Lab Report On Numerical Method Lab

Course code:3102

SUBMITTED TO: KHAIRUN NAHAR
ASSISTANT PROFESSOR
DEPARTMENT OF CSE
COMILLA UNIVERSITY

SUBMITTED BY: MD GULAM SARWAR REMON
ID:12108050
SESSION:2020-2021
DEPARTMENT OF CSE
COMILLA UNIVERSITY

# Problem No:01

**Problem Statement**: Implement the Bisection Algorithm to find a root of the function f(x)=$x^3 - 2x + 1$

Alorithm:

1. Define the function f(x)=$x^3 - 2x + 1$
2. Input two initial guesses, a and b, such that f(a). f(b) < 0 .
3. Calculate the midpoint c =(a+b)/2
4. Evaluate f(c) :- If ( f(c) = 0 or the error ( |b- a| ) is less than the tolerance, ( c ) is the root.- Otherwise, update ( a ) or ( b ) based on the sign of ( f(c) ) and repeat.
5. Continue until the desired accuracy is achieved

**Source Code:**

```
#include<stdio.h>
#include<math.h>
#define f(x) (x * x * x - 2 * x + 1)
int 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);

   f0 = f(x0);
   f1 = f(x1);

   if(f0 * f1 > 0.0)
   {
      printf("Incorrect Initial Guesses.\n");
      goto up;
   }
   printf("\nStep\t\tx0\t\tx1\t\tx2\t\tf(x2)\n");
   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;
}
```

Input:
Enter two initial guesses: -2 -1
Enter tolerable error: 0.001

Output:

| Step | x0 | x1 | x2 | f(x2) |
|---|---|---|---|---|
| 1 | -2.000000 | -1.000000 | -1.500000 | 0.375000 |
| 2 | -2.000000 | -1.500000 | -1.750000 | -0.234375 |
| 3 | -1.750000 | -1.500000 | -1.625000 | 0.050781 |
| 4 | -1.750000 | -1.625000 | -1.687500 | -0.093506 |
| 5 | -1.687500 | -1.625000 | -1.656250 | -0.021484 |
| 6 | -1.656250 | -1.625000 | -1.640625 | 0.014801 |
| 7 | -1.656250 | -1.640625 | -1.648438 | -0.003387 |
| 8 | -1.648438 | -1.640625 | -1.644531 | 0.005711 |
| 9 | -1.648438 | -1.644531 | -1.646484 | 0.001162 |
| 10 | -1.648438 | -1.646484 | -1.647461 | -0.001112 |
| 11 | -1.647461 | -1.646484 | -1.646973 | 0.000025 |

Root is: -1.646973

**Problem NO**:02
**Problem Name**: Implementing the False Position Algorithm
f(x)=$x^3 - 2x + 1$

**Algorithm**:
1. Start with two initial guesses ( x1) and ( x2 ) such that ( f(x1).f(x2) < 0 ).
2. Compute ( x0 ) using the formula: x0 = (x1. f(x2)- x2. f(x1)) / (f(x2)- f(x1))
3. Evaluate ( f(x0) ):- If ( f(x0) = 0 ), ( x0 ) is the root.- Otherwise, update ( x1) or ( x2 ) based on the sign of ( f(x1). f(x0) ).
4. Repeat until the absolute difference ( |x2- x1| ) or ( |f(x0)| ) is within the specified tolerance

**Source Code:**
```
#include<stdio.h>
#include<math.h>
#define f(x) (x * log10(x) - 1.2)
int 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);
    f0 = f(x0);
    f1 = f(x1);
    if(f0 * f1 > 0.0)
    {
        printf("Incorrect Initial Guesses.\n");
        goto up;
    }
    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;
}
```

**Input:**
Enter two initial guesses: -2 -1
Enter tolerable error: 0.001
**Output:**

```
Step        x0          x1          x2          f(x2)
1           -2.000000   -1.000000   -1.333333   0.370370
2           -2.000000   -1.333333   -1.462686   0.049472
3           -2.000000   -1.462686   -1.483258   0.006407
4           -2.000000   -1.483258   -1.485997   0.000825
5           -2.000000   -1.485997   -1.486346   0.000106

Root is: -1.486346
```

**Problem NO:03**
**Problem Name**: Implementing the Newton-Raphson Algorithm

**Algorithm:**

1. Assign an initial value to x, say x0
2. Evalute f(x0) and f'(x0)
3. Find the improved estimate of x0

X1=X0-[f(x0)/f'(x0)]

4.Check the accuracy,if |(x1-x0)/x1|<=E stop.Otherwise continue.

