COMPUTATIONAL METHODS LAB

ES-251

LAB MANUAL

MAHARAJA AGRASEN INSTITUTE OF TECHNOLOGY

MAIT

उद्यमेन हि सिध्यन्ति
कार्याणि न मनोरथैः

INSTITUTE VISION

To nurture young minds in a learning environment of high academic value and imbibe spiritual and ethical values with technological and management competence.

INSTITUTE MISSION

The Institute shall endeavor to incorporate the following basic missions in the teaching methodology:

Engineering Hardware – Software Symbiosis

Practical exercises in all Engineering and Management disciplines shall be carried out by Hardware equipment as well as the related software enabling deeper understanding of basic concepts and encouraging inquisitive nature.

Life – Long Learning

The Institute strives to match technological advancements and encourage students to keep updating their knowledge for enhancing their skills and inculcating their habit of continuous learning.

Liberalization and Globalization

The Institute endeavors to enhance technical and management skills of students so that they are intellectually capable and competent professionals with Industrial Aptitude to face the challenges of globalization.

Diversification

The Engineering, Technology and Management disciplines have diverse fields of studies with different attributes. The aim is to create a synergy of the above attributes by encouraging analytical thinking.

Digitization of Learning Processes

The Institute provides seamless opportunities for innovative learning in all Engineering and Management disciplines through digitization of learning processes using analysis, synthesis, simulation, graphics, tutorials and related tools to create a platform for multi-disciplinary approach.

Entrepreneurship

The Institute strives to develop potential Engineers and Managers by enhancing their skills and research capabilities so that they become successful entrepreneurs and responsible citizens.

1. COURSE DETAILS

1.1 COURSE OBJECTIVE

- To understand numerical methods to find roots of functions and first order unconstrained minimization of functions.
- To introduce concept of interpolation methods and numerical integration.
- To understand numerical methods to solve systems of algebraic equations and curve fitting by splines.
- To understand numerical methods for the solution of Ordinary and partial differential equations.

1.2 COURSE OUTCOMES

At the end of the course student will be able to:	
C.251.1	Utilize basic commands in Turbo C++
C.251.2	Determine the roots of the algebraic and transcendental equations using Turbo C++.
C.251.3	Develop Turbo C++ codes for implementation of interpolation formulae.
C.251.4	Develop Turbo C++ codes for solving numerical integration and differential equations

1.3 MAPPING COURSE OUTCOMES (CO) AND PROGRAM OUTCOMES (PO)

CO Course Outcomes	**Program Outcomes**											
	PO 1	**PO 2**	**PO 2**	**PO 4**	**PO 5**	**PO 6**	**PO 7**	**PO 8**	**PO 9**	**PO 10**	**PO 11**	**PO 12**
C.251.1	2	1	1	-	1	1	-	-	-	-	1	-
C.251.2	2	2	2	2	2	1	-	-	-	-	1	-
C.251.3	2	2	2	2	2	1	-	-	-	-	1	-
C.251.4	1	1	2	2	2	1	-	-	-	-	1	-

1.4 EVALUATION SCHEME

	Laboratory	
Components	**Internal**	**External**
Marks	40	60
Total Marks	100	

1.5 GUIDELINES FOR CONTINUOUS ASSESSMENT FOR EACH EXPERIMENT

- **Attendance and performance in experiments – 40 marks**
  - Practical performance
  - Results
  - Attendance and Viva Questions Answered
  - Timely Submission of Lab Record

The Rubrics for Experiment execution and Lab file and viva voce is given below:

Experiment Marks details:

LAC/ Sr No.	**Experiment Component (LAC)**	**Max. Marks**	**Grading Rubrics**	
			2 marks	**1 mark**
LAC 1	Practical Performance	2	Completeness of the program, exhibits proficiency in executing the program.	Not able to complete the program in the required way.
LAC 2	Results	2	Accuracy of results. Demonstrates excellent understanding of the concepts relevant to the program.	Results with moderate accuracy. Demonstrates partial understanding of the concepts relevant to the program.
LAC 3	Attendance and Viva Questions Answered	4	One mark for attendance.	
			Three marks for answering more than 75% questions.	
			Two marks for answering more than 50% questions.	
			One mark for answering less than 50% questions.	
LAC 4	Timely Submission of Lab Record	2	On time submission	Late submission

2. LIST OF EXPERIMENTS AS PER GGSIPU

Sr. No.	Title of Lab Experiments	CO
1.	Program for finding roots of f(x)=0 Newton Raphson method.	CO1, CO2
2.	Program for finding roots of f(x)=0 by bisection method.	CO1, CO2
3.	Program for finding roots of f(x)=0 by secant method.	CO1, CO2
4.	To implement Langrange's Interpolation formula.	CO1, CO3
5.	To implement Newton's Divided Difference formula.	CO1, CO3
6.	Program for solving numerical integration by Trapezoidal rule.	CO1, CO4
7.	Program for solving numerical integration by Simpson's 1/3 rule	CO1, CO4
8.	To implement Numerical Integration Simpson 3/8 rule.	CO1, CO4
9.	Inverse of a system of linear equations using Gauss-Jordan method.	CO1, CO4
10.	Program for solving ordinary differential equation by Runge-Kutta Method.	CO1, CO4

Experiment No: 1

Aim/Objective: Program for finding roots of f(x)=0 Newton Raphson Method.

Course Outcome: CO1

Software Used: Turbo C++

Theory:

The **Newton Raphson Method** is referred to as one of the most commonly used techniques for finding the roots of given equations. It can be efficiently generalised to find solutions to a system of equations. Moreover, we can show that when we approach the root, the method is quadratically convergent.

