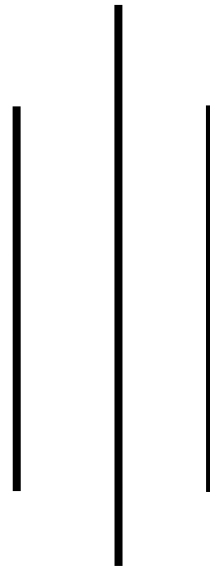




GREENFIELD NATIONAL COLLEGE

Bafal- Sitapaila, Kathmandu

Affiliated to [TU]



Lab Report of “Numerical Methods”

Submitted By:

Name: Manish karki

Semester: IV

Program: Bachelor of Computer Application

TU Reg. No: 6-2-717-6-2022

Submitted To:

Rabindra Pangen

Lecturer/ Faculty Member

BCA Department

Table of Contents

1. Bisection Method	1
2. Regula Falsi Method	3
3. Secant Method.....	5
4. Newton Raphson Method.....	7
5. Trapezoidal Method	10
6. Simpson's 1/3 Rule	12
7. Euler's Method.....	14
8. RK4 Method.....	16
9. Lagrange Interpolation Method	18
10. Gauss Jordan Method.....	20
11. Gauss Elimination Method.....	23
12. Jacobi Iteration Method	25
13. Gauss Seidel Method.....	27
14. Fixed Point Method.....	29
15. Newton's Forward Difference Polynomial	31
16. Newton's Backward Difference Interpolation	35

1.Bisection Method

Introduction

The bisection method is a numerical technique for finding the root of a function. It works by repeatedly dividing an interval $[a, b]$ in half and selecting the subinterval where the function changes sign. This process continues until the interval is sufficiently small, ensuring convergence to the root. It is simple and reliable but may be slow compared to other methods.

Algorithm

1. Start
2. Define function $f(x)$
3. Choose initial guesses x_0 and x_1 such that $f(x_0)f(x_1) < 0$
4. Choose pre-specified tolerable error e .
5. Calculate new approximated root as $x_2 = (x_0 + x_1)/2$
6. Calculate $f(x_0)f(x_2)$
 - a. if $f(x_0)f(x_2) < 0$ then $x_0 = x_0$ and $x_1 = x_2$
 - b. if $f(x_0)f(x_2) > 0$ then $x_0 = x_2$ and $x_1 = x_1$
 - c. if $f(x_0)f(x_2) = 0$ then goto (8)
7. if $|f(x_2)| > e$ then goto (5) otherwise goto (8)
8. Display x_2 as root.
9. Stop

Source Code

```
#include<stdio.h>
#include<math.h>
#define f(x) cos(x) - x * exp(x)
void main()
{
    float x0, x1, x2, f0, f1, f2, e;
    int step = 1;
    up:
    printf("\nEnter two initial guesses:\n");
    scanf("%f%f", &x0, &x1);
    printf("Enter tolerable error:\n");
    scanf("%f", &e);
    /* Calculating Functional Value */
    f0 = f(x0);
    f1 = f(x1);
    if( f0 * f1 > 0.0){
        printf("Incorrect Initial Guesses.\n");
        goto up;
    }
```

}

OUTPUT

Enter two initial guesses:

0

1

Enter tolerable error:

0.0001

Step	x_0	x_1	x_2	$f(x_2)$
1	0.000000	1.000000	0.500000	0.053222
2	0.500000	1.000000	0.750000	-0.856061
3	0.500000	0.750000	0.625000	-0.356691
4	0.500000	0.625000	0.562500	-0.141294
5	0.500000	0.562500	0.531250	-0.041512
6	0.500000	0.531250	0.515625	0.006475
7	0.515625	0.531250	0.523438	-0.017362
8	0.515625	0.523438	0.519531	-0.005404
9	0.515625	0.519531	0.517578	0.000545
10	0.517578	0.519531	0.518555	-0.002427
11	0.517578	0.518555	0.518066	-0.000940
12	0.517578	0.518066	0.517822	-0.000197
13	0.517578	0.517822	0.517700	0.000174
14	0.517700	0.517822	0.517761	-0.000012

Root is: 0.517761

2. Regula Falsi Method

Introduction

The Regula Falsi method (False Position method) is a numerical technique for finding the root of a function. It improves upon the bisection method by using a secant line to approximate the root. Instead of taking the midpoint, it finds the point where the secant line between $f(a)$ and $f(b)$ crosses the x-axis.

