book/chapter4/*]).

8.2.2. Scope

Now we must look again at the *scope* of the names of objects, which defines when and where a given name has a particular meaning. The different types of scope are the following:

- function scope
- file scope
- block scope
- function prototype scope

The easiest is *function scope*. This only applies to labels, whose names are visible throughout the function where they are declared, irrespective of the block structure. No two labels in the same function may have the same name, but because the name only has function scope, the same name can be used for labels in every function. Labels are *not* objects—they have no storage associated with them and the concepts of linkage and duration have no meaning for them.

Any name declared outside a function has *file scope*, which means that the name is usable at any point from the declaration on to the end of the source code file containing the declaration. Of course it is possible for these names to be temporarily hidden by declarations within compound statements. As we know, function definitions *must* be outside other functions, so the name introduced by any function definition will always have file scope.

A name declared inside a compound statement, or as a formal parameter to a function, has *block scope* and is usable up to the end of the associated `}` which closes the compound statement. Any declaration of a name within a compound statement hides any outer declaration of the same name until the end of the compound statement.

A special and rather trivial example of scope is *function prototype scope* where a declaration of a name extends only to the end of the function prototype. That means simply that this is wrong (same name used twice):

```
void func(int i, int i);
```

and this is all right:

```
void func(int i, int j);
```

The names declared inside the parentheses disappear outside them.

The scope of a name is completely independent of any storage class specifier that may be used in its declaration.

8.2.3. Linkage

We will briefly review the subject of *linkage* here, too. *Linkage* is used to determine what makes the same name declared in different scopes refer to the same thing. An object only ever has one name, but in many cases we would like to be able to refer to the same object from different scopes. A typical example is the wish to be able to call `printf` from several different places in a program, even if those places are not all in the same source file.

The Standard warns that declarations which refer to the same thing must all have compatible type, or the behaviour of the program will be undefined. A full description of compatible type is given later; for the moment you can take it to mean that, except for the use of the storage class specifier, the declarations must be identical. It's the responsibility of the programmer to get this right, though there will probably be tools available to help you check this out.

The three different types of linkage are:

- external linkage
- internal linkage
- no linkage

In an entire program, built up perhaps from a number of source files and libraries, if a name has *external linkage*, then every instance of a that name refers to the same object throughout the program.

For something which has *internal linkage*, it is only within a given source code file that instances of the same name will refer to the same thing.

Finally, names with *no linkage* refer to separate things.

8.2.4. Linkage and definitions

Every data object or function that is actually used in a program (except as the operand of a `sizeof` operator) *must have one and only one* corresponding *definition*. This is actually very important, although we haven't really come across it yet because most of our examples have used only data objects with automatic duration, whose declarations are axiomatically definitions, or functions which we have defined by providing their bodies.

This ‘exactly one’ rule means that for objects with external linkage there must be exactly one definition in the whole program; for things with internal linkage (confined to one source code file) there must be exactly one definition in the file where it is declared; for things with no linkage, whose declaration is always a definition, there is exactly one definition as well.

Now we try to draw everything together. The real questions are

1. How do I get the sort of linkage that I want?
2. What actually constitutes a definition?

We need to look into linkage first, then definitions.

How do you get the appropriate linkage for a particular name? The rules are a little complicated.

1. A declaration outside a function (file scope) which contains the static storage class specifier results in *internal linkage* for that name. (The Standard requires that function declarations which contain `static` *must* be at file scope, outside any block)
2. If a declaration contains the `extern` storage class specifier, or is the declaration of a function with no storage class specifier (or both), then:
    - If there is already a visible declaration of that identifier with file scope, the resulting linkage is the same as that of the visible declaration;
    - otherwise the result is *external linkage*.
3. If a file scope declaration is neither the declaration of a function nor contains an explicit storage class specifier, then the result is *external linkage*.
4. Any other form of declaration results in *no linkage*.
5. In any one source code file, if a given identifer has both internal and external linkage then the result is undefined.

These rules were used to derive the ‘linkage’ columns of Table 8.1 and Table 8.2, without the full application of rule 2—hence the use of the ‘probably external’ term. Rule 2 allows you to determine the precise linkage in those cases.

What makes a declaration into a definition?

- Declarations that result in no linkage are also definitions.
- Declarations that include an initializer are always definitions; this includes the ‘initialization’ of functions by providing their body. Declarations with block scope may only have initializers if they also have no linkage.
- Otherwise, the declaration of a name with file scope and with either no storage class specifier or with the `static` storage class specifier is a *tentative definition*. If a source code file contains one or more tentative definitions for an object, then if that file contains no actual definitions, a default definition is provided for that object as if it had an initializer of `0`. (Structures and arrays have all their elements initialized to `0`). Functions do not have tentative definitions.

A consequence of the foregoing is that unless you also provide an initializer, declarations that explicitly include the extern storage class specifier do *not* result in a definition.

8.2.5. Realistic use of linkage and definitions

The rules that determine the linkage and definition associated with declarations look quite complicated. The combinations used in practice are nothing like as bad; so let's investigate the usual cases.

The three types of accessibility that you will want of data objects or functions are:

- throughout the entire program,
- restricted to one source file,
- restricted to one function (or perhaps a single compound statement).

For the three cases above, you will want external linkage, internal linkage, and no linkage respectively. The recommended practice for the first two cases is to declare all of the names in each of the relevant source files *before* you define any functions. The recommended layout of a source file would be as shown in Figure 8.1.

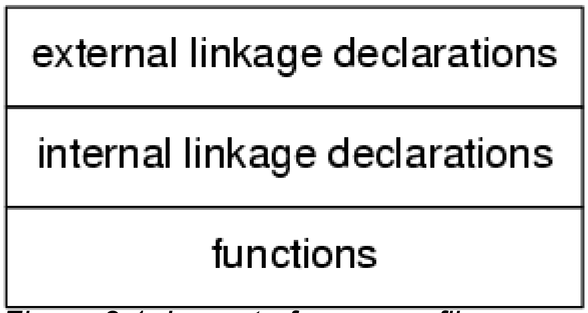


Figure 8.1. Layout of a source file

The external linkage declarations would be prefixed with extern, the internal linkage declarations with `static`. Here's an example.

```
/* example of a single source file layout */
#include <stdio.h>

/* Things with external linkage:
* accessible throughout program.
* These are declarations, not definitions, so
* we assume their definition is somewhere else.
*/

extern int important_variable;
extern int library_func(double, int);

/*
* Definitions with external linkage.
*/
extern int ext_int_def = 0;     /* explicit definition */
int tent_ext_int_def;           /* tentative definition */

/*
* Things with internal linkage:
* only accessible inside this file.
* The use of static means that they are also
* tentative definitions.
*/

static int less_important_variable;
static struct{
        int member_1;
        int member_2;
}local_struct;

/*
```

```
* Also with internal linkage, but not a tentative
* definition because this is a function.
*/
static void lf(void);

/*
* Definition with internal linkage.
*/
static float int_link_f_def = 5.3;

/*
* Finally definitions of functions within this file
*/

/*
* This function has external linkage and can be called
* from anywhere in the program.
*/
void f1(int a){}

/*
* These two functions can only be invoked by name from
* within this file.
*/
static int local_function(int a1, int a2){
        return(a1 * a2);
}

static void lf(void){
        /*
         * A static variable with no linkage,
         * so usable only within this function.
         * Also a definition (because of no linkage)
         */
        static int count;
        /*
         * Automatic variable with no linkage but
         * an initializer
         */
        int i = 1;

        printf("lf called for time no %d\n", ++count);
}
/*
* Actual definitions are implicitly provided for
* all remaining tentative definitions at the end of
* the file
*/
```

Example 8.2

We suggest that your re-read the preceding sections to see how the rules have been applied in Example 8.2.

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter8/health_warning.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter8/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter8/typedef.html*]

8.3. Typedef

<gbdirect>

Although typedef is thought of as being a storage class, it isn't really. It allows you to introduce synonyms for types which could have been declared some other way. The new name becomes equivalent to the type that you wanted, as this example shows.

```
typedef int aaa, bbb, ccc;
typedef int ar[15], arr[9][6];
typedef char c, *cp, carr[100];

/* now declare some objects */

/* all ints */
aaa     int1;
bbb     int2;
ccc     int3;

ar      yyy;    /* array of 15 ints */
arr     xxx;    /* 9*6 array of int */

c       ch;     /* a char */
cp      pnt;    /* pointer to char */
carr    chry;   /* array of 100 char */
```

The general rule with the use of typedef is to write out a declaration as if you were declaring variables of the types that you want. Where a declaration would have introduced names with particular types, prefixing the whole thing with typedef means that, instead of getting variables declared, you declare new type names instead. Those new type names can then be used as the prefix to the declaration of variables of the new type.

The use of `typedef` isn't a particularly common sight in most programs; it's typically found only in header files and is rarely the province of day-to-day coding.

It is sometimes found in applications requiring very high portability: there, new types will be defined for the basic variables of the program and appropriate `typedef`s used to tailor the program to the target machine. This can lead to code which C programmers from other environments will find difficult to interpret if it's used to excess. The flavour of it is shown below:

```
/* file 'mytype.h' */
typedef short   SMALLINT        /* range *******30000 */
typedef int     BIGINT          /* range ******* 2E9 */

/* program */
#include "mytype.h"

SMALLINT        i;
BIGINT          loop_count;
```

On some machines, the range of an int would not be adequate for a `BIGINT` which would have to be re- `typedef`'d to be `long`.

To re-use a name already declared as a `typedef`, its declaration must include at least one type specifier, which removes any ambiguity:

```
typedef int new_thing;
func(new_thing x){
        float new_thing;
        new_thing = x;
}
```

As a word of warning, `typedef` can only be used to declare the type of return value from a function, not the overall type of the function. The overall type includes information about the function's parameters as well as the type of its return value.

```
/*
* Using typedef, declare 'func' to have type
* 'function taking two int arguments, returning int'
*/
typedef int func(int, int);

/* ERROR */
func func_name{ /*....*/ }

/* Correct. Returns pointer to a type 'func' */
func *func_name(){ /*....*/ }

/*
* Correct if functions could return functions,
* but C can't.
*/
func func_name(){ /*....*/ }
```

If a `typedef` of a particular identifier is in scope, that identifer may not be used as the formal parameter of a function. This is because something like the following declaration causes a problem:

```
typedef int i1_t, i2_t, i3_t, i4_t;

int f(i1_t, i2_t, i3_t, i4_t)/*THIS IS POINT 'X'*/
```

A compiler reading the function declaration reaches point ‘X’ and still doesn't know whether it is looking at a function declaration, essentially similar to

```
int f(int, int, int, int) /* prototype */
```

or

```
int f(a, b, c, d) /* not a prototype */
```

—the problem is only resolvable (in the worst case) by looking at what follows point ‘X’; if it is a semicolon, then that was a declaration, if it is a `{` then that was a definition. The rule forbidding typedef names to be formal parameters means that a compiler can always tell whether it is processing a declaration or a definition by looking at the first identifier following the function name.

The use of typedef is also valuable when you want to declare things whose declaration syntax is painfully impenetrable, like ‘array of ten pointers to array of five integers’, which tends to cause panic even amongst the hardy. Hiding it in a typedef means you only have to read it once and can also help to break it up into manageable pieces:

```
typedef int (*a10ptoa5i[10])[5];
/* or */
typedef int a5i[5];
typedef a5i *atenptoa5i[10];
```

Try it out!

Previous section
[*http://publications.gbdirect.co.uk/c_book/chapter8/declarations_and_definitions.html*]
| Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter8/*] | Next section
[*http://publications.gbdirect.co.uk/c_book/chapter8/const_and_volatile.html*]

8.4. Const and volatile

<gbdirect>

These are new in Standard C, although the idea of `const` has been borrowed from C++. Let us get one thing straight: the concepts of `const` and `volatile` are *completely independent*. A common misconception is to imagine that somehow `const` is the opposite of `volatile` and vice versa. They are unrelated and you should remember the fact.

Since `const` declarations are the simpler, we'll look at them first, but only after we have seen where both of these type qualifiers may be used. The complete list of relevant keywords is

```
char      long      float     volatile
short     signed    double    void
int       unsigned  const
```

In that list, `const` and `volatile` are type qualifiers, the rest are *type specifiers*. Various combinations of type specifiers are permitted:

```
char, signed char, unsigned char
int, signed int, unsigned int
short int, signed short int, unsigned short int
long int, signed long int, unsigned long int
float
double
long double
```

A few points should be noted. All declarations to do with an `int` will be `signed` anyway, so signed is redundant in that context. If *any* other type specifier or qualifier is present, then the int part may be dropped, as that is the default.

The keywords `const` and `volatile` can be applied to any declaration, including those of structures, unions, enumerated types or `typedef` names. Applying them to a declaration is called *qualifying* the declaration—that's why const and volatile are called type qualifiers, rather than type specifiers. Here are a few representative examples:

```
volatile i;
volatile int j;
const long q;
const volatile unsigned long int rt_clk;
struct{
        const long int li;
        signed char sc;
}volatile vs;
```

Don't be put off; some of them are deliberately complicated: what they mean will be explained later. Remember that they could also be further complicated by introducing storage class specifications as well! In fact, the truly spectacular

```
extern const volatile unsigned long int rt_clk;
```

is a strong possibility in some real-time operating system kernels.

8.4.1. Const

Let's look at what is meant when `const` is used. It's really quite simple: `const` means that something is not modifiable, so a data object that is declared with `const` as a part of its type specification must not be assigned to in any way during the run of a program. It is very likely that the definition of the object will contain an initializer (otherwise, since you can't assign to it, how would it ever get a value?), but this is not always the case. For example, if you were accessing a hardware port at a fixed memory address and promised only to read from it, then it would be declared to be `const` but not initialized.

Taking the address of a data object of a type which isn't `const` and putting it into a pointer to the `const`-qualified version of the same type is both safe and explicitly permitted; you will be able to use the pointer to inspect the object, but not modify it. Putting the address of a const type into a pointer to the unqualified type is much more dangerous and consequently prohibited (although you can get around this by using a cast). Here is an example:

```
#include <stdio.h>
#include <stdlib.h>

main(){
        int i;
        const int ci = 123;

        /* declare a pointer to a const.. */
        const int *cpi;

        /* ordinary pointer to a non-const */
        int *ncpi;

        cpi = &ci;
        ncpi = &i;

        /*
         * this is allowed
         */
        cpi = ncpi;

        /*
         * this needs a cast
         * because it is usually a big mistake,
         * see what it permits below.
         */
        ncpi = (int *)cpi;

        /*
         * now to get undefined behaviour...
         * modify a const through a pointer
         */
        *ncpi = 0;

        exit(EXIT_SUCCESS);
}
```

Example 8.3

As the example shows, it is possible to take the address of a constant object, generate a pointer to a non-constant, then use the new pointer. This is an *error* in your program and results in undefined behaviour.

The main intention of introducing const objects was to allow them to be put into read-only store, and to permit compilers to do extra consistency checking in a program. Unless you defeat the intent by doing naughty things with pointers, a compiler is able to check that `const` objects are not modified explicitly by the user.

An interesting extra feature pops up now. What does this mean?

```
char c;
char *const cp = &c;
```

It's simple really; `cp` is a pointer to a `char`, which is exactly what it would be if the `const` weren't there. The `const` means that `cp` is not to be modified, although whatever it points to can be—the pointer is constant, not the thing that it points to. The other way round is

```
const char *cp;
```

which means that now cp is an ordinary, modifiable pointer, but the thing that it points to must not be modified. So, depending on what you choose to do, both the pointer and the thing it points to may be modifiable or not; just choose the appropriate declaration.

8.4.2. Volatile

After const, we treat `volatile`. The reason for having this type qualifier is mainly to do with the problems that are encountered in real-time or embedded systems programming using C. Imagine that you are writing code that controls a hardware device by placing appropriate values in hardware registers at known absolute addresses.

Let's imagine that the device has two registers, each 16 bits long, at ascending memory addresses; the first one is the control and status register (csr) and the second is a data port. The traditional way of accessing such a device is like this:

```
/* Standard C example but without const or volatile */
/*
* Declare the device registers
* Whether to use int or short
* is implementation dependent
*/

struct devregs{
        unsigned short  csr;    /* control & status */
        unsigned short  data;   /* data port */
};

/* bit patterns in the csr */
#define ERROR   0x1
#define READY   0x2
#define RESET   0x4

/* absolute address of the device */
#define DEVADDR ((struct devregs *)0xffff0004)

/* number of such devices in system */
#define NDEVS   4

/*
* Busy-wait function to read a byte from device n.
* check range of device number.
* Wait until READY or ERROR
* if no error, read byte, return it
* otherwise reset error, return 0xffff
*/
unsigned int read_dev(unsigned devno){

        struct devregs *dvp = DEVADDR + devno;

        if(devno >= NDEVS)
                return(0xffff);

        while((dvp->csr & (READY | ERROR)) == 0)
                ; /* NULL - wait till done */

        if(dvp->csr & ERROR){
                dvp->csr = RESET;
                return(0xffff);
        }

        return((dvp->data) & 0xff);
}
```

Example 8.4

The technique of using a structure declaration to describe the device register layout and names is very common practice. Notice that there aren't actually any objects of that type defined, so the declaration simply indicates the structure without using up any store.

To access the device registers, an appropriately cast constant is used as if it were pointing to such a structure, but of course it points to memory addresses instead.

However, a major problem with previous C compilers would be in the while loop which tests the status register and waits for the `ERROR` or `READY` bit to come on. Any self-respecting optimizing compiler would notice that the loop tests the same memory address over and over again. It would almost certainly arrange to reference memory once only, and copy the value into a hardware register, thus speeding up the loop. This is, of course, exactly what we don't want; this is one of the few places where we must look at the place where the pointer points, every time around the loop.

Because of this problem, most C compilers have been unable to make that sort of optimization in the past. To remove the problem (and other similar ones to do with when to write to where a pointer points), the keyword `volatile` was introduced. It tells the compiler that the object is subject to sudden change for reasons which cannot be predicted from a study of the program itself, and forces every reference to such an object to be a genuine reference.

Here is how you would rewrite the example, making use of `const` and `volatile` to get what you want.

```
/*
* Declare the device registers
* Whether to use int or short
* is implementation dependent
*/

struct devregs{
        unsigned short volatile csr;
        unsigned short const volatile data;
};

/* bit patterns in the csr */
#define ERROR   0x1
#define READY   0x2
#define RESET   0x4
```

```
/* absolute address of the device */
#define DEVADDR ((struct devregs *)0xffff0004)

/* number of such devices in system */
#define NDEVS   4

/*
* Busy-wait function to read a byte from device n.
* check range of device number.
* Wait until READY or ERROR
* if no error, read byte, return it
* otherwise reset error, return 0xffff
*/
unsigned int read_dev(unsigned devno){

        struct devregs * const dvp = DEVADDR + devno;

        if(devno >= NDEVS)
                return(0xffff);

        while((dvp->csr & (READY | ERROR)) == 0)
                ; /* NULL - wait till done */

        if(dvp->csr & ERROR){
                dvp->csr = RESET;
                return(0xffff);
        }

        return((dvp->data) & 0xff);
}
```

Example 8.5

The rules about mixing `volatile` and regular types resemble those for `const`. A pointer to a `volatile` object can be assigned the address of a regular object with safety, but it is dangerous (and needs a cast) to take the address of a `volatile` object and put it into a pointer to a regular object. Using such a derived pointer results in undefined behaviour.

If an array, union or structure is declared with `const` or `volatile` attributes, then all of the members take on that attribute too. This makes sense when you think about it—how could a member of a `const` structure be modifiable?

That means that an alternative rewrite of the last example would be possible. Instead of declaring the device registers to be `volatile` in the structure, the pointer could have been declared to point to a `volatile` structure instead, like this:

```
struct devregs{
      unsigned short  csr;    /* control & status */
      unsigned short  data;   /* data port */
};
volatile struct devregs *const dvp=DEVADDR+devno;
```

Since `dvp` points to a `volatile` object, it not permitted to optimize references through the pointer. Our feeling is that, although this would work, it is bad style. The `volatile` declaration belongs in the structure: it is the device registers which are `volatile` and that is where the information should be kept; it reinforces the fact for a human reader.

So, for any object likely to be subject to modification either by hardware or asynchronous interrupt service routines, the volatile type qualifier is important.

Now, just when you thought that you understood all that, here comes the final twist. A declaration like this:

```
volatile struct devregs{
      /* stuff */
}v_decl;
```

declares the type `struct devregs` and also a `volatile`-qualified object of that type, called `v_decl`. A later declaration like this

```
struct devregs nv_decl;
```

declares `nv_decl` which is *not* qualified with `volatile`! The qualification is *not* part of the type of `struct devregs` but applies only to the declaration of `v_decl`. Look at it this way round, which perhaps makes the situation more clear (the two declarations are the same in their effect):

```
struct devregs{
      /* stuff */
}volatile v_decl;
```

If you do want to get a shorthand way of attaching a qualifier to another type, you can use `typedef` to do it:

```
struct x{
      int a;
};
typedef const struct x csx;

csx const_sx;
struct x non_const_sx = {1};

const_sx = non_const_sx;        /* error - attempt to modify a const */
```

8.4.2.1. Indivisible Operations

Those of you who are familiar with techniques that involve hardware interrupts and other ‘real time’ aspects of programming will recognise the need for `volatile` types. Related to this area is the need to ensure that accesses to data objects are ‘atomic’, or uninterruptable. To discuss this is any depth would take us beyond the scope of this book, but we can at least outline some of the issues.

Be careful not to assume that any operations written in C are uninterruptable. For example,

```
extern const volatile unsigned long realtimeclock;
```

could be a counter which is updated by a clock interrupt routine. It is essential to make it `volatile` because of the asynchronous updates to it, and it is marked `const` because it should not be changed by anything other than the interrupt routine. If the program accesses it like this:

```
unsigned long int time_of_day;

time_of_day = real_time_clock;
```

there may be a problem. What if, to copy one `long` into another, it takes several machine instructions to copy the two words making up `real_time_clock` and `time_of_day`? It is possible that an interrupt will occur in the middle of the assignment and that in the worst case, when the low-order word of `real_time_clock` is `0xffff` and the high-order word is `0x0000`, then the low-order word of `time_of_day` will receive `0xffff`. The interrupt arrives and increments the low-order word of `real_time_clock` to `0x0` and then the high-order word to `0x1`, then returns. The rest of the assignment then completes, with `time_of_day` ending up containing `0x0001ffff` and `real_time_clock` containing the correct value, `0x00010000`.

This whole class of problem is what is known as a critical region, and is well understood by those who regularly work in asynchronous environments. It should be understood that Standard C takes no special precautions to avoid these problems, and that the usual techniques should be employed.

The header `'signal.h'` declares a type called `sig_atomic_t` which is guaranteed to be modifiable safely in the presence of asynchronous events. This means only that it can be modified by assigning a value to it; incrementing or decrementing it, or anything else which produces a new value depending on its previous value, is not safe.

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter8/typedef.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter8/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter8/sequence_points.html*]

8.5. Sequence points

Associated with, but distinct from, the problems of real-time programming are *sequence points*. These are the Standard's attempt to define when certain sorts of optimization may and may not be permitted to be in effect. For example, look at this program:

```
#include <stdio.h>
#include <stdlib.h>

int i_var;
void func(void);

main(){
        while(i_var != 10000){
                func();
                i_var++;
        }
        exit(EXIT_SUCCESS);
}

void
func(void){
        printf("in func, i_var is %d\n", i_var);
}
```

Example 8.6

The compiler might want to optimize the loop so that `i_var` can be stored in a machine register for speed. However, the function needs to have access to the correct value of `i_var` so that it can print the right value. This means that the register must be stored back into `i_var` at each function call (at least). When and where these conditions must occur are described by the Standard. At each sequence point, the side effects of all previous expressions will be completed. This is why you cannot rely on expressions such as:

```
a[i] = i++;
```

because there is no sequence point specified for the assignment, increment or index operators, you don't know when the effect of the increment on `i` occurs.

The sequence points laid down in the Standard are the following:

- The point of calling a function, after evaluating its arguments.
- The end of the first operand of the `&&` operator.
- The end of the first operand of the `||` operator.
- The end of the first operand of the `?:` conditional operator.
- The end of the each operand of the comma operator.
- Completing the evaluation of a full expression. They are the following:
    - Evaluating the initializer of an `auto` object.
    - The expression in an ‘ordinary’ statement—an expression followed by

semicolon.
- The controlling expressions in `do`, `while`, `if`, `switch` or `for` statements.
- The other two expressions in a for statement.
- The expression in a `return` statement.

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter8/const_and_volatile.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter8/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter8/summary.html*]

8.6. Summary

This is a chapter describing specialized areas of the language.

Undoubtedly, the issues of scope, linkage and duration are important. If you find the whole topic too much to digest, just learn the simple rules. The problem is that the Standard tries to be complete and unambiguous, so it has to lay down lots of rules. It's much easier if you just stick to the easy way of doing things and don't try to get too clever. Use Example 8.2 as a model if in doubt.

The use of `typedef` depends on your level of experience. Its most common use is to help avoid some of the more unpleasant aspects of complicated type declarations.

The use of `const` will be widespread in many programs. The idea of a pointer to something which is not modifiable is well and truly emphasized in the library function prototypes.

Only specialized applications will use `volatile`. If you work in the field of real-time programming, or embedded systems, this will matter to you. Otherwise it probably won't. The same goes for sequence points. How well the early compilers will support these last two features will be a very interesting question.

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter8/sequence_points.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter8/*]

Chapter 9

<gbdirect>

Libraries

Previous chapter [*http://publications.gbdirect.co.uk/c_book/chapter8/*] | Next chapter [*http://publications.gbdirect.co.uk/c_book/chapter10/*]

9.1. Introduction

There is no doubt that the Standard Committee's decision to define a set of library routines will prove to be a huge benefit to users of C. Previously there were *no* standard, accepted, definitions of library routines to provide support for the language. As a result, portability suffered seriously.

The library routines do not have to be present; they will only be present in a *hosted environment*—typically the case for applications programmers. Writers of embedded systems and the writers of the hosted environment libraries will not have the libraries present. They are using 'raw' C, in a *freestanding environment*, and this chapter will not be of much interest to them.

The descriptions (except for this introduction) are not meant to be read as a whole chapter, but as individual pieces. The material included here is meant more for information and convenient reference than as a full tutorial introduction. It would take a full book by itself to do real justice to the libraries.

9.1.1. Headers and standard types

A number of types and macros are used widely by the library functions. Where necessary, they are defined in the appropriate `#include` file for that function. The header will also declare appropriate types and prototypes for the library functions. Some important points should be noted here:

- All external identifiers and macro names declared in any of the library headers are reserved. They must not be used, or redefined, for any other purpose. In some cases they may be 'magic'—their names may be known to the compiler and cause it to use special methods to implement them.
- All identifiers that begin with an underscore are reserved.
- Headers may be included in any order, and more than once, but must be included outside of any external declaration or definition and before any use of the functions or macros defined inside them.
- Giving a 'bad value' to a function—say a null pointer, or a value outside the range of values expected by the function—results in undefined behaviour unless otherwise stated.

The Standard isn't quite as restrictive about identifiers as the list above is, but it's a brave move to make use of the loopholes. Play safe instead.

The Standard headers are:

```
<assert.h>   <locale.h>   <stddef.h>
<ctype.h>    <math.h>     <stdio.h>
<errno.h>    <setjmp.h>   <stdlib.h>
<float.h>    <signal.h>   <string.h>
<limits.h>   <stdarg.h>   <time.h>
```

A last general point is that many of the library routines may be implemented as macros, provided that there will be no problems to do with side-effects (as Chapter 7 [*http://publications.gbdirect.co.uk/c_book/chapter7/*] describes). The Standard guarantees that, if a function *is* normally implemented as a macro, there will also be a true function provided to do the same job. To use the real function, either undefine the macro name with `#undef`, or enclose its name in parentheses, which ensures that it won't be treated as a macro:

```
some function("Might be a macro\n");
(some function)("Can't be a macro\n");
```

9.1.2. Character set and cultural dependencies

The Committee has introduced features that attempt to cater for the use of C in environments which are not based on the character set of US ASCII and where there are cultural dependencies such as the use of comma or full stop to indicate the decimal point. Facilities have been provided (see Section 9.4 [*http://publications.gbdirect.co.uk/c_book/chapter9/localization.html*]) for setting a program's idea of its *locale*, which is used to control the behaviour of the library functions.

Providing full support for different native languages and customs is a difficult and poorly understood task; the facilities provided by the C library are only a first step on the road to a full solution.

In several places the ‘C locale’ is referred to. This is the only locale defined by the Standard and effectively provides support for the way that Old C worked. Other locale settings may provide different behaviour in implementation-defined ways.

9.1.3. The <stddef.h> Header

There are a small number of types and macros, found in `<stddef.h>`, which are widely used in other headers. They are described in the following paragraphs.

Subtracting one pointer from another gives a result whose type differs between different implementations. To allow safe use of the difference, the type is defined in `<stddef.h>` to be `ptrdiff_t`. Similarly, you can use `size_t` to store the result of `sizeof`.

For reasons which still escape us, there is an ‘implementation defined null pointer constant’ defined in `<stddef.h>` called `NULL`. Since the language explicitly defines the integer constant `0` to be the value which can be assigned to, and compared with, a null pointer, this would seem to be unnecessary. However, it is *very* common practice among experienced C programmers to write this sort of thing:

```
#include <stdio.h>
#include <stddef.h>
FILE *fp;

if((fp = fopen("somefile", "r")) != NULL){
        /* and so on */
```

There is also a macro called `offsetof` which can be used to find the offset, in bytes, of a structure member. The offset is the distance between the member and the start of the structure. It would be used like this:

```
#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>

main(){
        size_t distance;
        struct x{
                int a, b, c;
        }s_tr;

        distance = offsetof(s_tr, c);
        printf("Offset of x.c is %lu bytes\n",
                (unsigned long)distance);

        exit(EXIT_SUCCESS);
}
```

Example 9.1

The expression `s_tr.c` must be capable of evaluation as an address constant (see Chapter 6 [*http://publications.gbdirect.co.uk/c_book/chapter6/*]). If the member whose offset you want is a bitfield, then you're out of luck; offsetof has undefined behaviour in that case.

Note carefully the way that a `size_t` has to be cast to the longest possible unsigned type to ensure that not only is the argument to `printf` of the type that it expects (`%lu`is the format string for `unsigned long`), but also no precision is lost. This is all because the type of `size_t` is not known to the programmer.

The last item declared in `<stddef.h>` is `wchar_t`, an integral type large enough to hold a wide character from any supported extended character sets.

9.1.4. The <errno.h> Header

This header defines errno along with the macros `EDOM` and `ERANGE`, which expand to nonzero integral constant expressions; their form is additionally guaranteed to be acceptable to `#if` directives. The latter two are used by the mathematical functions to report which kind of errors they encountered and are more fully described later.

`errno` is provided to tell you when library functions have detected an error. It is not necessarily, as it used to be, an external variable, but is now a modifiable lvalue that has type `int`. It is set to zero at program start-up, but from then on never reset unless explicitly assigned to; in particular, the library routines never reset it. If an error occurs in a library routine, errno is set to a particular value to indicate what went wrong, and the routine returns a value (often −1) to indicate that it failed. The usual use is like this:

```
#include <stdio.h>
#include <stddef.h>
#include <errno.h>

errno = 0;
if(some_library_function(arguments) < 0){
        /* error processing code... */
        /* may use value of errno directly */
```

The implementation of `errno` is not known to the programmer, so don't try to do anything other than reset it or inspect its value. It isn't guaranteed to have an address, for example.

What's more, you should only check `errno` if the particular library function in use documents its effect on `errno`.

Other library functions are free to set it to arbitrary values after a call unless their description explicitly states what they do with it.

Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter9/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter9/diagnostics.html*]

9.2. Diagnostics

While you are debugging programs, it is often useful to check that the value of an expression is the one that you expected. The `assert` function provides such a diagnostic aid.

In order to use `assert` you must first include the header file `<assert.h>`. The function is defined as

```
#include <assert.h>

void assert(int expression)
```

If the expression evaluates to zero (i.e. false) then `assert` will write a message about the failing expression, including the name of the source file, the line at which the assertion was made and the expression itself. After this, the `abort` function is called, which will halt the program.

```
assert(1 == 2);

/* Might result in */

Assertion failed: 1 == 2, file silly.c, line 15
```

`Assert` is actually defined as a macro, not as a real function. In order to disable assertions when a program is found to work satisfactorily, defining the name `NDEBUG` *before* including `<assert.h>` will disable assertions totally. You should beware of side effects that the expression may have: when assertions are turned off with `NDEBUG`, the expression is *not* evaluated. Thus the following example will behave unexpectedly when debugging is turned off with the `#define NDEBUG`.

```
#define NDEBUG
#include <assert.h>

void
func(void)
{
        int c;
        assert((c = getchar()) != EOF);
        putchar(c);
}
```

Example 9.2

Note that assert returns no value.