Newton Raphson Method Formula

Let x_0 be the approximate root of f(x) = 0 and let $x_1 = x_0 + h$ be the correct root. Then $f(x_1) = 0$

$\Rightarrow f(x_0 + h) = 0$….(1)

By expanding the above equation using Taylor's theorem, we get:

$f(x_0) + hf^1(x_0) + \ldots = 0$

$\Rightarrow h = -f(x_0) / f'(x_0)$

Therefore, $x_1 = x_0 - f(x_0)/ f'(x_0)$

Now, x_1 is the better approximation than x_0.

Similarly, the successive approximations x_2, x_3, …., x_{n+1} are given by

xn+1=xn−f(xn)f′(xn)
This is called Newton Raphson formula.

Geometrical Interpretation of Newton Raphson Formula

The geometric meaning of Newton's Raphson method is that a tangent is drawn at the point $[x_0, f(x_0)]$ to the curve y = f(x).

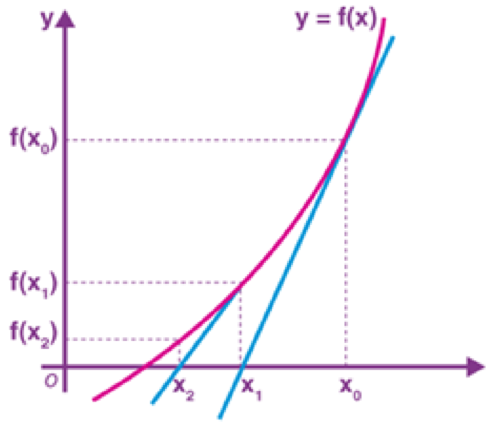

Figure – Newton Raphson Method

It cuts the x-axis at x_1, which will be a better approximation of the root. Now, drawing another tangent at [x_1, $f(x_1)$], which cuts the x-axis at x_2, which is a still better approximation, and the process can be continued till the desired accuracy is achieved.

Sample Problem:

$F(x)= x^3+5*x+3$ find approximation root using Newton Raphson Method
F(0)=3
F(-1)=-3
It means roots lie in between 0 and -1.

$F^1(x)= 3x^2+5$
$F^1(0)=5$

Using formula:
1st iteration
$x1= x0 – (F(x0)/F^1(x0))$
x1=0-(3/5)
x1=-0.6
2nd Iteration
x1=-0.6 , F(-0.6)= -0.784, $F^1(-0.6)= 6.08$
$x2= x1 – (F(x1)/(F^1(x1))$
x2= -0.6 – (-0.784/6.08)
x2= -0.47

3rd Iteration and so on to get closest root.

Code:

```
#include <iostream.h>
#include <conio.h>
#include <iomanip.h>
#include <math.h>
float f(float x)
{
return x*log10(x)-1.2;
}
float df(float x)
{
return log10(x) + 0.43429;
}
int main()
{
clrscr();
int itr, maxitr;
float h,x0,x1,aerr;
cout << "Enter x0,allowed error,"<< "maximum iterations" << endl;
cin >> x0 >> aerr >> maxitr;
for (itr=1;itr<=maxitr;itr++)
{
h = f(x0)/df(x0);
x1 = x0-h;
cout << "Iteration no." << setw(3) << itr
<< "X = " << setw(9) << setprecision(6)
<< x1 << endl;
if (fabs(h) < aerr)
{
cout << "After" << setw(3) << itr << "iterations, root = "
<< setw(8) << setprecision(6) << x1;
getch();
return 0;
}
x0 = x1;
}
cout << "Iterations not sufficient,"<< "solution does not converge" << endl;
getch();
return 0;
}
```

Experiment No: 2

Aim/Objective: Program for finding roots of f(x)=0 by Bisection method.

Course Outcome: CO1

Software Used: Turbo C++

Theory:

The **Bisection Method** is used to find the roots of a polynomial_equation. It separates the interval and subdivides the interval in which the root of the equation lies. The principle behind this method is the intermediate theorem for continuous functions. It works by narrowing the gap between the positive and negative intervals until it closes in on the correct answer. This method narrows the gap by taking the average of the positive and negative intervals. It is a simple method and it is relatively slow. The bisection method is also known as interval halving method, root-finding method, binary search method or dichotomy method.

Let us consider a continuous function "f" which is defined on the closed interval [a, b], is given with f(a) and f(b) of different signs. Then by intermediate theorem, there exists a point x belong to (a, b) for which f(x) = 0.

Bisection Method Algorithm

Follow the below procedure to get the solution for the continuous function:

For any continuous function f(x),

- Find two points, say a and b such that a < b and f(a)* f(b) < 0
- Find the midpoint of a and b, say "t"
- t is the root of the given function if f(t) = 0; else follow the next step
- Divide the interval [a, b] – If f(t)*f(a) <0, there exist a root between t and a – else if f(t) *f (b) < 0, there exist a root between t and b
- Repeat above three steps until f(t) = 0.

The bisection method is an approximation method to find the roots of the given equation by repeatedly dividing the interval. This method will divide the interval until the resulting interval is found, which is extremely small.

Sample Problem:

Given: $x^2-3 = 0$

Let $f(x) = x^2-3$

Now, find the value of f(x) at a= 1 and b=2.

$f(x=1) = 1^2-3 = 1 – 3 = -2 < 0$

$f(x=2) = 2^2-3 = 4 – 3 = 1 > 0$

The given function is continuous, and the root lies in the interval [1, 2].

Let "t" be the midpoint of the interval.

I.e., $t = (1+2)/2$

$t = 3 / 2$

$t = 1.5$

Therefore, the value of the function at “t” is

$f(t) = f(1.5) = (1.5)^2-3 = 2.25 – 3 = -0.75 < 0$

If $f(t)<0$, assume $a = t$.

and

If $f(t)>0$, assume $b = t$.

f(t) is negative, so a is replaced with t = 1.5 for the next iterations.