Algorithm

1. Start
2. Define function $f(x)$
3. Choose initial guesses x_0 and x_1 such that $f(x_0)f(x_1) < 0$
4. Choose pre-specified tolerable error e .
5. Calculate new approximated root as:
$$x_2 = x_0 - ((x_0 - x_1) * f(x_0)) / (f(x_0) - f(x_1))$$
6. Calculate $f(x_0)f(x_2)$
 - a. if $f(x_0)f(x_2) < 0$ then $x_0 = x_0$ and $x_1 = x_2$
 - b. if $f(x_0)f(x_2) > 0$ then $x_0 = x_2$ and $x_1 = x_1$
 - c. if $f(x_0)f(x_2) = 0$ then goto (8)
7. if $|f(x_2)| > e$ then goto (5) otherwise goto (8)
8. Display x_2 as root.
9. Stop

Source Code

```
#include <stdio.h>
#include <math.h>
#define EPSILON 0.0001 // Error tolerance
// Example function f(x) = x^3 - x - 1
double f(double x)
{
    return x * x * x - x - 1;
}

void regula_falsi(double a, double b)
{
    if (f(a) * f(b) >= 0)
    {
        printf("Invalid interval. f(a) and f(b) must have
opposite signs.\n");
        return;
    }

    double c;
    int step = 1;

    printf("Step\t a\t\t b\t\t c\t\t f(c)\n");
```

```

do
{
    c = (a * f(b) - b * f(a)) / (f(b) - f(a));

    printf("%d\t %.6f\t %.6f\t %.6f\t %.6f\n", step, a,
b, c, f(c));

    if (fabs(f(c)) < EPSILON)
        break;

    if (f(a) * f(c) < 0)
        b = c;
    else
        a = c;
    step++;
} while (1);

printf("\nApproximate Root: %.6f\n", c);
}

int main()
{
    double a, b;

    printf("Enter the interval (a, b): ");
    scanf("%lf %lf", &a, &b);

    regula_falsi(a, b);

    return 0;
}

```

OUTPUT

```

Enter the interval (a, b): 1
2
Step    a          b          c          f(c)
1       1.000000    2.000000    1.166667    -0.578704
2       1.166667    2.000000    1.253112    -0.285363
3       1.253112    2.000000    1.293437    -0.129542
4       1.293437    2.000000    1.311281    -0.056588
5       1.311281    2.000000    1.318989    -0.024304
6       1.318989    2.000000    1.322283    -0.010362
7       1.322283    2.000000    1.323684    -0.004404
8       1.323684    2.000000    1.324279    -0.001869
9       1.324279    2.000000    1.324532    -0.000793
10      1.324532    2.000000    1.324639    -0.000336
11      1.324639    2.000000    1.324685    -0.000143
12      1.324685    2.000000    1.324704    -0.000060

Approximate Root: 1.324704

```

3.Secant Method

Introduction

The secant method is a numerical root-finding algorithm that uses a secant line to approximate the root of a function by iteratively updating two initial guesses without requiring the function's derivative.

Algorithm

1. Start
2. Define function as $f(x)$
3. Input initial guesses (x_0 and x_1),
tolerable error (e) and maximum iteration (N)
4. Initialize iteration counter $i = 1$
5. If $f(x_0) = f(x_1)$ then print "Mathematical Error"
and goto (11) otherwise goto (6)
6. Calculate $x_2 = x_1 - (x_1 - x_0) * f(x_1) / (f(x_1) - f(x_0))$
7. Increment iteration counter $i = i + 1$
8. If $i \geq N$ then print "Not Convergent"
and goto (11) otherwise goto (9)
9. If $|f(x_2)| > e$ then set $x_0 = x_1$, $x_1 = x_2$
and goto (5) otherwise goto (10)
10. Print root as x_2
11. Stop

Source Code

```
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#define      f(x)      x*x*x - 2*x - 5
void main()
{
    float x0, x1, x2, f0, f1, f2, e;
    int step = 1, N;
    printf("\nEnter initial guesses:\n");
    scanf("%f%f", &x0, &x1);
    printf("Enter tolerable error:\n");
    scanf("%f", &e);
    printf("Enter maximum iteration:\n");
    scanf("%d", &N);
    printf("\nStep\t\tx0\t\tx1\t\tx2\t\tf(x2)\n");
    do{
        f0 = f(x0);
        f1 = f(x1);
        if(f0 == f1){
            printf("Mathematical Error.");
```

```

        exit(0);
    }
    x2 = x1 - (x1 - x0) * f1/(f1-f0);
    f2 = f(x2);
    printf("%d\t\t%f\t%f\t%f\t%f\n", step, x0, x1, x2,
f2);

    x0 = x1;
    f0 = f1;
    x1 = x2;
    f1 = f2;
    step = step + 1;
    if(step > N){
        printf("Not Convergent.");
        exit(0);
    }
}while(fabs(f2)>e);
printf("\nRoot is: %f", x2);
}

```

OUTPUT

```

Enter initial guesses:
1
2
Enter tolerable error:
0.0001
Enter maximum iteration:
10

Step          x0          x1          x2          f(x2)
1             1.000000    2.000000    2.200000    1.248001
2             2.000000    2.200000    2.088968    -0.062124
3             2.200000    2.088968    2.094233    -0.003554
4             2.088968    2.094233    2.094553    0.000012

Root is: 2.094553

```


4. Newton Raphson Method

Introduction

The Newton-Raphson method is an iterative numerical technique used to find approximations of the roots (or zeros) of a real-valued function. It starts with an initial guess and iteratively refines the guess.

