1. **Write a program in C to solve a root of the equation x^3- 4x + 1 using Bisection method.**

Algorithm:

Step-1: start.
Step-2: Define function f(x)
Step-3: Choose initial guesses x0 and x1 such that f(x0) f(x1) < 0.
Step-4: Choose pre-specified tolerable error e.
Step-5: Calculate new approximated root as x2 = (x0 + x1)/2
Step-6: Calculate f(x0) f(x2) a. if f(x0) f(x2) < 0 then x0 = x0 and x1 = x2 b. if f(x0) f(x2) > 0 then x0 = x2 and x1 = x1 c. if f(x0) f(x2) = 0 then go to (8)
Step-7: if |f(x2) | > e then go to (5) otherwise go to (8)
Step-8: Display x2 as root.
Step-9: Stop.

Source Code:

```
#include<stdio.h>
#include<math.h>
#define f(x) pow(x,3)-4*x+1
int main(){
int step = 1;
float x0, x1, x2, f0, f1, f2, e;
goto up;
printf("\nEnter two initial guesses: ");
scanf("%f%f", &x0, &x1);
printf("Enter tolerable error: ");
scanf("%f", &e);
f0 = f(x0);                                    /* Calculating Functional Value */
f1 = f(x1);
if( f0 * f1 > 0.0)    /* Checking whether given guesses bracket the root or not. */
{
printf("Incorrect Initial Guesses.\n");
goto up;
}
printf("\nStep\t\tx0\t\tx1\t\tx2\t\tf(x2)\n"); /* Implementing Bisection Method */
do {
x2 = (x0 + x1)/2;
f2 = f(x2);
printf("%d\t\t%f\t%f\t%f\t%f\n",step, x0, x1, x2, f2);
if( f0 * f2 < 0)
{
x1 = x2;
f1 = f2;
}
else
```

```
{
x0 = x2;
f0 = f2;
}
step = step + 1;
} while(fabs(f2)>e)
printf("\nRoot is: %f", x2);
return 0;
```

Output:

```
Enter two Initial Guesses: 0 1
Enter Tolerable Error: 0.0001
Step x0 x1 x2 f(x2)
1 0.000000 1.000000 0.500000 -0.875000
2 0.000000 0.500000 0.250000 0.015625
3 0.250000 0.500000 0.375000 -0.447266
4 0.250000 0.375000 0.312500 -0.219482
5 0.250000 0.312500 0.281250 -0.102753
6 0.250000 0.281250 0.265625 -0.043758
7 0.250000 0.265625 0.257813 -0.014114
8 0.250000 0.257813 0.253906 0.000744
9 0.253906 0.257813 0.255859 -0.006688
10 0.253906 0.255859 0.254883 -0.002973
11 0.253906 0.254883 0.254395 -0.001115
12 0.253906 0.254395 0.254150 -0.000185
13 0.253906 0.254150 0.254028 0.000279
14 0.254028 0.254150 0.254089 0.000047
Root is: 0.254089
```

2. **Write a program in C to solve a root of the equation x ^ 3 - 4x + 1 = 0 using Regula-Falsi Method**

Algorithm:

Step-1: start.
Step-2: Define function f(x).
Step-3: Choose initial guesses x0 and x1 such that f(x0) f(x1) < 0.
Step-4: Choose pre-specified tolerable error e.
Step-5: Calculate new approximated root as: x2 = x0 - ((x0-x1) * f(x0))/(f(x0) - f(x1))
Step-6: Calculate f(x0) f(x2)
    a. if f(x0) f(x2) < 0 then x0 = x0 and x1 = x2
    b. if f(x0) f(x2) > 0 then x0 = x2 and x1 = x1
    c. if f(x0) f(x2) = 0 then go to (8)
Step-7: if |f(x2) |>e then go to (5) otherwise go to (8)
Step-8: Display x2 as root.
Step-9: Stop