The iterations for the given functions are:

Iterations	**a**	**b**	**t**	**f(a)**	**f(b)**	**f(t)**
1	1	2	1.5	-2	1	-0.75
2	1.5	2	1.75	-0.75	1	0.062
3	1.5	1.75	1.625	-0.75	0.0625	-0.359
4	1.625	1.75	1.6875	-0.3594	0.0625	-0.1523
5	1.6875	1.75	1.7188	-01523	0.0625	-0.0457
6	1.7188	1.75	1.7344	-0.0457	0.0625	0.0081
7	1.7188	1.7344	1.7266	-0.0457	0.0081	-0.0189

So, at the seventh iteration, we get the final interval [1.7266, 1.7344]

Hence, 1.7344 is the approximated solution.

Code:

```
#include <iostream.h>
#include <conio.h>
#include <iomanip.h>
#include <math.h>
float f(float x)
{
return (x*x*x - 4*x - 9);
}
void bisect(float *x,float a,float b,int *itr)
{
*x = (a + b)/2;
++(*itr);
cout << "Iteration no." <<setw(3) << *itr
<< "X = " << setw(7) << setprecision(5)
<< *x << endl;
}
int main()
{
clrscr();
int itr = 0, maxitr;
float x, a, b, aerr, x1, fixed;
cout<<"This program is made by Sidharth"<<endl;
cout << "Enter the values of a,b,"
<< "allowed error, maximum iterations" << endl;
cin >> a >> b >> aerr >> maxitr;
cout << fixed;
bisect(&x,a,b,&itr);
do
{
if (f(a)*f(x) < 0)
b = x;
else
a = x;
bisect (&x1,a,b,&itr);
if (fabs(x1-x) < aerr)
{
cout << "After" << itr << "iterations, root"
<< "=" << setw(6) << setprecision(4)
<< x1 << endl;
getch();
return 0;
}
x = x1;
} while (itr < maxitr);
cout << "Solution does not converge,"
<< "iterations not sufficient" << endl;
getch();
return 0;
}
```

Experiment No: 3

Aim/Objective: Program for finding roots of f(x)=0 by secant method.

Course Outcome: CO1

Software Used: Turbo C++

Theory:

Secant method is also a recursive method for finding the root for the polynomials by successive approximation. It's similar to the **Regular-Falsi** method but here we don't need to check **$f(x_1)f(x_2)<0$** again and again after every approximation. In this method, the neighbourhood roots are approximated by secant line or chord to the function **f(x)**. It's also advantageous of this method that we don't need to differentiate the given function **f(x)**, as we do in **Newton-Raphson** method.

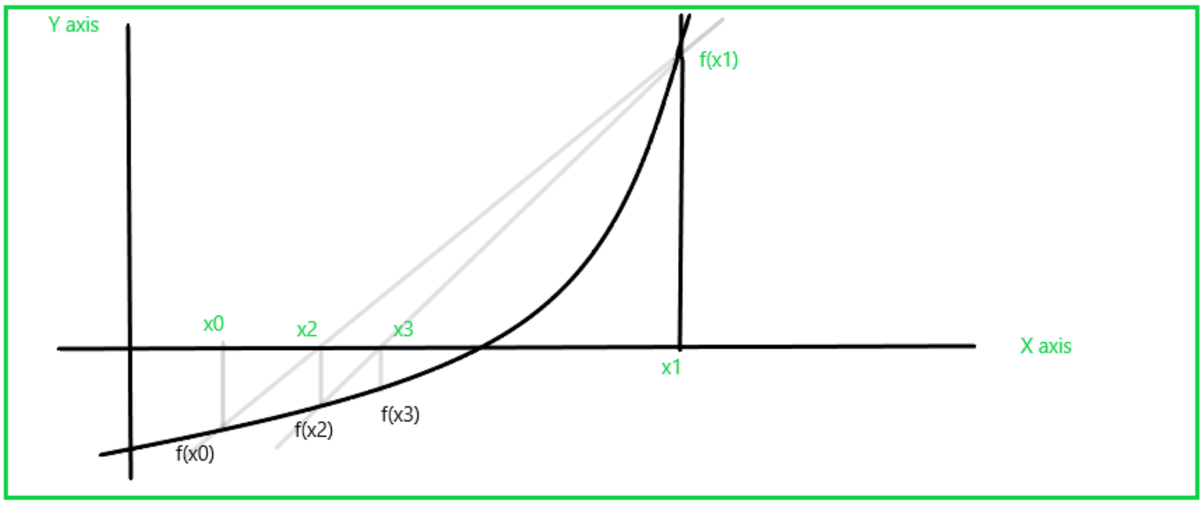

Figure – Secant Method

Now, we'll derive the formula for secant method. The equation of Secant line passing through two points is:

Here, m=slope

So, apply for $(x_1, f(x_1))$ and $(x_0, f(x_0))$

$Y - f(x_1) = [f(x_0)-f(x_1)/(x_0-x_1)]\ (x-x_1)$ Equation (1)

As we're finding root of function f(x) so, Y=f(x)=0 in Equation (1) and the point where the secant line cut the x-axis is,

$x= x_1 - [(x_0 - x_1)/ (f(x_0) - f(x_1)]f(x_1)$.

We use the above result for successive approximation for the root of function f(x). Let's say the first approximation is $x=x_2$:

$x_2= x_1 - [(x_0 - x_1)/ (f(x_0)-f(x_1))]f(x_1)$

Similarly, the second approximation would be $x =x_3$:

$x_3= x_2 - [(x_1-x_2)/ (f(x_1)-f(x_2))]f(x_2)$

And so on, till kth iteration,,

$x_{k+1}= x_k - [(x_{k-1} - x_k) / (f(x_{k-1}) - f(x_k))]f(x_k)$

Advantages of Secant Method:

- The speed of convergence of secant method is faster than that of Bisection and Regula falsi method.
- It uses the two most recent approximations of root to find new approximations, instead of using only those approximations which bound the interval to enclose root

Disadvantages of Secant Method:

- The Convergence in secant method is not always assured.
- If at any stage of iteration this method fails.
- Since convergence is not guaranteed, therefore we should put limit on maximum number of iterations while implementing this method on computer.