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter9/introduction.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter9/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter9/character_handling.html*]

9.3. Character handling

<gbdirect>

There are a variety of functions provided for testing and mapping characters. The testing functions, which are described first, allow you to test if a character is of a particular type, such as alphabetic, upper or lower case, numeric, a control character, a punctuation mark, printable or not and so on. The character testing functions return an integer, either zero if the character supplied is not of the category specified, or non-zero if it was. The functions all take an integer argument, which should either be an int, the value of which should be representable as `unsigned char`, or the integer constant `EOF`, as returned from functions such as `getchar()`. The behaviour is undefined if it is not.

These functions depend on the program's locale setting.

A *printing character* is a member of an implementation defined character set. Each printing character occupies one printing position. A *control character* is a member of an implementation defined character set, each of which is not a printing character. If the 7-bit ASCII character set is used, the printing characters are those that lie between space `(0x20)` and tilde `(0x7e)`, the control characters are those between `NUL (0x0)` and `US (0x1f)`, and the character `DEL (0x7f)`.

The following is a summary of all the character testing functions. The header `<ctype.h>` must be included before any of them is used.

`isalnum(int c)`
: True if `c` is alphabetic or a digit; specifically `(isalpha(c)||isdigit(c))`.

`isalpha(int c)`
: True if `(isupper(c)||islower(c))`.

  Also true for an implementation-defined set of characters which do not return true results from any of iscntrl, isdigit, ispunct or isspace. In the C locale, this extra set of characters is empty.

`iscntrl(int c)`
: True if `c` is a control character.

`isdigit(int c)`
: True if `c` is a decimal digit.

`isgraph(int c)`
: True if `c` is any printing character except space.

`islower(int c)`
: True if `c` is a lower case alphabetic letter. Also true for an implementation defined set of characters which do not return true results from any of `iscntrl`, `isdigit`, `ispunct` or `isspace`. In the C locale, this extra set of characters is empty.

`isprint(int c)`
: True if `c` is a printing character (including space).

`ispunct(int c)`
: True if `c` is any printing character that is neither a space nor a character which would return true from `isalnum`.

`isspace(int c)`

True if `c` is either a white space character (one of `' ' '\f' '\n' '\r' '\t' '\v'`) or, in other than the C locale, characters which would not return true from `isalnum`

`isupper(int c)`

True if `c` is an upper case alphabetic character.

Also true for an implementation-defined set of characters which do not return true results from any of `iscntrl`, `isdigit`, `ispunct` or `isspace`. In the C locale, this extra set of characters is empty.

`isxdigit(int c)`

True if `c` is a valid hexadecimal digit.

Two additional functions map characters from one set into another. The function `tolower` will, if given a upper case character as its argument, return the lower case equivalent. For example,

```
tolower('A') == 'a'
```

If `tolower` is given any character other than an upper case letter, it will return that character.

The converse function `toupper` maps lower case alphabetic letters onto their upper case equivalent.

For each, the conversion is only performed if there *is* a corresponding character in the alternate case. In some locales, not all upper case characters have lower case equivalents, and vice versa.

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter9/diagnostics.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter9/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter9/localization.html*]

9.4. Localization

This is where the program's idea of its current locale can be controlled. The header file `<locale.h>` declares the setlocale and localeconv functions and a number of macros:

```
LC_ALL
LC_COLLATE
LC_CTYPE
LC_MONETARY
LC_NUMERIC
LC_TIME
```

all of which expand to integral constant expressions and are used as values of the `category` argument to `setlocale` (other names may also be defined: they will all start with `LC_X` where `X` is an upper case letter), and the type

```
struct lconv
```

which is used for storing information about the formatting of numeric values. For members of type `char`, `CHAR_MAX` is used to indicate that the value is not available in the current locale.

`lconv` contains at least the following members:

`char *decimal_point`
: The character used for the decimal point in formatted non-monetary values. `"."` in the C locale.

`char *thousands_sep`
: The character used for separating groups of digits to the left of the decimal point in formatted non-monetary values. `""` in the C locale.

`char *grouping`
: Defines the number of digits in each group when formatting non-monetary values. The elements are interpreted as follows: A value of `CHAR_MAX` indicates that no further grouping is to be performed; `0` indicates that the previous element should be repeated for the remaining digits; if any other character is used, its integer value represents the number of digits that comprise the current group (the next character in the sequence is interpreted before grouping). `""` in the C locale. As an example, `"\3"` specifies that digits should be grouped in threes; the terminating null in the string signifies that the `\3` repeats.

`char *int_curr_symbol`
: The first three characters are used to hold the alphabetic international currency symbol for the current locale, the fourth character is used to separate the international currency symbol from the monetary quantity. `""` in the C locale.

`char *currency_symbol`
: The currency symbol for the current locale. `""` in the C locale.

`char *mon_decimal_point`
: The character used as the decimal point when formatting monetary values. `""` in the C locale.

`char *mon_thousands_sep`
: The digit group separator for formatted monetary values. `""` in the C locale.

`char *mon_grouping`
: Defines the number of digits in each group when formatting monetary values. Its elements are interpreted as those for grouping. `""` in the C locale.

`char *positive_sign`
: The string used to signify a non-negative monetary value. `""` in the C locale.

`char *negative_sign`
: The string used to signify a negative monetary value. `""` in the C locale.

`char int_frac_digits`
: The number of digits to be displayed after the decimal point in an internationally formatted monetary value. `CHAR_MAX` in the C locale.

`char frac_digits`
: The number of digits to be displayed after the decimal point in a non-internationally formatted monetary value. `CHAR_MAX` in the C locale.

`char p_cs_precedes`
: A value of `1` indicates that the `currency_symbol` should precede the value when formatting a non-negative monetary quantity; a value of `0` indicates that it should follow. `CHAR_MAX` in the C locale.

`char p_sep_by_space`
: A value of 1 indicates that the currency symbol is separated by a space from the value when formatting a non-negative monetary quantity; 0 indicates no space. CHAR_MAX in the C locale.

`char n_cs_precedes`
: As `p_cs_precedes` for negative monetary values. `CHAR_MAX` in the C locale.

`char n_sep_by_space`
: As `p_sep_by_space` for negative monetary values. `CHAR_MAX` in the C locale.

`char p_sign_posn`

Indicates the position of the `positive_sign` for a non-negative formatted monetary value according to the following:

- parentheses surround quantity and `currency_symbol`
- the string precedes the quantity and `currency_symbol`
- the string follows the quantity and `currency_symbol`
- the string precedes the `currency_symbol`
- the string follows the `currency_symbol`

`CHAR_MAX` in the C locale.

`char n_sign_posn`
: As `p_sign_posn` for negative monetary values. `CHAR_MAX` in the C locale.

9.4.1. The setlocale function

```
#include <locale.h>

char *setlocale(int category, const char *locale);
```

This function allows the program's idea of its locale to be set. All or parts of the locale can be set by providing values for `category` as follows:

`LC_ALL`
: Set entire locale.

`LC_COLLATE`
: Modify behaviour of `strcoll` and `strxfrm`.

`LC_CTYPE`
: Modify behaviour of character-handling functions.

`LC_MONETARY`
    Modify monetary formatting information returned by localeconv.
`LC_NUMERIC`
    Modify decimal-point character for formatted I/O and string conversion routines.
`LC_TIME`

    Modify behaviour of `strftime`.

    The values for locale can be:

`"C"`	Select the minimal environment for C translation
`""`	Select the implementation-defined ‘native environment’
implementation defined	Select other environments

When the program starts, it has an environment as if

```
setlocale(LC_ALL, "C");
```

has been executed.

The current string associated with a given category can be queried by passing a null pointer as the value for `locale`; if the selection can be performed, the string associated with the specified `category` for the new locale is returned. This string is such that if it is used in a subsequent call to `setlocale`, along with its associated category, that part of the program's locale will be restored. If the selection cannot be performed, a null pointer is returned and the locale is not changed.

9.4.2. The localeconv function

```
#include <locale.h>

struct lconv *localeconv(void);
```

The function returns a pointer to a structure of type `struct lconv`, set according to the current locale, which may be overwritten by subsequent calls to `localeconv` or `setlocale`. The structure must not be modified in any other way.

For example, if in the current locale monetary values should be represented as

`IR£1,234.56`	positive format
`(IR£1,234.56)`	negative format
`IRP 1,234.56`	international format

then the monetary members of `lconv` would have the values:

`int_curr_symbol`	`"IRP "`
`currency_symbol`	`"IR£"`
`mon_decimal_point`	`"."`
`mon_thousands_sep`	`","`
`mon_grouping`	`"\3"`
`postive_sign`	`""`
`negative_sign`	`""`
`int_frac_digits`	`2`

`frac_digits`	`2`
`p_cs_precedes`	`1`
`p_sep_by_space`	`0`
`n_cs_precedes`	`1`
`n_sep_by_space`	`0`
`p_sign_posn`	`CHAR_MAX`
`n_sign_posn`	`0`

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter9/character_handling.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter9/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter9/limits.html*]

9.5. Limits

<gbdirect>

Two header files `<float.h>` and `<limits.h>` define several implementation specific limits.

9.5.1. Limits.h

Table 9.1 gives the names declared, the allowable values, and a comment on what they mean. For example, the description of `SHRT_MIN` shows that in a given implementation the value must be less than or equal to −32767: this means that for maximum portability a program cannot rely on short variables being able to hold values more negative than −32767. Implementations may choose to support values which are more negative but must provide support for at least −32767.

Name	Allowable value	Comment
`CHAR_BIT`	(≥8)	bits in a `char`
`CHAR_MAX`	see note	max value of a `char`
`CHAR_MIN`	see note	min value of a `char`
`INT_MAX`	(≥+32767)	max value of an `int`
`INT_MIN`	(≤−32767)	min value of an `int`
`LONG_MAX`	(≥+2147483647)	max value of a `long`
`LONG_MIN`	(≤−2147483647)	min value of a `long`
`MB_LEN_MAX`	(≥1)	max number of bytes in a multibyte character
`SCHAR_MAX`	(≥+127)	max value of a `signed char`
`SCHAR_MIN`	(≤−127)	min value of a `signed char`
`SHRT_MAX`	(≥+32767)	max value of a `short`
`SHRT_MIN`	(≤−32767)	min value of a `short`
`UCHAR_MAX`	(≥255U)	max value of an `unsigned char`
`UINT_MAX`	(≥65535U)	max value of an `unsigned int`
`ULONG_MAX`	(≥4294967295U)	max value of an `unsigned long`
`USHRT_MAX`	(≥65535U)	max value of an `unsigned short`

Note: if the implementation treats `chars` as signed, then the values of `CHAR_MAX` and `CHAR_MIN` are the same as the equivalent `SCHAR` versions. If not, then the value of `CHAR_MIN` is zero and the value of `CHAR_MAX` is equal to the value of `UCHAR_MAX`.

Table 9.1. `<limits.h>`

9.5.2. Float.h

For floating point numbers, the file `<float.h>` contains a similar set of minimum values. (It is assumed that where no minimum value is specified, there is either no minimum, or the value depends on another value.)

Name	Allowable value	Comment
`FLT_RADIX`	(≥2)	the radix of exponent representation
`DBL_DIG`	(≥10)	the number of digits of precision in a `double`
`DBL_EPSILON`	(≤1E−9)	minimum positive number such that 1.0 + *x* ≠ 1.0
`DBL_MANT_DIG`	(—)	the number of base `FLT_RADIX` digits in the mantissa part of a `double`
`DBL_MAX`	(≥1E+37)	max value of a `double`
`DBL_MAX_10_EXP`	(≥+37)	max value of exponent (base 10) of a `double`
`DBL_MAX_EXP`	(—)	max value of exponent (base `FLT_RADIX`)) of a `double`
`DBL_MIN`	(≤1E−37)	min value of a `double`
`DBL_MIN_10_EXP`	(≤37)	minimum value of exponent (base 10) of a `double`
`DBL_MIN_EXP`	(—)	min value of exponent part of a `double` (base `FLT_RADIX`)
`FLT_DIG`	(≥6)	the number of digits of precision in a `float`
`FLT_EPSILON`	(≤1E−5)	minimum positive number such that 1.0 + x ≠ 1.0
`FLT_MANT_DIG`	(—)	the number of base `FLT_RADIX` digits in the mantissa of a `float`
`FLT_MAX`	(≥1E+37)	max value of a `float`
`FLT_MAX_10_EXP`	(≥+37)	max value (base 10) of exponent part of a `float`
`FLT_MAX_EXP`	(—)	max value (base `FLT_RADIX`) of exponent part of a `float`
`FLT_MIN`	(≤1E−37)	min value of a `float`
`FLT_MIN_10_EXP`	(≤−37)	min value (base 10) of exponent part of a `float`
`FLT_MIN_EXP`	(—)	min value (base `FLT_RADIX`) of exponent part of a `float`
`FLT_ROUNDS`	(0)	affects rounding of floating point addition: −1 indeterminate 0 towards zero 1 to nearest 2 towards +infinity 3 towards -infinity any other value is implementation defined.
`LDBL_DIG`	(≥10)	the number of digits of precision in a `long double`
`LDBL_EPSILON`	(≤1E−9)	minimum positive number such that 1.0 + x ≠= 1.0
`LDBL_MANT_DIG`	(—)	the number of base `FLT_RADIX` digits in the mantissa part of a `long double`

Name	Allowable value	Comment
`LDBL_MAX`	(≥1E+37)	max value of a `long double`
`LDBL_MAX_10_EXP`	(≥+37)	max value of exponent (base 10) of a `long double`
`LDBL_MAX_EXP`	(—)	max value of exponent (base `FLT_RADIX`) of a `long double`
`LDBL_MIN`	(≤1E−37)	minimum value of a `long double`
`LDBL_MIN_10_EXP`	(≤−37)	min value of exponent part (base 10) of a `long double`
`LDBL_MIN_EXP`	(—)	min value of exponent part of a `long double` (base `FLT_RADIX`)

Table 9.2. `<float.h>`

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter9/localization.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter9/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter9/maths_functions.html*]

9.6. Mathematical functions

If you are writing mathematical programs, involving floating point calculations and so on, then you will undoubtedly require access to the mathematics library. This set of functions all take `double` arguments, and return a double result. The functions and associated macros are defined in the include file `<math.h>`.

The macro `HUGE_VAL` is defined, which expands to a positive double expression, which is not necessarily representable as a `float`.

For all the functions, a *domain error* occurs if an input argument is outside the domain over which the function is defined. An example might be attempting to take the square root of a negative number. If this occurs, `errno` is set to the constant `EDOM`, and the function returns an implementation defined value.

If the result of the function cannot be represented as a double value then a *range error* occurs. If the magnitude of the result is too large, the functions return `±HUGE_VAL` (the sign will be correct) and `errno` is set to `ERANGE`. If the result is too small, `0.0` is returned and the value of `errno` is implementation defined.

The following list briefly describes each of the functions available:

`double acos(double x);`
: Principal value of the arc cosine of *x* in the range 0–π radians.
Errors: `EDOM` if *x* is not in the range −1–1.

`double asin(double x);`
: Principal value of the arc sine of *x* in the range -π/2–+π/2 radians.
Errors: `EDOM` if *x* is not in the range `-1-1`.

`double atan(double x);`
: Principal value of the arc tangent of *x* in the range -π/2–+π/2 radians.

`double atan2(double y, double x);`
: Principal value of the arc tangent of *y/x* in the range -π–+π radians, using the signs of both arguments to determine the quadrant of the return value.
Errors: `EDOM` may occur if both *x* and *y* are zero.

`double cos(double x);`
: Cosine of *x* (*x* measured in radians).

`double sin(double x);`
: Sine of *x* (*x* measured in radians).

`double tan(double x);`
: Tangent of *x* (*x* measured in radians). When a range error occurs, the sign of the resulting `HUGE_VAL` is not guaranteed to be correct.

`double cosh(double x);`
: Hyperbolic cosine of *x*.
Errors: `ERANGE` occurs if the magnitude of *x* is too large.

`double sinh(double x);`
: Hyperbolic sine of *x*.
Errors: `ERANGE` occurs if the magnitude of x is too large.

`double tanh(double x);`
: Hyperbolic tangent of `x`.

`double exp(double x);`
: Exponential function of *x*. Errors: `ERANGE` occurs if the magnitude of *x* is too

large.

`double frexp(double value, int *exp);`
: Break a floating point number into a normalized fraction and an integral power of two. This integer is stored in the object pointed to by *exp*.

`double ldexp(double x, int exp);`
: Multiply *x* by 2 to the power *exp*
: Errors: `ERANGE` may occur.

`double log(double x);`
: Natural logarithm of *x*.
: Errors: `EDOM` occurs if *x* is negative. `ERANGE` may occur if *x* is zero.

`double log10(double x);`
: Base-ten logarithm of *x*.
: Errors: `EDOM` occurs if *x* is negative. `ERANGE` may occur if *x* is zero.

`double modf(double value, double *iptr);`
: Break the argument value into integral and fractional parts, each of which has the same sign as the argument. It stores the integrbal part as a `double` in the object pointed to by *iptr*, and returns the fractional part.

`double pow(double x, double y);`
: Compute *x* to the power *y*.
: Errors: `EDOM` occurs if $x < 0$ and *y* not integral, or if the result cannot be represented if *x* is 0, and $y \leq 0$. `ERANGE` may also occur.

`double sqrt(double x);`
: Compute the square root of *x*.
: Errors: `EDOM` occurs if *x* is negative.

`double ceil(double x);`
: Smallest integer not less than *x*.

`double fabs(double x);`
: Absolute value of *x*.

`double floor(double x);`
: Largest integer not greater than *x*.

`double fmod(double x, double y);`
: Floating point remainder of *x/y*.
: Errors: If *y* is zero, it is implementation defined whether `fmod` returns zero or a domain error occurs.

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter9/limits.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter9/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter9/nonlocal_jumps.html*]

9.7. Non-local jumps

<gbdirect>

Provision is made for you to perform what is, in effect, a `goto` from one function to another. It isn't possible to do this by means of a `goto` and a label, since labels have only function scope. However, the macro `setjmp` and function `longjmp` provide an alternative, known as a *non-local goto*, or a *non-local jump*.

The file `<setjmp.h>` declares something called a `jmp_buf`, which is used by the cooperating macro and function to store the information necessary to make the jump. The declarations are as follows:

```
#include <setjmp.h>

int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
```

The `setjmp` macro is used to initialise the `jmp_buf` and returns zero on its initial call. The bizarre thing is that it returns *again*, later, with a non-zero value, when the corresponding `longjmp` call is made! The non-zero value is whatever value was supplied to the call of `longjmp`. This is best explained by way of an example:

```
#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>

void func(void);
jmp_buf place;

main(){
        int retval;

        /*
         * First call returns 0,
         * a later longjmp will return non-zero.
         */
        if(setjmp(place) != 0){
                printf("Returned using longjmp\n");
                exit(EXIT_SUCCESS);
        }

        /*
         * This call will never return - it
         * 'jumps' back above.
         */
        func();
        printf("What! func returned!\n");
}

void
func(void){
        /*
         * Return to main.
         * Looks like a second return from setjmp,
         * returning 4!
         */
        longjmp(place, 4);
        printf("What! longjmp returned!\n");
}
```

Example 9.3

The `val` argument to `longjmp` is the value seen in the second and subsequent ‘returns’ from `setjmp`. It should normally be something other than 0; if you attempt to return 0 via `longjmp`, it will be changed to 1. It is therefore possible to tell whether the `setjmp` was called directly, or whether it was reached by calling `longjmp`.

If there has been no call to `setjmp` before calling `longjmp`, the effect of `longjmp` is undefined, almost certainly causing the program to crash. The `longjmp` function is never expected to return, in the normal sense, to the instructions immediately following the call. All accessible objects on ‘return’ from `setjmp` have the values that they had when `longjmp` was called, except for objects of automatic storage class that do not have volatile type; if they have been changed between the `setjmp` and `longjmp` calls, their values are indeterminate.

The `longjmp` function executes correctly in the contexts of interrupts, signals and any of their associated functions. If `longjmp` is invoked from a function called as a result of a signal arriving while handling another signal, the behaviour is undefined.

It's a serious error to `longjmp` to a function which is no longer active (i.e. it has already returned or another `longjump` call has transferred to a `setjmp` occurring earlier in a set of nested calls).

The Standard insists that, apart from appearing as the only expression in an expression statement, `setjmp` may only be used as the entire controlling expression in an `if`, `switch`, `do`, `while`, or `for` statement. A slight extension to that rule is that as long as it is the whole controlling expression (as above) the `setjmp` call may be the subject of the `!` operator, or may be directly compared with an integral constant expression using one of the relational or equality operators. No more complex expressions may be employed. Examples are:

```
setjmp(place);                  /* expression statement */
if(setjmp(place)) ...           /* whole controlling expression */
if(!setjmp(place)) ...          /* whole controlling expression */
if(setjmp(place) < 4) ...       /* whole controlling expression */
if(setjmp(place)<;4 && 1!=2) ...  /* forbidden */
```

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter9/maths_functions.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter9/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter9/signal_handling.html*]

9.8. Signal handling

Two functions allow for asynchronous event handling to be provided. A *signal* is a condition that may be reported during program execution, and can be ignored, handled specially, or, as is the default, used to terminate the program. One function sends signals, another is used to determine how a signal will be processed. Many of the signals may be generated by the underlying hardware or operating system as well as by means of the signal-sending function `raise`.

The signals are defined in the include file `<signal.h>`.

`SIGABRT`
: Abnormal termination, such as instigated by the `abort` function. (Abort.)

`SIGFPE`
: Erroneous arithmetic operation, such as divide by 0 or overflow. (Floating point exception.)

`SIGILL`
: An ‘invalid object program’ has been detected. This usually means that there is an illegal instruction in the program. (Illegal instruction.)

`SIGINT`
: Interactive attention signal; on interactive systems this is usually generated by typing some ‘break-in’ key at the terminal. (Interrupt.)

`SIGSEGV`
: Invalid storage access; most frequently caused by attempting to store some value in an object pointed to by a bad pointer. (Segment violation.)

`SIGTERM`
: Termination request made to the program. (Terminate.)

Some implementations may have additional signals available, over and above this standard set. They will be given names that start `SIG`, and will have unique values, apart from the set above.

The function `signal` allows you to specify the action taken on receipt of a signal. Associated with each signal condition above, there is a pointer to a function provided to handle this signal. The signal function changes this pointer, and returns the original value. Thus the function is defined as

```
#include <signal.h>
void (*signal (int sig, void (*func)(int)))(int);
```

That is to say, `signal` is a function that returns a pointer to another function. This second function takes a single int argument and returns `void`. The second argument to `signal` is similarly a pointer to a function returning `void` which takes an `int` argument.

Two special values may be used as the `func` argument (the signal-handling function), `SIG_DFL`, the initial, default, signal handler; and `SIG_IGN`, which is used to ignore a signal. The implementation sets the state of all signals to one or other of these values at the start of the program.

If the call to `signal` succeeds, the previous value of `func` for the specified signal is returned. Otherwise, `SIG_ERR` is returned and `errno` is set.

When a signal event happens which is not being ignored, if the associated func is a pointer to a function, first the equivalent of `signal(sig, SIG_DFL)` is executed. This resets the signal handler to the default action, which is to terminate the program. If the signal was `SIGILL` then this resetting is implementation defined. Implementations may choose to 'block' further instances of the signal instead of doing the resetting.

Next, a call is made to the signal-handling function. If that function returns normally, then under most circumstances the program will resume at the point where the event occurred. However, if the value of `sig` was `SIGFPE` (a floating point exception), or any implementation defined computational exception, then the behaviour is undefined. The most usual thing to do in the handler for `SIGFPE` is to call one of the functions `abort`, `exit`, or `longjmp`.

The following program fragment shows the use of signal to perform a tidy exit to a program on receipt of the interrupt or 'interactive attention' signal.

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>


FILE *temp_file;
void leave(int sig);

main() {
        (void) signal(SIGINT,leave);
        temp_file = fopen("tmp","w");
        for(;;) {
                /*
                 * Do things....
                 */
                printf("Ready...\n");
                (void)getchar();
        }
        /* can't get here ... */
        exit(EXIT_SUCCESS);
}

/*
 * on receipt of SIGINT, close tmp file
 * but beware - calling library functions from a
 * signal handler is not guaranteed to work in all
 * implementations.....
 * this is not a strictly conforming program
 */

void
leave(int sig) {
        fprintf(temp_file,"\nInterrupted..\n");
        fclose(temp_file);
        exit(sig);
}
```

Example 9.4

It is possible for a program to send signals to itself by means of the `raise` function. This is defined as follows

```
include <signal.h>
int raise (int sig);
```

The signal sig is sent to the program.

`Raise` returns zero if successful, non-zero otherwise. The `abort` library function is essentially implementable as follows:

```
#include <signal.h>

void
abort(void) {
  raise(SIGABRT);
}
```

If a signal occurs for any reason other than calling abort or raise, the signal-handling function may only call signal or assign a value to a volatile static object of type `sig_atomic_t`. The type `sig_atomic_t` is declared in `<signal.h>`. It is the only type of object that can safely be modified as an atomic entity, even in the presence of asynchronous interrupts. This is a very onerous restriction imposed by the Standard, which, for example, invalidates the `leave` function in the example program above; although the function would work correctly in some environments, it does not follow the strict rules of the Standard.

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter9/nonlocal_jumps.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter9/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter9/stdarg.html*]

9.9. Variable numbers of arguments

It is often desirable to implement a function where the number of arguments is not known, or is not constant, when the function is written. Such a function is `printf`, described in Section 9.11 [*http://publications.gbdirect.co.uk/c_book/chapter9/formatted_io.html*]. The following example shows the declaration of such a function.

```
int f(int, ... );

int f(int, ... ) {
        .
        .
        .
}

int g() {
        f(1,2,3);
}
```

Example 9.5

In order to access the arguments within the called function, the functions declared in the `<stdarg.h>` header file must be included. This introduces a new type, called a `va_list`, and three functions that operate on objects of this type, called `va_start`, `va_arg`, and `va_end`.

Before any attempt can be made to access a variable argument list, `va_start` must be called. It is defined as

```
#include <stdarg.h>
void vastart(valist ap, parmN);
```

The `va_start` macro initializes `ap` for subsequent use by the functions `va_arg` and `va_end`. The second argument to `va_start`, *parmN* is the identifier naming the rightmost parameter in the variable parameter list in the function definition (the one just before the , ...). The identifier *parmN* must not be declared with `register` storage class or as a function or array type.

Once initialized, the arguments supplied can be accessed sequentially by means of the va arg macro. This is peculiar because the type returned is determined by an argument to the macro. Note that this is impossible to implement as a true function, only as a macro. It is defined as

```
#include <stdarg.h>
type va arg(va list ap, type);
```

Each call to this macro will extract the next argument from the argument list as a value of the specified type. The `va_list` argument must be the one initialized by `va_start`. If the next argument is not of the specified type, the behaviour is undefined. Take care here to avoid problems which could be caused by arithmetic conversions. Use of `char` or short as the second argument to `va_arg` is invariably an error: these types always promote up to one of `signed int` or `unsigned int`, and `float` converts to `double`. Note that it is implementation defined whether objects declared to have the types `char`, `unsigned char`, `unsigned short` and unsigned bitfields will promote to `unsigned int`, rather complicating the use of `va_arg`. This may be an area where some unexpected subtleties arise; only time will tell.

The behaviour is also undefined if `va_arg` is called when there were no further arguments.

The *type* argument must be a type name which can be converted into a pointer to such an object simply by appending a `*` to it (this is so the macro can work). Simple types such as `char` are fine (because `char *` is a pointer to a character) but array of char won't work (`char []` does not turn into 'pointer to array of char' by appending a `*`). Fortunately, arrays can easily be processed by remembering that an array name used as an actual argument to a function call is converted into a pointer. The correct *type* for an argument of type 'array of char' would be `char *`.

When all the arguments have been processed, the `va_end` function should be called. This will prevent the `va_list` supplied from being used any further. If va end is not used, the behaviour is undefined.

The entire argument list can be re-traversed by calling `va_start` again, after calling `va_end`. The `va_end` function is declared as

```
#include <stdarg.h>
void va_end(va list ap);
```

The following example shows the use of `va_start`, `va_arg`, and `va_end` to implement a function that returns the biggest of its integer arguments.

```
#include <stdlib.h>
#include <stdarg.h>
#include <stdio.h>

int maxof(int, ...) ;
void f(void);

main(){
        f();
        exit(EXIT SUCCESS);
}

int maxof(int n args, ...){
        register int i;
        int max, a;
        va_list ap;

        va_start(ap, n args);
        max = va_arg(ap, int);
        for(i = 2; i <= n_args; i++) {
                if((a = va_arg(ap, int)) > max)
                        max = a;
        }

        va_end(ap);
        return max;
}
```

```
void f(void) {
        int i = 5;
        int j[256];
        j[42] = 24;
        printf("%d\n",maxof(3, i, j[42], 0));
}
```

Example 9.6

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter9/signal_handling.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter9/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter9/input_and_output.html*]

9.10. Input and output

9.10.1. Introduction

One of the reasons that has prevented many programming languages from becoming widely used for 'real programming' is their poor support for I/O, a subject which has never seemed to excite language designers. C has avoided this problem, oddly enough, by having no I/O at all! The C language approach has always been to do I/O using library functions, which ensures that system designers can provide tailored I/O instead of being forced to change the language itself.

As C has evolved, a library package known as the 'Standard I/O Library' or stdio, has evolved with it and has proved to be both flexible and portable. This package has now become part of the Standard.

The old stdio package relied heavily on the UNIX model of file access, in particular the assumption that there is no distinction between unstructured binary files and files containing readable text. Many operating systems do maintain a distinction between the two, and to ensure that C programs can be written portably to run on both types of file model, the stdio package has been modified. There are changes in this area which affect many existing programs, although strenuous efforts were taken to limit the amount of damage.

Old C programs should still be able work unmodified in a UNIX environment.

9.10.2. The I/O model

The I/O model does not distinguish between the types of physical devices supporting the I/O. Each source or sink of data (file) is treated in the same way, and is viewed as a *stream* of bytes. Since the smallest object that can be represented in C is the character, access to a file is permitted at any character boundary. Any number of characters can be read or written from a movable point, known as the *file position indicator*. The characters will be read, or written, in sequence from this point, and the position indicator moved accordingly. The position indicator is initially set to the beginning of a file when it is opened, but can also be moved by means of positioning requests. (Where random access is not possible, the file position indicator is ignored.) Opening a file in append mode has an implementation defined effect on the stream's file position indicator.

The overall effect is to provide sequential reads or writes unless the stream was opened in append mode, or the file position indicator is explicitly moved.

There are two types of file, *text files* and *binary files*, which, within a program, are manipulated as *text streams* and *binary* streams once they have been opened for I/O. The stdio package does not permit operations on the contents of files 'directly', but only by viewing them as streams.

9.10.2.1. Text streams

The Standard specifies what is meant by the term *text stream*, which essentially considers a file to contain lines of text. A line is a sequence of zero or more characters terminated by a newline character. It is quite possible that the actual representation of lines in the external environment is different from this and there may be transformations of the data stream on the way in and out of the program; a common requirement is to translate the '`\n`' line-terminator into the sequence '`\r\n`' on output, and do the reverse on input. Other translations may also be necessary.

Data read in from a text stream is guaranteed to compare equal to the data that was earlier written out to the file if the data consists only of complete lines of printable characters and the control characters horizontal-tab and newline, no newline character is immediately preceded by space characters and the last character is a newline.

It is guaranteed that, if the last character written to a text file is a newline, it will read back as the same.

It is implementation defined whether the last line written to a text file must terminate with a newline character; this is because on some implementations text files and binary files are the same.

Some implementations may strip the leading space from lines consisting only of a space followed by a newline, or strip trailing spaces at the end of a line!

An implementation must support text files with lines containing at least 254 characters, including the terminating newline.

Opening a text stream in update mode may result in a binary stream in some implementations.

Writing on a text stream may cause some implementations to truncate the file at that point—any data beyond the last byte of the current write being discarded.

9.10.2.2. Binary streams

A binary stream is a sequence of characters that can be used to record a program's internal data, such as the contents of structures or arrays in binary form. Data read in from a binary stream will always compare equal to data written out earlier to the same stream, under the same implementation. In some circumstances, an implementation-defined number of `NUL` characters may be appended to a binary stream.

The contents of binary files are exceedingly machine specific, and not, in general, portable.

9.10.2.3. Other streams

Other stream types may exist, but are implementation defined.

9.10.3. The stdio.h header file

To provide support for streams of the various kinds, a number of functions and macros exist. The `<stdio.h>` header file contains the various declarations necessary for the functions, together with the following macro and type declarations:

`FILE`
: The type of an object used to contain stream control information. Users of stdio never need to know the contents of these objects, but simply manipulate pointers to them. It is not safe to copy these objects within the program; sometimes their addresses may be 'magic'.

`fpos_t`
: A type of object that can be used to record unique values of a stream's file

position indicator.

`_IOFBF _IOLBF _IONBF`
: Values used to control the buffering of a stream in conjunction with the `setvbuf` function.

`BUFSIZ`
: The size of the buffer used by the `setbuf` function. An integral constant expression whose value is at least 256.

`EOF`
: A negative integral constant expression, indicating the end-of-file condition on a stream i.e. that there is no more input.

`FILENAME_MAX`
: The maximum length which a filename can have, if there is a limit, or otherwise the recommended size of an array intended to hold a file name.

`FOPEN_MAX`
: The minimum number of files that the implementation guarantees may be held open concurrently; at least eight are guaranteed. Note that three predefined streams exist and may need to be closed if a program needs to open more than five files explicitly.

`L_tmpnam`
: The maximum length of the string generated by `tmpnam`; an integral constant expression.

`SEEK_CUR SEEK_END SEEK_SET`
: Integral constant expressions used to control the actions of `fseek`.

`TMP_MAX`
: The minimum number of unique filenames generated by `tmpnam`; an integral constant expression with a value of at least 25.

`stdin stdout stderr`
: Predefined objects of type (`FILE *`) referring to the standard input, output and error streams respectively. These streams are automatically open when a program starts execution.

9.10.4. Opening, closing and buffering of streams

9.10.4.1. Opening

A stream is connected to a file by means of the `fopen`, `freopen` or `tmpfile` functions. These functions will, if successful, return a pointer to a `FILE` object.

Three streams are available without any special action; they are normally all connected to the physical device associated with the executing program: usually your terminal. They are referred to by the names `stdin`, the *standard input*, `stdout`, the *standard output*, and `stderr`, the *standard error* streams. Normal keyboard input is from `stdin`, normal terminal output is to `stdout`, and error messages are directed to `stderr`. The separation of error messages from normal output messages allows the stdout stream to be connected to something other than the terminal device, and still to have error messages appear on the screen in front of you, rather than to be redirected to this file. These files are only fully buffered if they do not refer to interactive devices.

As mentioned earlier, the file position indicator may or may not be movable, depending on the underlying device. It is not possible, for example, to move the file position indicator on stdin if that is connected to a terminal, as it usually is.

All non-temporary files must have a *filename*, which is a string. The rules for what constitutes valid filenames are implementation defined. Whether a file can be simultaneously open multiple times is also implementation defined. Opening a new file may involve creating the file. Creating an existing file causes its previous contents to be discarded.

9.10.4.2. Closing

Files are closed by explicitly calling `fclose`, `exit` or by returning from `main`. Any buffered data is flushed. If a program stops for some other reason, the status of files which it had open is undefined.

9.10.4.3. Buffering

There are three types of buffering:

Unbuffered
: Minimum internal storage is used by stdio in an attempt to send or receive data as soon as possible.

Line buffered
: Characters are processed on a line-by-line basis. This is commonly used in interactive environments, and internal buffers are flushed only when full or when a newline is processed.

Fully buffered
: Internal buffers are only flushed when full.

The buffering associated with a stream can always be flushed by using `fflush` explicitly. Support for the various types of buffering is implementation defined, and can be controlled within these limits using `setbuf` and `setvbuf`.

9.10.5. Direct file manipulation

A number of functions exist to operate on files directly.