Source Code:

```
#include<stdio.h>
#include<math.h>
#define f(x) pow(x,3)-4*x+1
int main(){
float x0, x1, x2, f0, f1, f2, e; int step = 1;
up:
printf("\nEnter two initial guesses: ");
scanf("%f%f", &x0, &x1);
printf("Enter tolerable error: ");
scanf("%f", &e);
/* Calculating Functional Values */
f0 = f(x0);
f1 = f(x1);
if( f0*f1 > 0.0){
printf("Incorrect Initial Guesses.\n");
goto up;
}
/* Implementing Regula Falsi or False Position Method */
printf("\nStep\t\tx0\t\tx1\t\tx2\t\tf(x2)\n");
do{
x2 = x0 - (x0-x1) * f0/(f0-f1);
f2 = f(x2);
printf("%d\t\t%f\t%f\t%f\t%f\n",step, x0, x1, x2, f2);
if(f0*f2 < 0){
x1 = x2;
f1 = f2;
}
else{
```

```
x0 = x2;
f0 = f2;
}
step = step + 1;
}while(fabs(f2)>e);
printf("\nRoot is: %f", x2);
return 0;
}
```

Output:

```
Enter two Initial Guesses: 0 1
Enter Tolerable Error: 0.0001
Step x0 x1 x2 f(x2)
1 0.000000 1.000000 0.333333 -0.296296
2 0.000000 0.333333 0.257143 -0.011568
3 0.000000 0.257143 0.254202 -0.000382
4 0.000000 0.254202 0.254105 -0.000012
        Root is: 0.254105
```

3. **Write a program in C to solve a root of the equation x ^ 3 - 4x + 1 = 0 using Newton-Raphson Iterative Method**

Algorithm:

Step-1: Start
Step-2: Define function as f(x)
Step-3: Define first derivative of f(x) as g(x)
Step-4: Input initial guess (x0), tolerable error (e) and maximum iteration (N)
Step-5: Initialize iteration counter i = 1
Step-6: If g(x0) = 0 then print "Mathematical Error" and go to (12) otherwise go to (7)
Step-7: Calculate x1 = x0 - f(x0) / g(x0)
Step-8: Increment iteration counter i = i + 1
Step-9: If i >= N then print "Not Convergent" and go to (12) otherwise go to (10)
Step-10: If |f(x1)| > e then set x0 = x1 and go to (6) otherwise go to (11)
Step-11: Print root as x1
Step-12: Stop

Source Code:

```
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#define f(x) pow(x,3)-4*x+1
#define g(x) 3*pow(x,2)-4
int main(){
float x0, x1, f0, f1, g0, e;
int step = 1, N;
printf("\nEnter initial guess:");
scanf("%f", &x0);
printf("Enter tolerable error:");
scanf("%f", &e);
printf("Enter maximum iteration:");
scanf("%d", &N);
/* Implementing Newton Raphson Method */
printf("\nStep\t\tx0\t\tf(x0)\t\tx1\t\tf(x1)\n");
do{
g0 = g(x0);
f0 = f(x0);
if(g0 == 0.0){
printf("Mathematical Error.");
exit(0);
}
x1 = x0 - f0/g0;
printf("%d\t\t%f\t%f\t%f\t%f\n",step,x0,f0,x1,f1);
x0 = x1;
step = step+1;
if(step > N){
```

```
printf("Not Convergent.");
exit(0);
}
f1 = f(x1);
}while(fabs(f1)>e);
printf("\nRoot is: %f", x1);
return 0;
}
```

Output:

```
Enter initial guess:0
Enter tolerable error:0.0001
Enter maximum iteration:10
Step x0 f(x0) x1 f(x1)
1 0.000000 1.000000 0.250000 0.000000
2 0.250000 0.015625 0.254098 0.015625
Root is: 0.254098
```

4. **Write a program in C to solve the following set linear equations using Gauss Elimination Method**

x1 + x2 + x3 = 3
2x1+3x2+x3 = 6
x1- x2- x3 = - 3

Algorithm:

Step -1: Start.
Step -2: Read Number of Equation: n.
Step- 3: Read Augmented Matrix (A) of n by n+1 Size.
Step -4: Transform Augmented Matrix (A) to Upper Triangular Matrix by Row Operations.
Step -5: Obtain Solution by Back Substitution.
Step -6: Display Result.
Step -7: Stop.