Sample Problem:

A real root of the equation $f(x) = x^3 – 5x + 1 = 0$ lies in the interval (0, 1). Perform four iterations of the secant method.

Solution –

We have, $x_0 = 0$, $x_1 = 1$, $f(x_0) = 1$, $f(x_1) = – 3$

$x_2 = x_1 – [(x_0 – x_1) / (f(x_0) – f(x_1))]f(x_1)$

= 1 – [(0 – 1) / ((1-(-3))](-3)

= 0.25.

$f(x_2) = – 0.234375$

The second approximation is,

$x_3 = x_2 – [(x_1 – x_2) / (f(x_1) – f(x_2))]f(x_2)$

=(– 0.234375) – [(1 – 0.25)/(–3 – (– 0.234375))](– 0.234375)

= 0.186441

$f(x_3)$

The third approximation is,

x4 = x3 – [(x2 – x3) / (f(x2) – f(x3))]f(x3)

= 0.186441 – [(0.25 – 0.186441) / (– 0.234375) **– (0.074276)](– 0.234375)**

= 0.201736.

f(x4) = – 0.000470

The fourth approximation is,

x5 = x4 – [(x3 – x4) / (f(x3) – f(x4))]f(x4)

= 0.201736 – [(0.186441 – 0.201736) / (0.074276 – (– 0.000470)](– 0.000470)

= 0.201640

Code:

```
#include<iostream.h>
#include<iomanip.h>
#include<math.h>
#include<conio.h>
#include<stdlib.h>

float f(float x)
{
return  x*x*x - 2*x - 5;
}
```

```
int main()
{
clrscr ();

        float x0, x1, x2, f0, f1, f2, e;
        int itr = 1, N;

        cout<<"Enter first guess: ";
        cin>>x0;
        cout<<"Enter second guess: ";
        cin>>x1;
        cout<<"Enter tolerable error: ";
        cin>>e;
        cout<<"Enter maximum iteration: ";
        cin>>N;
        do
        {
                f0 = f(x0);
                f1 = f(x1);
                if(f0 == f1)
                {
                        cout<<"Mathematical Error.";
                        getch();
                        exit(0);
                }

                x2 = x1 - (x1 - x0) * f1/(f1-f0);
                f2 = f(x2);

                cout<<"Iteration-"<< itr<<":\t x2 = "<< setw(10)<< x2<<" and f(x2) = "<<
setw(10)<< f(x2)<< endl;

                x0 = x1;
                f0 = f1;
                x1 = x2;
                f1 = f2;
                itr = itr + 1;
                if(itr > N)
                {
                        cout<<"Not Convergent.";
                        getch();
                        exit(0);
                }
        }while(fabs(f2)>e);
        cout<< endl<<"Root is: "<< x2;
       getch();
        return 0;
}
```

Experiment – 4

Aim/Objective: To implement Lagrange's Interpolation formula
Course Outcome: CO2
Software used: Turbo C++
Theory:
Interpolation is a way of finding additional data points within a range of discrete data points. The Lagrange interpolation formula is a method for determining a polynomial, known as a Lagrange polynomial,that takes on specific values at random places. Lagrange's interpolation is a polynomial approximation to f of Nth degree (x). Interpolation is a technique for generating new values for any function from a set of existing values. Using this technique, we may find the unknown value on a point. When it comes to the linear interpolation formula, it can be used to find a new value from two provided points. The "n" set of numbers is required when comparing it to Lagrange's interpolation formula. The new value will then be found using Lagrange's approach.
Let $y = f(x)$ be a polynomial of nth degree passing through $(n+1)$ points (x_k, y_k), k=0,1,…,n.
Then,

$$\begin{aligned} y = a_0(x-x_1)(x-x_2)\cdots(x-x_n) + a_1(x-x_0)(x-x_2)\cdots(x-x_n) \quad &+ \cdots \\ + a_k(x-x_0)(x-x_1)\cdots(x-x_{k-1})(x-x_{k+1})\cdots(x-x_n) \quad &+ \cdots \\ + a_n(x-x_0)(x-x_1)\cdots(x-x_{n-1}) \quad & \quad (1) \end{aligned}$$

This equation will be satisfied by the given points $(x_k, y_k),\ k = 0, 1, \dots, n.$
For (x_0, y_0), we have $y_0 = a_0(x_0-x_1)(x_0-x_2)\cdots(x_0-x_n)$. Thus,

$$a_0 = \frac{y_0}{(x_0-x_1)(x_0-x_2)\cdots(x_0-x_n)}$$

Similarly, for (x_1, y_1), we have $y_1 = a_1(x_1-x_0)(x_1-x_2)\cdots(x_1-x_n)$. Hence,

$$a_1 = \frac{y_1}{(x_1-x_0)(x_1-x_2)\cdots(x_1-x_n)}$$

and so on. For the point (x_n, y_n) would thus be

$$a_n = \frac{y_n}{(x_n-x_0)(x_n-x_1)\cdots(x_n-x_{n-1})}$$

Applying these values in equation (1), we get

$$\begin{aligned} y = f(x) = &\frac{(x-x_1)(x-x_2)\cdots(x-x_n)}{(x_0-x_1)(x_0-x_2)\cdots(x_0-x_n)} y_0 \\ &+ \frac{(x-x_0)(x-x_1)\cdots(x-x_n)}{(x_1-x_0)(x_1-x_2)\cdots(x_1-x_n)} y_1 + \cdots \\ &+ \frac{(x-x_0)(x-x_1)\cdots(x-x_n)}{(x_n-x_0)(x_n-x_1)\cdots(x_n-x_{n-1})} y_n \end{aligned}$$

which is known as the $Lagrange's\ Interpolation\ formula\ for\ unequal\ intervals.$

Sample Problem:

x	-1	0	2	3
y	-8	3	1	12

Use the Lagrange formula to fit a polynomial to the following data and hence find $f(1)$.