```
#include <stdio.h>

int remove(const char *filename);
int rename(const char *old, const char *new);
char *tmpnam(char *s);
FILE *tmpfile(void);
```

`remove`
: Causes a file to be removed. Subsequent attempts to open the file will fail, unless it is first created again. If the file is already open, the operation of `remove` is implementation defined. The return value is zero for success, any other value for failure.

`rename`
: Changes the name of the file identified by `old` to `new`. Subsequent attempts to open the original name will fail, unless another file is created with the old name. As with `remove`, `rename` returns zero for a successful operation, any other value indicating a failure.

  If a file with the new name exists prior to calling `rename`, the behaviour is implementation defined.

  If `rename` fails for any reason, the original file is unaffected.

`tmpnam`
: Generates a string that may be used as a filename and is guaranteed to be different from any existing filename. It may be called repeatedly, each time generating a new name. The constant `TMP_MAX` is used to specify how many times `tmpnam` may be called before it can no longer find a unique name. `TMP_MAX` will be at least 25. If `tmpnam` is called more than this number of times, its behaviour is undefined by the Standard, but many implementations offer no practical limit.

If the argument s is set to `NULL`, then `tmpnam` uses an internal buffer to build the name, and returns a pointer to that. Subsequent calls may alter the same internal buffer. The argument may instead point to an array of at least `L_tmpnam` characters, in which case the name will be filled into the supplied buffer. Such a filename may then be created, and used as a temporary file. Since the name is generated by the function, it is unlikely to be very useful in any other context. Temporary files of this nature are not removed, except by direct calls to the remove function. They are most often used to pass temporary data between two separate programs.

`tmpfile`
: Creates a temporary binary file, opened for update, and returns a pointer to the stream of that file. The file will be removed when the stream is closed. If no file could be opened, `tmpfile` returns a null pointer.

9.10.6. Opening named files

Named files are opened by a call to the `fopen` function, whose declaration is this:

```
#include <stdio.h>
FILE *fopen(const char *pathname, const char *mode);
```

The `pathname` argument is the name of the file to open, such as that returned from `tmpnam`, or some program-specific filename.

Files can be opened in a variety of *modes*, such as *read* mode for reading data, *write* mode for writing data, and so on.

Note that if you only want to write data to a file, `fopen` will *create* the file if it does not already exist, or truncate it to zero length (losing its previous contents) if it did exist.

The Standard list of modes is shown in Table 9.3, although implementations may permit extra modes by appending extra characters at the end of the modes.

Mode	Type of file	Read	Write	Create	Truncate
"r"	text	yes	no	no	no
"rb"	binary	yes	no	no	no
"r+"	text	yes	yes	no	no
"r+b"	binary	yes	yes	no	no
"rb+"	binary	yes	yes	no	no
"w"	text	no	yes	yes	yes
"wb"	binary	no	yes	yes	yes
"w+"	text	yes	yes	yes	yes
"w+b"	binary	yes	yes	yes	yes
"wb+"	binary	yes	yes	yes	yes
"a"	text	no	yes	yes	no
"ab"	binary	no	yes	yes	no
"a+"	text	yes	yes	yes	no
"a+b"	binary	no	yes	yes	no
"ab+"	binary	no	yes	yes	no

Table 9.3. File opening modes

Beware that some implementations of binary files may pad the last record with `NULL` characters, so opening them with modes `ab`, `ab+` or `a+b` could position the file pointer beyond the last data written.

If a file is opened in append mode, *all* writes will occur at the end of the file, regardless of attempts to move the file position indicator with `fseek`. The initial position fo the file position indicator will be implementation defined.

Attempts to open a file in read mode, indicated by an '`r`' as the first character in the mode string, will fail if the file does not already exist or can't be read.

Files opened for update (‘`+`’ as the second or third character of mode) may be both read and written, but a read may not immediately follow a write, or a write follow a read, without an intervening call to one (or more) of `fflush`, `fseek`, `fsetpos` or `rewind`. The only exception is that a write may immediately follow a read if `EOF` was read.

It may also be possible in some implementations to omit the `b` in the binary modes, using the same modes for text and binary files.

Streams opened by fopen are fully buffered only if they are not connected to an interactive device; this ensures that prompts and responses are handled properly.

If `fopen` fails to open a file, it returns a null pointer; otherwise, it returns a pointer to the object controlling the stream. The `stdin`, `stdout` and `stderr` objects are not necessarily modifiable and it may not be possible to use the value returned from `fopen` for assignment to one of them. For this reason, `freopen` is provided.

9.10.7. Freopen

The `freopen` function is used to take an existing stream pointer and associate it with another named file:

```
#include <stdio.h>
FILE *freopen(const char *pathname,
              const char *mode, FILE *stream);
```

The `mode` argument is the same as for `fopen`. The `stream` is closed first, and any errors from the close are ignored. On error, `NULL` is returned, otherwise the new value for `stream` is returned.

9.10.8. Closing files

An open file is closed using `fclose`.

```
#include <stdio.h>

int fclose(FILE *stream);
```

Any unwritten data buffered for `stream` is flushed out and any unread data is thrown away. If a buffer had been automatically allocated for the stream, it is freed. The file is then closed.

Zero is returned on success, `EOF` if any error occurs.

9.10.9. Setbuf, setvbuf

These two functions are used to change the buffering strategy for an open stream:

```
#include <stdio.h>
```

```
int setvbuf(FILE *stream, char *buf,
              int type, size_t size);
void setbuf(FILE *stream, char *buf);
```

They must be used *before* the file is either read from or written to. The `type` argument defines how the `stream` will be buffered (see Table 9.4).

Value	Effect
`_IONBF`	Do not buffer I/O
`_IOFBF`	Fully buffer I/O
`_IOLBF`	Line buffer: flush buffer when full, when newline is written or when a read is requested.

Table 9.4. Type of buffering

The `buf` argument can be a null pointer, in which case an array is automatically allocated to hold the buffered data. Otherwise, the user can provide a buffer, but should ensure that its lifetime is at least as long as that of the `stream`: a common mistake is to use automatic storage allocated inside a compound statement; in correct usage it is usual to obtain the storage from `malloc` instead. The size of the buffer is specified by the `size` argument.

A call of `setbuf` is exactly the same as a call of `setvbuf` with `IOFBF` for the `type` argument, and `BUFSIZ` for the `size` argument. If `buf` is a null pointer, the value `_IONBF` is used for `type` instead.

No value is returned by `setbuf`, `setvbuf` returns zero on success, non-zero if invalid values are provided for `type` or `size`, or the request cannot be complied with.

9.10.10. Fflush

```
#include <stdio.h>

int fflush(FILE *stream);
```

If `stream` refers to a file opened for output or update, any unwritten data is 'written' out. Exactly what that means is a function of the host environment, and C cannot guarantee, for example, that data immediately reaches the surface of a disk which might be supporting the file. If the stream is associated with a file opened for input or update, any preceding `ungetc` operation is forgotten.

The most recent operation on the stream must have been an output operation; if not, the behaviour is undefined.

A call of `fflush` with an argument of zero flushes every output or update stream. Care is taken to avoid those streams that have not had an output as their last operation, thus avoiding the undefined behaviour mentioned above.

`EOF` is returned if an error occurs, otherwise zero.

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter9/stdarg.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter9/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter9/formatted_io.html*]

9.11. Formatted I/O

There are a number of related functions used for formatted I/O, each one determining the format of the I/O from a *format string*. For output, the format string consists of plain text, which is output unchanged, and embedded *format specifications* which call for some special processing of one of the remaining arguments to the function. On input, the plain text must match what is seen in the input stream; the format specifications again specify what the meaning of remaining arguments is.

Each format specification is introduced by a `%` character, followed by the rest of the specification.

9.11.1. Output: the printf family

For those functions performing output, the format specification takes the following form, with optional parts enclosed in brackets:

```
%<flags><field width><precision><length>conversion
```

The meaning of *flags*, *field width*, *precision*, *length*, and *conversion* are given below, although tersely. For more detail, it is worth looking at what the Standard says.

flags

Zero or more of the following:

`-`
: Left justify the conversion within its field.

`+`
: A signed conversion will always start with a plus or minus sign.

space
: If the first character of a signed conversion is not a sign, insert a space. Overridden by `+` if present.

`#`
: Forces an alternative form of output. The first digit of an octal conversion will always be a `0`; inserts `0X` in front of a non-zero hexadecimal conversion; forces a decimal point in all floating point conversions even if one is not necessary; does not remove trailing zeros from `g` and `G` conversions.

`0`
: Pad `d`, `i`, `o`, `u`, `x`, `X`, `e`, `E`, `f`, `F` and `G` conversions on the left with zeros up to the field width. Overidden by the `-` flag. If a precision is specified for the `d`, `i`, `o`, `u`, `x` or `X` conversions, the flag is ignored. The behaviour is undefined for other conversions.

field width

A decimal integer specifying the minimum output field width. This will be exceeded if necessary. If an asterisk is used here, the next argument is converted to an integer and used for the value of the field width; if the value is negative it is treated as a `-` flag followed by a positive field width. Output that would be less than the field width is padded with spaces (zeros if the *field* *width* integer starts with a zero) to fit. The padding is on the left unless the left-adjustment flag is specified.

precision

This starts with a period '`.`'. It specifies the minimum number of digits for `d`, `i`, `o`, `u`, `x`, or `X` conversions; the number of digits after the decimal point for `e`, `E`, `f` conversions; the maximum number of digits for `g` and `G` conversions; the number of characters to be printed from a string for `s` conversion. The amount of padding overrides the `field width`. If an asterisk is used here, the next argument is converted to an integer and used for the value of the field width. If the value is negative, it is treated as if it were missing. If only the period is present, the precision is taken to be zero.

length

`h` preceding a specifier to print an integral type causes it to be treated as if it were a `short`. (Note that the various sorts of short are always promoted to one of the flavours of int when passed as an argument.) `l` works like `h` but applies to a `long` integral argument. `L` is used to indicate that a `long double` argument is to be printed, and only applies to the floating-point specifiers. These are cause undefined behaviour if they are used with the 'wrong' type of conversion.

conversion

See Table 9.5.

Specifier	**Effect**	**Default precision**
`d`	signed decimal	1
`i`	signed decimal	1
`u`	unsigned decimal	1
`o`	unsigned octal	1
`x`	unsigned hexadecimal (`0`–`f`)	1
`X`	unsigned hexadecimal (`0`–`F`)	1
	Precision specifies minimum number of digits, expanded with leading zeros if necessary. Printing a value of zero with zero precision outputs no characters.	
`f`	Print a `double` with *precision* digits (rounded) after the decimal point. To suppress the decimal point use a *precision* of explicitly zero. Otherwise, at least one digit appears in front of the point.	6
`e`, `E`	Print a `double` in exponential format, rounded, with one digit before the decimal point, *precision* after it. A *precision* of zero suppresses the decimal point. There will be at least two digits in the exponent, which is printed as `1.23e15` in `e` format, or `1.23E15` in `E` format.	6
`g,G`	Use style `f`, or `e` (`E` with `G`) depending on the exponent. If the exponent is less than −4 or ≥ *precision*, `f` is not used. Trailing zeros are suppressed, a decimal point is only printed if there is a following digit.	unspecified
`c`	The `int` argument is converted to an unsigned char and the resultant character printed.	
`s`	Print a string up to *precision* digits long. If *precision* is not specified, or is greater than the length of the string, the string must be `NUL` terminated.	infinite
`p`	Display the value of a (`void *`) pointer in a system-dependent way.	
`n`	The argument must be a pointer to an integer. The number of characters output so far by this call will be written into the integer.	

Specifier	Effect	Default precision
`%`	A `%`	—

Table 9.5. Conversions

The functions that use these formats are described in Table 9.6. All need the inclusion of `<stdio.h>`. Their declarations are as shown.

```
#include <stdio.h>

int fprintf(FILE *stream, const char *format, ...);
int printf(const char *format, ...);
int sprintf(char *s, const char *format, ...);

#include <stdarg.h>     /* as well as stdio.h */
int vfprintf(FILE *stream, const char *format, va list arg);
int vprintf(const char *format, va list arg);
int vsprintf(char *s, const char *format, va list arg);
```

Name	Purpose
`fprintf`	General formatted output as described. Output is written to the file indicated by `stream`.
`printf`	Identical to `fprintf` with a first argument equal to `stdout`.
`sprintf`	Identical to `fprintf` except that the output is not written to a file, but written into the character array pointed to by `s`.
`vfprintf`	Formatted output as for `fprintf`, but with the variable argument list replaced by arg which must have been initialized by `va_start`. `va_end` is not called by this function.
`vprintf`	Identical to `vfprintf` with a first argument equal to `stdout`.
`vsprintf`	Formatted output as for `sprintf`, but with the variable argument list replaced by `arg` which must have been initialized by `va_start`. `va_end` is not called by this function.

Table 9.6. Functions performing formatted output

All of the above functions return the number of characters output, or a negative value on error. The trailing null is *not* counted by `sprintf` and `vsprintf`.

Implementations must permit at least 509 characters to be produced by any single conversion.

9.11.2. Input: the scanf family

A number of functions exist analogous to the `printf` family, but for the purposes of input instead. The most immediate difference between the two families is that the `scanf` group needs to be passed *pointers* to their arguments, so that the values read can be assigned to the proper destinations. Forgetting to pass a pointer is a very common error, and one which the compiler cannot detect—the variable argument list prevents it.

The format string is used to control interpretation of a stream of input data, which generally contains values to be assigned to the objects pointed to by the remaining arguments to `scanf`. The contents of the format string may contain:

white space
: This causes the input stream to be read up to the next non-white-space character.

ordinary character

Anything except white-space or `%` characters. The next character in the input stream *must* match this character.

conversion specification

This is a `%` character, followed by an optional `*` character (which suppresses the conversion), followed by an optional nonzero decimal integer specifying the maximum field width, an optional `h`, `l` or `L` to control the length of the conversion and finally a non-optional conversion specifier. Note that use of `h`, `l`, or `L` will affect the type of pointer which must be used.

Except for the specifiers `c`, `n` and `[`, a field of input is a sequence of non-space characters starting at the first non-space character in the input. It terminates at the first conflicting character or when the input field width is reached.

The result is put into wherever the corresponding argument points, unless the assignment is suppressed using the `*` mentioned already. The following conversion specifiers may be used:

`d i o u x`

Convert a signed integer, a signed integer in a form acceptable to `strtol`, an octal integer, an unsigned integer and a hexadecimal integer respectively.

`e f g`

Convert a `float` (*not* a double).

`s`

Read a string, and add a null at the end. The string is terminated by whitespace on input (which is not read as part of the string).

`[`

Read a string. A list of characters, called the *scan set* follows the `[`. A `]` delimits the list. Characters are read until (but not including) the first character which is *not* in the scan set. If the first character in the list is a circumflex `^`, then the scan set includes any character not in the list. If the initial sequence is `[^]` or `[]`, the `]` is not a delimiter, but part of the list and another `]` will be needed to end the list. If there is a minus sign (`-`) in the list, it must be either the first or the last character; otherwise the meaning is implementation defined.

`c`

Read a single character; white space is significant here. To read the first non-white space character, use `%1s`. A field width indicates that an array of characters is to be read.

`p`

Read a (`void *`) pointer previously written out using the `%p` of one of the `printfs`.

`%`

A `%` is expected in the input, no assignment is made.

`n`

Return as an integer the number of characters read by this call so far.

The size specifiers have the effect shown in Table 9.7.

Specifier	Modifies	Converts
`l`	`d i o u x`	long int
`h`	`d i o u x`	short int
`l`	`e f`	double
`L`	`e f`	long double

Table 9.7. Size specifiers

The functions are described below, with the following declarations:

```
#include <stdio.h>
```

```
int fscanf(FILE *stream, const char *format, ...);
int sscanf(const char *s, const char *format, ...);
int scanf(const char *format, ...);
```

`Fscanf` takes its input from the designated stream, `scanf` is identical to `fscanf` with a first argument of `stdin`, and `sscanf` takes its input from the designated character array.

If an input failure occurs before any conversion, EOF is returned. Otherwise, the number of successful conversions is returned: this may be zero if no conversions are performed.

An input failure is caused by reading `EOF` or reaching the end of the input string (as appropriate). A conversion failure is caused by a failure to match the proper pattern for a particular conversion.

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter9/input_and_output.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter9/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter9/character_io.html*]

9.12. Character I/O

A number of functions provide for character oriented I/O. Their declarations are:

```
#include <stdio.h>
/* character input */
int fgetc(FILE *stream);
int getc(FILE *stream);
int getchar(void);
int ungetc(int c, FILE *stream);

/* character output */
int fputc(int c, FILE *stream);
int putc(int c, FILE *stream);
int putchar(int c);

/* string input */
char *fgets(char *s, int n, FILE *stream);
char *gets(char *s);

/* string output */
int fputs(const char *s, FILE *stream);
int puts(const char *s);
```

Their descriptions are as follows.

9.12.1. Character input

These read an `unsigned char` from the input stream where specified, or otherwise `stdin`. In each case, the next character is obtained from the input stream. It is treated as an `unsigned char` and converted to an `int`, which is the return value. On End of File, the constant `EOF` is returned, and the end-of-file indicator is set for the associated stream. On error, EOF is returned, and the error indicator is set for the associated stream. Successive calls will obtain characters sequentially. The functions, if implemented as macros, may evaluate their `stream` argument more than once, so do not use side effects here.

There is also the supporting `ungetc` routine, which is used to push back a character on to a stream, causing it to become the next character to be read. This is not an output operation and can never cause the external contents of a file to be changed. A `fflush`, `fseek`, or `rewind` operation on the stream between the pushback and the read will cause the pushback to be forgotten. Only one character of pushback is guaranteed, and attempts to pushback `EOF` are ignored. In every case, pushing back a number of characters then reading or discarding them leaves the file position indicator unchanged. The file position indicator is decremented by every successful call to `ungetc` for a binary stream, but unspecified for a text stream, or a binary stream which is positioned at the beginning of the file.

9.12.2. Character output

These are identical in description to the input functions already described, except performing output. They return the character written, or `EOF` on error. There is no equivalent to End of File for an output file.

9.12.3. String output

These write strings to the output file; `stream` where specified, otherwise `stdout`. The terminating null is not written. Non-zero is returned on error, zero otherwise. *Beware*: `puts` appends a newline to the string output; `fputs` does not!

9.12.4. String input

`Fgets` reads a string into the array pointed to by `s` from the stream `stream`. It stops on either `EOF` or the first newline (which it reads), and appends a null character. At most n−1 characters are read (leaving room for the null).

Gets works similarly for the stream stdin, but discards the newline!

Both return s if successful, or a null pointer otherwise. In each case, if EOF is encountered before any characters have been read, the array is unchanged and a null pointer is returned. A read error in the middle of a string leaves the array contents undefined and a null pointer is returned.

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter9/formatted_io.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter9/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter9/unformatted_io.html*]

9.13. Unformatted I/O

This is simple: only two functions provide this facility, one for reading and one for writing:

```
#include <stdio.h>

size_t fread(void *ptr, size_t size, size_t nelem, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nelem, FILE *stream);
```

In each case, the appropriate read or write is performed on the data pointed to by `ptr`. Up to `nelem` elements, of size `size`, are transferred. Failure to transfer the full number is an error only when writing; End of File can prevent the full number on input. The number of elements actually transferred is returned. To distinguish between End of File on input, or an error, use `feof` or `ferror`.

If `size` or `nelem` is zero, `fread` does nothing except to return zero.

An example may help.

```
#include <stdio.h>
#include <stdlib.h>

struct xx{
        int xx_int;
        float xx_float;
}ar[20];

main(){

        FILE *fp = fopen("testfile", "w");

        if(fwrite((const void *)ar,
                sizeof(ar[0]), 5, fp) != 5){

                fprintf(stderr,"Error writing\n");
                exit(EXIT_FAILURE);
        }

        rewind(fp);

        if(fread((void *)&ar[10],
                sizeof(ar[0]), 5, fp) != 5){

                if(ferror(fp)){
                        fprintf(stderr,"Error reading\n");
                        exit(EXIT_FAILURE);
                }
                if(feof(fp)){
                        fprintf(stderr,"End of File\n");
                        exit(EXIT_FAILURE);
                }
        }
        exit(EXIT_SUCCESS);
}
```

Example 9.7

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter9/character_io.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter9/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter9/random_access_io.html*]

9.14. Random access functions

<gbdirect>

The file I/O routines all work in the same way; unless the user takes explicit steps to change the file position indicator, files will be read and written sequentially. A read followed by a write followed by a read (if the file was opened in a mode to permit that) will cause the second read to start immediately following the end of the data just written. (Remember that `stdio` insists on the user inserting a buffer-flushing operation between each element of a read-write-read cycle.) To control this, the Random Access functions allow control over the implied read/write position in the file. The file position indicator is moved without the need for a read or a write, and indicates the byte to be the subject of the next operation on the file.

Three types of function exist which allow the file position indicator to be examined or changed. Their declarations and descriptions follow.

```
#include <stdio.h>

/* return file position indicator */
long ftell(FILE *stream);
int fgetpos(FILE *stream, fpos_t *pos);

/* set file position indicator to zero */
void rewind(FILE *stream);

/* set file position indicator */
int fseek(FILE *stream, long offset, int ptrname);
int fsetpos(FILE *stream, const fpos_t *pos);
```

`Ftell` returns the current value (measured in characters) of the file position indicator if `stream` refers to a binary file. For a text file, a 'magic' number is returned, which may only be used on a subsequent call to `fseek` to reposition to the current file position indicator. On failure, `-1L` is returned and `errno` is set.

`Rewind` sets the current file position indicator to the start of the file indicated by `stream`. The file's error indicator is reset by a call of `rewind`. No value is returned.

`Fseek` allows the file position indicator for stream to be set to an arbitrary value (for binary files), or for text files, only to a position obtained from ftell, as follows:

- In the general case, the file position indicator is set to offset bytes (characters) from a point in the file determined by the value of `ptrname`. `Offset` may be negative. The values of `ptrname` may be `SEEK_SET`, which sets the file position indicator relative to the beginning of the file, `SEEK_CUR`, which sets the file position indicator relative to its current value, and `SEEK_END`, which sets the file position indicator relative to the end of the file. The latter is not necessarily guaranteed to work properly on binary streams.
- For text files, `offset` must either be zero or a value returned from a previous call to `ftell` for the same stream, and the value of `ptrname` must be `SEEK_SET`.
- `Fseek` clears the end of file indicator for the given stream and erases the memory of any `ungetc`. It works for both input and output.

- Zero is returned for success, non-zero for a forbidden request.

Note that for `ftell` and `fseek` it must be possible to encode the value of the file position indicator into a `long`. This may not work for very long files, so the Standard introduces `fgetpos` and `fsetpos` which have been specified in a way that removes the problem.

`Fgetpos` stores the current file position indicator for stream in the object pointed to by pos. The value stored is 'magic' and only used to return to the specified position for the same stream using fsetpos.

`Fsetpos` works as described above, also clearing the stream's end-of-file indicator and forgetting the effects of any ungetc operations.

For both functions, on success, zero is returned; on failure, non-zero is returned and errno is set.

9.14.1. Error handling

The standard I/O functions maintain two indicators with each open stream to show the end-of-file and error status of the stream. These can be interrogated and set by the following functions:

```
#include <stdio.h>

void clearerr(FILE *stream);

int feof(FILE *stream);

int ferror(FILE *stream);

void perror(const char *s);
```

`Clearerr` clears the error and EOF indicators for the stream.

`Feof` returns non-zero if the `stream`'s EOF indicator is set, zero otherwise.

`Ferror` returns non-zero if the stream's error indicator is set, zero otherwise.

`Perror` prints a single-line error message on the program's standard output, prefixed by the string pointed to by `s`, with a colon and a space appended. The error message is determined by the value of errno and is intended to give some explanation of the condition causing the error. For example, this program produces the error message shown:

```
#include <stdio.h>
#include <stdlib.h>

main(){

        fclose(stdout);
        if(fgetc(stdout) >= 0){
                fprintf(stderr, "What - no error!\n");
                exit(EXIT_FAILURE);
        }
        perror("fgetc");
        exit(EXIT_SUCCESS);
}

/* Result */
```

```
fgetc: Bad file number
```

Example 9.8

Well, we didn't say that the message had to be very meaningful!

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter9/unformatted_io.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter9/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter9/general_utilities.html*]

9.15. General Utilities

These all involve the use of the header `<stdlib.h>`, which declares a number of types and macros and several functions of general use. The types and macros are as follows:

`size_t`
: Described at the start of this chapter.

`div_t`
: This is the type of the structure returned by `div`.

`ldiv_t`
: This is the type of the structure returned by `ldiv`.

`NULL`
: Again, described at the start of this chapter.

`EXIT_FAILURE`
`EXIT_SUCCESS`
: These may be used as arguments to `exit`.

`MB_CUR_MAX`
: The maximum number of bytes in a multibyte character from the extended character set specified by the current locale.

`RAND_MAX`
: This is the maximum value returned by the `rand` function.

9.15.1. String conversion functions

Three functions take a string as an argument and convert it to a number of the type shown below:

```
#include <stdlib.h>

double atof(const char *nptr);
long atol(const char *nptr);
int atoi(const char *nptr);
```

For each of the functions, the number is converted and the result returned. None of them guarantees to set `errno` (although they may do in some implementations), and the results of a conversion which overflows or cannot be represented is undefined.

More sophisticated functions are:

```
#include <stdlib.h>

double strtod(const char *nptr, char **endptr);
long strtol(const char *nptr, char **endptr, int base);
unsigned long strtoul(const char *nptr,char **endptr, int base);
```

All three functions work in a similar way. Leading white space is skipped, then a `subject sequence`, resembling an appropriate constant, is found, followed by a sequence of unrecognized characters. The trailing null at the end of a string is always unrecognized. The subject sequence can be empty. The subject sequences are determined as follows:

`strtod`
: Optional `+` or `-`, followed by a digit sequence containing an optional decimal point character, followed by an optional exponent. No floating suffix will be recognized. If there is no decimal point present, it is assumed to follow the digit sequence.

`strtol`
: Optional `+` or `-`, followed by a digit sequence. The digits are taken from the decimal digits or an upper or lower case letter in the range `a`–`z` of the English alphabet; the letters are given the values 10–35 respectively. The base argument determines which values are permitted, and may be zero, or otherwise 2–36. Only 'digits' with a value less than that of `base` are recognized. A base of 16 permits the characters `0x` or `0X` to follow the optional sign. A `base` of zero permits the input of characters in the form of a C integer constant. No integer suffix will be recognized.

`strtoul`
: Identical to `strtol` but with no sign permitted.

If `endptr` is non-null, the address of the first unrecognized character is stored in the object that it points to. If the subject sequence is empty or has the wrong form, this is the value of `nptr`.

If a conversion can be performed, the functions convert the number and return its value, taking into account a leading sign where permitted. Otherwise they return zero. On overflow or error the action is as follows:

`strtod`
: On overflow, returns ±`HUGE_VAL` according to the sign of the result; on underflow, returns zero. In either case, `errno` is set to `ERANGE`.

`strtol`
: On overflow, `LONG_MAX` or `LONG_MIN` is returned according to the sign of the result, `errno` is set to `ERANGE`.

`strtoul`
: On overflow, `ULONG_MAX` is returned, `errno` is set to `ERANGE`.

If the locale is not the "C" locale, there may be other subject sequences recognised depending on the implementation.

9.15.2. Random number generation

Provision for pseudo-random number generation is made by the following functions.

```
#include <stdlib.h>

int rand(void);
void srand(unsigned int seed);
```

`Rand` returns a pseudo-random number in the range `0` to `RAND_MAX`, which has a value of at least `32767`.

`Srand` allows a given starting point in the sequence to be chosen according to the value of `seed`. If `srand` is not called before `rand`, the value of the `seed` is taken to be `1`. The same sequence of values will always be returned from `rand` for a given value of seed.

The Standard describes an algorithm which may be used to implement `rand` and `srand`. In practice, most implementations will probably use this algorithm.

9.15.3. Memory allocation

These functions are used to allocate and free storage. The storage so obtained is only guaranteed to be large enough to store an object of the specified type and aligned appropriately so as not to cause addressing exceptions. No further assumptions can be made.

```
#include <stdlib.h>

void *malloc(size_t size);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);

void *free(void *ptr);
```

All of the memory allocation functions return a pointer to allocated storage of size `size` bytes. If there is no free storage, they return a null pointer. The differences between them are that calloc takes an argument `nmemb` which specifies the number of elements in an array, each of whose members is `size` bytes, and so allocates a larger piece of store (in general) than `malloc`. Also, the store allocated by `malloc` is not initialized, whereas `calloc` sets all bits in the storage to zero. This is not necessarily the equivalent representation of floating-point zero, or the null pointer.

`Realloc` is used to change the size of the thing pointed to by `ptr`, which may require some copying to be done and the old storage freed. The contents of the object pointed to by ptr is unchanged up to the smaller of the old and the new sizes. If `ptr` is null, the behaviour is identical to `malloc` with the appropriate size.

`Free` is used to free space previously obtained with one of the allocation routines. It is permissible to give `free` a null pointer as the argument, in which case nothing is done.

If an attempt is made to free store which was never allocated, or has already been freed, the behaviour is undefined. In many environments this causes an addressing exception which aborts the program, but this is not a reliable indicator.

9.15.4. Communication with the environment

A miscellany of functions is found here.