Algorithm

1. Start
2. Define function as $f(x)$
3. Define first derivative of $f(x)$ as $g(x)$
4. Input initial guess (x_0), tolerable error (e) and maximum iteration (N)
5. Initialize iteration counter $i = 1$
6. If $g(x_0) = 0$ then print "Mathematical Error"
and goto (12) otherwise goto (7)
7. Calculate $x_1 = x_0 - f(x_0) / g(x_0)$
8. Increment iteration counter $i = i + 1$
9. If $i \geq N$ then print "Not Convergent"
and goto (12) otherwise goto (10)
10. If $|f(x_1)| > e$ then set $x_0 = x_1$
and goto (6) otherwise goto (11)
11. Print root as x_1
12. Stop

Source Code

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#define f(x) 3 * x - cos(x) - 1
#define g(x) 3 + sin(x)
void main()
{
    float x0, x1, f0, f1, g0, e;
    int step = 1, N;
    printf("\nEnter initial guess:\n");
    scanf("%f", &x0);
    printf("Enter tolerable error:\n");
    scanf("%f", &e);
```

```

printf("Enter maximum iteration:\n");
scanf("%d", &N);
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\t%f\t\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);
}

```

OUTPUT

Enter initial guess:

1

Enter tolerable error:

0.0001

Enter maximum iteration:

10

Step	x_0	$f(x_0)$	x_1	$f(x_1)$
1	1.000000	1.459698	0.620016	0.000000
2	0.620016	0.046179	0.607121	0.046179

Root is: 0.607121

5.Trapezoidal Method

Introduction

The Trapezoidal Method is a numerical integration technique used to approximate the definite integral of a function. It is based on dividing the integral into small trapezoidal segments and summing their areas.

Algorithm

1. Start
2. Define function $f(x)$
3. Read lower limit of integration, upper limit of integration and number of sub- interval
4. Calculate: $\text{step size} = (\text{upper limit} - \text{lower limit}) / \text{number of sub interval}$
5. Set: $\text{integration value} = f(\text{lower limit}) + f(\text{upper limit})$
6. Set: $i = 1$
7. If $i > \text{no. of sub interval}$ then goto
8. Calculate: $k = \text{lower limit} + i * h$
9. Calculate: $\text{Integration value} = \text{Integration Value} + 2 * f(k)$
10. Increment i by 1 i.e. $i = i + 1$ and go to step 7
11. Calculate: $\text{Integration value} = \text{Integration value} * \text{step size} / 2$
12. Display Integration value as required answer
13. Stop

Source Code

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
#define f(x) 1/(1+pow(x,2))
int main()
{
    float lower, upper, integration=0.0, stepSize, k;
    int i, subInterval;
    printf("Enter lower limit of integration: ");
    scanf("%f", &lower);
    printf("Enter upper limit of integration: ");
    scanf("%f", &upper);
```

```
printf("Enter number of sub intervals: ");
scanf("%d", &subInterval);
stepSize = (upper - lower)/subInterval;
integration = f(lower) + f(upper);
for(i=1; i<= subInterval-1; i++){
    k = lower + i*stepSize;
    integration = integration + 2 * f(k);
}
integration = integration * stepSize/2;
printf("\nRequired value of integration is: %.3f",
integration);
return 0;
}
```

OUTPUT

```
Enter lower limit of integration: 0
Enter upper limit of integration: 1
Enter number of sub intervals: 5

Required value of integration is: 0.784
```

6.Simpson's 1/3 Rule

Introduction

Simpson's 1/3 Rule is a numerical method used to approximate the definite integral of a function. It provides a more accurate result compared to the Trapezoidal Rule by approximating the function with a parabolic segment instead of straight-line segments.

Algorithm

1. Start
2. Define function $f(x)$
3. Read lower limit of integration, upper limit of integration and number of sub interval
4. Calculate: $\text{step size} = (\text{upper limit} - \text{lower limit}) / \text{number of sub interval}$
5. Set: $\text{integration value} = f(\text{lower limit}) + f(\text{upper limit})$
6. Set: $i = 1$
7. If $i > \text{number of sub interval}$ then goto
8. Calculate: $k = \text{lower limit} + i * h$
9. If $i \bmod 2 = 0$ then
 $\text{Integration value} = \text{Integration Value} + 2 * f(k)$
 Otherwise
 $\text{Integration Value} = \text{Integration Value} + 4 * f(k)$
 End If
10. Increment i by 1 i.e. $i = i + 1$ and go to step 7
11. Calculate: $\text{Integration value} = \text{Integration value} * \text{step size} / 3$
12. Display Integration value as required answer
13. Stop