Source Code:

```
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#define SIZE 10
int main()
{
float a[SIZE][SIZE], x[SIZE], ratio;
int i,j,k,n;
printf("Enter number of Eqaction: ");
scanf("%d", &n);
for(i=1;i<=n;i++)
{
for(j=1;j<=n+1;j++)
{
printf("a[%d][%d] = ",i,j);
scanf("%f", &a[i][j]);
}
}
/* Applying Gauss Elimination */
for(i=1;i<=n-1;i++)
{
if(a[i][i] == 0.0)
{
printf("Mathematical Error!");
exit(0);
}
for(j=i+1;j<=n;j++)
```

```
{
ratio = a[j][i]/a[i][i];
for(k=1;k<=n+1;k++)
{
a[j][k] = a[j][k] - ratio*a[i][k];
}
}
}
/* Obtaining Solution by Back Substitution */
x[n] = a[n][n+1]/a[n][n];
for(i=n-1;i>=1;i--)
{
x[i] = a[i][n+1];
for(j=i+1;j<=n;j++)
{
x[i] = x[i] - a[i][j]*x[j];
}
x[i] = x[i]/a[i][i];
}
/* Displaying Solution */
printf("\nSolution:\n");
for(i=1;i<=n;i++)
{
printf("x[%d] = %f\n",i, x[i]);
}
return (0);
}
```

Output:

```
Enter number of Equation: 3
a [1][1] = 1
a [1][2] = 1
a [1][3] = 1
a [1][4] = 3
a [2][1] = 2
a [2][2] = 3
a [2][3] = 1
a [2][4] = 6
a [3][1] = 1
a [3][2] = -1
a [3][3] = -1
a [3][4] = -3
Solution:
x [1] = 0.000
x [2] = 1.500
x [3] = 1.500
```

5. **Write a program in C to solve the following set linear equations using Gauss Jordon Method**

x1 + x2 + x3 = 3
2x1+3x2+x3 = 6
x1- x2- x3 = - 3

Algorithm:

Step-1: Start
Step-2: Read Number of Unknowns: n.
Step-3: Read Augmented Matrix (A) of n by n+1 Size.
Step-4: Transform Augmented Matrix (A) to Diagonal Matrix by row Operations.
Step-5: Obtain Solution by Making All Diagonal Elements to 1.
Step-6: Display Result
Step-7: Stop.

Source Code:

```
#include<stdio.h>
#include<math.h>
#define SIZE 10
int main()
{
float a[SIZE][SIZE], x[SIZE], ratio;
int i,j,k,n;
printf("Enter number of unknowns: ");
scanf("%d", &n);
printf("Enter coefficients of Augmented Matrix:\n");
for(i=1;i<=n;i++)
{
for(j=1;j<=n+1;j++)
{
printf("a[%d][%d] = ",i,j);
scanf("%f", &a[i][j]);
}
}
/* Applying Gauss Jordan Elimination */
for(i=1;i<=n;i++)
{
if(a[i][i] == 0.0)
{
printf("Mathematical Error!");
exit(0);
}
for(j=1;j<=n;j++)
{
if(i!=j)
{
```

```
ratio = a[j][i]/a[i][i];
for(k=1;k<=n+1;k++)
{
a[j][k] = a[j][k] - ratio*a[i][k];
}
}
}
}
for(i=1;i<=n;i++)
{
x[i] = a[i][n+1]/a[i][i];
}
printf("\nSolution:\n");
for(i=1;i<=n;i++)
{
printf("x[%d] = %f\n",i, x[i]);
}
return (0);
}
```