```
#include <stdlib.h>

void abort(void);
int atexit(void (*func)(void));
void exit(int status);
char *getenv(const char *name);
int system(const char *string);
```

`abort`
: Causes abnormal program termination to occur, by raising the `SIGABRT` signal. Abnormal termination is only prevented if the signal is being caught, and the signal handler does not return. Otherwise, output files may be flushed and temporary files may be removed according to implementation definition, and an ‘unsuccessful termination’ status returned to the host environment. This function cannot return.

`atexit`
: The argument `func` becomes a function to be called, without arguments, when the program terminates. Up to at least 32 such functions may be registered, and are called on program termination in reverse order of their registration. Zero is returned for success, non-zero for failure.

`exit`
: Normal program termination occurs when this is called. First, all of the functions

registered using `atexit` are called, but beware—by now, `main` is considered to have returned and *no* objects with automatic storage duration may safely be used. Then, all the open output streams are flushed, then closed, and all temporary files created by `tmpfile` are removed. Finally, the program returns control to the host environment, returning an implementation-defined form of successful or unsuccessful termination status depending on whether the argument to `exit` was `EXITSUCCESS` or `EXIT FAILURE` respectively. For compatibility with Old C, zero can be used in place of EXITSUCCESS, but other values have implementation-defined effects. Exit *cannot* return.

`getenv`

The implementation-defined *environment list* is searched to find an item which corresponds to the string pointed to by name. A pointer to the item is returned—it points to an array which must not be modified by the program, but may be overwritten by a subsequent call to getenv. A null pointer is returned if no item matches.

The purpose and implementation of the environment list depends on the host environment.

`system`

An implementation-defined command processor is passed the string string. A null pointer will cause a return of zero if no command processor exists, non-zero otherwise. A non-null pointer causes the command to be processed. The effect of the command and the value returned are implementation defined.

9.15.5. Searching and sorting

Two functions exist in this category: one for searching an already sorted list, the other for sorting an unsorted list. They are completely general, handling arrays of arbitrary size with elements of arbitrary size.

To enable them to compare two elements, the user provides a comparison function, which is called with pointers to two of the elements as its arguments. It returns a value less than, equal to or greater than zero depending on whether the first pointer points to an element considered to be less than, equal to or greater than the object pointed to by the second pointer, respectively.

```
#include <stdlib.h>

void *bsearch(const void *key, const void *base,
        size_t nmemb, size_t size,
        int (*compar)(const void *, const void *));

void *qsort(const void *base, size_t nmemb,
        size_t size,
        int (*compar)(const void *, const void *));
```

For both functions, `nmemb` is the number of elements in the array, `size` is the size in bytes of an array element and `compar` is the function to be called to compare them. `Base` is a pointer to the base of the array.

`Qsort` will sort the array into ascending order.

`Bsearch` assumes that the array is already sorted and returns a pointer to any element it finds that compares equal to the object pointed to by `key`. A null pointer is returned if no match is found.

9.15.6. Integer arithmetic functions

These provide ways of finding the absolute value of an integral argument and the quotient and remainder of a division, for both `int` and `long` types.

```
#include <stdlib.h>

int abs(int j);
long labs(long j);

div_t div(int numerator, int denominator);
ldiv_t ldiv(long numerator, long denominator);
```

`abs`
`labs`
: These return the absolute value of their argument—choose the appropriate one for your needs. The behaviour is undefined if the value cannot be represented—this can happen in two's complement systems where the most negative number has no positive equivalent.

`div`
`ldiv`
: These divide the `numerator` by the `denominator` and return a structure of the indicated type. In each case the structure will contain a member called `quot` which contains the quotient of the division truncated towards zero, and a member called rem which will contain the remainder. The type of each member is `int` for `div` and `long` for `ldiv`. Provided that the result could be represented, `quot*denominator+rem == numerator`.

9.15.7. Functions using multibyte characters

The `LC_CTYPE` category of the current locale affects the behaviour of these functions. For an encoding that is state-dependent, each function is put in its initial state by a call in which its character pointer argument, `s`, is a null pointer. The internal state of the function is altered as necessary by subsequent calls when s is not a null pointer. If `s` is a null pointer, the functions return a non-zero value if encodings are state-dependent, otherwise zero. If the `LC_CTYPE` category is changed, the shift state of the functions will become indeterminate.

The functions are:

```
#include <stdlib.h>

int mblen(const char *s, size_t n);
int mbtowc(wchar_t *pwc, const char *s, size_t n);
int wctomb(char *s, wchar_t wchar);
size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);
size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);
```

`mblen`
: Returns the number of bytes that are contained in the multibyte character pointed to by `s`, or −1 if the first `n` bytes do not form a valid multibyte character. If s points to the null character, zero is returned.

`mbtowc`
: Converts the multibyte character pointed to by s to the corresponding code of type `wchar_t` and stores the result in the object pointed to by `pwc`, unless `pwc` is a null pointer. Returns the number of bytes successfully converted, or −1 if the first `n` bytes do not form a valid multibye character. No more than `n` bytes pointed to by `s` are examined. The value returned will not be more than `n` or `MB_CUR_MAX`.

`wctomb`

Converts the code whose value is in `wchar` to a sequence of bytes representing the corresponding multibyte character, and stores the result in the array pointed to by s, if s is not a null pointer. Returns the number of bytes that are contained in the multibyte character, or −1 if the value in wchar does not correspond to a valid multibyte character. At most, MB_CUR_MAX bytes are processed.

`mbstowcs`

Converts the sequence of multibyte characters, beginning in the initial shift state, in the array pointed to by `s`, into a sequence of corresponding codes which are then stored in the array pointed to by `pwcs`. Not more than n values will be placed in pwcs. Returns −1 if an invalid multibyte character is encountered, otherwise returns the number of array elements modified, excluding the terminating null-code.

If the two objects overlap, the behaviour is undefined.

`wcstombs`

Converts the sequence of codes pointed to by `pwcs` to a sequence of multibyte characters, beginning in the initial shift state, which are then stored in the array pointed to by `s`. Conversion stops when either a null-code is encountered or `n` bytes have been written to `s`. Returns −1 if a code is encountered which does not correspond to a valid multibyte character, otherwise the number of bytes written, excluding the terminating null-code.

If the two objects overlap, the behaviour is undefined.

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter9/random_access_io.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter9/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter9/string_handling.html*]

9.16. String handling

Numerous functions exist to handle strings. In C, a string is an array of characters terminated by a null. In all cases, the functions expect a pointer to the first character in the string. The header `<string.h>` declares these functions.

9.16.1. Copying

The functions for this purpose are:

```
#include <string.h>

void *memcpy(void *s1, const void *s2, size_t n);
void *memmove (void *s1, const void *s2, size_t n);
char *strcpy(char *s1, const char *s2);
char *strncpy(char *s1, const char *s2, size_t n);
char *strcat(char *s1, const char *s2);
char *strncat(char *s1, const char *s2, size_t n);
```

`memcpy`
: This copies `n` bytes from the place pointed to by `s2` to the place pointed to by `s1`. If the objects overlap, the result is undefined. The value of `s1` is returned.

`memmove`
: Identical to `memcpy`, but works even for overlapping objects. It may be marginally slower, though.

`strcpy`
`strncpy`
: Both of these copy the string pointed to by `s2` into the string pointed to by `s1`, including the trailing null. `Strncpy` will copy at most `n` characters, and pad with trailing nulls if `s2` is shorter than `n` characters. If the strings overlap, the behaviour is undefined. They return `s1`.

`strcat`
`strncat`
: Both append the string in `s2` to `s1`, overwriting the null at the end of `s1`. A final null is always written. At most `n` characters are copied from `s2` by `strncat`, which means that for safety the destination string should have room for its original length (not counting the null) plus `n + 1` characters. They return `s1`.

9.16.2. String and byte comparison

These comparison functions are used to compare arrays of bytes. This obviously includes the traditional C strings, which are an array of `char` (bytes) with a terminating null. All of these functions work by comparing a byte at a time, and stopping either when two bytes differ (in which case they return the sign of the difference between the two bytes), or the arrays are considered to be equal: no differences were found, and the length of the arrays was equal to the specified amount, or the null was found at the end of a string comparison.

For all except `strxfrm`, the value returned is less than, equal to or greater than zero depending on whether the first object was considered to be less than, equal to or greater than the second.

```
#include <string.h>

int memcmp(const void *s1, const void *s2, size_t n);
int strcmp(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t n);
size_t strxfrm(char *to, const char *from,
int strcoll(const char *s1, const char *s2);
```

`memcmp`
: Compares the first `n` characters in the objects pointed to by `s1` and `s2`. It is very dodgy to compare structures in this way, because unions or 'holes' caused by alignment padding can contain junk.

`strcmp`
: Compares the two strings. This is one of the most commonly used of the string-handling functions.

`strncmp`
: As for `strcmp`, but compares at most `n` characters.

`strxfrm`
: The string in from is converted (by some magic), and placed wherever to points. At most `maxsize` characters (including the trailing null) are written into the destination. The magic guarantees that two such transformed strings will give the same comparison with each other for the user's current locale when using strcmp, as when `strcoll` is applied to the original two strings.

  In all cases, the length of the resulting string (not counting its terminating null) is returned. If the value is equal to or greater than maxsize, the contents of *to is undefined. If `maxsize` is zero, `s1` may be a null pointer.

  If the two objects overlap, the behaviour is undefined.

`strcoll`
: This function compares the two strings according to the collating sequence specified by the current locale.

9.16.3. Character and string searching functions

```
#include <string.h>

void *memchr(const void *s, int c, size_t n);
char *strchr(const char *s, int c);
size_t strcspn(const char *s1, const char *s2);
char *strpbrk(const char *s1, const char *s2);
char *strrchr(const char *s, int c);
size_t strspn(const char *s1, const char *s2);
char *strstr(const char *s1, const char *s2);
char *strtok(const char *s1, const char *s2);
```

`memchr`
: Returns a pointer to the first occurrence in the initial `n` characters of `*s` of the `(unsigned char)c`. Returns null if there is no such occurrence.

`strchr`
: Returns a pointer to the first occurrence of `(char)c` in `*s`, including the null in the search. Returns null if there is no such occurrence.

`strcspn`

Returns the length of the initial part of the string `s1` which contains no characters from `s2`. The terminating null is not considered to be part of `s2`.

`strpbrk`
: Returns a pointer to the first character in `s1` which is any of the characters in `s2`, or null if there is none.

`strrchr`
: Returns a pointer to the last occurrence in `s1` of `(char)c` counting the null as part of `s1`, or null if there is none.

`strspn`
: Returns the length of the initial part of `s1` consisting entirely of characters from `s1`.

`strstr`
: Returns a pointer to the first occurrence in `s1` of the string `s2`, or null if there is none.

`strtok`
: Breaks the string in `s1` into 'tokens', each delimited by one of the characters from `s2` and returns a pointer to the first token, or null if there is none. Subsequent calls with `(char *)0` as the value of `s1` return the next token in sequence, with the extra fun that `s2` (and hence the delimiters) may differ on each subsequent call. A null pointer is returned if no tokens remain.

9.16.4. Miscellaneous functions

```
#include <string.h>

void *memset(void *s, int c, size_t n);
char *strerror(int errnum);
size_t strlen(const char *s);
```

`memset`
: Sets the `n` bytes pointed to by `s` to the value of `(unsigned char)c`. Returns `s`.

`strlen`
: Returns the length of the string `s` not counting the terminating null. This is a very widely used function.

`strerror`
: Returns a pointer to a string describing the error number `errnum`. This string may be changed by subsequent calls to `strerror`. Useful for finding out what the values in `errno` mean.

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter9/general_utilities.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter9/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter9/date_and_time.html*]

9.17. Date and time

These functions deal with either 'elapsed' or 'calendar' time. They share the `<time.h>` header, which declares the functions as necessary and also the following:

`CLOCKS_PER_SEC`
: This is the number of 'ticks' per second returned by the `clock` function.

`clock_t`
`time_t`
: These are arithmetic types used to represent different forms of time.

`struct tm`
: This structure is used to hold the values representing a calendar time. It contains the following members, with the meanings as shown.

```
int tm_sec      /* seconds after minute [0-61] (61 allows for 2 leap-seconds)*/
int tm_min      /* minutes after hour [0-59] */
int tm_hour     /* hours after midnight [0-23] */
int tm_mday     /* day of the month [1-31] */
int tm_mon      /* month of year [0-11] */
int tm_year     /* current year-1900 */
int tm_wday     /* days since Sunday [0-6] */
int tm_yday     /* days since January 1st [0-365] */
int tm_isdst    /* daylight savings indicator */
```

The `tm_isdst` member is positive if daylight savings time is in effect, zero if not and negative if that information is not available.

The time manipulation functions are the following:

```
#include <time.h>

clock_t clock(void);
double difftime(time_t time1, time_t time2);
time_t mktime(struct tm *timeptr);
time_t time(time_t *timer);
char *asctime(const struct tm *timeptr);
char *ctime(const time_t *timer);
struct tm *gmtime(const time_t *timer);
struct tm *localtime(const time_t *timer);
size_t strftime(char *s, size_t maxsize,
  const char *format,
  const struct tm *timeptr);
```

The functions `asctime`, `ctime`, `gmtime`, `localtime`, and `strftime` all share static data structures, either of type `struct tm` or `char []`, and calls to one of them may overwrite the data stored by a previous call to one of the others. If this is likely to cause problems, their users should take care to copy any values needed.

`clock`
: Returns the best available approximation to the time used by the current invocation of the program, in 'ticks'. `(clock_t)-1` is returned if no value is available. To find the actual time used by a run of a program, it is necessary to find the difference between the value at the start of the run and the time of interest—there is an implementation-defined constant factor which biases the value returned from clock. To determine the time in seconds, the value returned should be divided by `CLOCKS_PER_SEC`.

`difftime`
: This returns the difference in seconds between two calendar times.

`mktime`
: This returns the calendar time corresponding to the values in a structure pointed to by `timeptr`, or `(time_t)-1` if the value cannot be represented.

The `tm_wday` and `tm_yday` members of the structure are ignored, the other members are not restricted to their usual values. On successful conversion, the members of the structure are all set to appropriate values within their normal ranges. This function is useful to find out what value of a `time_t` corresponds to a known date and time.

`time`
: Returns the best approximation to the current calendar time in an unspecified encoding. `(time_t)-1` is returned if the time is not available.

`asctime`
: Converts the time in the structure pointed to by `timeptr` into a string of the form

```
Sun Sep 16 01:03:52 1973\n\0
```

the example being taken from the Standard. The Standard defines the algorithm used, but the important point to notice is that all the fields within that string are of constant width and relevant to most English-speaking communities. The string is stored in a static structure which may be overwritten by a subsequent call to one of the other time-manipulation functions (see above).

`ctime`
: Equivalent to `asctime(localtime(timer))`. See `asctime` for the return value.

`gmtime`
: Returns a pointer to a `struct tm` set to represent the calendar time pointed to by `timer`. The time is expressed in terms of Coordinated Universal Time (UTC) (formerly Greenwich Mean Time). A null pointer is returned if UTC is not available.

`localtime`
: Converts the time pointed to by `timer` into local time and puts the results into a `struct tm`, returning a pointer to that structure.

`strftime`
: Fills the character array pointed to by `s` with at most `maxsize` characters. The `format` string is used to format the time represented in the structure pointed to `timeptr`. Characters in the format string (including the terminating null) are copied unchanged into the array, unless one of the following format directives is found—then the value specified below is copied into the destination, *as appropriate to the locale*.

`%a`	abbreviated weekday name
`%A`	full weekday name
`%b`	abbreviated month name
`%B`	full month name
`%c`	date and time representation
`%d`	decimal day of month number 01–31
`%H`	hour 00–23 (24 hour format)
`%I`	hour 01–12 (12 hour format)
`%j`	day of year 001–366
`%m`	month 01–12
`%M`	minute 00–59
`%p`	local equivalent of 'AM' or 'PM'
`%S`	second 00–61
`%U`	week number in year 00–53 (Sunday is first day of week
`%w`	weekday, 0–6 (Sunday is 0)
`%W`	week number in year 00–53 (Monday is first day of week
`%x`	local date representation
`%X`	local time representation
`%y`	year without century prefix 00–99
`%Y`	year with century prefix
`%Z`	timezone name, or no characters if no timezone exists
`%%`	a `%` character

The total number of characters copied into `*s` is returned, excluding the null. If there was not room (as determined by `maxsize`) for the trailing null, zero is returned.

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter9/string_handling.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter9/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter9/summary.html*]

9.18. Summary

It will almost certainly be the standardization of the run-time library that has the most effect on the portability of C programs. Prospective users of C really should read through this chapter carefully and familiarize themselves with its contents. The lack of a widely implemented, portable library was historically the biggest single barrier to portability.

If you are writing programs for embedded systems, bad luck! The library is not defined for stand-alone applications, but in practice we can expect suppliers to produce a stand-alone library package too. It will probably come without the file handling, but there is no reason why, say, the string-handling functions should not work just as well in hosted and unhosted environments.

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter9/date_and_time.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter9/*]

Chapter 10

Complete Programs in C

Previous chapter [*http://publications.gbdirect.co.uk/c_book/chapter9/*] | Next chapter [*http://publications.gbdirect.co.uk/c_book/answers/*]

10.1. Putting it all together

Having considered the language and the libraries defined by the Standard, all that now remains is to demonstrate what complete programs look like. This chapter contains some example programs which illustrate how to combine these elements to build programs.

However, just before these examples are presented there is one more aspect of the C language to discuss.

Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter10/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter10/arguments_to_main.html*]

10.2. Arguments to main

<gbdirect>

For those writing programs which will run in a hosted environment, arguments to main provide a useful opportunity to give parameters to programs. Typically, this facility is used to direct the way the program goes about its task. It's particularly common to provide file names to a program through its arguments.

The declaration of main looks like this:

```
int main(int argc, char *argv[]);
```

This indicates that `main` is a function returning an integer. In hosted environments such as DOS or UNIX, this value or *exit status* is passed back to the command line interpreter. Under UNIX, for example, the exit status is used to indicate that a program completed successfully (a zero value) or some error occurred (a non-zero value). The Standard has adopted this convention; `exit(0)` is used to return ‘success’ to its host environment, any other value is used to indicate failure. If the host environment itself uses a different numbering convention, `exit` will do the necessary translation. Since the translation is implementation-defined, it is now considered better practice to use the values defined in `<stdlib.h>: EXIT_SUCCESS` and `EXIT_FAILURE`.

There are at least two arguments to `main`: `argc` and `argv`. The first of these is a count of the arguments supplied to the program and the second is an array of pointers to the strings which are those arguments—its type is (almost) ‘array of pointer to `char`’. These arguments are passed to the program by the host system's command line interpreter or job control language.

The declaration of the `argv` argument is often a novice programmer's first encounter with pointers to arrays of pointers and can prove intimidating. However, it is really quite simple to understand. Since `argv` is used to refer to an array of strings, its declaration will look like this:

```
char *argv[]
```

Remember too that when it is passed to a function, the name of an array is converted to the address of its first element. This means that we can also declare `argv` as `char **argv;` the two declarations are equivalent in this context.

Indeed, you will often see the declaration of `main` expressed in these terms. This declaration is exactly equivalent to that shown above:

```
int main(int argc, char **argv);
```

When a program starts, the arguments to main will have been initialized to meet the following conditions:

- `argc` is greater than zero.
- `argv[argc]` is a null pointer.
- `argv[0]` through to `argv[argc-1]` are pointers to strings whose meaning will be determined by the program.
- `argv[0]` will be a string containing the program's name or a null string if that is not

available. Remaining elements of `argv` represent the arguments supplied to the program. In cases where there is only support for single-case characters, the contents of these strings will be supplied to the program in lower-case.

To illustrate these points, here is a simple program which writes the arguments supplied to `main` on the program's standard output.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
        while(argc--)
                printf("%s\n", *argv++);
        exit(EXIT_SUCCESS);
}
```

Example 10.1

If the program name is `show_args` and it has arguments `abcde`, `text`, and `hello` when it is run, the state of the arguments and the value of `argv` can be illustrated like this:

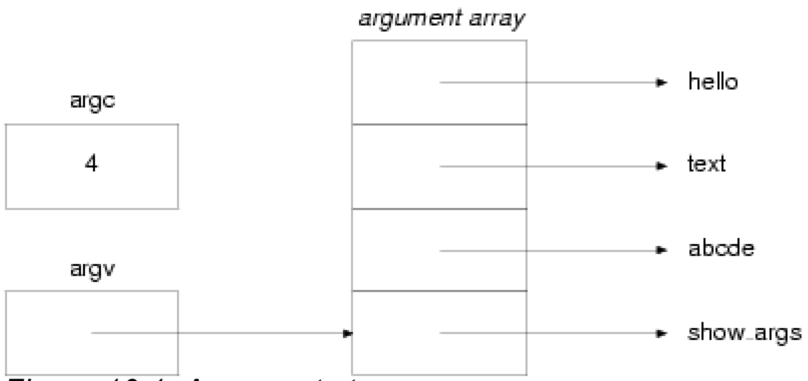


Figure 10.1. Arguments to a program

Each time that `argv` is incremented, it is stepped one item further along the array of arguments. Thus after the first iteration of the loop, `argv` will point to the pointer which in turn points to the `abcde` argument. This is shown in Figure 10.2.

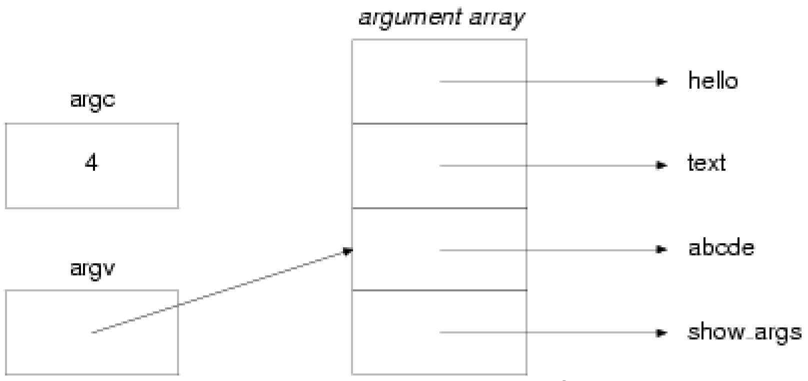


Figure 10.2. Arguments to a program after incrementing `argv`

On the system where this program was tested, a program is run by typing its name and then the arguments, separated by spaces. This is what happened (the `$` is a prompt):

```
$ show_args abcde text hello
show_args
abcde
text
hello
```

```
$
```

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter10/putting_it_together.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter10/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter10/interpreting_program_arguments.html*]

10.3. Interpreting program arguments

The loop used to examine the program arguments in the example above is a common C idiom wh
you will see in many other programs. An additional common idiom is to use ‘options’ to control the
behaviour of the program (these are also sometimes called switches or flags). Arguments which
start with a ‘-’ are taken to introduce one or more single-letter option indicators, which can be run
together or provided separately:

```
progname -abxu file1 file2
progname -a -b -x -u file1 file2
```

The idea is that each of the options selects a particular aspect from the program's repertoire of
features. An extension to that idea is to allow options to take arguments; if the `-x` option is specifi
to take an argument, then this is how it might be used:

```
progname -x arg file1
```

so that the `arg` argument is associated with the option. The `options` function below automates t
processing of this style of use, with the additional (common but preferably considered obsolescen
support for the provision of option arguments immediately following the option letter, as in:

```
progname -xarg file1
```

In either of the above cases, the options routine returns the character ‘`x`’ and sets a global pointer
`OptArg`, to point to the value `arg`.

To use this routine, a program must supply a list of valid option letters in the form of a string; whe
letter in this string is followed by a ‘`:`’ this indicates that the option letter is to be followed by an
argument. When the program is run, it is then simply a question of repeatedly calling the `option`
routine until no more option letters remain to be found.

It seems to be a fact of life that functions which scan text strings looking for various combinations
patterns within them end up being hard to read; if it's any consolation they aren't all that easy to w
either. The code that implements the options is definitely one of the breed, although by no means
one of the worst:

```
/*
* options() parses option letters and option arguments from the argv lis
* Succesive calls return succesive option letters which match one of
* those in the legal list. Option letters may require option arguments
* as indicated by a ':' following the letter in the legal list.
* for example, a legal list of "ab:c" implies that a, b and c are
* all valid options and that b takes an option argument. The option
* argument is passed back to the calling function in the value
* of the global OptArg pointer. The OptIndex gives the next string
* in the argv[] array that has not already been processed by options().
*
* options() returns -1 if there are no more option letters or if
* double SwitchChar is found. Double SwitchChar forces options()
* to finish processing options.
*
* options() returns '?' if an option not in the legal set is
* encountered or an option needing an argument is found without an
* argument following it.
*
*/

#include <stdio.h>
#include <string.h>

static const char SwitchChar = '-';
static const char Unknown = '?';

int OptIndex = 1;       /* first option should be argv[1] */
char *OptArg = NULL;    /* global option argument pointer */

int options(int argc, char *argv[], const char *legal)
{
        static char *posn = "";  /* position in argv[OptIndex] */
        char *legal_index = NULL;
        int letter = 0;

        if(!*posn){
                /* no more args, no SwitchChar or no option letter ? */
                if((OptIndex >= argc) ||
                        (*(posn = argv[OptIndex]) != SwitchChar) ||
                        !*++posn)
                                return -1;
                /* find double SwitchChar ? */
                if(*posn == SwitchChar){
                        OptIndex++;
                        return -1;
                }
        }
        letter = *posn++;
        if(!(legal_index = strchr(legal, letter))){
                if(!*posn)
                        OptIndex++;
                return Unknown;
        }
        if(*++legal_index != ':'){
                /* no option argument */
                OptArg = NULL;
                if(!*posn)
                        OptIndex++;
        } else {
                if(*posn)
                        /* no space between opt and opt arg */
                        OptArg = posn;
                else
                        if(argc <= ++OptIndex){
                                posn = "";
                                return Unknown;
                        } else
                                OptArg = argv[OptIndex];
                posn = "";
                OptIndex++;
        }
        return letter;
}
```

Example 10.2

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter10/arguments_to_main.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter10/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter10/pattern_matching_example.html*]

10.4. A pattern matching program

<gbdirect>

This section presents a complete program which makes use of option letters as program argumer to control the way it performs its job.

The program first processes any arguments that resemble options; the first argument which is not option is remembered for use as a 'search string'. Any remaining arguments are used to specify fi names which are to be read as input to the program; if no file names are provided, the program reads from its standard input instead. If a match for the search string is found in a line of input tex that whole line is printed on the standard output.

The `options` function is used to process all option letters supplied to the program. This program recognises five options: `-c`, `-i`, `-l`, `-n`, and `-v`. None of these options is required to be followed l an option argument. When the program is run with one or more of these options its behaviour is modified as follows:

`-c`
: the program prints a count of the total number of matching lines it found in the input file(s). N lines of text are printed.

`-i`
: when searching for a match, the case of letters in both the input lines and string is ignored.

`-l`
: each line of text printed on the output is prefixed with the line number being examined in the current input file.

`-n`
: each line of text printed on the output is prefixed with the name of the file that contained the line.

`-v`
: the program prints only lines which do not match the string supplied.

When the program finishes, it returns an exit status to indicate one of the following situations:

`EXIT_SUCCESS`
: at least one match was found.

`EXIT_FAILURE`
: no match was found, or some error occurred.

The program makes extensive use of standard library functions to do all of the hard work. For example, all of the file handling is performed by calls to `stdio` functions. Notice too that the real heart of the program, the string matching, is simply handled by a call to the `strstr` library functio

Here is the code for the whole program. Of course, to get this to work you would need to compile together with the code for the options routine presented above.

```
/*
* Simple program to print lines from a text file which contain
* the "word" supplied on the command line.
*
*/

#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>
#include <ctype.h>

/*
* Declarations for the pattern program
*
*/

#define CFLAG 0x001     /* only count the number of matching lines */
#define IFLAG 0x002     /* ignore case of letters */
#define LFLAG 0x004     /* show line numbers */
#define NFLAG 0x008     /* show input file names */
#define VFLAG 0x010     /* show lines which do NOT match */

extern int OptIndex;    /* current index into argv[] */
extern char *OptArg;    /* global option argument pointer */

/*
* Fetch command line switches from arguments to main()
*/

int options(int, char **, const char *);

/*
* Record the required options ready to control program beaviour
*/

unsigned set_flags(int, char **, const char *);

/*
* Check each line of the input file for a match
*/

int look_in(const char *, const char *, unsigned);

/*
* Print a line from the input file on the standard output
* in the format specified by the command line switches
*/

void print_line(unsigned mask, const char *fname,
                int lnno, const char *text);


static const char
                /* Legal options for pattern */
        *OptString = "cilnv",
                /* message when options or arguments incorrect */
        *errmssg = "usage: pattern [-cilnv] word [filename]\n";

int main(int argc, char *argv[])
{
        unsigned flags = 0;
        int success = 0;
        char *search_string;

        if(argc < 2){
                fprintf(stderr, errmssg);
                exit(EXIT_FAILURE);
        }
        flags = set_flags(argc, argv, OptString);

        if(argv[OptIndex])
                search_string = argv[OptIndex++];
        else {
                fprintf(stderr, errmssg);
                exit(EXIT_FAILURE);
        }

        if(flags & IFLAG){
                /* ignore case by dealing only with lowercase */
                char *p;
                for(p = search_string ; *p ; p++)
                        if(isupper(*p))
                                *p = tolower(*p);
        }

        if(argv[OptIndex] == NULL){
                /* no file name given, so use stdin */
                success = look_in(NULL, search_string, flags);
        } else while(argv[OptIndex] != NULL)
                success += look_in(argv[OptIndex++],
                                search_string, flags);

        if(flags & CFLAG)
                printf("%d\n", success);

        exit(success ? EXIT_SUCCESS : EXIT_FAILURE);
}

unsigned set_flags(int argc, char **argv, const char *opts)
{
        unsigned flags = 0;
        int ch = 0;

        while((ch = options(argc, argv, opts)) != -1){
                switch(ch){
                        case 'c':
                                flags |= CFLAG;
                                break;
                        case 'i':
                                flags |= IFLAG;
                                break;
                        case 'l':
                                flags |= LFLAG;
                                break;
                        case 'n':
                                flags |= NFLAG;
                                break;
                        case 'v':
                                flags |= VFLAG;
                                break;
                        case '?':
                                fprintf(stderr, errmssg);
                                exit(EXIT_FAILURE);
                }
        }
        return flags;
```

```
}


int look_in(const char *infile, const char *pat, unsigned flgs)
{
        FILE *in;
        /*
         * line[0] stores the input line as read,
         * line[1] is converted to lower-case if necessary
         */
        char line[2][BUFSIZ];
        int lineno = 0;
        int matches = 0;

        if(infile){
                if((in = fopen(infile, "r")) == NULL){
                        perror("pattern");
                        return 0;
                }
        } else
                in = stdin;

        while(fgets(line[0], BUFSIZ, in)){
                char *line_to_use = line[0];
                lineno++;
                if(flgs & IFLAG){
                        /* ignore case */
                        char *p;
                        strcpy(line[1], line[0]);
                        for(p = line[1] ; *p ; *p++)
                                if(isupper(*p))
                                        *p = tolower(*p);
                        line_to_use = line[1];
                }

                if(strstr(line_to_use, pat)){
                        matches++;
                        if(!(flgs & VFLAG))
                                print_line(flgs, infile, lineno, line[0]
                } else if(flgs & VFLAG)
                        print_line(flgs, infile, lineno, line[0]);
        }
        fclose(in);
        return matches;
}

void print_line(unsigned mask, const char *fname,
                        int lnno, const char *text)
{
        if(mask & CFLAG)
                return;
        if(mask & NFLAG)
                printf("%s:", *fname ? fname : "stdin");
        if(mask & LFLAG)
                printf(" %d :", lnno);
        printf("%s", text);
}
```

Example 10.3

Previous section
[*http://publications.gbdirect.co.uk/c_book/chapter10/interpreting_program_arguments.html*] | Chap
contents [*http://publications.gbdirect.co.uk/c_book/chapter10/*] | Next section
[*http://publications.gbdirect.co.uk/c_book/chapter10/ambitious_example.html*]

10.5. A more ambitious example

<gbdirect>

Finally here is a set of programs designed to cooperate and manipulate a single data file in a coherent, robust fashion.

The programs are intended to help keep track of a ladder of players who compete against each other at some game, squash or chess perhaps.

Each player has a rank from one to n, where n is the number of players who play, one being the highest rank on the ladder. Players lower down the ladder may challenge players above them and, if the lower ranked player wins, he or she moves up taking the rank of the player who loses. The loser in such a situation, and any other players between challenger and loser, are then moved down one rank. If a challenger does not win, the rankings on the ladder remain unchanged.

To provide some measure of equilibrium in the rankings, a player may challenge any higher ranked player, but only wins over players ranked three (or less) higher will allow the challenger to move up the rankings. This ensures that new players added to the bottom of the ladder are forced to play more than one game to reach the top of the ladder!

There are three basic tasks which are required to record all the information needed to keep such a ladder going:

- Printing the ladder.
- Addition of new players.
- Recording of results.

The design to be used here provides a separate program to perform each of these tasks. Having made this decision it is clear that a number of operations needed by each program will be common to all three. For example, all three will need to read player records from the data file, at least two will need to write player records into the data file.

This suggests that a good approach would be to design a ‘library’ of functions which manipulate player records and the data file which may in turn be combined to make up the programs which maintain the ladder.

Before this can be done it will be necessary to define the data structure which represents player records. The minimum information necessary to record for each player consists of player name and rank. However, to allow for more interesting statistics to be compiled about the ladder let us chose to also keep a record of games won, games lost and the time when the last game was played. Clearly this disparate set of information is best collected together in a structure.