Source Code

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
#define f(x) 1/(1+x*x)
int main()
{
    float lower, upper, integration=0.0, stepSize, k;
    int i, subInterval;
    printf("Enter lower limit of integration: ");
    scanf("%f", &lower);
    printf("Enter upper limit of integration: ");
    scanf("%f", &upper);
```

```

printf("Enter number of sub intervals: ");
scanf("%d", &subInterval);
stepSize = (upper - lower)/subInterval;
integration = f(lower) + f(upper);
for(i=1; i<= subInterval-1; i++)
{
    k = lower + i*stepSize;
    if(i%2==0){
        integration = integration + 2 * f(k);
    }
    else{
        integration = integration + 4 * f(k);
    }
}
integration = integration * stepSize/3;
printf("\nRequired value of integration is: %.3f",
integration);
return 0;
}

```

OUTPUT

```

Enter lower limit of integration: 0
Enter upper limit of integration: 1
Enter number of sub intervals: 5

Required value of integration is: 0.749

```

7.Euler's Method

Introduction

Euler's Method (RK 1st order) is a numerical technique used to approximate the solution of first-order ordinary differential equations (ODEs). It estimates the next value of y using the slope at the current point.

Algorithm

1. Start
2. Define function $f(x,y)$
3. Read values of initial condition(x_0 and y_0),
number of steps (n) and calculation point (x_n)
4. Calculate step size (h) = $(x_n - x_0)/n$
5. Set $i=0$
6. Loop
 - $y_n = y_0 + h * f(x_0 + i*h, y_0)$
 - $y_0 = y_n$
 - $i = i + 1$
7. While $i < n$
8. Display y_n as result
9. Stop

Source Code

```
#include<stdio.h>
#include<conio.h>
#define f(x,y) x+y
int main()
{
    float x0, y0, xn, h, yn, slope;
    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);
    h = (xn-x0)/n;
    printf("\nx0\ty0\tslope\ty_n\n");
    printf("-----\n");
    for(i=0; i < n; i++)
```



```

{
    slope = f(x0, y0);
    yn = y0 + h * slope;
    printf("%.4f\t%.4f\t%.4f\t%.4f\n", x0, y0, slope, yn);
    y0 = yn;
    x0 = x0+h;
}
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 = 1
Enter number of steps: 10

x0      y0      slope  yn
-----
0.0000  1.0000  1.0000  1.1000
0.1000  1.1000  1.2000  1.2200
0.2000  1.2200  1.4200  1.3620
0.3000  1.3620  1.6620  1.5282
0.4000  1.5282  1.9282  1.7210
0.5000  1.7210  2.2210  1.9431
0.6000  1.9431  2.5431  2.1974
0.7000  2.1974  2.8974  2.4872
0.8000  2.4872  3.2872  2.8159
0.9000  2.8159  3.7159  3.1875

Value of y at x = 1.00 is 3.187

```

8.RK4 Method

Introduction

The Runge-Kutta 4th Order Method (RK4) is a numerical technique used to solve first-order ordinary differential equations (ODEs) with high accuracy. It improves upon lower-order methods by taking a weighted average of slopes at different points within each step.

Algorithm

1. Start
2. Define function $f(x,y)$
3. Read values of initial condition(x_0 and y_0), number of steps (n) and calculation point (x_n)
4. Calculate step size (h) = $(x_n - x_0)/n$
5. Set $i=0$
6. Loop
 - $k_1 = h * f(x_0, y_0)$
 - $k_2 = h * f(x_0+h/2, y_0+k_1/2)$
 - $k_3 = h * f(x_0+h/2, y_0+k_2/2)$
 - $k_4 = h * f(x_0+h, y_0+k_3)$
 - $k = (k_1+2*k_2+2*k_3+k_4)/6$
 - $y_n = y_0 + k$
 - $i = i + 1$
 - $x_0 = x_0 + h$
 - $y_0 = y_n$
 - While $i < n$
7. Display y_n as result
8. Stop

Source Code

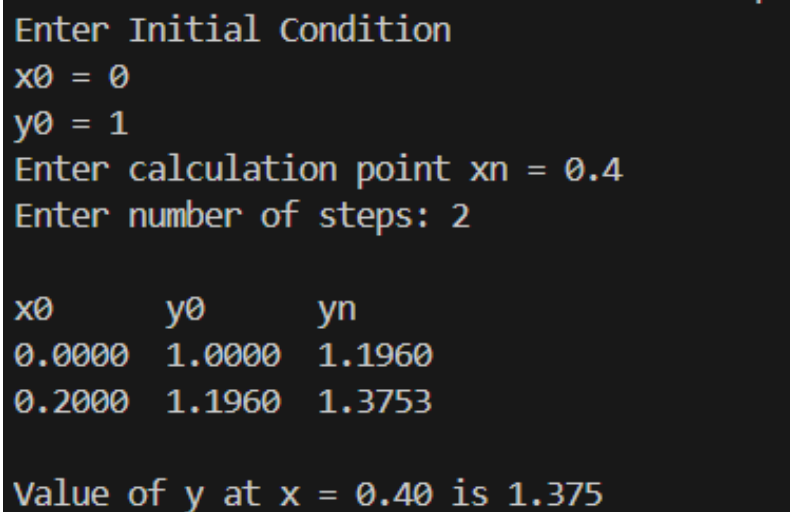
```
#include<stdio.h>
#include<conio.h>
#define f(x,y) (y*y-x*x)/(y*y+x*x)
int main()
{
    float x0, y0, xn, h, yn, k1, k2, k3, k4, k;
    int i, n;
    printf("Enter Initial Condition\n");
    printf("x0 = ");
    scanf("%f", &x0);
    printf("y0 = ");
    scanf("%f", &y0);
    printf("Enter calculation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

```

scanf("%f", &xn);
printf("Enter number of steps: ");
scanf("%d", &n);
h = (xn-x0)/n;
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 = 0.4
Enter number of steps: 2

x0      y0      yn
0.0000  1.0000  1.1960
0.2000  1.1960  1.3753

Value of y at x = 0.40 is 1.375