Solution:
By Lagrange's formula

$$
\begin{aligned}
f(x) &= \frac{(x-0)(x-2)(x-3)}{(-1-0)(-1-2)(-1-3)}(-8) + \frac{(x+1)(x-2)(x-3)}{(0+1)(0-2)(0-3)}(3) \\
&\quad + \frac{(x+1)(x-0)(x-3)}{(2+1)(2-0)(2-3)}(1) \quad + \frac{(x+1)(x-0)(x-2)}{(3+1)(3-0)(3-2)}(12) \\
&= 2x^3 - 6x^2 + 3x + 3
\end{aligned}
$$

Therefore, $f(1) = 2 - 6 + 3 + 3 = 2.$

Code:

```
#include <iostream.h>
#include <iomanip.h>
#include <conio.h>
#define MAX 100
int main()
{
clrscr();
float ax [MAX+1],ay[MAX+1],nr,dr,x,y=0;
int i,j,n;
cout << "Lagrange by Sidharth"<<endl<< "Enter the value of n" << endl;
cin >> n;
cout << "Enter the set of values" << endl;
for (i=0;i<=n;i++)
cin >> ax[i] >> ay[i];
cout << "Enter the value of x for which value of y is wanted" << endl;
cin >> x;
for (i=0;i<=n;i++)
{
nr=dr=1;
for(j=0;j<=n;j++)
if (j!=i)
{
nr *= x-ax[j];
dr *= ax[i]-ax[j];
}
y += (nr/dr)*ay[i];
}
cout << "When x="
<< setw(4) << setprecision(3)
<< x << "\t y="
<< setw(7) << setprecision(3)
<< y << endl;
getch();
return 0;
}
```

<u>Experiment – 5</u>

Aim/Objective: To implement Newton's Divided Difference formula
Course Outcome: CO2
Software Used: Turbo C++
Theory:
Let $f(x_0), f(x_1), \dots, f(x_n)$ be the values of $f(x)$ corresponding to the arguments $x_0, x_1, \dots, x_n$ which are not equally spaced. By the definition of the first divided difference,

$$[x, x_0] = \frac{f(x) - f(x_0)}{x - x_0}$$

$$f(x) = f(x_0) + (x - x_0)[x, x_0] \qquad (1)$$

Now,

$$[x, x_0, x_1] = \frac{[x, x_0] - [x_0, x_1]}{x - x_1}$$

Then, $$[x, x_0] = [x_0, x_1] + (x - x_1)[x, x_0, x_1]$$

Using this value of $[x, x_0]$ in (1), we get

$$f(x) = f(x_0) + (x - x_0)[x_0, x_1] + (x - x_0)(x - x_1)[x, x_0, x_1] \qquad (2)$$

Again,

$$[x, x_0, x_1, x_2] = \frac{[x, x_0, x_1] - [x_0, x_1, x_2]}{x - x_2}$$

and so

$$[x, x_0, x_1] = [x_0, x_1, x_2] + (x - x_2)[x, x_0, x_1, x_2] \qquad (3)$$

Equation (2) then gives us

$$\begin{aligned} y = y_0 &+ (x - x_0)[x_0, x_1] + (x - x_0)(x - x_1)[x_0, x_1, x_2] \\ &+ (x - x_0)(x - x_1)(x - x_2)[x, x_0, x_1, x_2] \qquad (4) \end{aligned}$$

Proceeding in this way, we obtain

$$\begin{aligned} y = y_0 &+ (x - x_0)[x_0, x_1] + (x - x_0)(x - x_1)[x_0, x_1, x_2] \\ &+ (x - x_0)(x - x_1)(x - x_2)[x_0, x_1, x_2, x_3] + \cdots \quad + (x - x_0)(x - x_1)\cdots(x \\ &- x_n)[x, x_0, x_1, \dots, x_n] \qquad (5) \end{aligned}$$

This formula is called Newton's general interpolation formula with divided differences, the last term being the remainder term after (n+1) terms.

Sample Problem:
Using the following table find $f(x)$ as a polynomial in x. Find $f(1)$.

x	-1	0	3	6	7
$f(x)$	3	-6	39	822	1611

Solution: The divided difference table is made as

x	$f(x)$	$1st\ differenc$	$2nd\ differenc$	$3rd\ differenc$	$4th\ differenc$
-1	3				
		-9			
0	-6		6		
		15		5	
3	39		41		1
		261		13	
6	822		132		
		789			
7	1611				

Hence, the equation (5) gives

$$f(x) = 3 + (x+1)(-9) + x(x+1)(6) + x(x+1)(x-3)(5) + x(x+1)(x-3)(x-6) \quad = \quad x^4 - 3x^3 + 5x^2 - 6$$

Therefore, $f(1) = -3$

Code:

```
#include<iostream.h>
#include<conio.h>
int main()
{
int x[10],y[10],p[10];
int k,f,n,i,j=1,f1=1,f2=0;
clrscr();
cout<<"enter the no. of observations\n";
cin>>n;
cout<<"enter the different values of x\n";
for(i=1;i<=n;i++)
{
cin>>x[i];
}
cout<<"enter the corresponding values of y\n";
for(i=1;i<=n;i++)
{
cin>>y[i];
}
f=y[1];
cout<<"enter the value of 'k' in f(k) you want to evaluate\n";
cin>>k;
do
{
for(i=1;i<=n-1;i++)
{
p[i]=(((y[i+1]-y[i]))/(x[i+j]-x[i]));
y[i]=p[i];
}
f1=1;
for(i=1;i<=j;i++)
{
f1*=(k-x[i]);
} f2+=(y[1]*f1);
n--;
j++;
} while(n!=1);
f+=f2;
cout<<"f("<<k<<")="<<f;
getch();
return 0;
}
```