The player structure declaration together with the declarations of the player library functions are combined together in the `player.h` header file. The data file is maintained as lines of text, each line corresponding to a record; this requires input and output conversions to be performed but is a useful technique if the conversions don't cost too much in performance terms.

```
/*
*
* Declarations and definitions for functions which manipulate player
* records which form the basis of the ladder
*
*/

#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define NAMELEN 12              /* max. for player name length */

#define LENBUF 256              /* max. for input buffer length */

#define CHALLENGE_RANGE 3       /* number of higher ranked players who may
                                * be challenged to move up in rank
                                */

extern char *OptArg;
```

```
typedef struct {
        char    name[NAMELEN+1];
        int     rank;
        int     wins;
        int     losses;
        time_t  last_game;
} player;

#define NULLPLAYER (player *)0

extern const char *LadderFile;

extern const char *WrFmt;       /* used when writing records */
extern const char *RdFmt;       /* used when reading records */

/*
* Declarations for routines used to manipulate the player records
* and the ladder file which are defined in player.c
*
*/

int     valid_records(FILE *);
int     read_records(FILE *, int, player *);
int     write_records(FILE *, player *, int);
player *find_by_name(char *, player *, int);
player *find_by_rank(int, player *, int);
void    push_down(player *, int, int, int);
int     print_records(player *, int);
void    copy_player(player *, player *);
int     compare_name(player *, player *);
int     compare_rank(player *, player *);
void    sort_players(player *, int);
```

Example 10.4

Here is the code for the `player.c` file implementing the generic functions which manipulate player records and the data file. These functions can be combined with more specific routines to make up the three programs required to maintain the ladder.

Notice that to manipulate the player records, each program is required to read the entire data file into a dynamically allocated array. Before this array is written back to the data file, it is assumed that the records it contains will have been sorted into rank order. If the records do not remain sorted, the `push_down` function will produce some 'interesting' results!

```
/*
* Generic functions to manipulate the ladder data file and
* player records.
*
*/

#include "player.h"

const char *LadderFile = "ladder";

const char *WrFmt = "%s %d      %d      %d      %ld\n";
const char *RdFmt = "%s %d      %d      %d      %ld";


/* note use of string-joining */
const char *HeaderLine =
        "Player Rank Won Lost Last Game\n"
        "===============================================\n";

const char *PrtFmt = "%-12s%4d %4d %4d %s\n";

/* return the number of records in the data file */

int valid_records(FILE *fp)
{
        int i = 0;
        long plrs = 0L;
        long tmp = ftell(fp);
        char buf[LENBUF];

        fseek(fp, 0L, SEEK_SET);

        for(i = 0; fgets(buf, LENBUF, fp) != NULL ; i++)
                ;

        /* Restore the file pointer to original state */

        fseek(fp, tmp, SEEK_SET);

        return i;
}

/* read num player records from fp into the array them */

int read_records(FILE *fp, int num, player *them)
{
        int i = 0;
        long tmp = ftell(fp);

        if(num == 0)
                return 0;

        fseek(fp, 0L, SEEK_SET);

        for(i = 0 ; i < num ; i++){
                if(fscanf(fp, RdFmt, (them[i]).name,
                                &((them[i]).rank),
                                &((them[i]).wins),
                                &((them[i]).losses),
                                &((them[i]).last_game)) != 5)
                        break;          /* error on fscanf! */
        }

        fseek(fp, tmp, SEEK_SET);
        return i;
}

/* write num player records to the file fp from the array them */

int write_records(FILE *fp, player *them, int num)
{
        int i = 0;

        fseek(fp, 0L, SEEK_SET);

        for(i = 0 ; i < num ; i++){
                if(fprintf(fp, WrFmt, (them[i]).name,
                                (them[i]).rank,
                                (them[i]).wins,
                                (them[i]).losses,
                                (them[i]).last_game) < 0)
                        break;          /* error on fprintf! */
        }

        return i;
}

/*
* return a pointer to the player in array them
* whose name matches name
*/

player *find_by_name(char * name, player *them, int num)
{
        player *pp = them;
        int i = 0;

        for(i = 0; i < num; i++, pp++)
```

```
                if(strcmp(name, pp->name) == 0)
                        return pp;

        return NULLPLAYER;
}

/*
* return a pointer to the player in array them
* whose rank matches rank
*/

player *find_by_rank(int rank, player *them, int num)
{
        player *pp = them;
        int i = 0;

        for(i = 0; i < num; i++, pp++)
                if(rank == pp->rank)
                        return pp;

        return NULLPLAYER;
}

/*
* reduce by one the ranking of all players in array them
* whose ranks are now between start and end
*/

void push_down(player *them, int number, int start, int end)
{
        int i;
        player *pp;

        for(i = end; i >= start; i--){
        if((pp = find_by_rank(i, them, number)) == NULLPLAYER){
                fprintf(stderr,
                        "error: could not find player ranked %d\n", i);
                free(them);
                exit(EXIT_FAILURE);
        } else
                (pp->rank)++;
        }
}

/* pretty print num player records from the array them */

int print_records(player *them, int num)
{
        int i = 0;

        printf(HeaderLine);

        for(i = 0 ; i < num ; i++){
                if(printf(PrtFmt,
                        (them[i]).name, (them[i]).rank,
                        (them[i]).wins, (them[i]).losses,
                        asctime(localtime(&(them[i]).last_game))) < 0)
                        break;          /* error on printf! */
        }

        return i;
}

/* copy the values from player from to player to */

void copy_player(player *to, player *from)
{
        if((to == NULLPLAYER) || (from == NULLPLAYER))
                return;

        *to = *from;
        return;
}

/* compare the names of player first and player second */

int compare_name(player *first, player *second)
{
        return strcmp(first->name, second->name);
}

/* compare the ranks of player first and player second */

int compare_rank(player *first, player *second)
{
        return (first->rank - second->rank);
}

/* sort num player records in the array them */

void sort_players(player *them, int num)
{
        qsort(them, num, sizeof(player), compare_rank);
}
```

Example 10.5

This code, when tested, was compiled into an object file which was then linked (together with an object file containing the code for the `options` function) with one of the following three programs to for the ladder maintenance utilities.

Here is the code for the simplest of those utilities, `showlddr` which is contained in the file `showlddr.c`.

This program takes a single option, `-f`, which you will notice takes an option argument. The purpose of this argument is to allow you to print a ladder data file with a name other than the default file name, `ladder`.

The player records in the data file should be stored pre-sorted but, just to be safe, `showlddr` sorts them before it prints them out.

```
/*
* Program to print the current ladder status.
*
*/

#include "player.h"

const char *ValidOpts = "f:";

const char *Usage = "usage: showlddr [-f ladder_file]\n";

char *OtherFile;

int main(int argc, char *argv[])
{
        int number;
        char ch;
        player *them;
        const char *fname;
        FILE *fp;

        if(argc == 3){
                while((ch = options(argc, argv, ValidOpts)) != -1){
                        switch(ch){
                                case 'f':
                                        OtherFile = OptArg;
                                        break;
                                case '?':
                                        fprintf(stderr, Usage);
                                        break;
                        }
                }
        } else if(argc > 1){
                fprintf(stderr, Usage);
                exit(EXIT_FAILURE);
        }

        fname = (OtherFile == 0)? LadderFile : OtherFile;
        fp = fopen(fname, "r+");

        if(fp == NULL){
                perror("showlddr");
                exit(EXIT_FAILURE);
        }

        number = valid_records (fp);

        them = (player *)malloc((sizeof(player) * number));

        if(them == NULL){
                fprintf(stderr,"showlddr: out of memory\n");
                exit(EXIT_FAILURE);
        }

        if(read_records(fp, number, them) != number){
                fprintf(stderr, "showlddr: error while reading"
                                        " player records\n");
                free(them);
                fclose(fp);
                exit(EXIT_FAILURE);
        }

        fclose(fp);

        sort_players(them, number);

        if(print_records(them, number) != number){
                fprintf(stderr, "showlddr: error while printing"
                                        " player records\n");
                free(them);
                exit(EXIT_FAILURE);
        }

        free(them);
        exit(EXIT_SUCCESS);
}
```

Example 10.6

Of course the `showlddr` program works only if there is an existing data file containing player records in the correct format. The program newplyr creates such a file if one does not already exist and then adds a new player record, in the correct format to that file.

Typically, new players are added at the bottom of the rankings but for the odd occasion where this really may not make sense, `newplyr` also allows a player to be inserted into the middle of the rankings.

A player may only appear once on the ladder (unless a pseudonym is used!) and there can only be one player at any one rank. Thus the program checks for duplicate entries and if the new player is to be inserted into a middling rank, moves other players already on the ladder out of the way.

As with the `showlddr` program, `newplyr` recognises a `-f` option as a request to add the new player to a file named by the option argument rather than the default file, ladder. In addition, newplyr requires two options, `-n` and `-r`, each with option arguments to specify both the new player's name and initial ranking respectively.

```
/*
* Program to add a new player to the ladder.
* You are expected to assign a realistic
* ranking value to the player.
*
*/

#include "player.h"

const char *ValidOpts = "n:r:f:";
```

```
char *OtherFile;

static const char *Usage = "usage: newplyr -r rank -n name [-f file]\n";

/* Forward declaration of function defined in this file */

void record(player *extra);

int main(int argc, char *argv[])
{
        char ch;
        player dummy, *new = &dummy;

        if(argc < 5){
                fprintf(stderr, Usage);
                exit(EXIT_FAILURE);
        }

        while((ch = options(argc, argv, ValidOpts)) != -1){
                switch(ch){
                case 'f':
                        OtherFile=OptArg;
                        break;
                case 'n':
                        strncpy(new->name, OptArg, NAMELEN);
                        new->name[NAMELEN] = 0;
                        if(strcmp(new->name, OptArg) != 0)
                                fprintf(stderr,
                                        "Warning: name truncated to %s\n", new->name);
                        break;
                case 'r':
                        if((new->rank = atoi(OptArg)) == 0){
                                fprintf(stderr, Usage);
                        exit(EXIT_FAILURE);
                        }
                        break;
                case '?':
                        fprintf(stderr, Usage);
                        break;
                }
        }

        if((new->rank == 0)){
                fprintf(stderr, "newplyr: bad value for rank\n");
                exit(EXIT_FAILURE);
        }

        if(strlen(new->name) == 0){
                fprintf(stderr,
                        "newplyr: needs a valid name for new player\n");
                exit(EXIT_FAILURE);
        }

        new->wins = new->losses = 0;
        time(& new->last_game); /* make now the time of the "last game" */

        record(new);

        exit(EXIT_SUCCESS);
}

void record(player *extra)
{
        int number, new_number, i;
        player *them;
        const char *fname =(OtherFile==0)?LadderFile:OtherFile;
        FILE *fp;

        fp = fopen(fname, "r+");
        if(fp == NULL){
                if((fp = fopen(fname, "w")) == NULL){
                        perror("newplyr");
                        exit(EXIT_FAILURE);
                }
        }

        number = valid_records (fp);
        new_number = number + 1;

        if((extra->rank <= 0) || (extra->rank > new_number)){
                fprintf(stderr,
                        "newplyr: rank must be between 1 and %d\n",
                        new_number);
                exit(EXIT_FAILURE);
        }

        them = (player *)malloc((sizeof(player) * new_number));

        if(them == NULL){
                fprintf(stderr,"newplyr: out of memory\n");
                exit(EXIT_FAILURE);
        }

        if(read_records(fp, number, them) != number){
                fprintf(stderr,
                        "newplyr: error while reading player records\n");
                free(them);
                exit(EXIT_FAILURE);
        }

        if(find_by_name(extra->name, them, number) != NULLPLAYER){
                fprintf(stderr,
                        "newplyr: %s is already on the ladder\n",
                        extra->name);
                free(them);
                exit(EXIT_FAILURE);
        }

        copy_player(&them[number], extra);

        if(extra->rank != new_number)
                push_down(them, number, extra->rank, number);

        sort_players(them, new_number);

        if((fp = freopen(fname, "w+", fp)) == NULL){
                perror("newplyr");
                free(them);
                exit(EXIT_FAILURE);
        }

        if(write_records(fp, them, new_number) != new_number){
                fprintf(stderr,
                        "newplyr: error while writing player records\n");
                fclose(fp);
                free(them);
                exit(EXIT_FAILURE);
        }
        fclose(fp);
        free(them);
}
```

Example 10.7

The only remaining utility required is one for recording the results of games played. The `result` program performs this task.

As with the previous two utilities, `result` will accept a `-f` option together with a file name to specify an alternative to the default player record file.

Unlike the `newplyr` utility, `result` interactively prompts the user for the names of the winning and losing players. The program insists that the names supplied should be those of existing players.

Given a valid pair of names, a check is then made to see if the loser is higher ranked than winner and whether or not the winner is ranked close enough for the victory to alter the rankings.

If a change in the standings is in order, the victor takes the loser's rank and the loser (as well as any other player on an intervening rank) is demoted one rank.

Here is the code for the `result` utility.

```
/*
* Program to record a result in the ladder
*
*/

#include "player.h"

/* Forward declarations for functions defined in this file */

char *read_name(char *, char *);
void move_winner(player *, player *, player *, int);

const char *ValidOpts = "f:";

const char *Usage = "usage: result [-f file]\n";

char *OtherFile;

int main(int argc, char *argv[])
{
        player *winner, *loser, *them;
        int number;
        FILE *fp;
        const char *fname;
        char buf[LENBUF], ch;

        if(argc == 3){
                while((ch = options(argc, argv, ValidOpts)) != -1){
                        switch(ch){
                                case 'f':
                                        OtherFile = OptArg;
                                        break;
                                case '?':
                                        fprintf(stderr, Usage);
                                        break;
                        }
                }
        } else if(argc > 1){
                fprintf(stderr, Usage);
                exit(EXIT_FAILURE);
        }

        fname = (OtherFile == 0)? LadderFile : OtherFile;
        fp = fopen(fname, "r+");

        if(fp == NULL){
                perror("result");
                exit(EXIT_FAILURE);
        }

        number = valid_records (fp);

        them = (player *)malloc((sizeof(player) * number));

        if(them == NULL){
                fprintf(stderr,"result: out of memory\n");
                exit(EXIT_FAILURE);
        }
        if(read_records(fp, number, them) != number){
                fprintf(stderr,
                        "result: error while reading player records\n");
                fclose(fp);
                free(them);
                exit(EXIT_FAILURE);
        }

        fclose(fp);

        if((winner = find_by_name(read_name(buf, "winner"), them, number))
                == NULLPLAYER){
                fprintf(stderr,"result: no such player %s\n",buf);
                free(them);
                exit(EXIT_FAILURE);
        }

        if((loser = find_by_name(read_name(buf, "loser"), them, number))
                == NULLPLAYER){
                fprintf(stderr,"result: no such player %s\n",buf);
                free(them);
                exit(EXIT_FAILURE);
        }

        winner->wins++;
        loser->losses++;

        winner->last_game = loser->last_game = time(0);

        if(loser->rank < winner->rank)
                if((winner->rank - loser->rank) <= CHALLENGE_RANGE)
                        move_winner(winner, loser, them, number);

        if((fp = freopen(fname, "w+", fp)) == NULL){
                perror("result");
                free(them);
                exit(EXIT_FAILURE);
        }

        if(write_records(fp, them, number) != number){
                fprintf(stderr,"result: error while writing player records\n");
                free(them);
                exit(EXIT_FAILURE);
        }
        fclose(fp);
        free(them);
        exit(EXIT_SUCCESS);
}

void move_winner(player *ww, player *ll, player *them, int number)
{
        int loser_rank = ll->rank;

        if((ll->rank - ww->rank) > 3)
                return;

        push_down(them, number, ll->rank, (ww->rank - 1));
        ww->rank = loser_rank;
        sort_players(them, number);
        return;
}

char *read_name(char *buf, char *whom)
{
        for(;;){
                char *cp;
                printf("Enter name of %s : ",whom);
                if(fgets(buf, LENBUF, stdin) == NULL)
                        continue;
                /* delete newline */
                cp = &buf[strlen(buf)-1];
                if(*cp == '\n')
                        *cp = 0;
                /* at least one char? */
                if(cp != buf)
                        return buf;
        }
}
```

Example 10.8

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter10/pattern_matching_example.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter10/*] | Next section [*http://publications.gbdirect.co.uk/c_book/chapter10/afterword.html*]

10.6. Afterword

The programs shown in this chapter should help to to get a feel for what middle-of-the-road C programs look like, using the language and libraries defined in the Standard.

What do we mean by ‘middle-of-the-road’? Simply this: they have been designed, implemented, tested and documented in a way appropriate for small, self-contained programs that have no real need to show high levels of robustness and reliability. Many programs don't need to meet demanding criteria; to do more to them would be over-engineering. Clearly, it is entirely dependent on the eventual purpose for which the program is intended.

There are situations which place very high demands on the software that is in use; programs to meet these requirements are very carefully engineered and have much higher amounts of effort put into reviewing, testing and the control of access to the source code than would be appropriate for simple illustrative example programs. C is also used in these application areas. The source code of programs that meet such high requirements tends to look distinctively different; the language is the same, but the amount of error checking and correction is typically much higher. We have *not* tried to illustrate that type of program.

Whichever environment you work in, we hope that this book has helped you in your understanding of C. Good luck!

Previous section [*http://publications.gbdirect.co.uk/c_book/chapter10/ambitious_example.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/chapter10/*]

Answers to Exercises

Previous chapter [*http://publications.gbdirect.co.uk/c_book/chapter10/*]

Chapter 1

Exercise 1.2 [http://publications.gbdirect.co.uk/c_book/chapter1/exercises.html#exercise-2

```
#include <stdio.h>
#include <stdlib.h>

main(){
   int this_number, divisor, not_prime;
   int last_prime;

   this_number = 3;
   last_prime = 3;

   printf("1, 3 is a prime pair\n");

   while(this_number < 10000){
      divisor = this_number / 2;
      not_prime = 0;
      while(divisor > 1){
         if(this_number % divisor == 0){
            not_prime = 1;
            divisor = 0;
         }
         else
            divisor = divisor-1;
      }

      if(not_prime == 0){
         if(this_number == last_prime+2)
            printf("%d, %d is a prime pair\n",
               last_prime, this_number);
         last_prime = this_number;
      }
      this_number = this_number + 1;
   }
   exit(EXIT_SUCCESS);
}
```

Exercise 1.3 [http://publications.gbdirect.co.uk/c_book/chapter1/exercises.html#exercise-3

```
#include <stdio.h>
#include <stdlib.h>

main(){
   printf("Type in a string: ");
   printf("The value was: %d\n", getnum());
   exit(EXIT_SUCCESS);
}

getnum(){
   int c, value;;

   value = 0;
   c = getchar();
   while(c != '\n'){
      value = 10*value + c - '0';
      c = getchar();
   }
   return (value);
}
```

Exercise 1.4 [*http://publications.gbdirect.co.uk/c_book/chapter1/exercises.html#exercise-4*

```
#include <stdio.h>
#include <stdlib.h>

/* array size */
#define NUMBER  10

main(){
   int arr[NUMBER], count, lo, hi;

   count = 0;
   while(count < NUMBER){
      printf("Type in a string: ");
      arr[count] = getnum();
      count = count+1;
   }
   lo = 0;
   while(lo < NUMBER-1){
      hi = lo+1;
      while(hi < NUMBER){
         int tmp;
         if(arr[lo] > arr[hi]){
            tmp = arr[lo];
            arr[lo] = arr[hi];
            arr[hi] = tmp;
         }
         hi = hi + 1;
      }
      lo = lo + 1;
   }
   /* now print them */
   count = 0;
   while(count < NUMBER){
      printf("%d\n", arr[count]);
      count = count+1;
   }
   exit(EXIT_SUCCESS);
}

getnum(){
   int c, value;;
   value = 0;
   c = getchar();
   while(c != '\n'){
      value = 10*value + c - '0';
      c = getchar();
   }
   return (value);
}
```

Exercise 1.5 [http://publications.gbdirect.co.uk/c_book/chapter1/exercises.html#exercise-5

```
#include <stdio.h>
#include <stdlib.h>

/*
* To print an int in binary, hex, decimal,
* we build an array of characters and print it out
* in order.
* The values are found least significant digit first,
* and printed most significant digit first.
*/
#define NDIG    32      /* assume max no. of digits */

int getnum(void);

main(){
   int val, i, count;
   char chars[NDIG];

   i = getnum();

   /* print in binary */
   val = i;
   count = 0;
   do{
      chars[count] = val % 2;
      val = val / 2;
      count = count + 1;
   }while(val);
   count = count - 1; /* just incremented above */

   while(count >= 0){
      printf("%d", chars[count]);
      count = count - 1;
   }
   printf("\n");

   /* print in decimal */
   val = i;
   count = 0;
   do{
      chars[count] = val % 10;
      val = val / 10;
      count = count + 1;
   }while(val);
   count = count - 1; /* just incremented above */

   while(count >= 0){
      printf("%d", chars[count]);
      count = count - 1;
   }
   printf("\n");

   /* print in hex */
   val = i;
   count = 0;
   do{
      chars[count] = val % 16;
      val = val / 16;
      count = count + 1;
   }while(val);
   count = count - 1; /* just incremented above */

   while(count >= 0){
      if(chars[count] < 10)
         printf("%d", chars[count]);
      else{
         /* assume 'A' - 'F' consecutive */
         chars[count] = chars[count]-10+'A';
         printf("%c", chars[count]);
      }
      count = count - 1;
   }
   printf("\n");
   exit(EXIT_SUCCESS);
}

getnum(){
   int c, value;;

   value = 0;
   c = getchar();
   while(c != '\n'){
      value = 10*value + c - '0';
      c = getchar();
   }
   return (value);
}
```

Chapter contents [*http://publications.gbdirect.co.uk/c_book/answers/*] | Next section [*http://publications.gbdirect.co.uk/c_book/answers/chapter_2.html*]

Chapter 2

Exercise 2.1 [*http://publications.gbdirect.co.uk/c_book/chapter2/variable_declaration.html#exercise-1*]

Trigraphs are used when the input device used, or the host system's native character set, do not support enough distinct characters for the full C language.

Exercise 2.2 [*http://publications.gbdirect.co.uk/c_book/chapter2/variable_declaration.html#exercise-2*]

Trigraphs would not be used in a system that has enough distinct characters to allocate a separate one to each of the C language symbols. For maximum portability, one might see a trigraph representation of a C program being distributed, on the grounds that most systems which do not use ASCII will be able to read ASCII coded data and translate it into their native codeset. A Standard C compiler could then compile such a program directly.

Exercise 2.3 [*http://publications.gbdirect.co.uk/c_book/chapter2/variable_declaration.html#exercise-3*]

White space characters are not equivalent to each other inside strings and character constants. Newline is special to the preprocessor.

Exercise 2.4 [*http://publications.gbdirect.co.uk/c_book/chapter2/variable_declaration.html#exercise-4*]

To continue a long line. Especially in systems that have an upper limit on physical line length.

Exercise 2.5 [*http://publications.gbdirect.co.uk/c_book/chapter2/variable_declaration.html#exercise-5*]

They become joined.

Exercise 2.6 [*http://publications.gbdirect.co.uk/c_book/chapter2/variable_declaration.html#exercise-6*]

Because the `*/` which apparently terminates the inner comment actually terminates the outer comment.

Exercise 2.7 [*http://publications.gbdirect.co.uk/c_book/chapter2/variable_declaration.html#exercise-7*]

31 characters for internal variables, six for external variables. The six character names must not rely on distinction between upper and lower case, either.

Exercise 2.8 [*http://publications.gbdirect.co.uk/c_book/chapter2/variable_declaration.html#exercise-8*]

A declaration introduces a name and a type for something. It does not necessarily reserve any storage.

Exercise 2.9 [*http://publications.gbdirect.co.uk/c_book/chapter2/variable_declaration.html#exercise-9*]

A definition is a declaration that also reserves storage.

Exercise 2.10 [*http://publications.gbdirect.co.uk/c_book/chapter2/real_types.html#exercise-10*]

It is always the case that the largest range of values can be held in a long double, although it may not actually be any different from one of the smaller floating point types.

Exercise 2.11 [*http://publications.gbdirect.co.uk/c_book/chapter2/real_types.html#exercise-11*]

The same answer holds true for the type with the greatest precision: long double. C does not permit the language implementor to use the same number of bits for, say, double and long double, then to allocate more bits for precision in one type and more for range in the other.

Exercise 2.12 [*http://publications.gbdirect.co.uk/c_book/chapter2/real_types.html#exercise-12*]

There can never be problems assigning a shorter floating point type to a longer one.

Exercise 2.13 [*http://publications.gbdirect.co.uk/c_book/chapter2/real_types.html#exercise-13*]

Assigning a longer floating type to a shorter one can result in overflow and undefined behaviour.

Exercise 2.14 [*http://publications.gbdirect.co.uk/c_book/chapter2/real_types.html#exercise-14*]

Undefined behaviour is completely unpredictable. Anything may happen. Often, nothing seems to happen except that erroneous arithmetic values are produced.

Exercise 2.15 [*http://publications.gbdirect.co.uk/c_book/chapter2/expressions_and_arithmetic.html#exercise-15*]

a. `Signed int` (by the integral promotions).
b. This cannot be predicted without knowing about the implementation. If an `int` can hold all of the values of an `unsigned char` the result will be `int`, again by the integral promotions. Otherwise, it will have to be `unsigned int`.
c. `Unsigned int`.
d. `Long`.
e. `Unsigned long`.
f. `Long`.
g. `Float`.
h. `Float`.
i. `Long double`.

Exercise 2.16 [*http://publications.gbdirect.co.uk/c_book/chapter2/constants.html#exercise-16*]

a. `i1 % i2`
b. `i1 % (int)f1`
c. If either operand is negative, the sign is implementation defined, otherwise it is positive. This means that, even if both operands are negative, you can't predict the sign.
d. Two—unary negate, binary subtract.
e. `i1 &= 0xf;`
f. `i1 |= 0xf;`
g. `i1 &= ~0xf;`
h. `i1 = ((i2 >> 4) & 0xf) | ((i2 & 0xf) << 4);`
i. The result is unpredictable. You must never use the same variable more than once in an expression if the expression changes its value.

Exercise 2.17 [*http://publications.gbdirect.co.uk/c_book/chapter2/exercises.html#exercise-17*]

a.
```
(c = (( u * f) + 2.6L);
(int = ((float) + long double);
(int = (long double));
(int);
```

Note: the integral promotion of `char` to `int` might be to `unsigned int`, depending on the implementation.

b.
```
(u += (((--f) / u) % 3));
(unsigned += ((float / unsigned) % int));
(unsigned += (float % int));
(unsigned += float);
(unsigned);
```

c.
```
(i <<= (u * (- (++f))));
(int <<= (unsigned * (- float)));
(int <<= (unsigned * float));
(int <<= float);
(int);
```

The rules for the shift operators state the right-hand operand is always converted to `int`. However, this does not affect the result, whose type is always determined by the type of the left-hand operand. This is doubly so for the current example, since an assignment operator is being used.

d.
```
(u = (((i + 3) + 4) + 3.1));
```

The rules state that the subexpressions involving + can be arbitrarily regrouped, as long as no type changes would be introduced. The types are:

```
(unsigned = (((int + int) + int) + double))
```

so the leftmost two additions can be regrouped. Working from the left:

```
(unsigned = ((int + int) + double));
(unsigned = (int + double));
(unsigned = double);
(unsigned);
```

e. `(u = (((3.1 + i) + 3 ) + 4));`

See the comments above on regrouping.

```
(unsigned = (((double + int) + int) + int));
```

The two rightmost additions can be regrouped.

```
(unsigned = ((double + int) + int));
(unsigned = (double + int));
(unsigned = double);
(unsigned);
```

f.
```
(c = ((i << (- (--f))) & 0xf));
(char = ((int << (- (--float))) & int ));
(char = ((int << (- float)) & int ));
(char = ((int << float) & int));
(char = (int & int));
(char);
```

Previous section [*http://publications.gbdirect.co.uk/c_book/answers/chapter_1.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/answers/*] | Next section [*http://publications.gbdirect.co.uk/c_book/answers/chapter_3.html*]

Chapter 3

Exercise 3.1 [*http://publications.gbdirect.co.uk/c_book/chapter3/exercises.html#exercise-1*

They all give an `int` result with a value of `1` for true and `0` for false.

Exercise 3.2 [*http://publications.gbdirect.co.uk/c_book/chapter3/exercises.html#exercise-2*

They all give an `int` result with a value of `1` for true and `0` for false.

Exercise 3.3 [*http://publications.gbdirect.co.uk/c_book/chapter3/exercises.html#exercise-3*

They guarantee an order of evaluation: left to right, and stop as soon as the overall result can be determined.

Exercise 3.4 [*http://publications.gbdirect.co.uk/c_book/chapter3/exercises.html#exercise-4*

`Break` can be used to turn a `switch` statement into a set of exclusive choices of action.

Exercise 3.5 [*http://publications.gbdirect.co.uk/c_book/chapter3/exercises.html#exercise-5*

`Continue` has no special meaning in a `switch` statement, but only to an outer `do`, `while` or `fo`
statement.

Exercise 3.6 [*http://publications.gbdirect.co.uk/c_book/chapter3/exercises.html#exercise-6*

Inside a `while` statement, the use of continue may cause the update of the loop control variable
be missed. It is, of course, the responsibility of the programmer to get this right.

Exercise 3.7 [*http://publications.gbdirect.co.uk/c_book/chapter3/exercises.html#exercise-7*

Because the scope of a label doesn't extend outside the function that it lives in, you can't use got
to jump from one function to another. Using the `longjmp` library routine, described in Chapter 9 [*http://publications.gbdirect.co.uk/c_book/chapter9/*], a form of function-to-function jump is supported, but not a completely general one.

Previous section [*http://publications.gbdirect.co.uk/c_book/answers/chapter_2.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/answers/*] | Next section [*http://publications.gbdirect.co.uk/c_book/answers/chapter_4.html*]

Chapter 4

<gbdirect>

Exercise 4.1 [*http://publications.gbdirect.co.uk/c_book/chapter4/exercises.html#exercise-1*]

```
#include <stdio.h>
#include <stdlib.h>

main(){
  int i, abs_val(int);;

  for(i = -10; i <= 10; i++)
    printf("abs of %d is %d\n", i, abs_val(i));
  exit(EXIT_SUCCESS);
}

int
abs_val(int x){

  if(x < 0)
    return(-x);
  return(x);
}
```

Exercise 4.2 [*http://publications.gbdirect.co.uk/c_book/chapter4/exercises.html#exercise-2*]

There are two files that form the answer to this exercise. This is the first.

```
#include <stdio.h>
#include <stdlib.h>

int curr_line(void), curr_col(void);
void output(char);

main(){
  printf("line %d\n", curr_line());
  printf("column %d\n", curr_col());

  output('a');
  printf("column %d\n", curr_col());

  output('\n');
  printf("line %d\n", curr_line());
  printf("column %d\n", curr_col());
  exit(EXIT_SUCCESS);
}
```

The second file contains the functions and static variables.

```
#include <stdio.h>

int curr_line(void), curr_col(void);
void output(char);

static int lineno=1, colno=1;

int
curr_line(void){
  return(lineno);
}

int
curr_col(void){
  return(colno);
}

void
output(char a){
  putchar(a);
  colno++;
  if(a == '\n'){
    colno = 1;
    lineno++;
  }
}
```

Exercise 4.3 [*http://publications.gbdirect.co.uk/c_book/chapter4/exercises.html#exercise-3*]

The recursive function:

```
#include <stdio.h>
#include <stdlib.h>

void recur(void);

main(){
  recur();
  exit(EXIT_SUCCESS);
}

void
recur(void){
  static ntimes;

  ntimes++;
    if(ntimes < 100)
      recur();
  printf("%d\n", ntimes);
  ntimes--;
}
```

Exercise 4.4 [*http://publications.gbdirect.co.uk/c_book/chapter4/exercises.html#exercise-4*]

And finally, the largest of all of the answers.

```
#include <stdio.h>
#include <stdlib.h>

#define PI 3.141592
#define INCREMENT (PI/20)
#define DELTA .0001

double sine(double), cosine(double);
static unsigned int fact(unsigned int n);
static double pow(double x, unsigned int n);

main(){
  double arg = 0;

  for(arg = 0; arg <= PI; arg += INCREMENT){
    printf("value %f\tsine %f\tcosine %f\n", arg, sine(arg), cosine(arg)
  }
  exit(EXIT_SUCCESS);
}

static unsigned int
fact(unsigned int n){
  unsigned int answer;

  answer = 1;
  while(n > 1)
    answer *= n--;

  return(answer);
}

static double
pow(double x, unsigned int n){
  double answer;

  answer = 1;
  while(n){
    answer *= x;
    n--;
  }
  return(answer);
}

double
sine(double x){
  double difference, thisval, lastval;
  unsigned int term;
  int sign;

  sign = -1;
  term = 3;

  thisval = x;
  do{
    lastval = thisval;
    thisval = lastval + pow(x, term)/fact(term) * sign;
    term += 2;
    sign = -sign;
    difference = thisval - lastval;
    if(difference < 0)
      difference = -difference;
    }while(difference > DELTA && term < 16);

  return(thisval);
}

double
cosine(double x){
double difference, thisval, lastval;
  unsigned int term;
  int sign;

  sign = -1;
  term = 2;

  thisval = 1;
  do{
    lastval = thisval;
    thisval = lastval + pow(x, term)/fact(term)  * sign;
    term += 2;
    sign = -sign;
    difference = thisval - lastval;
    if(difference < 0)
      difference = -difference;
  }while(difference > DELTA && term < 16);

  return(thisval);
}
```

Previous section [*http://publications.gbdirect.co.uk/c_book/answers/chapter_3.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/answers/*] | Next section [*http://publications.gbdirect.co.uk/c_book/answers/chapter_5.html*]

Chapter 5

<gbdirect>

Exercise 5.1 [*http://publications.gbdirect.co.uk/c_book/chapter5/exercises.html#exercise-1*

`0-9.`

Exercise 5.2 [*http://publications.gbdirect.co.uk/c_book/chapter5/exercises.html#exercise-2*

Nothing. It is guaranteed to be a valid address and can be used to check a pointer against the enc of the array.