Output:

```
Enter number of Equation: 3
a [1][1] = 1
a [1][2] = 1
a [1][3] = 1
a [1][4] = 3
a [2][1] = 2
a [2][2] = 3
a [2][3] = 1
a [2][4] = 6
a [3][1] = 1
a [3][2] = -1
a [3][3] = -1
a [3][4] = -3
Solution:
x [1] = 0.000
x [2] = 1.500
x [3] = 1.500
```

6. **Write a program in C to solve the following set linear equations using Gauss Seidel Method**

2x1 + 4x2 + 2x3 = 15

2x1+x2+2x3 = -5

4x1- x2- 2x3 = 0

Algorithm:

Step -1: Start.

Step -2: Arrange given system of linear equations in diagonally dominant form

Step -3: Read tolerable error (e)

Step -4: Convert the first equation in terms of first variable, second equation in

Step -5: Set initial guesses for x0, y0, z0 and so on

Step -6: Substitute value of y0, z0 ... from step 5 in first equation obtained from step 4 to calculate. new value of x1. Use x1, z0, u0 .... in second equation obtained from step 4 to calculate new value of y1. Similarly, use x1, y1, u0... to find new z1 and so on.

Step -7: If| x0 - x1| > e and | y0 - y1| > e and | z0 - z1| > e and so, on then go to step 9.

Step -8: Set x0=x1, y0=y1, z0=z1 and so on and go to step 6

Step -9: Print value of x1, y1, z1 and so on

Step -10: Stop.

Source Code:

```
#include<stdio.h>
#include<math.h>
#define f1(x,y,z) (5-z-y)/2
#define f2(x,y,z) (15-2*z-3*x)/5
#define f3(x,y,z) (8-y-2*x)/4
/* Main function */
int main()
{
float x0=0, y0=0, z0=0, x1, y1, z1, e1, e2, e3, e;
int count=1;
printf("Enter tolerable error:\n");
scanf("%f", &e);
printf("\nCount\tx\ty\tz\n");
do
{
/* Calculation */
x1 = f1(x0, y0, z0);
y1 = f2(x1, y0, z0);
z1 = f3(x1, y1, z0);
printf("%d\t%0.4f\t%0.4f\t%0.4f\n",count, x1,y1,z1);
/* Error */
e1 = fabs(x0-x1);
```

```
e2 = fabs(y0-y1);
e3 = fabs(z0-z1);
count++;
/* Set value for next iteration */
x0 = x1;
y0 = y1;
z0 = z1;
}while(e1>e && e2>e && e3>e);
printf("\nSolution: x=%0.3f, y=%0.3f and z = %0.3f\n",x1,y1,z1);
return 0;
}
```

Output:

```
Enter tolerable error: 0.001
Count x y z
1 2.5000 1.5000 0.3750
2 1.5625 1.9125 0.7406
3 1.1734 1.9997 0.9134
4 1.0435 2.0086 0.9761
5 1.0077 2.0050 0.9949
6 1.0001 2.0020 0.9995
7 0.9993 2.0007 1.0002
Solution: x=0.999, y=2.001 and z = 1.000
```

7. **Write a program in C to solve the following differential equation by Runge-Kutta 2nd Order method or Heun's Method. dy / dx + xy = 0, y(0)=1 from x=0 to x = 0.25**

Algorithm:

Step -1: Start.
Step -2: Define function f(x,y)
Step -3: Read values of initial condition (x0 and y0),
number of steps (n) and calculation point (xn)
Step- 4: Calculate step size (h) = (xn - x0)/n
Step -5: Set i=0
Step -6: Loop
k1=h*f(x0,y0);
k2=h*f(x0+h/2,y0+k1/2);
yn = y0 + k2
i = i + 1
x0 = x0 + h
y0 = yn
While i < n
Step -7: Display yn as result
Step -8: Stop.