```

9. Lagrange Interpolation Method

Introduction

The Lagrange Interpolation Method is a numerical technique used to find the polynomial that passes through a given set of data points. It constructs an interpolating polynomial as a linear combination of Lagrange basis polynomials, ensuring that the polynomial passes exactly through all given points.

Algorithm

1. Start
2. Read number of data (n)
3. Read data X_i and Y_i for $i=1$ to n
4. Read value of independent variables say x_p whose corresponding value of dependent say y_p is to be determined.
5. Initialize: $y_p = 0$
6. For $i = 1$ to n
 - Set $p = 1$
 - For $j = 1$ to n
 - If $i \neq j$ then
 - Calculate $p = p * (x_p - X_j) / (X_i - X_j)$
 - End If
 - Next j
 - Calculate $y_p = y_p + p * Y_i$
 - Next i
7. Display value of y_p as interpolated value.
8. Stop

Source Code

```
#include<stdio.h>
#include<conio.h>
void main()
{
    float x[100], y[100], xp, yp=0, p;
    int i,j,n;
    printf("Enter number of data: ");
    scanf("%d", &n);
    printf("Enter data:\n");
    for(i=1;i<=n;i++)
    {
        printf("x[%d] = ", i);
        scanf("%f", &x[i]);
        printf("y[%d] = ", i);
        scanf("%f", &y[i]);
    }
}
```

```

printf("Enter interpolation point: ");
scanf("%f", &xp);
for(i=1;i<=n;i++)
{
    p=1;
    for(j=1;j<=n;j++)
    {
        if(i!=j)
        {
            p = p* (xp - x[j])/(x[i] - x[j]);
        }
    }
    yp = yp + p * y[i];
}
printf("Interpolated value at %.3f is %.3f.", xp, yp);
}

```

OUTPUT

```

Enter number of data: 5
Enter data:
x[1] = 1
y[1] = 1
x[2] = 2
y[2] = 1.1
x[3] = 2
y[3] = 1.3
x[4] = 3
y[4] = 4
x[5] = 5
y[5] = 3
Enter interpolation point: 2.23
Interpolated value at 2.230 is 1.100.

```

10. Gauss Jordan Method

Introduction

The Gauss-Jordan Method is a numerical technique used to solve systems of linear equations by transforming the augmented matrix into reduced row echelon form (RREF) using row operations. It is an extension of Gaussian elimination, where back-substitution is eliminated by converting the matrix directly into an identity matrix.

Algorithm

1. Start
2. Read Number of variables(x,y,z): n
3. Read Augmented Matrix (A) of n by n+1 Size
4. Transform Augmented Matrix (A) to Diagonal Matrix by Row Operations.
5. Obtain Solution by Making All Diagonal Elements to 1.
6. Display Result.
7. Stop

Source Code

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define SIZE 10

int main()
{
    double a[SIZE][SIZE + 1], x[SIZE], ratio;
    int i, j, k, n;

    printf("Enter number of variables: ");
    scanf("%d", &n);

    printf("Enter coefficients of Augmented Matrix:\n");
    for (i = 0; i < n; i++)
    {
```