Exercise 5.3 [*http://publications.gbdirect.co.uk/c_book/chapter5/exercises.html#exercise-3*

Only when they point into the same array, or to the same object.

Exercise 5.4 [*http://publications.gbdirect.co.uk/c_book/chapter5/exercises.html#exercise-4*

It can safely be used to hold the value of a pointer to any sort of object.

Exercise 5.5 [*http://publications.gbdirect.co.uk/c_book/chapter5/exercises.html#exercise-5*

a.
```
int
st_eq(const char *s1, const char * s2){

  while(*s1 && *s2 && (*s1 == *s2)){
    s1++; s2++;
  }

  return(*s1-*s2);
}
```

b.
```
const char *
find_c(char c, const char *cp){

  while(*cp && *cp != c)
     cp++;
  if(*cp)
    return(cp);

  return(0);
}
```

c.
```
const char *
sub_st(const char *target, const char *sample){
  /*
   * Try for a substring starting with
   * each character in sample.
  */
  while(*sample){
    const char *targ_p, *sample_p;

    targ_p = target;
    sample_p = sample;
    /* string compare */
    while(*targ_p && *sample_p && (*targ_p == *sample_p)){
        targ_p++; sample_p++;
    }
    /*
    * If at end of target, have substring!
    */
    if(*targ_p == 0)
      return(sample);
    /* otherwise try next place */
      sample++;
    }
  return(0);      /* no match */
}
```

Exercise 5.6 [*http://publications.gbdirect.co.uk/c_book/chapter5/exercises.html#exercise-6*

No answer can be given.

Previous section [*http://publications.gbdirect.co.uk/c_book/answers/chapter_4.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/answers/*] | Next section [*http://publications.gbdirect.co.uk/c_book/answers/chapter_6.html*]

Chapter 6

Exercise 6.1 [*http://publications.gbdirect.co.uk/c_book/chapter6/exercises.html#exercise-1*

```
struct {
  int a,b;
};
```

Exercise 6.2 [*http://publications.gbdirect.co.uk/c_book/chapter6/exercises.html#exercise-2*

Without a tag or any variables defined, the structure declaration is of little use. It cannot be referre to later.

Exercise 6.3 [*http://publications.gbdirect.co.uk/c_book/chapter6/exercises.html#exercise-3*

```
struct int_struc{
  int a,b;
}x,y;
```

Exercise 6.4 [*http://publications.gbdirect.co.uk/c_book/chapter6/exercises.html#exercise-4*

```
struct int_struc z;
```

Exercise 6.5 [*http://publications.gbdirect.co.uk/c_book/chapter6/exercises.html#exercise-5*

```
p = &z;
p->a = 0;
```

Exercise 6.6 [*http://publications.gbdirect.co.uk/c_book/chapter6/exercises.html#exercise-6*

Explicitly, for example

```
struct x;
```

or implicitly,

```
struct x *p;
```

when no outer declaration exists.

Exercise 6.7 [*http://publications.gbdirect.co.uk/c_book/chapter6/exercises.html#exercise-7*

It is not treated as a pointer, but as a short-hand way of initializing the individual array elements.

Exercise 6.8 [*http://publications.gbdirect.co.uk/c_book/chapter6/exercises.html#exercise-8*

Nothing unusual at all, the string is treated as a literal constant of type `const char *`.

Exercise 6.9 [*http://publications.gbdirect.co.uk/c_book/chapter6/exercises.html#exercise-9*

Yes. It is easier!

Previous section [*http://publications.gbdirect.co.uk/c_book/answers/chapter_5.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/answers/*] | Next section [*http://publications.gbdirect.co.uk/c_book/answers/chapter_7.html*]

Chapter 7

<gbdirect>

Exercise 7.1 [*http://publications.gbdirect.co.uk/c_book/chapter7/exercises.html#exercise-1*

```
#define MAXLEN 100
```

Exercise 7.2 [*http://publications.gbdirect.co.uk/c_book/chapter7/exercises.html#exercise-2*

In expressions, there may be precedence problems. A safer definition would be `#define VALUE (100+MAXLEN)`.

Exercise 7.3 [*http://publications.gbdirect.co.uk/c_book/chapter7/exercises.html#exercise-3*

```
#define REM(a,b) ((a)%(b))
```

Exercise 7.4 [*http://publications.gbdirect.co.uk/c_book/chapter7/exercises.html#exercise-4*

```
#define REM(a,b) ((long)(a)%(long)(b))
```

Exercise 7.5 [*http://publications.gbdirect.co.uk/c_book/chapter7/exercises.html#exercise-5*

It generally signifies a library header file.

Exercise 7.6 [*http://publications.gbdirect.co.uk/c_book/chapter7/exercises.html#exercise-6*

It generally signifies a private header file.

Exercise 7.7 [*http://publications.gbdirect.co.uk/c_book/chapter7/exercises.html#exercise-7*

By using the conditional compilation directives. Examples are shown in the text.

Exercise 7.8 [*http://publications.gbdirect.co.uk/c_book/chapter7/exercises.html#exercise-8*

It uses long int in place of int and unsigned long int, in place of unsigned int using the arithmetic environment provided by the translator, not the target. It must provide at least the ranges described in `<limits.h>`.

Previous section [*http://publications.gbdirect.co.uk/c_book/answers/chapter_6.html*] | Chapter contents [*http://publications.gbdirect.co.uk/c_book/answers/*]

The C Book — Disclaimer and Copyright Notice

<gbdirect>

CH-101

CHEMISTRY-I

Time Allotted: 3 Hours Full Marks: 70

The questions are of equal value.
The figures in the margin indicate full marks.
Candidates are required to give their answers in their own words as far as practicable.

GROUP A
(Multiple Choice Type Questions)

1. Answer any *ten* questions. $10\times1 = 10$

(i) The unit of ionic mobility is

(A) cm^2 volt sec^{-1} (B) cm^2 $volt^{-1}$ sec^{-1}
(C) cm^{-2} volt sec (D) none of these

(ii) A living system is thermodynamically an example of

(A) an isolated system (B) a closed system
(C) an adiabatically closed system (D) an open system

(iii) The number of isomers are possible for the complex $[Pt(NH_3)_4(NCS)_2]^{+2}$ are

(A) 3 (B) 4 (C) 5 (D) 6

(iv) When rate constant K has the unit moles $liter^{-1}$ sec^{-1}, the order of the reaction is

(A) 0 (B) 1 (C) 2 (D) 3

(v) The conductance of an ion in aqueous solution depends on its

(A) charge only (B) speed only
(C) charge and speed (D) charge, speed and hydration

(vi) Which of the following possesses an octane rating of 100?

(A) petrol (B) LPG (C) CNG (D) iso-octane

(vii) But-2-ene is more stable than But-1-ene due to

(A) inductive effect (B) resonance effect

(C) hyperconjugative effect (D) electromeric effect

(viii) On adding a little indium to germanium metal, we get

(A) rectifier (B) insulator

(C) n-type semiconductor (D) p-type semiconductor

(ix) Which one is most nucleophilic species?

(A) H_2O (B) O^-H

(C) CH_3O^- (D) NH_3

(x) Which of the following defects arise due to misplacement of ions in a crystal lattice?

(A) Schottky defect (B) Frenkel defect

(C) metal excess defect (D) non-stoichiometric defect

(xi) Producer gas is a mixture of

(A) carbon monoxide and hydrogen (B) carbon monoxide and nitrogen

(C) carbon dioxide and hydrogen (D) carbon dioxide and nitrogen

(xii) The electrode potential of a standard hydrogen electrode is ____.

(A) 0.1 volt (B) 1 volt

(C) 0.01 volt (D) 0 volt

(xiii) XeF_4 has the shape of

(A) tetrahedral (B) square planar

(C) trigonal bypiramidal (D) trigonal

GROUP B
(Short Answer Type Questions)

Answer any *three* questions. $3\times5 = 15$

2. Write down the difference between Schottky defect and Frenkel defect. 5

3. If 50 gm ice, initially at –5 °C is heated to water vapour finally at 127 °C, calculate entropy change for the entire process. (specipic heat of ice and water vapour are 9.0 and 8 cal / mole respectively). 5

4. What is Kirchhoff's equation? Deduce it. 5

5. (a) Distinguish between HTC and LTC of coal. 3
(b) What is aviation fuel? 2

6. What do you understand by polymerization? Write down the structure and use of Nylon-6, 6 and PVC. 1+2+2

7.(a) Give one example of nucleophile. 1
(b) Give reaction and mechanism of nitration in benzene ring. 2
(c) Write note on electromeric effect. 2

GROUP C
(Long Answer Type Questions)

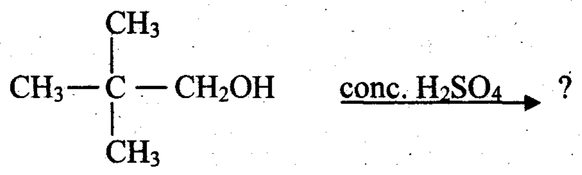
Answer any *three* questions. 3×15 = 45

8. (a) Show that PdV – VdP is not a perfect differential. 3
(b) State and explain Hess's law of thermochemistry. Show that it is a corollary to the first law of thermodynamics. 5
(c) Derive Maxwell's relation $(dS / dV)_T = (dP / dT)_V$. 3
(d) Derive that $C_P - C_V = \alpha^2 VT/\beta$, where α is called the thermal expansion coefficient and β is called the compressibility factor. 4

9. (a) Draw the conductometric titration curve for the titration of HCl vs. NaOH and explain the salient features in the curve. 4
(b) What is Kohlrausch's Law? If the ionic conductance at infinite dilution of NaCl, HCl and CH_3COONa are 126.45, 426.16 and 91.0 respectively, what will be the $\lambda\alpha$ of acetic acid? 2+3
(c) Discuss the physicochemical principle involved in the measurement of pH of an aqueous solution by Hydrogen electrode method. 6

10.(a) Write a comparative short note on S_N2 and S_N1 reaction covering (i) rate equation (ii) mechanism (iii) potential energy diagram and (iv) implication of stereochemistry, if any. 4
(b) Explain the order of the acid strength HCOOH > CH_3COOH > phenol > ethanol. 4

(c) Write down the product of the following reaction with mechanism 4



(d) Explain the transition state theory for a reaction. 3

11.(a) Why a good motor engine fuel is not good diesel engine fuel or vice versa? 2

(b) (i) Differentiate between addition and condensation polymerization. 4+1

(ii) What is glass transition temperature in polymer.

(c) Give example of n-type and p-type dopent. 2

(d) Name the monomers of the following polymers : 2

(i) PVC, (ii) Teflon, (iii) Nylon 6, 6 and (iv) Bakelite.

(e) Equal number of polymer molecules with M1=100000 and M2=10000 are mixed. Calculate Mn and Mw. 3

(f) What do you mean by degree of polymerization? 1

12. Write short notes on any *three* of the following: 3×5

(a) Joule-Thomson expansion and inversion temperature

(b) Clausius-Clapeyron equation

(c) Octane number and cetane number

(d) Reference electrode

(f) Biodiesel.

MAULANA ABUL KALAM AZAD UNIVERSITY OF TECHNOLOGY, WEST BENGAL

CH-101

CHEMISTRY – I

Time Allotted: 3 Hours Full Marks: 70

The questions are of equal value.
The figures in the margin indicate full marks.
Candidates are required to give their answers in their own words as far as practicable.
All symbols are of usual significance.

GROUP A
(Multiple Choice Type Questions)

1. Answer any *ten* questions. $10 \times 1 = 10$

(i) In P-V diagram for expression of work done for expansion of volume in reversible process

(A) slope of adiabatic > slope of isothermal
(B) slope of adiabatic < slope of isothermal
(C) slope of adiabatic = slope of isothermal
(D) none of these

(ii) At inversion temperature, Joule-Thomson coefficient is

(A) zero (B) positive
(C) negative (D) all of these

(iii) Which defect causes decrease in the density of the crystal?

(A) Interstitial (B) Schottky
(C) Frenkel (D) F-centre

(iv) The cell reaction is spontaneous if the cell potential is

(A) zero (B) positive (C) negative (D) infinite

(v) Which of the following is a macromolecule?

(A) Nylon-66 (B) Bakelite (C) Polyester (D) Chlorophyle

(vi) Example of an electrophile is

(A) $AlCl_3$ (B) NH_3 (C) CH_3OH (D) CN^-

(vii) The highest ranking of coal is

(A) Anthracite (B) Bituminous
(C) Lignite (D) Peat

(viii) Which of the following is not a polymer?

(A) Silk (B) DNA (C) TNT (D) Starch

(ix) The role of a positive catalyst is to

(A) decrease activation energy (B) decrease enthalpy
(C) increase enthalpy (D) increase activation energy

(x) Which one of the following compounds is the most reactive towards S_N^2 mechanism?

(A) n-butyl chloride (B) t-butyl chloride
(C) 2-chlorobutane (D) 2-methyl-1-chloropropane

(xi) Which of the following is not a renewable source of energy?

(A) Solar (B) Wind
(C) Natural gas (D) Ocean tidal energy

(xii) CO_2 is isostructural with

(A) C_2H_2 (B) $HgCl_2$ (C) NH_3 (D) NO_2

(xiii) Intra-molecular H-bonding can be observed in

(A) o-nitro phenol (B) p-nitro phenol
(C) HF molecule (D) CH_3OH

GROUP B
(Short Answer Type Questions)

Answer any *three* questions. $3\times5 = 15$

2. (a) The rate constant of a reaction become doubled when temperature changes from 27°C to 37°C. Calculate the activation energy for the reaction. 3

   (b) Define homogeneous catalyst with example. 2

3. Why molecular weight of a polymer molecule is expressed in terms of average? Deduce expression for weighted average polymer molecular weight (M_w). 2+3

4. Why carboxylic acid is stronger acid than phenol? All the carbon-carbon distance in benzene are equal – Explain. 3+2

5. Define transport number and ionic mobility. State Hittorf's rule. 2+3

6. Discuss Schottky defects. Why ZnO is white but changes colour on heating? 3+2

7. (a) Define fuels. 1

   (b) The percentage composition of a coal sample is C = 80%, H = 4%, O = 3%, N = 3%, S = 2%, ash = 5% and rest moisture. Calculate the quantity of air needed for complete combustion of 1 kg of coal if 40% excess of air is supplied. 4

GROUP C
(Long Answer Type Questions)

Answer any *three* questions. $3\times15 = 45$

8. (a) Deduce the expression for the maximum work done when n moles of an ideal gas expand isothermally and reversibly. 5

   (b) Show that the decrease in Gibb's free energy at constant temperature and pressure is equal to the total work over and above the mechanical work. 5

   (c) Deduce the Gibb's- Helmholtz equation in any one of its standard forms. 5

9. (a) Give the name of the products obtained with their temperature range in fractional distillation of crude petroleum. 5
   (b) Explain p-type and n-type semiconductor with example. 5
   (c) What is carbonization of coal? Distinguish between HTC and LTC? 2+3

10.(a) Write a short note on substitution reaction. 3
   (b) Write down the order of stability for carbanion and justify the answer. 4
   (c) Specify the process for the conversion. 3

   Toluene $\rightarrow$ 2, 4, 6 trinitrotoluene

   (d) What is solvolysis reaction? What will be the product when solvent is methanol? 5

11.(a) A sample of human blood contains 60 gms of albumin with M.wt = 69000 and 40 gms of globine with M.wt = 162000 find Mn and Mw? 4
   (b) Why hyper conjugation is called 'no bond resonance'? In what way does hyper conjugation differ from the usual process of resonance? 4
   (c) Nitration of phenol is faster than benzene – Explain. 2
   (d) Rate constant at 300 K and 310 K are 4.5×10^{-5} s^{-1} and 9×10^{-5} s^{-1}. Find activation energy and the frequency factor. What is the order of the reaction? 5

12.(a) Write short note on 'Synthetic petrol'. 4
   (b) Write short note on 'bio gas'. 3
   (c) Calculate the EMF of the following cell: 3

   $Ni|Ni^{2+}(IM)\,||Pb^{2+}(IM)|\,Pb$ at 25°C. Given standard electrode potentials of Ni and Pb are –0.24V and –0.13V respectively at 25°C.

   (d) Define gross and net calorific values of fuel. 2
   (e) Explain pseudounimolecular reaction with example. 3

ES-101

BASIC ELECTRICAL AND ELECTRONICS ENGINEERING

Time Allotted: 3 Hours　　　　　　　　　　　　　　　　　　Full Marks: 70

The questions are of equal value.
The figures in the margin indicate full marks.
Candidates are required to give their answers in their own words as far as practicable.

PART - I (Electrical)

(*Use blue colour answer book for this part*)

GROUP A

(Multiple Choice Type Questions)

1. Answer any *five* questions　　　　　　　　　　　　　　　　$5\times1 = 5$

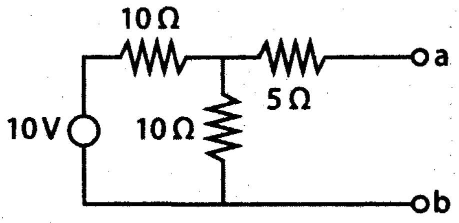
(i) Force experienced by a small conductor of length L, carrying current I, placed in a magnetic field B, is at an angle θ with respect to B is given by

(A) BIL　　(B) BIL sin θ　　(C) BIL cos θ　　(D) zero

(ii) Three resistance of 4 ohm, 6 ohm and 8 ohm are connected in parallel. The maximum power dissipation will occur in

(A) 4 ohm　　(B) 6 ohm
(C) 8 ohm　　(D) equal in all resistor

(iii) For the circuit shown the Thevenin's voltage and resistance as shown at ab are



(A) 5 V, 10 ohm　　(B) 10 V, 10 ohm　　(C) 5V, 5 ohm　　(D) 15 V, 15 ohm

(iv) Inductive resistance of a coil of inductance will be proportional to

(A) 62.8 ohm (B) 628 ohm (C) 0.2 ohm (D) 20 ohm

(v) The power factor of a purely inductive will be proportional to

(A) zero (B) one (C) infinity (D) 0.5

(vi) The form factor current is 1, its shape is

(A) sinusoidal (B) triangle (C) square (D) sawtooth

(vii) The unit of m.m.f is

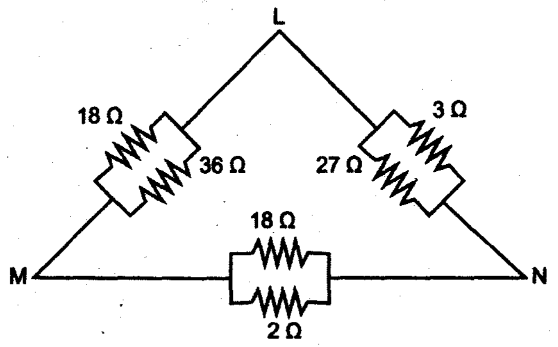
(A) AT/m (B) N/Wb (C) both (A) and (B) (D) Wb/m^2

GROUP B

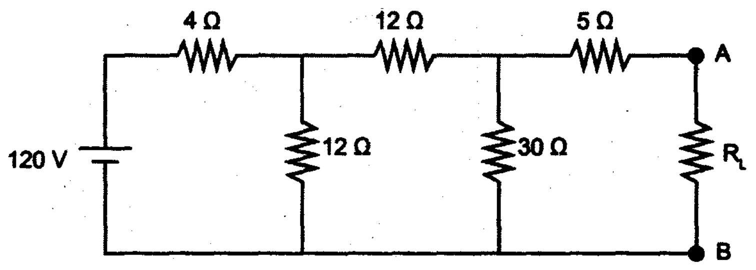
(Short Answer Type Questions)

Answer any *two* questions. 2×5 = 10

2. A network of resistance is formed as given in the figure. Compute the resistance measured between L and M.



3. Obtain the maximum power transferred to R_L in the circuit and also the value of R_L.



4. Two impedances $Z_1 = (47.92 + j76.73)\ \Omega$ and $Z_2 = (10 - j5)\ \Omega$ are connected in parallel across a 200 volt, 50 Hz supply. Find the current through each impedance and total current. What is the phase difference angle of each branch current with respect to the applied voltage?

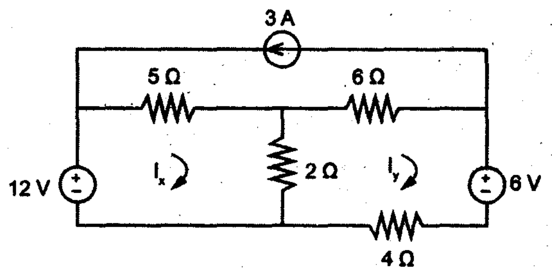
5. Derive an expression for the lifting power of an electromagnet.

GROUP C

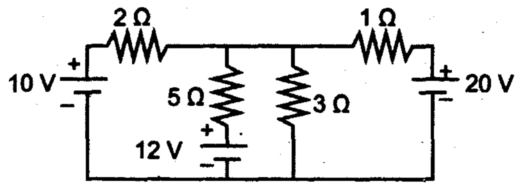
(Long Answer Type Questions)

Answer any *two* questions. $2\times10=20$

6. (a) Using Mesh analysis, determine the currents I_x and I_y in the network shown below 5



(b) Determine the voltage across 3 Ω resistor by applying Thevenin's Theorem in the following network. 5



7. (a) A coil of resistance 10 Ω and inductance 0.02 H is connected in series with another coil of resistance 6 Ω and inductance 15 mH across a 230 V, 50 Hz supply. 6

Calculate (i) impedance of the circuit

(ii) the voltage drop across each coil and

(iii) the total power consumed by the circuit.

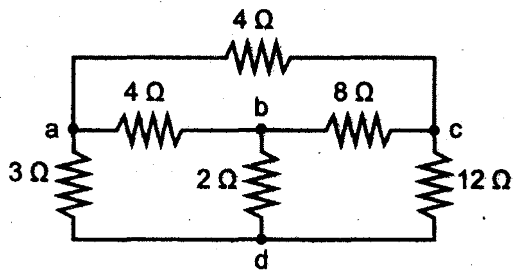
(b) Define Power factor. Show that the active power of a purely capacitive circuit over a complete cycle is zero. 1+3

8. (a) State Faraday's Laws of Electromagnetic Induction. Show that $M = K\sqrt{L_1 L_2}$ where M is the mutual inductance between the coils L_1 and L_2, and K is the co-efficient of coupling. 2+3

(b) A coil of 250 turns carrying a current of 2A produces a flux of 0.3 mWb. When the current is reduced to zero in 2 ms, the voltage induced in a nearby coil is 60 V. Calculate the self-inductance of each coil and the manual inductance between the two coils. Assume co-efficient of coupling to be 0.7. 5

9. (a) Explain Delta (Δ) – Star (Y) conversion and Star (Y) – Delta (Δ) conversion, for a purely resistive circuit. $2^1/_2+2^1/_2$

(b) Reduce the network given below to obtain the equivalent resistance as seen between nodes *ad*. 5



PART - II (Electronics)

(*Use green colour answer book for this part*)

GROUP A

(Multiple Choice Type Questions)

1. Answer any *five* questions $5\times1 = 5$

(i) Fermi level of a p-type semiconductors lies

(A) Near the conduction band edge (B) near the valence band edge
(C) at the middle of the band gap (D) none of these

(ii) With the rise in temperature reverse saturation current

(A) increases linearly (B) increases exponentially
(C) decreases linearly (D) decreases exponentially

(iii) Zener diodes are used as

(A) reference voltage elements (B) reference current elements
(C) reference resistance (D) both (A) and (B)

(iv) With both junctions reverse biased the transistor operates in

(A) active region (B) cut-off region
(C) saturation region (D) inverted region

(v) The ripple factor for a half wave rectifier is

(A) 0.482 (B) 0.41
(C) 1.21 (D) 1.11

(vi) If $\alpha = 0.98$ then $\beta =$

(A) 0.49 (B) 49
(C) 50 (D) 0.5

GROUP B

(Short Answer Type Questions)

Answer any *two* questions. $2 \times 5 = 10$

2. (i) Explain the drift and diffusion current for a semiconductor. 3

   (ii) What is meant by intrinsic semiconductor? 2

3. At 300 K, the intrinsic carrier concentration of Si is 1.5×10^{16} m^{-3}. If the electron and hole mobility are 0.13 and 0.05 m^{2}V^{-1} s^{-1}, calculate the intrinsic resistivity of Si at 300 K. 5

4. Distinguish between zener break down and avalanche break down. 5

GROUP C

(Long Answer Type Questions)

Answer any *two* questions. $2 \times 10 = 20$

5. (i) Define Fermi level. 1

   (ii) What is the position of Fermi level in an intrinsic semiconductor? How does its position change when (a) donors and (b) acceptors are added to the semiconductor? 2

   (iii) Draw the energy band diagram of a (a) forward biased pn junction diode (b) reverse biased pn junction diode (c) unbiased pn junction diode. 3

   (iv) Determine the resistivity of germanium (a) in intrinsic condition at 300K (b) with donor impurity of 1 in 10^{7} (c) with acceptor impurity of 1 in 10^{8}. Given that for germanium at 300K, $n_i = 2.5 \times 10^{13}$ cm^{-3}, $\mu_n = 3800$ cm^{2} /V–s, $\mu_p = 1800$ cm^{2}/V–s and number of germanium atoms / cm^{-3} $= 4.4 \times 10^{22}$. 4

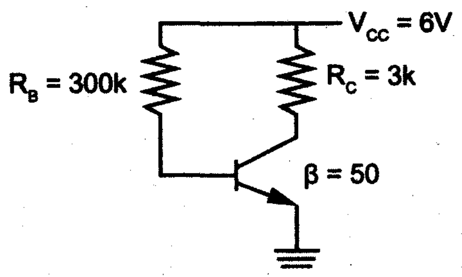
6. (i) Explain the operation of a half-wave rectifier with the help of circuit diagram. Obtain a mathematical expression for the efficiency of the half-wave rectifier and show that its ripple factor is 1.21. 2+4

(ii) A diode having internal resistance 20 Ω is used for half-wave rectification. The ac input voltage is 6 sin ω and load resistance is 600 Ω. Obtain (a) dc output voltage (b) ac input power (c) ripple factor and (d) the efficiency of the rectifier. 4

7. (i) Draw the circuit diagram and output characteristics of a common emitter transistor showing different regions. 1+2

(ii) Explain the concept of thermal run-away and Q-point. 2+2

(iii) Calculate V_{CE} and I_C in the circuit below. Assume $V_{BE} = 0.7$ V. 3



8. Write short notes on any *two* of the following : 2×5 = 10

(a) Zener diode as a voltage regulator

(b) Junction capacitances

(c) Stability factors

(d) Varactor diode

MAULANA ABUL KALAM AZAD UNIVERSITY OF TECHNOLOGY, WEST BENGAL

ES-101

BASIC ELECTRICAL AND ELECTRONIC ENGINEERING - I

Time Allotted: 3 Hours Full Marks: 70

The questions are of equal value.
The figures in the margin indicate full marks.
Candidates are required to give their answers in their own words as far as practicable.
All symbols are of usual significance.

PART – I (Electrical)

GROUP A

(Multiple Choice Type Questions)

1. Answer any *five* questions. $5\times1 = 5$

(i) Inductive reactance of a coil of inductance 0.2H at 50Hz is

(A) 62.8 Ω (B) 628 Ω (C) 6.28 Ω (D) 4 Ω

(ii) If a voltage source is to be neglected then the terminals across the source will be

(A) open circuited (B) short circuited
(C) both (A) and (B) (D) none of these

(iii) The form factor of a waveform is 1, its shape is

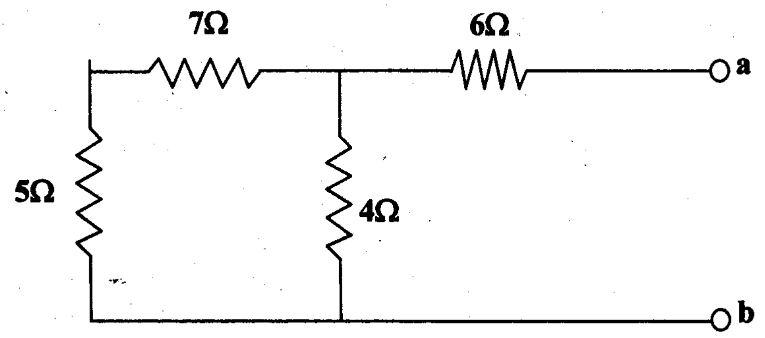
(A) sinusoidal (B) triangular

(C) square (D) sawtooth

(iv) The equivalent resistance across the terminal a-b will be

(A) 9 Ω (B) 3 Ω

(C) 6 Ω (D) 2 Ω



(v) An inductive coil with impedance Z = (5 + j10), it's conductance will be

(A) 0.2 ℧ (B) 2 ℧

(C) 4 ℧ (D) 0.4 ℧

(vi) The bandwidth of a series resonant a. c. circuit is equal to

(A) $\frac{1}{2\pi R}$ (B) $\frac{L}{2\pi R}$

(C) $\frac{R}{2\pi L}$ (D) $\frac{1}{RLC}$

(vii) Kirchhoff's current law is used for

(A) loop analysis (B) node analysis

(C) finding out equivalent resistance (D) none of these

GROUP B

(Short Answer Type Questions)

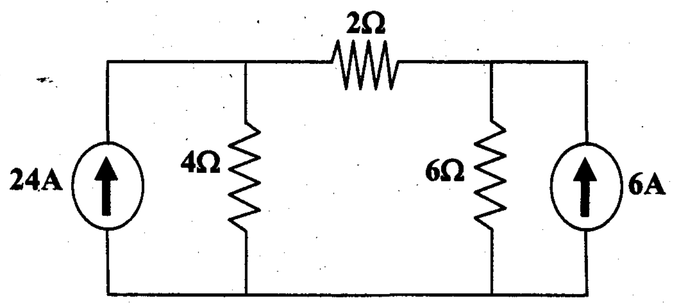
Answer any *two* questions. $2\times5=10$

2. Derive an expression for the resonant frequency of a parallel circuit, one branch consisting of a coil of inductance L in series with resistance R and the other branch of capacitance C.

3. Two coils having self inductances L_1 and L_2 and mutual inductance between them is M. Derive a mathematical expression for co-efficient of coupling k for these coils.