Experiment – 6

Aim/Objective: Program for solving numerical integration by Trapezoidal rule
Course Outcome: CO2
Software used: Turbo C++
Theory:
The process of evaluating a definite integral $I = \int_a^b f(x)dx$ from a set of numerical values of the integrand $f(x)$ is known as *quadrature.*
Let $f(x)$ be a function defined in $[a, b]$. Partition the interval $[a, b]$ into n equal parts of step-length h, i.e., $b - a = nh$. Thus, $x_0 = a, x_1 = a + h, x_2 = a + 2h, \dots, x_n = b$ and $y_0, y_1, y_2, \dots, y_n$ are the corresponding values of $y = f(x)$ at $x = x_k, k = 0,1,2, \dots, n$. Therefore,

$$I = \int_a^b f(x)dx = \int_0^n f(x_0 + uh)h du$$

where $u = (x - x_0)/h$ and, so, $dx = hdu$. Hence,

$$I = h\int_0^n \left[y_0 + u\Delta y_0 + \frac{u(u-1)}{2!}\Delta^2 y_0 + \dots + \frac{u(u-1)\dots(u-n+1)}{n!}\Delta^n y_0\right] du$$
$$= h\left[ny_0 + \frac{n^2}{2}\Delta y_0 + \left(\frac{n^3}{3} - \frac{n^2}{2}\right)\frac{\Delta^2 y_0}{2!} + \left(\frac{n^4}{4} - n^3 + n^2\right)\frac{\Delta^3 y_0}{3!} + \cdots \right.$$
$$\left. + \, up\ to\ (n+1)\ terms\right] \quad (1)$$

This is known as *general quadrature formula* or the *Newton – Cote formula.*
Put $n = 1$ in equation (1) and neglecting the second and higher order difference, we get,

$$\int_{x_0}^{x_0+h} ydx = h\left(y_0 + \frac{1}{2}\Delta y_0\right) = h\left(y_0 + \frac{y_1 - y_0}{2}\right) = \frac{h}{2}[y_0 + y_1]$$

Similarly,

$$\int_{x_0+h}^{x_0+2h} ydx = \frac{h}{2}(y_1 + y_2) \vdots \int_{x_0+(n-1)h}^{x_0+nh} ydx = \frac{h}{2}(y_{n-1} + y_n)$$

Adding these n integrals, we get

$$\int_a^b f(x)dx = \frac{h}{2}[y_0 + 2(y_1 + y_2 + \dots + y_{n-1}) + y_n]$$

This is called the Trapezoidal rule.

Sample Problem:
Evaluate the following integral using the trapezoidal rule with $h = 0.2$.

$$I = \int_0^1 \frac{1}{1+x^2}dx$$

Solution:
Let $y = 1/(1 + x^2)$. Given $a = 0$, $b = 1$ and $h = 0.2$. Therefore, we can make the following table

x	0	0.2	0.4	0.6	0.8	1.0
$f(x)$	1.00000	0.96154	0.86207	0.73529	0.60976	0.50000

By Trapezoidal rule,

$$\int_0^1 \frac{1}{1+x^2}dx = \frac{h}{2}[y_0 + 2(y_1 + y_2 + y_3 + y_4) + y_5]$$
$$= \frac{0.2}{2}[1 + 2(0.96154 + 0.86207 + 0.73529 + 0.60976) + 0.5] \quad = 0.783732$$

Code:

```
#include <iostream.h>
#include <iomanip.h>
#include <conio.h>
float y(float x)
{
 return 1/(1+x*x);
}
int main()
{
 clrscr();
 float x0,xn,h,s;
 int i,n;
 cout << "Enter x0,xn,no. of subintervals" << endl;
 cin >> x0 >> xn >> n;
 h = (xn-x0)/n;
 s = y(x0)+y(xn);
 for (i=1;i<=n-1;i++)
 s += 2*y(x0+i*h);
 cout << "Value of integral is"
 << setw(6) << setprecision(4)
 << (h/2)*s << endl;
 getch();
 return 0;
}
```

Experiment – 7

Aim/Objective: Program for solving numerical integration by Simpson's 1/3rd rule
Course Outcome: CO2
Software used: Turbo C++
Theory:
Simpson's rule is one of the numerical methods which is used to evaluate the definite integral. Simpson's 1/3rd rule is an extension of the trapezoidal rule in which the integrand is approximated by a second-order polynomial. Simpson rule can be derived from the various way using Newton's divided difference polynomial, Lagrange polynomial and the method of coefficients.
Simpson's Rule Formula
Simpson's rule methods are more accurate than the other numerical approximations and its formula for n+1 equally spaced subdivision is given by;

$$\int_a^b f(x)dx = \frac{\Delta x}{3}[f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + \cdots \ldots\ldots. + 2f(x_{n-2}) + 4f(x_{n-1}) + f(x_n)]$$

Where n is the even number, $\Delta x = (b - a)/n$ and $x_i = a + i\Delta x$
If we have f(x) = y, which is equally spaced between [a, b] and if $a = x_0$, $x_1 = x_0 + h$, $x_2 = x_0 + 2h$ …., $x_n = x_0 + nh$, where h is the difference between the terms. Or we can say that $y_0 = f(x_0)$, $y_1 = f(x_1)$, $y_2 = f(x_2)$,……,$y_n = f(x_n)$ are the analogous values of y with each value of x.
Simpson's 1/3 Rule

$$I = \int_a^b f(x)dx$$
$$= \frac{h}{3}[(Y_0 + Y_n) + 4(Y_1 + Y_3 + Y_5 + \cdots \ldots\ldots + Y_{n-1}) + 2(Y_2 + Y_4 + \cdots \ldots\ldots + Y_{n-2})]$$