```

    for (j = 0; j <= n; j++)
    {
        printf("a[%d][%d] = ", i, j);
        scanf("%lf", &a[i][j]);
    }
}

for (i = 0; i < n; i++)
{
    if (a[i][i] == 0.0)
    {
        printf("Mathematical Error: Division by zero
detected!\n");
        exit(1);
    }

    for (j = 0; j < n; j++)
    {
        if (i != j)
        {
            ratio = a[j][i] / a[i][i];
            for (k = 0; k <= n; k++)
            {
                a[j][k] = a[j][k] - ratio * a[i][k];
            }
        }
    }
}

for (i = 0; i < n; i++)
{

```

```

        x[i] = a[i][n] / a[i][i];
    }

    printf("\nSolution:\n");
    for (i = 0; i < n; i++)
    {
        printf("x[%d] = %.3lf\n", i + 1, x[i]);
    }

    return 0;
}

```

OUTPUT

```

Enter number of variables: 3
Enter coefficients of Augmented Matrix:
a[0][0] = 5
a[0][1] = 4
a[0][2] = 3
a[0][3] = 4
a[1][0] = 56
a[1][1] = 7
a[1][2] = 4
a[1][3] = 3
a[2][0] = 2
a[2][1] = 4
a[2][2] = 5
a[2][3] = 6

Solution:
x[1] = -0.061
x[2] = 0.394
x[3] = 0.909

```


11. Gauss Elimination Method

Introduction

The Gauss Elimination Method is a numerical technique used to solve systems of linear equations by transforming the augmented matrix into upper triangular form using row operations. The solution is then obtained through back-substitution.

Algorithm

1. Start
2. Read Number of variables(x,y,z): n
3. Read Augmented Matrix (A) of n by n+1 Size
4. Transform Augmented Matrix (A) to Upper Trainagular Matrix by Row Operations.
5. Obtain Solution by Back Substitution.
6. Display Result.
7. Stop

Source Code

```
#include<stdio.h>
#include<conio.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 variables: ");
    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]);
        }
    }
    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];
    }}
}
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];
}
printf("\nSolution:\n");
for(i=1;i<=n;i++){
    printf("x[%d] = %0.3f\n",i, x[i]); }
return(0);
}

```

OUTPUT

```

Enter number of variables: 3
a[1][1] = 5
a[1][2] = 4
a[1][3] = 3
a[1][4] = 24
a[2][1] = 7
a[2][2] = 4
a[2][3] = -1
a[2][4] = 20
a[3][1] = 8
a[3][2] = -5
a[3][3] = 4
a[3][4] = 24

Solution:
x[1] = 2.552
x[2] = 1.103
x[3] = 2.276

```

12. Jacobi Iteration Method

Introduction

The Jacobi Iteration Method is an iterative numerical technique used to solve systems of linear equations. It updates each variable independently in each iteration using values from the previous iteration until convergence is achieved. It is useful for diagonally dominant or strictly dominant coefficient matrices.

Algorithm

1. Start
2. Arrange given system of linear equations in diagonally dominant form
3. Read tolerable error (ϵ)
4. Convert the first equation in terms of first variable, second equation in terms of second variable and so on.
5. Set initial guesses for x_0, y_0, z_0 and so on
6. Substitute value of $x_0, y_0, z_0 \dots$ from step 5 in equation obtained in step 4 to calculate new values x_1, y_1, z_1 and so on
7. If $|x_0 - x_1| > \epsilon$ and $|y_0 - y_1| > \epsilon$ and $|z_0 - z_1| > \epsilon$ and so on then goto step 9
8. Set $x_0=x_1, y_0=y_1, z_0=z_1$ and so on and goto step 6
9. Print value of x_1, y_1, z_1 and so on
10. Stop

Source Code

```
#include<stdio.h>
#include<math.h>
#define f1(x,y,z) (-1+2*y-3*z)/5
#define f2(x,y,z) (2+3*x-z)/9
#define f3(x,y,z) (-3+2*x-y)/7
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\tty\ttz\n");
    do{
        x1 = f1(x0,y0,z0);
        y1 = f2(x0,y0,z0);
        z1 = f3(x0,y0,z0);
        printf("%d\t%0.4f\t%0.4f\t%0.4f\n",count, x1,y1,z1);
        e1 = fabs(x0-x1);
        e2 = fabs(y0-y1);
```

```

    e3 = fabs(z0-z1);
    count++;
    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:
0.0001

```

Count	x	y	z
1	-0.2000	0.2222	-0.4286
2	0.1460	0.2032	-0.5175
3	0.1917	0.3284	-0.4159
4	0.1809	0.3323	-0.4207
5	0.1854	0.3293	-0.4244
6	0.1863	0.3312	-0.4226
7	0.1861	0.3313	-0.4226