4. State and prove Maximum Power Transfer Theorem.

5. Applying Superposition theorem compute the current through 2 Ω resistor.



GROUP C

(Long Answer Type Questions)

Answer any *two* questions. $2\times10=20$

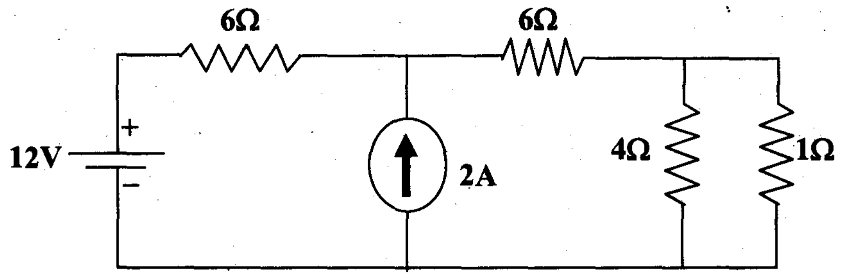
6. (a) A capacitor of 100 μF is connected across a 200 V, 50 Hz single phase supply. Calculate (i) the reactance of the capacitor, (ii) r.m.s. value of current, (iii) the maximum current. 3

(b) What is meant by bandwidth? With a neat sketch of waveform find out the expression for the bandwidth of a resonant circuit. 7

7. (a) Give an example of passive element. 1

(b) State and explain Thevenin's Theorem. 3

(c) Find the current through 1 Ω resistor using Thevenin's Theorem. 6



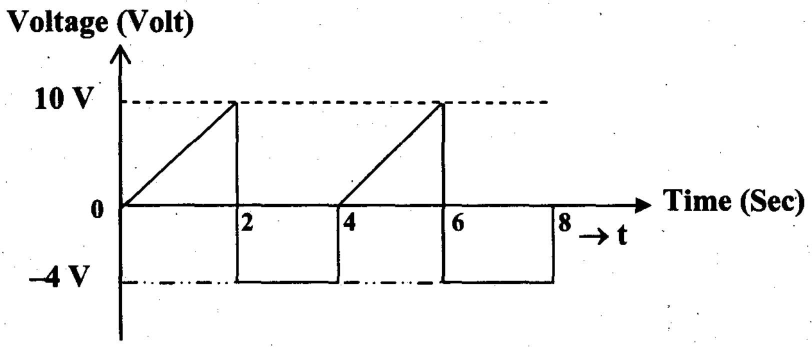
8. (a) State and explain Bio-Savart's law. 3

(b) What is meant by hysteresis in a magnetic circuit? Draw the B-H curve. 3

(c) The coil of a moving coil instrument is wound with 50 turns of wire. The flux density in the gap is 0.06 wb/m^2 and the effective length of the coil side in the gap is 4 cm. Find the force acting on each side of the coil when the current is 40 mA. 4

9. (a) Prove that the current in purely capacitive circuit leads the applied voltage by an angle 90° and draw their waveforms. Also calculate the average power of capacitive circuit. 5

(b) Find the Form Factor of the given waveform. 5



PART – II (Electronic)

GROUP A
(Multiple Choice Type Questions)

1. Answer any *five* questions. $5 \times 1 = 5$

(i) The temperature coefficient of resistance of a pure semiconductor is-

(A) negative (B) positive
(C) constant (D) none of these

(ii) Unit of diffusion constant for silicon in SI unit is

(A) $m^2 / V.s$ (B) m^2 / s
(C) m / s (D) V / s

(iii) If line frequency is 60 Hz, the output frequency of a bridge rectifier is

(A) 30 Hz (B) 60 Hz
(C) 120 Hz (D) 240 Hz

(iv) Without a DC source, a clipper acts like a

(A) rectifier (B) clamper
(C) chopper (D) demodulator

(v) Zener diodes are used as

(A) reference voltage elements (B) reference current elements
(C) reference resistance (D) both (A) and (B)

(vi) A transistor acts like a diode and a

(A) voltage source (B) current source
(C) power supply (D) resistance

(vii) A BJT is a

(A) voltage controlled device (B) current controlled device
(C) power controlled device (D) none of these

(viii) For an npn transistor, I_{CBO} approximately doubles for temperature rise of every

(A) 5°C (B) 7°C

(C) 10°C (D) none of these

GROUP B

(Short Answer Type Questions)

Answer any *two* questions. 2×5 = 10

2. Define the mobility of charge carriers in a semiconductor. Obtain expressions for the electrical conductivity of (i) an intrinsic, (ii) an n-type.

3. Draw the energy band diagram of a (a) forward biased pn junction diode, (b) reverse biased pn junction diode, (c) unbiased pn junction diode.

4. A full wave bridge rectifier is fed from a 15 V r.m.s. source and is connected across a 100 ohm load. Calculate PIV, RMS current draw from the supply and average D.C. current across the load.

5. What is meant by d.c. operating point or Q point in the context of transistor characteristics? What is load line? Why is transistor biasing necessary?

GROUP C

(Long Answer Type Questions)

Answer any *two* questions. 2×10 = 20

6. (a) Write the difference between metal, insulator and semiconductor. 4

(b) Why does extrinsic semiconductor behave as good conductor? 3

(c) What do you mean by depletion region of p-n junction diode? 3

7. (a) What is ripple factor? Evaluate the ripple factor and efficiency of full-wave rectifier. 5

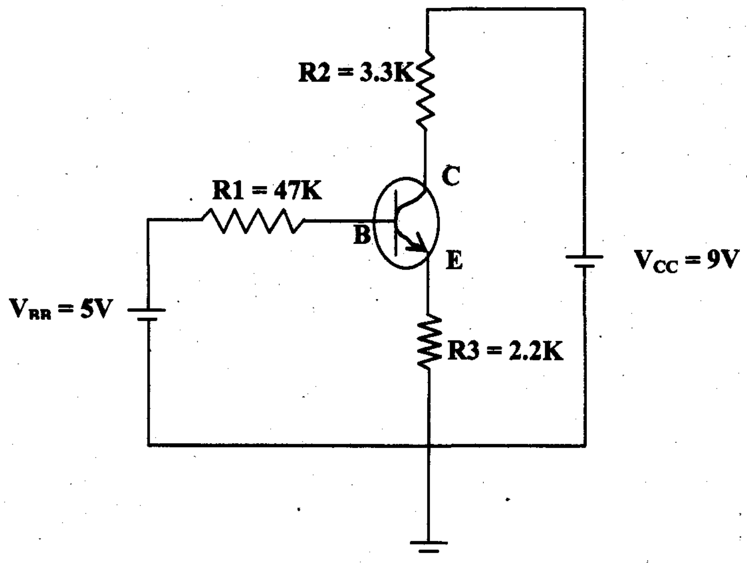
(b) A silicon diode having internal resistance $R_F = 30\ \Omega$ is used for half-wave rectification. The input ac voltage is $V_i = 6 \sin\omega t$ and load resistance is $500\ \Omega$. Find 5
(i) dc output voltage,
(ii) ac input power,
(iii) the efficiency of the rectifier.

8. (a) Explain why the collector region is larger than that of the emitter and base in a transistor? 3

(b) Why n-p-n and p-n-p transistors are called bipolar transistors? 2

(c) Show that the collector current is given by : $I_C = \beta I_B + (1 + \beta)I_{CO}$. 5

9. Draw the common-base input characteristics of a transistor. What is early effect? Refer to the following circuit $V_{BE,sat} = 0.85$ V and $V_{CE,sat} = 0.22$ V. If $h_{FE} = 110$, is the transistor operating in the saturation region? 3+2+5



HU-101

ENGLISH LANGUAGE AND TECHNICAL COMMUNICATION

Time Allotted: 3 Hours Full Marks: 70

The questions are of equal value.
The figures in the margin indicate full marks.
Candidates are required to give their answers in their own words as far as practicable.

GROUP A
(Multiple Choice Type Questions)

1. Answer all questions. 10×1 = 10

(i) The word 'extravagant' is the opposite of

(A) expensive (B) unlimited
(C) thrifty (D) proud

(ii) The synonym of the word 'tyranny' is

(A) misrule (B) cruelty
(C) tension (D) madness

(iii) To make the house more comfortable, we have decided to have central heating _________.

(A) installed (B) established
(C) settled (D) involved

(iv) To put down in black and white means

(A) to distinguish clearly (B) to write down on paper
(C) to seal the envelope (D) to tell the truth

(v) Synonym of 'auxiliary' is:

(A) conductive (B) associate
(C) capacity (D) authority

(vi) A place where government records are kept is called

(A) museum	(B) archives
(C) herbarium	(D) planetarium

(vii) The single word for 'a remedy for all diseases' is

(A) amnesty	(B) panacea
(C) amateur	(D) anarchy

(viii) One living with the same time with another is

(A) temporary	(B) contemporary
(C) permanent	(D) simultaneous

(ix) I request you to ____________ your crime. (Choose the most effective word to make a meaningful sentence)

(A) apologies	(B) agree
(C) confess	(D) pardon

(x) The Sun set before we reached the village. (Improve the sentence)

(A) had set	(B) would set
(C) would have set	(D) no improvement

GROUP B
(Short Answer Type Questions)

2. Read the following passage carefully and answer the questions that follow: 15

The function of the universities is not merely to send out technically skilled and professionally competent men, but it is their duty to produce in them the quality of compassion. The quality which enables the individuals to treat one another in a truly democratic spirit. Our religions have proclaimed from the very beginning that each human individuals is to be regarded as a spark of the divine. Tat twam asi, that art thou, is the teaching of the "Upanishads". The Buddhists declared that each individual has in him a spark of the divine and can become a

Bodhisattva. These proclamations by themselves are not enough. The minds and hearts of the people'require to be attended. We must strike to become democratic not merely in the political sense of the term but also in the social and economic sense. It is essential to bring about this democratic change, this democratic temper, the kind of outlook by a proper study of the humanities, including philosophy and religion. There is a great verse which says that in this poison-tree of Samsara are two fruits of incomparable value. They are the enjoyment of great books and the company of good souls. If we want to absorb the fruits of a great literature, we read them, not as we do cricket stories but read them with concentration. Our generations in its rapid travel has not achieved the habit of reading the great books and has lost the habit of being influenced by the great classics of our country. If these principles of democracy in our Construction are to become habits of mind and pattern of behavior, principles which change the very character of the individual and the nature of the society it can be done only by the study of great literature, of philosophy and religion. That is, even though our country needs great scientists, great technologists, great engineers, we should not neglect to make them humanists. While we retain science and technology, we remember that science and technology are not all.

(a) What, according to the author, is the duty of universities? 1

(b) How is compassion defined? 1

(c) Why is proper study of the humanities essential? 2

(d) What does the study of great literature achieve for us? 2

(e) Explain any *two* of the following in your own words: 2×2

(i) Truly democratic spirit

(ii) Cricket stories

(iii) Humanists

(f) Make précis of the passage given above and give a suitable title. 5

GROUP C
(Long Answer Type Questions)

Answer any *three* questions. $3 \times 15 = 45$

3. Write an essay of not more than 200 words on any *one* of the following topics: 15

(a) The technological miracles of the 21st century.

(b) Time as a tyrant of modern life.

(c) Is attainment of peace the ultimate goal of life?

4. Write a letter to the Editor of a local newspaper expressing your views against ragging and urging more stringent measures to stop this menace with reference to recent tragic death. 15

5. Write a report for a local newspaper on the two-day Seminar on "Industrialisation in West Bengal" hosted by your college, giving details of the presentation made by various eminent speakers on the subject. 15

6. You are the General Secretary of the Students' Union of your college. Your Principal has asked you to investigate the library facilities in your college. Prepare a letter report addressed to the Principal. 15

7. Comment on the moment of transformation of the thief in Ruskin Bond's short story 'The Thief'. 15

MAULANA ABUL KALAM AZAD UNIVERSITY OF TECHNOLOGY, WEST BENGAL

HU-101

ENGLISH LANGUAGE AND TECHNICAL COMMUNICATION

Time Allotted: 3 Hours   Full Marks: 70

The questions are of equal value.
The figures in the margin indicate full marks.
Candidates are required to give their answers in their own words as far as practicable.
All symbols are of usual significance.

GROUP A
(Multiple Choice Type Questions)

1. Answer *all* questions. 10×1 = 10

(i) The synonym of "ethnic" is

(A) coloured   (B) racial
(C) prejudiced   (D) tribal

(ii) The antonym of "global" is

(A) international   (B) worldwide
(C) zonal   (D) local

(iii) We went ______ a holiday to Gopalpur-on-Sea last December.

(A) on   (B) to
(C) around   (D) from

(iv) I wish you __________ that photo. It was a rare one of my grandparents' wedding in 1928.

(A) hasn't taken (B) haven't taken
(C) hadn't taken (D) won't have taken

(v) People usually _________ to the radio in the early 1980s.

(A) listen (B) listened
(C) had listened (D) have listened

(vi) If I _________ king, I would be the happiest man on earth.

(A) was (B) were
(C) am (D) be

(vii) I recommend that he ________ at once.

(A) apologize (B) apologizes
(C) apologized (D) apologizing

(viii) He succeeded _________ perseverance and sheer hard work.

(A) instead of (B) by dint of
(C) by reason of (D) in the event of

(ix) The victim ___________ me, "Could you show me the way?"

(A) commanded (B) suggested
(C) asked (D) adviced

(x) Despite the millions of dollars spent on improvements, the transport system in India remains ___________ and continues to __________ the citizens who depend on it.

(A) primitive....inconvenience (B) bombastic......upset
(C) suspicious.....connect (D) outdated....elate

GROUP B
(Short Answer Type Questions)

Answer *all* questions. 15

2. *Read the following passage carefully and answer the following questions in your words.*

If HD is today's revolution, 3D is going to be tomorrow's. There appear to be a slew of 3D products–3D televisions, 3D monitors, 3D video games, 3D movies, 3D glasses and even 3D goggles that add a third dimension to games you play on your iPhone or iPod–slowly popping up on store shelves globally, though very few are visible in the Indian market yet. 3D TVs are leading the pack.

Today's 3D displays are mainly stereoscopic models that alternately telecast images meant for the left and right eye stored in the even and odd fields, respectively, of the video signal. Or a shutter glass worn by the viewer alternately shuts off each eye, so that the eye sees only the video stream meant for it. The brain then follows its natural process of merging the two streams of data to present a 3D view to the user, just as it would merge the views of the right and left eye when we see a scene around us naturally.

While this was the basic technology behind 3D movies we saw decades ago, it has now come to our living rooms, thanks to the advanced display technology, lighter and stylish shutter glasses, and more capable broad cast techniques.

LED models, Plasma models, Special cameras for easy shooting of 3D content, real-time 2D-to-3D converters and more such technologies are coming up. Auto-stereoscopic models which do not require glasses are also expected in the future. In fact, mobile phone makers such as NTT Docomo, LG and HTC have already introduced devices that feature glassless 3D displays.

However, due to high price tags and insufficient content, 3D products are yet to catch up in India.

(Electronics For You, June 2011)

(a) Give examples of 3D products which are slowly popping up globally. Which product is leading the pack? 2

(b) What is the function of Shutter glasses? 2

(c) How does the brain present a 3D view to the user? 2

(d) Mention the upcoming technologies in the field of 3D. What are Auto stereoscopic models? 2

(e) Mention the present status of 3D products in India. 2

(f) Write a précis of the given passage. 5

GROUP C
(Long Answer Type Questions)

Answer any *three* questions. $3\times15=45$

3. An MNC company is looking for a competent software designer. Eligible candidates must apply with their CV to the HR of the company located at Bangalore within fifteen days from the date of advertisement. Salary would not be a bar for the desirable candidate. Draft an application (with a CV) in the form of a letter. 7+8

4. Do you feel that the attitude towards love and marriage has undergone any change with the change of time? Justify your points making a comparative study with "Marriage is a Private Affair". 6+9

5. You have purchased a vacuum cleaner, BPL make (Model: EURO 2020), from Vishal Electronics, Lal Bazaar. On trying to use it you find, however, it refuses to start. Write a letter of complaint to the firm complaining about the malfunctioning product. 15

6. Write an essay in around 300 words on any *one* of the following topics: 15

(a) Electronic Gadgets in Our Everyday Lives

(b) Efficacy of "Clean India" campaign

(c) The Uses of Paper

7. Earthcare Books is organizing a programme of "Meet Your Favourite Author" on 23rd December, featuring the children's writer, Ruskin Bond, in which over 200 children from various schools will participate. The Manager is calling a meeting on 15th November at 3 p.m. to discuss the organization of this event. Draft the e-mail he sends his employees with a brief agenda of the forthcoming meeting. 15

8. Imagine that you are starting a Web Café with scanning, printing and word processing facilities. It is going to be the only web café of its kind in the area. Moreover, you are providing 25% rebate to the first 200 customers. As the General Manager of the café, write a sales letter to be distributed to your prospective customers. 15

M-101

MATHEMATICS-I

Time Allotted: 3 Hours Full Marks: 70

The questions are of equal value.
The figures in the margin indicate full marks.
Candidates are required to give their answers in their own words as far as practicable.

GROUP A
(Multiple Choice Type Questions)

1. Answer any *ten* questions. 10×1 = 10

(i) $\int_0^{\pi/2} \sin^5\theta\, d\theta =$

(A) $\frac{8}{15}$ (B) $\frac{8\pi}{15}$ (C) $\frac{8}{15}$ (D) $\frac{4}{15}$

(ii) If $u(x,y) = yf(\frac{x^2}{y^2})$ then $x\frac{\delta u}{\delta x} + y\frac{\delta u}{\delta y} =$

(A) 0 (B) $2u(x,y)$ (C) $u(x,y)$ (D) 2

(iii) The value of $\int_c (xdx - dy)$, where c is the line joining (0,1) to (1,0) is

(A) $\frac{3}{2}$ (B) $\frac{1}{2}$ (C) 0 (D) $\frac{2}{3}$

(iv) Component of the vector $2\hat{i} + 5\hat{j} + 7\hat{k}$ on $\hat{i} - 2\hat{j} + 2\hat{k}$ is

(A) $\sqrt{78}$ (B) 3 (C) 6 (D) 2

(v) The value of t for which $(x+3y)\hat{i} + (y-2z)\hat{j} + (x+tz)\hat{k}$ is solenoidal is

(A) 2 (B) −2 (C) 0 (D) 1

(vi) If $x = r\cos\theta$ and $y = r\sin\theta$ then $\dfrac{\partial(r,\theta)}{\partial(x,y)} =$

(A) r (B) 1 (C) $\dfrac{1}{r}$ (D) 0

(vii) $f(x,y) = \dfrac{x^3+y^3}{\sqrt{x^2+y^2}}$ is a homogeneous function of degree

(A) 0 (B) 2 (C) 1 (D) $\dfrac{1}{2}$

(viii) If $A = \begin{bmatrix} -1 & 1 & -1 \\ 3 & -3 & 3 \\ 5 & -5 & 5 \end{bmatrix}$ then A is

(A) idempotent (B) nilpotent (C) involutary (D) none of these

(ix) If $y = \tan^{-1} x$ then

(A) $(1+x^2)y_1 = 1$ (B) $(1+x^2)y_2 = 1$ (C) $(1+x^2)y_1 = 0$ (D) $(1+x^2)y_1 = 2$

(x) If A is real skew-symmetric matrix such that $A^2 + 1 = 0$, then A is

(A) singular (B) unit matrix (C) orthogonal (D) none of these

(xi) The Sequence $\left\{1, \frac{1}{3}, \frac{1}{3^2}, \ldots, \frac{1}{3^n}, \ldots \infty\right\}$ is

(A) divergent (B) oscillatory (C) convergent (D) none of these

(xii) For a function $f(x)$ the expression $\dfrac{h^n(1-\theta)^{(n-1)}}{(n-1)!} f^n(a+\theta h)$ is known as

(A) Lagrange's remainder (B) Cauchy's remainder

(C) Maclaurin's remainder (D) Taylor's remainder

GROUP B
(Short Answer Type Questions)

Answer any *three* questions. $3\times5 = 15$

2. Using Laplace's method of expansion, prove that :

$$\begin{vmatrix} x & y & -u & -v \\ y & x & v & u \\ u & v & x & y \\ -v & -u & y & x \end{vmatrix} = (x^2+v^2-y^2-u^2)^2$$

3. For what values of x is the following infinite series convergent?

$$\sum_{n=1}^{\infty} \frac{(n+1)^n x^n}{n^{(n+1)}} \quad (x>0).$$

4. If α, β, γ are the angles which a vector makes with the co-ordinate axes, prove that $\sin^2\alpha + \sin^2\beta + \sin^2\gamma = 2$.

5. If $y = x^{n-1}\log x$, using Leibnitz's theorem show that $y_n = \dfrac{(n-1)!}{x}$.

6. Using Green's theorem evaluate $\oint_c \{(\cos x\sin y - xy)dx + \sin x\cos y\, dy\}$ where c is the circle $x^2+y^2=1$.

GROUP C
(Long Answer Type Questions)

Answer any *three* questions. $3\times15 = 45$

7. (a) If $A = \begin{bmatrix} 1 & 2 & 2 \\ 2 & 1 & 2 \\ 2 & 2 & 1 \end{bmatrix}$ then show that $A^2 - 4A - 5I_3 = 0$. Hence find A^{-1}. 5

(b) If $y = e^{m\sin^{-1}x}$, then show that 5

(i) $(1-x^2)y_2 - xy_1 - m^2y = 0$ (ii) $(1-x^2)y_{n+2} - (2n+1)xy_{n+1} - (n^2+m^2)y_n = 0$

Also find $(y_n)_0$.

(c) Is Rolle's Theorem applicable to the function $f(x) = (x-p)^m(x-q)^n$, $x \in [p,q]$, where m, n are positive integers? If so, find the constant c of Rolle's Theorem, where c has its usual meaning. 5

8. (a) State D'Alembert's Ratio test. Applying this test, examine the convergence of the following series 2+3

$$1+\frac{2^a}{2!}+\frac{3^a}{3!}+\frac{4^a}{4!}.......\infty \quad (a>0)$$

(b) Show that $\left[\vec{a}+\vec{b},\vec{b}+\vec{c},\vec{c}+\vec{a}\right]=2\left[\vec{a},\vec{b},\vec{c}\right]$. 4

(c) If $f(v^2-x^2, v^2-y^2, v^2-z^2)=0$, where v is a function of x, y, z then show that 6

$$\frac{1}{x}\frac{\partial v}{\partial x}+\frac{1}{y}\frac{\partial v}{\partial y}+\frac{1}{z}\frac{\partial v}{\partial z}=\frac{1}{v}$$

9. (a) Determine the conditions under which the system of equations 5

$x+y+z=1 \qquad x+2y-z=k \qquad 5x+7y+az=K^2$

admits (i) only one solution, (ii) no solution, (iii) many solutions.

(b) Verify the divergence theorem for the vector function $\vec{F}=4xz\hat{i}-y^2\hat{j}+yz\hat{k}$ taken over a cube bounded by $x=0, x=1; y=0, y=1; z=0, z=1$. 6

(c) If $I_n=\int_0^{\pi/2} x^n \sin x\, dx, (n>1)$ then prove that $I_n+n(n-1)I_{n-2}=n\left(\frac{\pi}{2}\right)^{n-1}$ 4

10.(a) Verify Lagrange's Mean Value Theorem at [–1, 1] for 5

$$f(x)=x\sin\frac{1}{x}, \quad x\neq 0$$
$$=0, \quad x=0$$

(b) If $u=xf\left(\frac{y}{x}\right)+g\left(\frac{y}{x}\right)$ then show that 5

$$x\frac{\partial u}{\partial x}+y\frac{\partial u}{\partial y}=xf\left(\frac{y}{x}\right) \text{ and } x^2\frac{\partial^2 u}{\partial x^2}+2xy\frac{\partial^2 u}{\partial x\partial y}+y^2\frac{\partial^2 u}{\partial y^2}=0.$$

(c) Find the rank of the following matrix $\begin{bmatrix}2 & 3 & 16 & 5\\ 4 & 5 & 6 & 7\\ 2 & 0 & 1 & 3\\ 8 & 8 & 23 & 15\end{bmatrix}$ 5

11. (a) Find the extremum of the following function : x^3+y^3-3axy. 5

(b) Show that $\nabla\phi$ is irrotational, where $\phi=x^2y+2xy+z^2$. 5

(c) Evaluate $\int_0^a\int_0^x\int_0^y x^3y^2z\, dzdydx$. 5

MAULANA ABUL KALAM AZAD UNIVERSITY OF TECHNOLOGY, WEST BENGAL

M-101

MATHEMATICS-I

Time Allotted: 3 Hours Full Marks: 70

The questions are of equal value.
The figures in the margin indicate full marks.
Candidates are required to give their answers in their own words as far as practicable.
All symbols are of usual significance.

GROUP A
(Multiple Choice Type Questions)

1. Answer any *ten* questions. $10\times1 = 10$

(i) If 2, 3, 5 are the three eigenvalues of a 3rd order matrix A, then the value of det (A) is

(A) 30 (B) –30
(C) 0 (D) none of these

(ii) The matrix $\begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}$ is

(A) symmetric (B) skew-symmetric
(C) singular (D) orthogonal

(iii) 0 is an Eigenvalue of matrix if the matrix is

(A) non-singular	(B) orthogonal
(C) skew-symmetric	(D) singular

(iv) The function $f(x) = |x-2|$ satisfies Rolle's Theorem in the interval

(A) [3, 4]	(B) [0, 4]
(C) [–3, 3]	(D) [–1, 4]

(v) The value of $\int_0^{\frac{\pi}{2}} \sin^8 x\,dx = ?$

(A) $\frac{35}{128}$	(B) $\frac{35}{256}$
(C) $\frac{35\pi}{128}$	(D) $\frac{35\pi}{256}$

(vi) $\frac{\partial(x^y)}{\partial y} = ?$

(A) x^y	(B) $x^y \log y$
(C) $x^y \log x$	(D) does not exist

(vii) If $x = r\cos\theta$ and $y = r\sin\theta$ then $\frac{\partial(r,\theta)}{\partial(x,y)} = ?$

(A) r	(B) 1
(C) $\frac{1}{r}$	(D) none of these

(viii) The necessary condition that (a, b) is a ________ point of $f(x, y)$ if $f_x(a,b) = 0 = f_y(a,b)$

(A) maximum	(B) stationary
(C) saddle point	(D) minimum

(ix) The series $\sum_{n=1}^{\infty} \frac{1}{n^p}$ is convergent if

(A) $p < 1$ (B) $p > 0$
(C) $p > 1$ (D) $p < 0$

(x) The series $1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \ldots$ is

(A) absolutely convergent (B) conditionally convergent
(C) oscillatory (D) none of these

(xi) The value of λ for which the vector function $\vec{f} = (x+3y)\hat{i} + (y-2z)\hat{j} + (x+\lambda z)\hat{k}$ is solenoidal is

(A) –2 (B) 1
(C) 3 (D) 2

(xii) In the mean value theorem $f(h) = f(0) + hf'(\theta h)$, $0 < \theta < 1$, if $f(x) = \frac{1}{1+x}$ and $h = 3$, then the value of θ is

(A) 1 (B) $\frac{1}{3}$
(C) $\frac{1}{\sqrt{2}}$ (D) none of these

GROUP B
(Short Answer Type Questions)

Answer any *three* questions. $3\times5 = 15$

2. If $y = e^{m\sin^{-1}x}$, then prove that $(1-x^2)y_{n+2} - (2n+1)xy_{n+1} - (n^2+m^2)y_n = 0$. Find y_n for $x = 0$. 5

3. Using M.V.T. prove that $1+\frac{x}{2\sqrt{1+x}}<\sqrt{1+x}<1+\frac{x}{2}$. 5

4. Expanding the determinant by Laplace's method in terms of minors of 2nd order formed from the first two rows. 5

5. Using M.V.T. prove that $x>\tan^{-1}x>\frac{x}{1+x^2},\ 0<x<\frac{\pi}{2}$. 5

6. State D' Alembert's ratio test for convergence of an infinite series. Examine the convergence and divergence of the series. 5

$$1+\frac{x}{2}+\frac{x^2}{5}+\frac{x^3}{10}+\ldots\ldots\ldots\infty$$

GROUP C
(Long Answer Type Questions)

Answer any *three* questions. $3\times15=45$

7. (a) Expanding the determinant by Laplace's method in terms of minors of 2nd order formed from the first two, prove that 5

$$\begin{vmatrix} 0 & a & b & c \\ -a & 0 & d & e \\ -b & -d & 0 & f \\ -c & -e & -f & 0 \end{vmatrix}=(af-be+cd)^2.$$

(b) Find the eigenvalues and the corresponding eigenvectors of the matrix 5

$$A=\begin{pmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 0 & 0 & 1 \end{pmatrix}.$$

(c) If $y=(x^2-1)^n$, then prove that $(x^2-1)y_{n+2}+2xy_{n+1}-n(n+1)y_n=0$. 5

8. (a) If $u = \phi(x, y)$, and $x = r\cos\theta$, $y = r\sin\theta$ then if the variables are changed from x, y to r, θ then show that 6

(i) $\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial u}{\partial y}\right)^2 = \left(\frac{\partial u}{\partial r}\right)^2 + \frac{1}{r^2}\left(\frac{\partial u}{\partial \theta}\right)^2$

(ii) $\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = \frac{\partial^2 u}{\partial r^2} + \frac{1}{r}\frac{\partial u}{\partial r} + \frac{1}{r^2}\frac{\partial^2 u}{\partial \theta^2}$.

(b) Prove that the series $x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + ...(-1)^{n+1}\frac{x^n}{n} + ...\infty$ is absolutely convergent when $|x| < 1$ and conditionally convergent when $x = 1$. 5

(c) Prove that the series $\sum_{n=1}^{\infty}\frac{1}{n^p}$ converges for $p > 1$ and diverges for $p \le 1$. 4

9. (a) If $u = \tan^{-1}\frac{x^2 + y^2}{x - y}$, show that $x\frac{\partial u}{\partial x} + y\frac{\partial u}{\partial y} = \frac{1}{2}\sin 2u$ 5

(b) $\begin{vmatrix} 1 & \alpha & \alpha^2 & -\beta\gamma \\ 1 & \beta & \beta^2 & -\gamma\alpha \\ 1 & \gamma & \gamma^2 & -\alpha\beta \end{vmatrix} = 0$ 5

(c) If $u = x^2 - 2y$, $v = x + y + z$, $w = x - 2y + 3z$, find $\frac{\partial(u, v, w)}{\partial(x, y, z)}$. 5

10.(a) Show that $\vec{\nabla} r^n = n r^{n-2}\,\vec{r}$, where $\vec{r} = \hat{i}x + \hat{j}y + \hat{k}z$ 5

(b) Evaluate $\iint\sqrt{4x^2 - y^2}\,dxdy$ over the triangle formed by the straight lines $y = 0$, $x = 1$ and $y = x$. 5

(c) Verify Stokes theorem for $\vec{F} = (2x - y)\hat{i} - yz^2\hat{j} - y^2z\hat{k}$, where S is the upper half surface of the sphere $x^2 + y^2 + z^2 = 1$ and C is its boundary. 5

11.(a) Show that $\vec{f} = (6xy + z^2)\hat{i} + (3x^2 - z)\hat{j} + (3xz^2 - y)\hat{k}$ is irrotational. Hence find a scalar function ϕ such that $\vec{f} = \nabla\phi$. 5

(b) Given that 6

$$f(x,y) = \begin{cases} \dfrac{xy(x^2 - y^2)}{x^2 + y^2}, & \text{for} \quad (x,y) \neq (0,0) \\ 0, & \text{for} \quad (x,y) = (0,0). \end{cases}$$

Show that $f_{xy}(0,0) \neq f_{yx}(0,0)$.

(c) Evaluate 4

$$\int_C (3xy\,dx - y^2 dy)$$

Where C is the arc of the parabola $y = 2x^2$ from (0, 0) to (1, 2).

ME-101

ENGINEERING MECHANICS

Time Allotted: 3 Hours Full Marks: 70

The questions are of equal value.
The figures in the margin indicate full marks.
Candidates are required to give their answers in their own words as far as practicable.

GROUP A
(Multiple Choice Type Questions)

1. Answer any *ten* questions. 10×1 = 10

(i) The work done against any conservative forces is stored in the body in the form of

(A) energy (B) potential energy (C) elastic energy (D) strain energy

(ii) The magnitude of two forces, which when acting at right angle produce resultant force of $\sqrt{10}$ kg and when acting at 60° produce resultant of $\sqrt{13}$ kg. These forces are

(A) 2 and $\sqrt{6}$ kg (B) 3 and 1 (C) $\sqrt{5}$ and $\sqrt{5}$ (D) 2 and 5

(iii) If three forces acting in one plane upon a rigid body, keep it in equilibrium, then they must either

(A) meet in a point (B) be all parallel
(C) at least two of them must meet (D) all the above are correct

(iv) A projectile is fired at an angle θ to the vertical. Its horizontal range will be maximum when θ is

(A) 0° (B) 30° (C) 45° (D) 60°

(v) Varignon's theorem is related with

(A) moment of forces(s) (B) friction
(C) deformation characteristics of rigid bodies (D) none of the above

(vi) Strain energy is the

(A) maximum energy which can be stored in a body
(B) energy stored in a body when stressed to the elastic limit
(C) energy stored in a body when stressed to the breaking point
(D) none of these

(vii) The CG of a solid hemisphere lies on the central radius

(A) at distance 3r/2 from the plane base (B) at distance 3r/4 from plane base
(C) at distance 3r/5 from the plane base (D) at distance 3r/8 from plane base

(viii) If **i** and **j** are two Cartesian unit vectors then

(A) **i**·**j** = 1 (B) **i**·**j** = 0 (C) **i**·**j** = 2 (D) none of these

(ix) An elevator weighing 980 N attains an upward velocity of 3m/s in 3 s following a uniform acceleration. The tension in the cable that supports the elevator is

(A) 1000 N (B) 1080 N (C) 880 N (D) 1150 N

(x) If momentum of a body is doubled, its kinetic energy will

(A) get doubled (B) get halved (C) remain same (D) get quadrupled

(xi) The condition of equilibrium of co-planar non-concurrent forces are

(A) $\sum F_x = 0 ; \sum F_y = 0$ (B) $\sum F_x = 0 ; \sum F_y = 0 ; \sum M = 0$
(C) $\sum F_y = 0 ; \sum M = 0$ (D) $\sum F_x = 0 ; \sum M = 0$

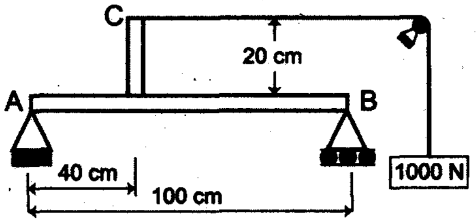
(xii) The equation of motion of a particle is $S = 2t^3 - t^2 - 2$ where S is the displacement in metres and t is time in seconds. The acceleration of the particle after 1 second will be

(A) 8 m/s^2 (B) 9 m/s^2 (C) 10 m/s^2 (D) 5 m/s^2

GROUP B
(Short Answer Type Questions)

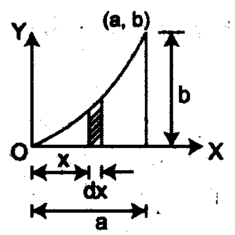
Answer any *three* questions. 3×5 = 15

2. A string is connected at point C of a structure AB, passing through a frictionless pulley and at the free end of the string a weight is suspended as shown in Figure. Determine the reaction forces developed at point A and B. Neglect the mass of the structure AB. 5



3. What is meant by toughness? What is meant by resilience? Draw a stress-strain diagram of a mild steel specimen and show region of modulus of toughness and modulus of resilience. 1+1+3

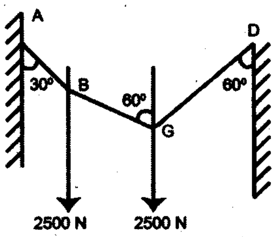
4. By integration determine the co-ordinate of the centroid of the plane area under the curve $y = kx^2$ and x axis, between (0, 0) and (a, b) of the given Figure. 5



5. Given a force F = 10i + 5j + Ak N. If this force is to have a rectangular component of 8 N along a line having unit vector r = 0.6i + 0.8k, what should be the value of A? What is the angle between F and r? 3+2

6. (a) State Lami's theorem. 1+4

(b) Two equal loads of 2500 N are supported by a flexible string ABCD at points B and D as shown in the figure. Find the tensions in the portions AB, BC, CD of the string.