Where, $h = \frac{b-a}{n}$, and n is even number equally spaced between $[a, b]$
Sample Problem
Evaluate the following integral using the Simpson's 1/3 rule

$$I = \int_0^1 \frac{1}{1 + x^2}dx$$

Let us divide the range [0,1] into six equal parts by taking h = 0.2

If $x_0 = 0$ then $y_0 = 1$

If $x_1 = x_0 + h = 1/6$, then $y_1 = 0.9731$

If $x_2 = x_0 + 2h = 2/6$, then, $y_2 = 0.90$

If $x_3 = x_0 + 3h = 3/6$, then $y_3 = 0.801$

If $x_4 = x_0 + 4h = 4/6$, then $y_4 = 0.694$

If $x_5 = x_0 + 5h = 5/6$,then $y_5 = 0.599$

If $x_6 = x_0 + 6h = 6/6$, then $y_6 = 0.5$

We know by Simpson's ⅓ rule;

$\int_a^b f(x)dx =$ h/3 $[(y_0 + y_n) + 4(y_1 + y_3 + y_5 + \ldots. + y_{n-1}) + 2(y_2 + y_4 + y_6 + \ldots.. + y_{n-2})]$

Therefore,

$\int_0^1 logxdx$ = (0.166/3) [(1 + 0.5) + 4*(0.973+0.801+0.599) + 2*(0.9+0.694)]

= 0.784 (approx.)

Code:

```
#include <iostream.h>
#include <iomanip.h>
#include <conio.h>
float y(float x)
{
return 1/(1+x*x);
}
int main()
{
clrscr();
float x0,xn,h,s;
int i,n;
cout << "Enter x0,xn, no.of subintervals"
<< endl;
cin >> x0 >> xn >> n;
cout << fixed;
h = (xn-x0)/n;
s = y(x0)+y(xn)+4*y(x0+h);
for (i=3;i<=n-1;i+=2)
s += 4*y(x0+i*h)+2*y(x0+(i-1)*h);
cout << "Value of integral is"
<< setw(6) << setprecision(4)
<< (h/3)*s << endl;
getch();
return 0;
}
```

Experiment – 8

Aim/Objective: To implement Numerical Integration by Simpson 3/8th rule.
Course Outcome: CO2
Software Used: Turbo C++
Theory:
Simpson's 3/8th Rule is a Numerical technique to find the definite integral of a function within a given interval.
It's so called because the value 3/8 appears in the formula. The function is divided into many sub-intervals and each interval is approximated by a cubic curve. And the area is then calculated to find the integral. The more is the number of sub-intervals used, the better is the approximation.

Simpson's 3/8th Rule Formula

$\int_a^b f(x)dx$ = 3h/8 [(y_0 + y_n) + 3(y_1 + y_2 + y_4 + y_5 + …. + y_{n-1}) + 2(y_3 + y_6 + y_9 + ….. + y_{n-3})]
Where, $h = \frac{b-a}{n}$, and n should be multiple of 3.
Sample Problem: Evaluate the following integral using the Simpson's 3/8th rule.

$$I = \int_0^1 \frac{1}{1+x^2} dx$$

Let us divide the range [0,1] into six equal parts by taking h = 0.2

If x_0 = 0 then y_0 = 1

If x_1 = x_0 + h = 1/6, then y_1 = 0.9731

If x_2 = x_0 + 2h = 2/6, then, y_2 = 0.90

If x_3 = x_0 + 3h = 3/6, then y_3 =0.801

If x_4 = x_0 + 4h = 4/6, then y_4 = 0.694

If x_5 = x_0 + 5h = 5/6,then y_5 = 0.599

If x_6 = x_0 + 6h = 6/6, then y_6 = 0.5

We know by Simpson's 3/8th rule;

$\int_0^1 \frac{1}{1+x^2} dx$ = 3h/8 [(y_0 + y_n) + 3(y_1 + y_2 + y_4 + y_5 + …. + y_{n-1}) + 2(y_3 + y_6 + y_9 + ….. + y_{n-3})]
=(3*0.166)/8[(1+0.5)+3(0.973+0.9+0.694+0.599)+2(0.801)]
=0.78372

Code:

```
# include <iostream.h>
# include <conio.h>
float y(float x)
{
return 1/(1+x*x);
}
void main()
{
float result=1;
float x0,xn,h,s;
int i,j,n;
clrscr();
cout<<"\n\n Enter the range - ";
cout<<"\n\n Lower Limit x0 - ";
cin>>x0;
cout<<"\n\n Upper Limit xn - ";
cin>>xn;
cout<<"\n\n Enter number of subintervals - ";
cin>>n;
h=(xn-x0)/n;
s=0;
s=y(x0)+y(xn);
for(i=1;i<n;i++)
{
if(i%3==0)
{
s+=2*y(x0+i*h);
}
else
{
s+=3*y(x0+(i)*h);
}
}
result=s*3*h/8;
cout<<"\n\n\n\n Value of the integral is \t"<<result;
getch();
}
```

Experiment – 9

Aim/Objective: Inverse of a system of linear equations using Gauss-Jordan method.
Course Outcome: CO3
Software used: Turbo C++
Theory:
The Gauss-Jordan method is used to analyze different systems of linear simultaneous equations that arise in engineering and science. This method finds its application in examining a network under sinusoidal steady state, output of a chemical plant, electronic circuits consisting invariant elements, and more.
The Gauss-Jordan method is focused on reducing the system of equations to a diagonal matrix form by row operations such that the solution is obtained directly. Further, it reduces the time and effort invested in back-substitution for finding the unknowns.