Source Code:

```
#include<stdio.h>
#include<math.h>
#define f(x,y) -(x*y)
int main()
{
float x0, y0, xn, h, yn, k1, k2, k3, k4, k;
int i, n;
printf("Enter Initial Condition\n");
printf("x0 = ");
scanf("%f", &x0);
printf("y0 = ");
scanf("%f", &y0);
printf("Enter calculation point xn = ");
scanf("%f", &xn);
printf("Enter number of steps: ");
scanf("%d", &n);
/* Calculating step size (h) */
h = (xn-x0)/n;
/* Runge Kutta Method */
printf("\nx0\ty0\tyn\n");
for (i=0; i < n; i++)
{
k1=h*f (x0, y0);
```

```
k2=h*f(x0+h/2,y0+k1/2);
yn = y0 + k2;
printf("%0.4f\t%0.4f\t%0.4f\n",x0,y0,yn);
x0 = x0+h;
}
/* Displaying result */
printf("\nValue of y at x = %0.2f is %0.3f",xn, yn);
return 0;
}
```

Output:

```
Enter Initial Condition
x0 = 0
y0 = 1
Enter calculation point xn = 0.25
Enter number of steps: 2
x0 y0 yn
0.0000 1.0000 0.9922
0.1250 1.0000 0.9775
Value of y at x = 0.25 is 0.978
```

8. **Write a program in C to solve the following differential equation by Runge-Kutta 4nd Order method or Heun's Method. dy / dx + xy = 0, y(0)=1 from x=0 to x = 0.25**

Algorithm:

Step- 1: Start.
Step-2: Define function f(x,y)
Step- 3: Read values of initial condition (x0 and y0), number of steps (n) and calculation point (xn)
Step- 4: Calculate step size (h) = (xn - x0)/n
Step- 5: Set i=0
Step- 6: Loop
    k1 = h * f(x0, y0)
    k2 = h * f(x0+h/2, y0+k1/2)
    k3 = h * f(x0+h/2, y0+k2/2)
    k4 = h * f(x0+h, y0+k3)
    k = (k1+2*k2+2*k3+k4)/6
    yn = y0 + k
    i = i + 1
    x0 = x0 + h
    y0 = yn
    While i < n
Step- 7: Display yn as result
Step- 8: Stop

Source Code:

```
#include<stdio.h>
#include<math.h>
#define f(x,y) -(x*y)
int main()
{
float x0, y0, xn, h, yn, k1, k2, k3, k4, k;
int i, n;
printf("Enter Initial Condition\n");
printf("x0 = ");
scanf("%f", &x0);
printf("y0 = ");
scanf("%f", &y0);

printf("Enter calculation point xn = ");
scanf("%f", &xn);
printf("Enter number of steps: ");
scanf("%d", &n);
/* Calculating step size (h) */
h = (xn-x0)/n;
```

```
/* Runge Kutta Method */
printf("\nx0\ty0\tyn\n");
for(i=0; i < n; i++)
{
k1 = h * (f(x0, y0));
k2 = h * (f((x0+h/2), (y0+k1/2)));
k3 = h * (f((x0+h/2), (y0+k2/2)));
k4 = h * (f((x0+h), (y0+k3)));
k = (k1+2*k2+2*k3+k4)/6;
yn = y0 + k;
printf("%0.4f\t%0.4f\t%0.4f\n",x0,y0,yn);
x0 = x0+h;
y0 = yn;
}
printf("\nValue of y at x = %0.2f is %0.3f",xn, yn);
return 0;
}
```

Output:

```
Enter Initial Condition
x0 = 0
y0 = 1
Enter calculation point xn = 0.25
Enter number of steps: 2
x0 y0 yn
0.0000 1.0000 0.9922
0.1250 1.0000 0.9775
Value of y at x = 0.25 is 0.978
```

9. **Write a C program to find out the approximate definite integral of the given equation: - $\int xe2x40$ using Simpson's 1/3 rule.**