```

Solution: x=0.186, y=0.331 and z = -0.423

```

13. Gauss Seidel Method

Introduction

The Gauss-Seidel method is an iterative algorithm used to solve a system of linear equations by successively substituting the latest values into the equations to refine the solution.

Algorithm

1. Start
2. Arrange given system of linear equations in diagonally dominant form
3. Read tolerable error (ϵ)
4. Convert the first equation in terms of first variable, second equation in terms of second variable and so on.
5. Set initial guesses for x_0 , y_0 , z_0 and so on
6. Substitute value of y_0 , z_0 ... from step 5 in first equation obtained from step 4 to calculate new value of x_1 . Use x_1 , z_0 , u_0 ... in second equation obtained from step 4 to calculate new value of y_1 . Similarly, use x_1 , y_1 , u_0 ... to find new z_1 and so on.
7. If $|x_0 - x_1| > \epsilon$ and $|y_0 - y_1| > \epsilon$ and $|z_0 - z_1| > \epsilon$
8. and so on then goto step 9
9. Set $x_0 = x_1$, $y_0 = y_1$, $z_0 = z_1$ and so on and goto step 6
10. Print value of x_1 , y_1 , z_1 and so on
11. Stop

Source Code

```
#include<stdio.h>
#include<math.h>
#define f1(x,y,z)    (-1+2*y-3*z)/5
#define f2(x,y,z)    (2+3*x-z)/9
#define f3(x,y,z)    (-3+2*x-y)/7
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{
        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);
        e1 = fabs(x0-x1);
        e2 = fabs(y0-y1);
        e3 = fabs(z0-z1);
```

```

        count++;
        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:
0.0001

Count    x          y          z
1        -0.2000  0.1556  -0.5079
2         0.1670  0.3343  -0.4286
3         0.1909  0.3335  -0.4217
4         0.1864  0.3312  -0.4226
5         0.1861  0.3312  -0.4227

Solution: x=0.186, y=0.331 and z = -0.423

```

14. Fixed Point Method

Introduction

The Fixed Point Iteration Method is a numerical technique used to find an approximate root of a given equation. It is based on transforming the equation $f(x)=0$ into the form $x=g(x)$ and iterating it until convergence. The method converges if $|g'(x)| < 1$ in the given interval.

Algorithm

START

Step 1. Define the function $G(x) = (a_3x^3 + a_2x^2 + a_1x + a_0) / (-a_1)$ This is derived from rearranging $f(x) = a_3x^3 + a_2x^2 + a_1x + a_0 = 0$

Step 2: Input: a. Coefficients of the polynomial (a_3, a_2, a_1, a_0) b. Initial guess (x_0) c. Desired precision (E)

Step 3: Repeat until convergence:

- a. Compute next approximation: $x_1 = G(x_0)$
- b. Calculate relative error: $Er = |x_1 - x_0| / |x_1|$
- c. If $|Er| < E$, exit loop
- d. Set $x_0 = x_1$ and continue to next iteration

Step 4: Output the root (x_1)

END

Source Code:

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
#define G(x) (a3 * x * x * x + a2 * x * x + a0) / (-a1)
float a0, a1, a2, a3;
int main()
{
    float x0, x1, E, Er;
    printf("Enter coefficients a3,a2,a1 and a0");
    scanf("%f%f%f%f", &a3, &a2, &a1, &a0);
    printf("Enter initial guess and E\n");
    scanf("%f%f", &x0, &E);
```

```

while (1)
{
    x1 = G(x0);
    Er = (x1 - x0) / x1;
    if (fabs(Er) < E)
    {
        printf("Root=%f\n", x1);
        break;
    }
    x0 = x1;
}
getch();
}

```

Output:

```

Enter coeffients a3,a2,a1 and a0 2
3
4
1
Enter initial guessss and E
1
0.001
Root=-0.305801

```


15. Newton's Forward Difference Polynomial

Introduction

Newton's Forward Difference Polynomial is used for interpolating values based on given data points. It constructs a polynomial using forward differences.

Algorithm

1. Start
2. Read the value of n (the number of data points)
3. Read the value of x_p (the point at which the interpolated value is to be calculated)
4. For i from 0 to $n-1$:
 - a. Read the values of $x[i]$ and $fx[i]$ (the data points)
5. Calculate the step size (h):
 - a. Set $h = x[1] - x[0]$
6. Calculate the value of s :
 - a. Set $s = (x_p - x[0]) / h$
7. Initialize the forward difference table:
 - a. For i from 0 to $n-1$:
 - i. Set $fd[i] = fx[i]$
8. Calculate the forward differences:
 - a. For j from 1 to $n-1$:
 - i. For i from $n-1$ down to j :
 1. Set $fd[i] = fd[i] - fd[i - 1]$
9. Initialize the interpolated value (v):
 - a. Set $v = fd[0]$
10. Calculate the interpolated value
 - a. For i from 1 to $n-1$:
 - i. Set $p = 1$
 - ii. For k from 0 to $i-1$:
 1. Set $p = p * (s - k)$
 - iii. Update $v = v + (fd[i] * p) / \text{factorial}(i)$
11. Output the interpolated value:
 - a. Print the value of v
12. Define the factorial function:
 - a. Create a recursive function to calculate the factorial of a number