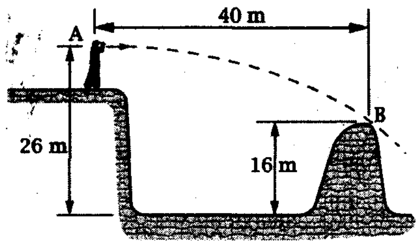
GROUP C

(Long Answer Type Questions)

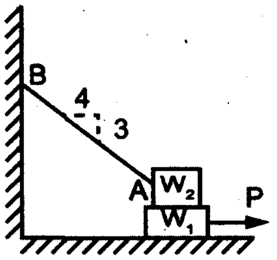
Answer any *three* questions. 3×15 = 45

7. (a) A force of 200 N is directed along the line drawn from the point P(5,2,4) to the point Q(3,–5,6). Determine the moment of this force about a point A(4,3,2). The distances are in meters. 7

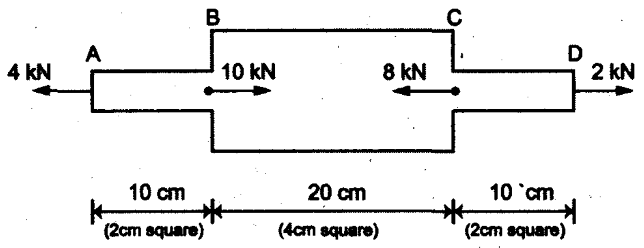
(b) Reference to Figure, with what minimum horizontal velocity *u* can a boy throw a rock at A and have it just clear the obstruction at B? 8



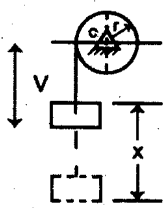
8. (a) A block of weight W_1 = 200 kgf rests on a horizontal surface and supports on top of it another block of weight W_2 = 50 kgf. The block W_2 is attached to a vertical wall by the inclined string AB. Find the magnitude of the horizontal force P applied to the lower block as shown in Figure, which will be necessary to cause slipping to impend the coefficient of static friction for all contiguous surfaces which is $\mu = 0.3$. 8



(b) A steel rod ABCD of stepped section is loaded as shown in figure. The loads are assumed to act along the centre line of the rod. Estimate the displacement of *D* relative to *A*. Assume *E* = 2×10^5 N/mm^2. 7

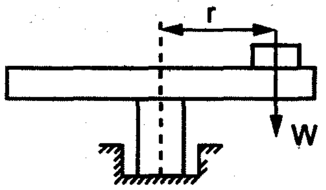


9. (a) Determine the velocity V of the falling weight W of the system as shown in figure, as a function of displacement from the initial position of rest. Assume weight of the cylinder as $2W$. 5

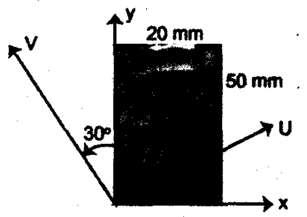


(b) Prove that the volumetric strain of a rectangular bar is the algebraic sum of strains of length, width and height. 5

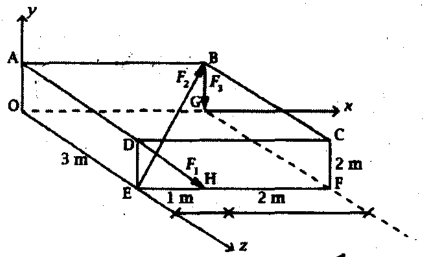
(c) A small block of weight W rests on a horizontal turntable at a distance r from the axis of rotation as shown in figure. If the coefficient of friction between the block and surface of the turntable is μ, find the maximum uniform speed that the block can have due to rotation of the turntable without slipping off. 5



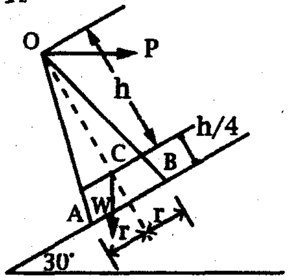
10.(a) For a rectangle shown in Figure, compute I_u, I_v and I_{uv} with respect to u-v axes inclined to x-y axes by 30°. Determine principal axes and second moment of area about the principal axes. 7



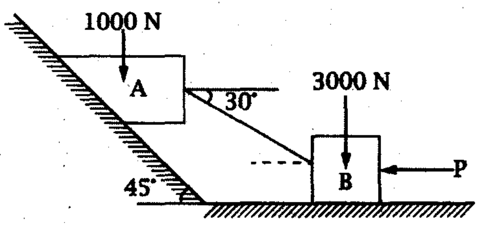
(b) The forces F_1, F_2 and F_3 act on the box as shown in Figure 10. The magnitude of the given forces are 19 N, 23 N and 46 N respectively. Determine the resultant of the forces and its magnitude. 8



11.(a) A solid right circular cone of altitude h = 12 cm and radius r = 3 cm has its cg C on its geometric axis at a distance h/4 above the base. This cone rests on the inclined plane AB which makes an angle of 30° with the horizontal and for which the angle of friction is 0.5. A horizontal force P is applied to the vertex O of the cone and acts in the vertical plane of the figure. Find the maximum and minimum values of P consistent with equilibrium of the cone of weight W = 10 kgf. 8



(b) A block A weighing 1000 N rests on a rough inclined plane whose inclination to the horizontal is 45°. The block is connected to another block B weighing 3000 N resting on a rough horizontal plane, by a weightless rigid bar inclined at an angle 30° to the horizontal as shown in figure. Find the horizontal force that has to be applied on the block B to just move the block A up the slope. Assume coefficients of friction for all contact surfaces is 0.26. 7



MAULANA ABUL KALAM AZAD UNIVERSITY OF TECHNOLOGY, WEST BENGAL

ME-101

ENGINEERING MECHANICS

Time Allotted: 3 Hours Full Marks: 70

The questions are of equal value.
The figures in the margin indicate full marks.
Candidates are required to give their answers in their own words as far as practicable.
All symbols are of usual significance.

GROUP A
(Multiple Choice Type Questions)

1. Answer *all* questions. 10×1 = 10

(i) A number of forces acting at a point will be in equilibrium if

(A) their total sum is zero
(B) two resolved parts in two directions at right angles are equal
(C) sum of resolved parts in any two perpendicular directions are both zero
(D) all of them are inclined equally
(E) none of these

(ii) Given $\vec{F}_1 = 5\hat{j} + 4\hat{k}$ and $\vec{F}_2 = 3\hat{i} + 6\hat{k}$. The magnitude of the scalar product of these vectors is

(A) 15 (B) 12 (C) 24 (D) 30

(iii) The work done against any conservative force is stored in the body in the form of

(A) energy (B) potential energy

(C) elastic energy (D) strain energy

(iv) Equation of motion of a particle is $S = 2t^3 - t^2 - 2$, where S is displacement in meter and t is in sec. Acceleration of the particle after 1 sec will be

(A) 8 m/sec^2 (B) 9 m/sec^2 (C) 10 m/sec^2 (D) 5 m/sec^2

(v) The values of $\hat{i} \cdot \hat{i}$ and $\hat{i} \times \hat{i}$ are

(A) 1 and 0 (B) 1 and 1 (C) 0 and 0 (D) 0 and 1

(vi) A body is resting on a plane inclined at angle of 30° to horizontal. What force would be required to slide it down, if the coefficient of friction between body and plane is 0.3

(A) zero (B) 1 kg

(C) 5 kg (D) would depend on weight of body

(vii) Null vector is known as

(A) negative vector (B) unit vector

(C) zero vector (D) all of these

(viii) Relative velocity of $\vec{A}$ with respect to $\vec{B}$ is defined as

(A) $\vec{V}_{A|B} = \vec{V}_B - \vec{V}_A$ (B) $\vec{V}_{A|B} = \vec{V}_A - \vec{V}_B$

(C) $\vec{V}_{A|B} = \vec{V}_B + \vec{V}_A$ (D) None of these

(ix) Frictional force has the following relation with the normal reaction between the two connecting surfaces

(A) $F = \mu N$ (B) $F = \mu^2 N$ (C) $F = \mu/N$ (D) None of these

(x) Three forces $\sqrt{3}p$, p and $2p$ acting on a particle are in equilibrium. If the angle between the first and second be 90°, the angle between the second and the third will be

(A) 30° (B) 60° (C) 120° (D) 150°

GROUP B
(Short Answer Type Questions)

Answer any *three* questions. $3 \times 5 = 15$

2. A ball of weight Q = 120N rests in trough ABC, as shown in Fig. A. Determine the forces exerted on the sides, on the vertical face at D and on the inclined face at E, if all the surfaces are perfectly smooth. 5

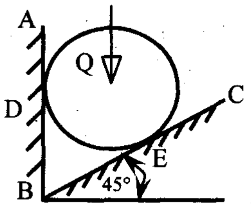


Fig A

3. The line of action of the 500-N force runs through the points A(–7, –2) and B(8, 6) as shown in Fig. B. Find the 'x' and 'y' scalar components of force F. 5

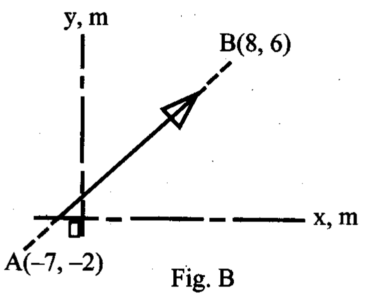


Fig. B

4. The acceleration of a particle is given by '$a = 4t - 30$', where 'a' is in m/sec^2 and t is in seconds. Determine the velocity and displacement as functions of time. The initial displacement at $t = 0$ is $s_0 = -5$m and initial velocity is $v_0 = 3$m/s. 5

5. A force given by F = 3i + 2j – 4k is applied at the point P (1, –1, 2). Find the moment of the force F about the point O (2, –1, 3) and about origin. 3+2

6. Two blocks of weights P and Q are connected by a flexible but inextensible cord and supported as shown in Fig. C. If the co-efficient of friction between 5

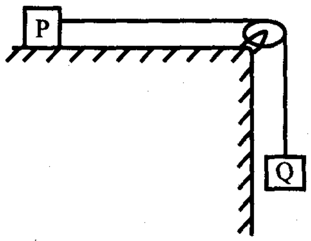


Fig. C

the block P and the horizontal surface is μ (mu) and all other friction is negligible, find (a) the acceleration of the system and (b) the tensile force S in the cord. The following numerical data are given P = 53.4 N; Q = 26.7 N $\mu = \frac{1}{3}$.

7. In Fig. D, a lever is attached to a spindle by means of a square key 6 mm × 6 mm by 2.5 cm long. If the average shear stress in the key is not to exceed 700 N/Cm^2, what is the safe value of the load P applied to the end of the lever? 5

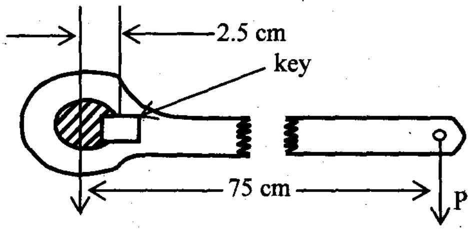


Fig. D

GROUP C
(Long Answer Type Questions)

Answer any *three* questions. $3 \times 15 = 45$

8. (a) A force F of 100 kN is applied at the origin O of the axes x-y-z. The line of action of F passes through a point A whose coordinates are 3 m, 4 m and 5 m. Find (i) x, y and z scalar components of F, (ii) the projection F_{XY} of F on the x-y plane and (iii) the projection F_{OB} of F along the line OB. 2.5+2.5+2

(b) Weight W and 2W are supported in a vertical plane by a string and pulleys arranged as shown in Fig. E. Find the magnitude of an additional weight Q applied on the left which will give a downward acceleration $a = 0.1\,g$ to the weight W. Neglect friction and inertia of pulleys. 8

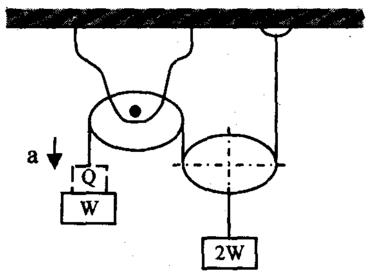


Fig. E

9. (a) Given initial velocity v_o and angle of projection θ of a projectile. Find the equation that defines y as a function of x. Eliminate time from the kinematic equation. 5

(b) A ball is dropped vertically on to a 20° inclined plane at 'A'. The direction of rebound forms an angle of 35° with vertical. Knowing that the ball strikes the inclined plane at 'B' as shown in Fig. F. Determine 10

(i) The velocity of rebound at 'A'

(ii) The time required for the ball to travel from A to B.

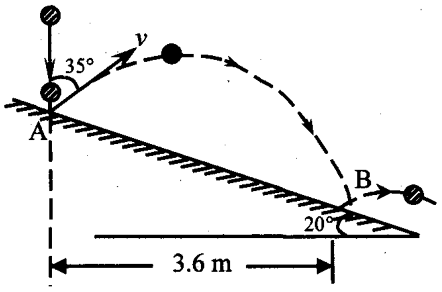


Fig. F

10.(a) Determine the centroid of the area shown in the Fig. G with respect to the axis shown. 8

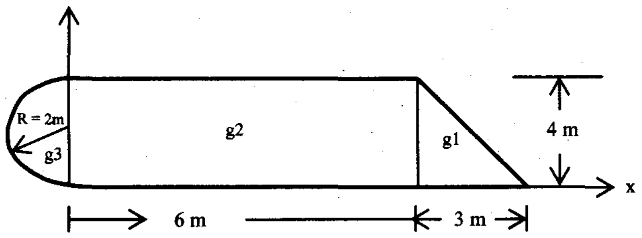


Fig. G

(b) Find the centroid of the unequal angle 200 * 150 * 12 mm, shown in Fig. H 7

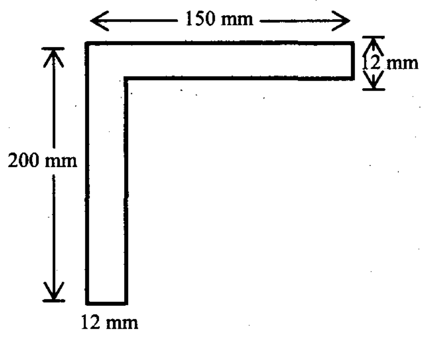


Fig . H

11.(a) A cable supporting a 6 m high vertical post. The post is anchored to the ground as shown in Fig. I. If the tensile force in the cable is 15 kN, find its moment about z-axis passing through the base of the post. 7

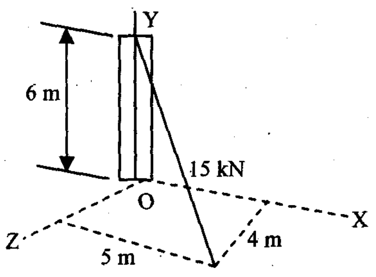


Fig. I

(b) Three flat blocks are positioned on the 30° incline as shown in Fig. J and a force P parallel to the incline is applied to the middle block. The upper block is prevented from moving by a wire which attaches it to the fixed support. The coefficient of static friction for each of the three pairs of mating surfaces is shown. Determine the maximum value of P which may have before any slipping takes place. 8

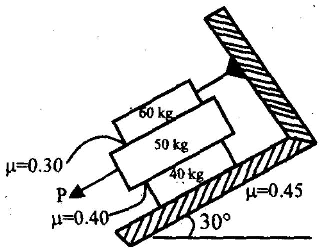


Fig. J

12.(a) A body having mass 5 kg is released at point A from rest, down the incline shown in the Fig. K. Find the velocity when the body will reach to position B, using work energy principle. 6

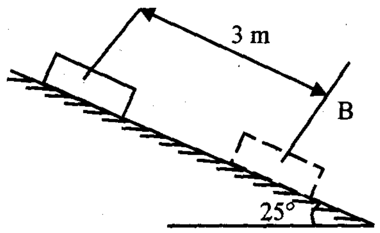


Fig. K

(b) A right circular cylindrical tank containing water spins about its vertical geometric axis oo′ at such speed that the free water surface is a paraboloid ACB, as shown in Fig. L. What will be the depth of water in the tank when it comes to rest? 9

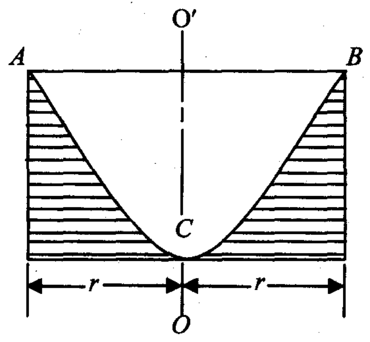


Fig. L

PH-101

PHYSICS -I

Time Allotted: 3 Hours   Full Marks: 70

The questions are of equal value.
The figures in the margin indicate full marks.
Candidates are required to give their answers in their own words as far as practicable.

GROUP A
(Multiple Choice Type Questions)

1. Answer any *ten* questions. $10 \times 1 = 10$

(i) Superposition of two mutually perpendicular SHMs of equal time period and equal amplitude and phase difference π form

(A) ellipse (B) circle
(C) straight line (D) parabola

(ii) The amount of power supplied to a system is equal to the rate of dissipation of energy in

(A) forced vibration (B) damped vibration
(C) simple harmonic motion (D) oscillatory motion

(iii) For large values of damping constant the Q-factor

(A) increases (B) decreases
(C) remains same (D) becomes zero

(iv) Dispersive power of grating increases with increase of

(A) number of lines per centimeter (B) order number
(C) number of lines and order number (D) intensity of incident light

(v) In Young's double slit experiment using two identical slits, the intensity of the maximum at the centre of the screen is '*I*'. What will be the intensity at the centre of the screen if one on the slits is closed?

(A) I (B) $2I$

(C) $\frac{I}{4}$ (D) $\frac{I}{2}$

(vi) A nicol prism can act as a

(A) polarizer (B) both polarizer and analyzer

(C) analyzer (D) laser

(vii) Emission of photon due to transition of an electron from higher to lower energy level caused external energy is known as

(A) stimulated absorption (B) spontaneous emission

(C) stimulated emission (D) population inversion

(viii) In Ruby laser the host crystal is

(A) $CaCO_3$ (B) MnO_2

(C) Al_2O_3 (D) Fe_3O_4

(ix) The Compton shift is maximum when scattering angle is

(A) 45° (B) 90°

(C) 180° (D) 60°

(x) In holography, three dimensional image of the object may be produced

(A) by using single lens (B) by using a mirror and a lens

(C) by using two lenses (D) without using lens

(xi) de Broglie wavelength of a gas molecule at temperature T is:

(A) $\frac{h}{\sqrt{3mkT}}$ (B) $\frac{2h}{\sqrt{3mkT}}$

(C) $\frac{h^2}{\sqrt{3mkT}}$ (D) $\frac{h/2}{\sqrt{3mkT}}$

(xii) The Compton shift and Compton wavelength λ_c of a particle are equal if the angle of scattering is

(A) 0° (B) 90°
(C) 180° (D) 45°

(xiii) According to Rayleigh-Jean's Law, the energy density of a blackbody radiation is

(A) inversely proportional to the fourth power of wavelength
(B) directly proportional to the fourth power of wavelength
(C) inversely proportional to the fifth power of wavelength
(D) directly proportional to the fifth power of frequency

(xiv) If a and r be respectively the lattice constant and radius of an atom in a BCC structure then,

(A) $r = \frac{\sqrt{3}}{4}a$ (B) $r = \frac{\sqrt{2}}{4}a$
(C) $r = a$ (D) $r = \frac{a}{2}$

(xiv) If λ_L and λ_K are characteristic wavelengths belonging to L and K lines in X ray spectrum, then

(A) $\lambda_{L.} > \lambda_K$ (B) $\lambda_{L.} < \lambda_K$
(C) $\lambda_{L.} = \lambda_K$ (D) $\lambda_{L.} = 4\lambda_K$

GROUP B
(Short Answer Type Questions)

Answer any *three* questions. $3\times5 = 15$

2. (a) What are Miller Indices? Find the Miller Indices of a crystal plane whose intercepts are [a, 2b, c] in simple cubic crystal. 1+2+2
   (b) What is ultraviolet catastrophe?

3. (a) Distinguish between spontaneous and stimulated emission? 3+2
   (b) State Moseley's Law.

4. (a) What do you mean by Q-factor of a damped oscillator? Derive the relation between Q-factor and relaxation time. 3+2
   (b) If the natural angular frequenecy of a simple harmonic oscillator of mass 2g is 0.8rad.s^{-1}. It undergoes critically damped motion when taken to a viscous medium. Find the damping force on the oscillator when its speed is 0.2cm.s^{-1}.

5. (a) In a Newton's ring experiment the diameter of 5th dark ring is 0.366 cm and the diameter of the 15th dark ring 0.590 cm. Find the radius of the plano-convex lens if the wave length of light used is 5890 A^0. 2+3
   (b) A monochromatic radiation of wavelength 2×10^{-12} m is incident on free stationary electrons. What is the wavelength of the beam which is scattered directly backwards? What is the energy gained by the scattered electron?

6. (a) State Malus' law. 2+2+1
   (b) In damped harmonic motion, calculate the time in which the energy falls to e^{-1} times of its initial value.
   (c) Draw the amplitude resonance curves for different values of damping.

GROUP C
(Long Answer Type Questions)

Answer any *three* questions. $3 \times 15 = 45$

7. (a) Prove that the intensity of secondary maxima formed for Fraunhofer diffraction at a single slit are of decreasing order. 4+3+4+4
   (b) Explain missing order in N-slit diffraction.
   (c) State Rayleigh criterion. Write a short note on resolving power of a grating.
   (d) A mica sheet of refractive index 1.58 is introduced in one of the interfering beams and the central fringe gets shifted by 0.2 cm. The distance between the sources is 0.1cm and the screen is placed at a distance 50 cm from the sources. Determine the thickness of the mica sheet.

8. (a) What do you mean by black body? State Kirchhoff's law of blackbody radiations. 4+3+4+4
   (b) Establish Newton's law of cooling from the Stefan-Boltzmann law of Blackbody radiation.
   (c) Starting from de Broglie's hypothesis show that the group velocity associated with a particle is the same as the particle velocity.
   (d) If an electron is subjected to a potential difference of V volts then prove that the corresponding de Broglie wavelength is $\frac{12.26}{\sqrt{v}}$ Å.

9. (a) A particle is subjected to a harmonic restoring force and a damping force. Its equation of motion is given 5+3+3+4

$$\frac{d^2x}{dt^2}+2b\frac{dx}{dt}+\omega_0^2x=0$$

Under the condition of small damping, find the expression for displacement as a function of time.

(b) Draw the displacement – time graph for large and critical damping and compare their relative shift.

(c) Two simple harmonic oscillators of different masses oscillate separately under the action of same restoring force at frequencies 3Hz and 5Hz. Calculate the ratio of their masses.

(d) Find the amplitude, phase and instantaneous velocity of the vibrational motion represented by

$$x=a\cos\omega t+\frac{a}{2}\cos(\omega t+\frac{\pi}{2})+\frac{a}{4}\cos(\omega t+\pi)+\frac{a}{8}\cos(\omega t+\frac{3\pi}{2})$$

10.(a) What is holography? Give the differences between ordinary photography and holography. 5+6+2+2

(b) Define Einstein's A and B co-efficients and obtain a relation between them.

(c) What is optical pumping?

(d) A tube of 20 cm length filled with a solution of 15g of cane sugar in 100cc of water placed in the path of a polarized light. Find the angle of rotation of the plane of polarization if the specific rotation of cane sugar is 65°.

11.(a) Define lattice constant in case of crystal system. 1+2+3+3+2 +4

(b) What is the physical significance of Miller indices?

(c) Deduce the interplaner spacing of a simple cubic lattice of side 'a'.

(d) The distance between (d_{100}) plane in BCC structure is 0.335 nm. What is the size of the unit cell?

(e) What is the difference between crystal grating and optical grating? Write Bragg's equation in crystal structure.

(f) A beam of X-rays is incident on NaCl crystal with lattice spacing 0.282 nm. Calculate the wavelength of X-rays if the first order Bragg's reflection is observed at a glancing angle of $8^\circ 35'$. Also find the maximum order of diffraction possible.

MAULANA ABUL KALAM AZAD UNIVERSITY OF TECHNOLOGY, WEST BENGAL

PH-101

PHYSICS – I

Time Allotted: 3 Hours | Full Marks: 70

The questions are of equal value.
The figures in the margin indicate full marks.
Candidates are required to give their answers in their own words as far as practicable.
All symbols are of usual significance.

GROUP A
(Multiple Choice Type Questions)

1. Answer any *ten* questions. 10×1 = 10

(i) Time period of a simple pendulum having infinite length will be

(A) finite non-zero value (B) infinite
(C) zero (D) cannot be estimated

(ii) The quality factor Q for an L-C-R circuit is

(A) $\frac{\omega R}{L}$ (B) $\frac{\omega L}{R}$
(C) $\frac{\omega}{LR}$ (D) $\frac{R}{\omega L}$

(iii) According to Moseley's law, frequency of a spectral line depends on

(A) mass (B) atomic number

(C) atomic weight (D) molecular weight

(iv) The energy relaxation time (τ) of a damped oscillator with damping coefficient (k) is

(A) $\tau = 1/k$ (B) $\tau = 1/2k$

(C) $\tau = k$ (D) $\tau = 2k$

(v) In Young's double slit experiment the two coherent sources are produced by

(A) division of wave front (B) division of amplitude

(C) all of these (D) none of these

(vi) The resolving power of a grating having number of rulings N, of order n is

(A) n/N (B) n^2/N

(C) $n+N$ (D) nN

(vii) When a white light is incident on a plane diffracting grating the central maxima will be

(A) Dark (B) White

(C) Blue (D) Red

(viii) Newton's ring experiment is based on

(A) division of amplitude (B) division of wave front

(C) division of frequency (D) division of phase angle

(ix) According to Wien's displacement law

(A) $\lambda_m T =$ Constant (B) $\dfrac{\lambda m}{T} =$ Constant

(C) $\lambda_m T^2 =$ Constant (D) $\dfrac{\lambda m}{T^2} =$ Constant

(x) Polarization conclusively proves that light waves are

(A) longitudinal (B) progressive

(C) stationary (D) transverse

(xi) The relativistic energy momentum relation is

(A) $p^2 = E^2 + m_0^2c^2$ (B) $E^2 = p^2c^2 + m_0^2c^4$

(C) $E^2 = p^2 + m_0^2c^4$ (D) $p^2 = E^2c^2 + m_0^2c^4$

(xii) An α-particle is 4 times heavier than a proton. If a proton and an α-particle are moving with the same velocity how their De-Broglie wavelengths are related?

(A) $\lambda_p = \lambda_\alpha$ (B) $\lambda_p = \lambda_\alpha / 2$

(C) $\lambda_p = \lambda_\alpha / 4$ (D) $\lambda_p = 4\lambda_\alpha$

(xiii) Assuming that the atoms in a crystal are spheres of equal size and touching each other, it can be shown that the atomic radius of BCC lattice is equal to

(A) $\frac{a}{\sqrt{2}}$ (B) $\sqrt{3}\frac{a}{2}$

(C) $\sqrt{3}\frac{a}{4}$ (D) $\frac{a}{2\sqrt{2}}$

(xiv) In lasing action, the spontaneous emission does not depend on

(A) the number of atoms present in the excited state

(B) the intensity of the incident light

(C) both of (A) and (B)

(D) none of these

(xv) If we measure the energy of a particle accurately then the uncertainty of the measurement of time becomes

(A) 0 (B) π

(C) $\frac{\pi}{2}$ (D) ∞

GROUP B
(Short Answer Type Questions)

Answer any *three* questions. $3 \times 5 = 15$

2. (a) Draw and explain the electrical equivalent circuit corresponding to damped vibration. 2

   (b) What is critical damping? 1

   (c) Establish the relation between the logarithmic decrement and quality factor of a damped oscillatory system. 2

3. (a) What is missing order in double slit diffraction pattern? – Explain graphically. 2

   (b) What do you mean by resolving power for plane transmission grating? 2

   (c) Why the Newton's rings are circular in nature? 1

4. (a) The refractive indices of double refracting crystal for ordinary and extraordinary rays are 1.584 and 1.592 respectively, for wavelength λ = 5600 Å. Determine the thickness of the crystal to produce half wave plate. 2

   (b) Describe briefly the principle of operation of a laser. 3

5. (a) What is Compton effect? 1

   (b) Calculate the Compton shift in wavelength for an electron. Explain the origin of unmodified lines in Compton scattering. 3+1

6. (a) Calculate atomic radius for a FCC crystal. What is Bravais lattice? 1+1

   (b) The interplaner spacing of a plane in a crystal is 1.2 Å and the angle for the first-order Bragg's reflection is 30°. Determine the energy of the X-ray beams in eV. 3

GROUP C
(Long Answer Type Questions)

Answer any *three* questions. $3 \times 15 = 45$

7. (a) An oscillator executing S.H.M. has zero displacement at time t = 0. If the displacement are 0.1 cm and 0.15 cm at instant t = 0.1 and 0.2 seconds respectively, calculate the frequency and amplitude of oscillation. 3

(b) If a particle executing simple harmonic motion simultaneously at two perpendicular directions with the same amplitude and frequency then find out the condition that the Lissajous figure will be a circle. 4

(c) Write down the differential equation for a series L-C-R circuit driven by a sinusoidal voltage. Indentify the natural frequency of this circuit. Find out the condition that this circuit will show an oscillatory decay and find out the relaxation time. 4

(d) Show that at velocity resonance, the velocity is in phase with the driving force. 4

8. (a) If the distance between two coherent sources of light with wavelength λ is d and D is the source screen distance then show that fringe width separation $x = \frac{D\lambda}{d}$. 4

(b) State the condition to be fulfilled for the production of sustained interference fringes. 3

(C) Show that in Newton's Ring experiment the radii of bright fringes are proportional to the square roots of odd natural numbers. 4

(d) In a Newton's ring experiment, the diameter of 5th dark ring is 0.336 cm and the diameter of the 15th dark ring is 0.590 cm. Find the radius of the Plano-convex lens if the wavelength of the light used is 5890 Å. 4

9. (a) What is the difference between single-slit and double-slit diffraction pattern? 2

(b) Show that the intensity of secondary maxima formed by single-slit Fraunhofer diffraction process is nearly 4.5% of the principal maxima. 4

(c) A parallel beam of light of wavelength 5890 Å falls normally on a plane transmission grating having 4250 line/cm. Find the angle of diffraction for maximum intensity in first order. 3

(d) Calculate the polarizing angle for a light ray travelling from water of refractive index 1.33 to glass of refractive index 1.53. 3

(e) Discuss briefly how Nicol prism can be used as a polarizer. 3

10.(a) What is the basic principle behind holography? 2

(b) Discuss the process of construction and reconstruction of hologram. 3+3

(c) In a He-Ne laser transition from E_3 to E_2 level gives a laser emission of wavelength 632.8 nm. If the energy of E_2 level is 15.2×10^{-19} J, how much pumping energy is required, if there is no energy loss? (Given that Planck's constant $h = 6.625 \times 10^{-34}$ J). 4

(d) What is the role of optical resonators in LASER? 3

11.(a) Show that in a cubic lattice of side '*a*', the interplaner spacing between consecutive parallel planes of Miller indices (*h k l*) is 5

$$d_{hkl} = \frac{a}{\sqrt{(h^2 + k^2 + l^2)}}.$$

(b) Explain the origin of characteristics of X-rays. 3

(c) Copper has FCC structure and atomic radius of 0.1278 nm. Calculate its density and the interplaner spacing for (3 2 1) planes. Take the atomic weight of cooper as 63.50. 3+2

(d) State Moseley's law. 2