Algorithm:

Step-01: Start,
Step-02: Define f(X) as per the given equation $\int xe2x40$
Step-03: Take input the intervals n, lower limit l, upper limit u.
Step-04: Calculate the step size(h) using formula h = (u-l)/n.
Step-05: Initialize the sum = f(l) + f(u),
Step-06: Start a for loop initializing i=1, taking condition i<n and increment the i = i+2 [For odd term sum],
Step-07: Inside loop calculate the sum of odd term of f(X) using the formula odsum=odsum+f(l+i*h),
Step-08: End the for loop,
Step-09: Start another for loop initializing i=2, taking condition i<n and increment the i = i+2 [For even term sum],
Step-10: Inside loop calculate the sum of even term of f(X) using the formula. evsum=evsum+f(l+i*h),
Step-11: End the for loop,
Step-12: Calculate the sum again using the formula. sum=sum+4*odsum+2*evsum;
Step-13: Calculate the final result using the formula result=(h/3)*sum,
Step-14: Display the result,
Step-15: Stop.

Source Code:

```
#include<stdio.h>
#include<math.h>
#define f(x) (x*exp(x*2))
int main()
{
int i,n;
float l,u,h,sum=0,odsum=0,evsum=0,result;
printf("Simpson's 1/3 Method\n\n");
printf("Enter the intervals:- ");
scanf("%d",&n);
printf("Enter the lower and upper limit::\n");
scanf("%f%f",&l,&u);
h=(u-l)/n;
sum=f(l)+f(u);
for(i=1;i<n;i+=2)
{
odsum=odsum+f(l+i*h);
}
for(i=2;i<n;i+=2)
{
```

```
evsum=evsum+f(l+i*h);
}
sum=sum+4*odsum+2*evsum;
result=(h/3)*sum;
printf("\nThe approximate definite integral of x*e^2x = %f\n",result);
return 0;
}
```

Output:

```
Enter the intervals:- 50
Enter the lower and upper limit::
0
4
The approximate definite integral of x*e^2x = 5216.954590
```

10. **Write a C program to find out the approximate definite integral of the given equation: - $\int xe2x40$ using Trapezoidal rule.**

Algorithm:

Step-01: Start,
Step-02: Define f(X) as per the given equation- $\int xe2x40$,
Step-03: Take input the intervals n, lower limit l, upper limit u.
Step-04: Calculate the step size(h) using formula h = (u-l)/n.
Step-05: Initialize the sum = f(l) + f(u),
Step-06: Start a for loop initializing i=2, taking condition i<n and increment the i =i+1 [For mid-term sum],
Step-07: Inside loop calculate the sum of all term of f(X) except 1st term [f(l)] and last term [f(u)] using the formula midsum = midsum + f(l+i*h),
Step-08: End the for loop,
Step-09: Calculate the sum again using the formula sum=sum+2*midsum,
Step-10: Calculate the final result using the formula result=(h/2)*sum,
Step-11: Display the result,
Step-12: Stop.

Source Code:

```
#include<stdio.h>
#include<math.h>
#define f(x) (x*exp(x*2))
int main()
{
int i,n;
float l,u,h,sum=0,midsum=0,result;
printf("Trapezoidal Rule to find approximate definite integral.\n\n");
printf("Enter the intervals:- ");
scanf("%d",&n);
printf("Enter the lower and upper limit::\n");
scanf("%f%f",&l,&u);
h=(u-l)/n;
sum=f(l)+f(u);
for(i=2;i<n;i++)
{
midsum=midsum+f(l+i*h);
}
sum=sum+2*midsum;
result=(h/2)*sum;
printf("\nThe approximate definite integral of x*e^2x = %f\n",result);
}
```

Output:

```
Enter the intervals: - 50
Enter the lower and upper limit:
0
4
The approximate definite integral of x*e^2x = 5216.954590
```