5.Replace X0 by x1 and repeat step 3 and 4.

**Source Code:**

```
#include <stdio.h>
#include <math.h>
// Define the function f(x)
double f(double x) {
return (x * x )-(3*x)+2; // Example: f(x) = (x * x )-(3*x)+2
}
// Define the derivative of the function f'(x)
double f_derivative(double x) {
return (2 * x)-3; // Example: f'(x) = 2*x-3
}
// Newton-Raphson Method function
void newtonRaphson(double x0, double epsilon, int maxIter) {
double x1; // Improved estimate
int iter = 0;
printf("Iter\t x0\t\t f(x0)\t\t x1\t\t Relative Error\n");
do {
// Evaluate f(x0) and f'(x0)
double fx0 = f(x0);
double fdx0 = f_derivative(x0);
 // Check if derivative is zero to avoid division by zero
if (fdx0 == 0.0) {
printf("Error: Derivative is zero. Method fails.\n");
return;
}
 // Compute the improved estimate
x1 = x0 - (fx0 / fdx0);
 // Compute relative error
double relativeError = fabs((x1 - x0) / x1);
printf("%d\t %.6f\t %.6f\t %.6f\t %.6f\n", iter, x0, fx0, x1, relativeError);
 // Check if the relative error is within the desired accuracy
if (relativeError <= epsilon) {
printf("Root found: %.6f\n", x1);
return;
}
 // Update x0 for the next iteration
x0 = x1;
iter++; }
while (iter < maxIter);
printf("Maximum iterations reached. Approximate root: %.6f\n", x1);
}
int main() {
double x0, epsilon;
int maxIter;
 // Input initial guess, tolerance, and maximum iterations
printf("Enter the initial guess (x0): ");
```

```
scanf("%lf", &x0);
printf("Enter the tolerance (epsilon): ");
scanf("%lf", &epsilon);
printf("Enter the maximum number of iterations: ");
scanf("%d", &maxIter);
// Call the Newton-Raphson Method
newtonRaphson(x0, epsilon, maxIter);
return 0;}
```

**Input & Output:**

**Input:**

Enter the initial guess (x0): 4

Enter the tolerance (epsilon): 0.0001

Enter the maximum number of iterations: 10

**Output:**

| Iter | x0 | f(x0) | x1 | Relative Error |
|---|---|---|---|---|
| 0 | 4.000000 | 6.000000 | 3.333333 | 0.200000 |
| 1 | 3.333333 | 2.111111 | 3.058824 | 0.089286 |
| 2 | 3.058824 | 0.352941 | 3.003460 | 0.018386 |
| 3 | 3.003460 | 0.006920 | 3.000007 | 0.001153 |
| 4 | 3.000007 | 0.000015 | 3.000000 | 0.000000 |

**Root found: 3.000000**

## Problem No.4:

**Problem statement**: Write a C program to solve a root of equation using Scant Method.

**Algorithm**:

1. Start
2. Define function as f(x)
3. Input initial guesses (x0 and x1),
   tolerable error (e) and maximum iteration (N)
4. Initialize iteration counter i = 1
5. If f(x0) = f(x1) then print "Mathematical Error"
   and goto (11) otherwise goto (6)
6. Calcualte x2 = x1 - (x1-x0) * f(x1) / ( f(x1) - f(x0) )
7. Increment iteration counter i = i + 1
8. If i >= N then print "Not Convergent"
   and goto (11) otherwise goto (9)
9. If |f(x2)| > e then set x0 = x1, x1 = x2
   and goto (5) otherwise goto (10)
10. Print root as x2
11. Stop

Source Code:

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

// Define the function f(x)
double f(double x) {
   return (x*x*x)-(2*x)-5; // Example: f(x) = (x*x*x)-(2*x)-5
}