In the Gauss-Jordan C program, the given matrix is diagonalized using the following step-wise procedure.

1. The element in the first column and the first row is reduced 1, and then the remaining elements in the first column are made 0 (zero).
2. The element in the second column and the second row is made 1, and then the other elements in the second column are reduced to 0 (zero).
3. Similarly, steps 1 and 2 are repeated for the next 3rd, 4th and following columns and rows.
4. The overall diagonalization procedure is done in a sequential manner, performing only row operations.

Code:

```
#include <iostream.h>
#include <iomanip.h>
#include <conio.h>
#define N 4
int main()
{
float a[N][N+1],t;
int i,j,k;
cout << "Enter the elements of the"
<< "augmented matrix rowwise" << endl;
for (i=0;i<N;i++)
for (j=0;j<N+1;j++)
cin >> a[i][j];
cout << fixed;
for (j=0;j<N;j++)
for (i=0;i<N;i++)
if (i!=j)
{
t = a[i][j]/a[j][j];
for (k=0;k<N+1;k++)
a[i][k] -= a[j][k]*t;
}
cout << "The diagonal matrix is:-" << endl;
for (i=0;i<N;i++)
{
for (j=0;j<N+1;j++)
cout << setw(9) << setprecision(4) << a[i][j];
cout << endl;
}
cout << "The solution is:- " << endl;
for (i=0;i<N;i++)
cout << "x[" << setw(3) << i+1 << "] ="
<< setw(7) << setprecision(4)
<< a[i][N]/a[i][i] << endl;
getch();
return 0;
}
```

Experiment – 10

Aim/Objective: Program for solving ordinary differential equation by Runge-Kutta Method.
Course Outcome: CO4
Software used: Turbo C++
Theory:
Runge-Kutta method is a popular iteration method of approximating solution of ordinary differential equations. Developed around 1900 by German mathematicians C Runge and M. W. Kutta, this method is applicable to both families of explicit and implicit functions. The Runge-Kutta method is based on solution procedure of initial value problem in which the initial conditions are known. Based on the order of differential equation, there are different Runge-Kutta methods which are commonly referred to as: RK2, RK3, and RK4 methods. The most widely known member of the Runge–Kutta family is generally referred to as "RK4", "classical Runge– Kutta method" or simply as "the Runge–Kutta method".

Algorithm For Runge-Kutta Method: Runge-Kutta Method for 4th order

```
Step 1: Read x1,y1 initial values.
Step 2: Read a, value at which function value is to be found.
Step 3: Read n, the number of steps.
Step 4: count=0
Step 5: h=(a-x1)/n
Step 6: write x1,y1
Step 7: s1=f(x1,y1)
Step 8: s2=f(x1+h/2,y1+s1*h/2)
Step 9: s3=f(x1+h/2,y1+s2*h/2)
Step 10: s4=f(x1+h,y1+s3*h)
Step 11: y2=y1+(s1+2*s2+2*s3+s4)*h/6
Step 12: x2=x1+h
Step 13: write x2,y2
Step 14: count=count+1
Step 15: If count<n . then
x1=x2
y1=y2
go to step step 7
endif
Step 16: write x2,y2
Step 17: stop
```

Sample Problem: Using the fourth-order Runge-Kutta formulas, find Y(0,2) given that $Y' = x + y, Y(0) = 1$.

Given, $f(x, y) = x + y, x_0 = 0, Y_0 = 1$
Suppose, h = 0.1, hence $x_1 = 0.1$ *and* $x_2 = 0.2$
Let us apply the fourth order Runge-Kutta formula in the interval (0,0.1),

$$k_1 = hf(x_0, y_0) = 0.1(0 + 1) = 0.1$$
$$k_2 = hf\left(x_0 + \frac{h}{2}, y_0 + \frac{k_1}{2}\right) = 0.1f(0.05,1.05) = 0.11$$

$$k_3 = hf\left(x_0 + \frac{h}{2}, y_0 + \frac{k_2}{2}\right) = 0.1f(0.05, 1.055) = 0.1105$$
$$k_4 = hf(x_0 + h, y_0 + k_3) = 0.1f(0.1, 1.1105) = 0.12105$$

Then, $Y_1 = Y(0.1) = Y_0 + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) = 1.110342$
Now, we start with $(x_1, y_1) = (0.1, 1.110342)$ and again apply the formulas.

We get
$$k_1 = hf(x_1, y_1) = 0.1210342$$
$$k_2 = hf\left(x_1 + \frac{h}{2}, y_1 + \frac{k_1}{2}\right) = 0.1320859$$
$$k_3 = hf\left(x_1 + \frac{h}{2}, y_1 + \frac{k_2}{2}\right) = 0.13263848$$
$$k_4 = hf(x_1 + h, y_1 + k_3) = 0.144298048$$
Then, $Y_2 = Y(0.2) = Y_1 + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) = 1.2428055$
Thus, the required valuer of y is 1.2428055 at x = 0.2.

Code:

```
#include <iostream.h>
#include <iomanip.h>
#include <conio.h>
float f(float x, float y)
{
return x+y*y;
}
int main()
{
float x0,y0,h,xn,x,y,k1,k2,k3,k4,k;
cout << "Enter the values of x0,y0,"
<< "h,xn" << endl;
cin >> x0 >> y0 >> h >> xn;
x = x0; y = y0;
while (1)
{
if (x == xn) break;
k1 = h*f(x,y);
k2 = h*f(x+h/2,y+k1/2);
k3 = h*f(x+h/2,y+k2/2);
k4 = h*f(x+h,y+k3);
k = (k1+(k2+k3)*2+k4)/6;
x += h; y += k;
cout << "When x = " << setprecision(4)
<< setw(8) << x
<< " y = " << setw(8) << y << endl;
}
getch();
return 0;
}
```