13.Stop

Source Code:

```
#include <stdio.h>

float v = 0, p, xp;
int n, i;
float x[10], fx[10], h, s;
float fd[10];

int factorial(int num);
int main()
{
    printf("Enter the number of points\n");
    scanf("%d", &n);

    printf("Enter the value at which interpolated value is to be calculated\n");
    scanf("%f", &xp);

    for (i = 0; i < n; i++)
    {
        printf("Enter the value of x and fx at i=%d\n", i);
        scanf("%f %f", &x[i], &fx[i]);
    }
    h = x[1] - x[0];
    s = (xp - x[0]) / h;

    for (i = 0; i < n; i++)
    {
```

```

    fd[i] = fx[i];
}

for (int j = 1; j < n; j++)
{
    for (i = n - 1; i >= j; i--)
    {
        fd[i] = (fd[i] - fd[i - 1]);
    }
}

v = fd[0];

for (i = 1; i < n; i++)
{
    p = 1;
    for (int k = 0; k < i; k++)
    {
        p *= (s - k);
    }
    v += (fd[i] * p) / factorial(i);
}

printf("Interpolation value = %f\n", v);
return 0;
}

int factorial(int num)
{
    if (num <= 1)

```

```
    return 1;  
    return num * factorial(num - 1);  
}
```

Output:

```
Enter the number of points  
3  
Enter the value at which interpolated value is to be calculated  
2.5  
Enter the value of x and fx at i=0  
1  
4  
Enter the value of x and fx at i=1  
2  
4  
Enter the value of x and fx at i=2  
3  
9  
Interpolation value = 5.875000
```

16. Newton's Backward Difference Interpolation

Introduction

Newton's Backward Difference Interpolation is used when the value to be interpolated lies near the end of the given dataset. It uses backward differences to construct the polynomial.

Algorithm

1. Start
2. Read the value of n (the number of data points)
3. Read the value of x_p (the point at which the interpolated value is to be calculated)
4. For i from 0 to $n-1$:
 - a. Read the values of $x[i]$ and $fx[i]$ (the data points)
5. Calculate the step size (h):
 - a. Set $h = x[1] - x[0]$
6. Calculate the value of s :
 - a. Set $s = (x_p - x[0]) / h$
7. Initialize the forward difference table:
 - a. For i from 0 to $n-1$:
 - i. Set $fd[i] = fx[i]$
8. Calculate the forward differences:
 - a. For j from 1 to $n-1$:
 - i. For i from $n-1$ down to j :
 - Set $fd[i] = fd[i] - fd[i - 1]$
9. Initialize the interpolated value (v):
 - a. Set $v = fd[0]$
10. Calculate the interpolated value:
 - a. For i from 1 to $n-1$:
 - i. Set $p = 1$
 - ii. For k from 0 to $i-1$:
 - Set $p = p * (s - k)$
 - iii. Update $v = v + (fd[i] * p) / \text{factorial}(i)$
11. Output the interpolated value:
 - a. Print the value of v
12. Define the factorial function:

- a. Create a recursive function to calculate the factorial of a number
- 13.Stop

Source Code:

```
#include <stdio.h>
```

```
// Function prototype for factorial
```

```
int factorial(int num);
```

```
int main()
```

```
{
```

```
    float v = 0, p, xp;
```

```
    int n, i;
```

```
    float x[10], fx[10], h, s;
```

```
    float fd[10];
```

```
    printf("Enter the number of points\n");
```

```
    scanf("%d", &n);
```

```
    printf("Enter the value at which interpolated value is to be calculated\n");
```

```
    scanf("%f", &xp);
```

```
    for (i = 0; i < n; i++)
```

```
    {
```

```
        printf("Enter the value of x and fx at i=%d\n", i);
```

```
        scanf("%f %f", &x[i], &fx[i]);
```

```
    }
```

```
    h = x[1] - x[0];
```

```
    s = (xp - x[0]) / h;
```

```

for (i = 0; i < n; i++)
{
    fd[i] = fx[i];
}

for (int j = 1; j < n; j++)
{
    for (i = n - 1; i >= j; i--)
    {
        fd[i] = (fd[i] - fd[i - 1]);
    }
}

v = fd[0];

for (i = 1; i < n; i++)
{
    p = 1;
    for (int k = 0; k < i; k++)
    {
        p *= (s - k);
    }
    v += (fd[i] * p) / factorial(i);
}

printf("Interpolation value = %f\n", v);
return 0;
}

```

```
int factorial(int num)
{
    if (num <= 1)
        return 1;
    return num * factorial(num - 1);
}
```

Output:

```
Enter the number of points
3
Enter the value at which interpolated value is to be calculated
2.5
Enter the value of x and fx at i=0
1
1
Enter the value of x and fx at i=1
2.5
2.5
Enter the value of x and fx at i=0
1
2.5
2.5
Enter the value of x and fx at i=0
1
1
Enter the value of x and fx at i=1
2
4
Enter the value of x and fx at i=2
3
9
Interpolation value = 6.250000
```