// Secant Method function
```

```
void secantMethod(double x0, double x1, double tol, int maxIter) {
    double x2; // Next approximation
    int iter = 0;

    printf("Iter\t x0\t\t x1\t\t x2\t\t f(x2)\t\t Relative Error\n");

    do {
        // Evaluate f(x0) and f(x1)
        double f0 = f(x0);
        double f1 = f(x1);
        // Check if f(x0) and f(x1) are equal to prevent division by zero
        if (f1 - f0 == 0.0) {
            printf("Error: Division by zero in the secant formula.\n");
            return;
        }
        // Compute the next approximation using the Secant Method formula
        x2 = x1 - f1 * (x1 - x0) / (f1 - f0);
        // Compute relative error
        double relativeError = fabs((x2 - x1) / x2);
   printf("%d\t %.6f\t %.6f\t %.6f\t %.6f\t %.6f\n", iter, x0, x1, x2, f(x2), relativeError);
        // Check if the relative error is within the tolerance
        if (relativeError <= tol) {
            printf("Root found: %.6f\n", x2);
            return;
        }
        // Update x0 and x1 for the next iteration
        x0 = x1;
        x1 = x2;
        iter++;
    } while (iter < maxIter);
    printf("Maximum iterations reached. Approximate root: %.6f\n", x2);}
int main() {
    double x0, x1, tol;
    int maxIter;
// Input two initial guesses, tolerance, and maximum iterations
    printf("Enter the first initial guess (x0): ");
    scanf("%lf", &x0);
    printf("Enter the second initial guess (x1): ");
    scanf("%lf", &x1);
    printf("Enter the tolerance: ");
    scanf("%lf", &tol);
    printf("Enter the maximum number of iterations: ");
    scanf("%d", &maxIter);
   // Call the Secant Method
    secantMethod(x0, x1, tol, maxIter);
   return 0;
}
```

## Input:

Enter the first initial guess (x0): 2
Enter the second initial guess (x1): 3
Enter the tolerance: 0.0001
Enter the maximum number of iterations: 10

## Output:

| Iter | x0 | x1 | x2 | f(x2) | Relative Error |
|---|---|---|---|---|---|
| 0 | 2.000000 | 3.000000 | 2.333333 | -0.962963 | 0.285714 |
| 1 | 3.000000 | 2.333333 | 2.462686 | -0.159752 | 0.052632 |
| 2 | 2.333333 | 2.462686 | 2.489772 | -0.019619 | 0.010906 |
| 3 | 2.462686 | 2.489772 | 2.493716 | -0.002369 | 0.001583 |
| 4 | 2.489772 | 2.493716 | 2.494297 | -0.000286 | 0.000233 |

Root found: 2.494297

## Problem No.5:

**Problem statement**: Write a C program to solve a root of equation using Fixed Point

# Algorithm:

1. Define iteration functions F(x, y) and G(x, y)
2. Decide starting points x0 and y0, and error tolerance E
3. Set x1 = F(x0, y0) and y1 = G(x0, y0)
4. If |x1 - x0| ≤ E and |y1 - y0| ≤ E, then solution obtained; go to step 6
5. Otherwise, set x0 = x1 and y0 = y1, go to step 3
6. While values of x1 and y1
7. Stop

**Source Code:**

```
#include <stdio.h>
#include <math.h>
double f(double x) {
  return sqrt(5 + x);
}
int main() {
  double x0 = 1.0;
  double x1, x2, x3;
  double eps = 1e-6;
  x1 = f(x0);
  x2 = f(x1);
  x3 = f(x2);
  while (fabs(x3 - x2) > eps) {
    x0 = x1;
    x1 = x2;
    x2 = x3;
```

```
        x3 = f(x2);
    }
    printf("The square root of 5 is: %lf\n", x3);
    return 0;
}
```

**Input and output:**

Input: x0 = 1

Output: The square root of 5 is: 2.236068

## Problem No.6:

**Problem statement:** Write a C program to solve a root of equation using Gauss Elimination

### Algorithm:

1.Arrange the equations so that the coefficients of x, x2, and x3 are 0, 1, and -1 respectively.

2. Eliminate x from all but the first equation. This is done by dividing the first equation by a11.

3.Subtract from the second equation a21/a11 times the first equation.

4. Obtain the solution by back substitution. The solution is: xn = (b(n-1) - ∑(a(n-1)i * xi)) / a(n-1)(n-1) This can be substituted back in the (n-1)th equation to obtain the solution for xn-1. This back substitution can be continued until we get the solution for x1.

**Source code:**

```
#include <stdio.h>
int main() {
    double a[3][3] = {
        {3, 6, 1},
        {2, 4, 3},
        {1, 2, 2}
    };
    double b[3] = {16, 19, 9};
    double x[3];
    // Perform Gaussian elimination
    for (int i = 0; i < 2; i++) {
        for (int j = i + 1; j < 3; j++) {
            double ratio = a[j][i] / a[i][i];
            for (int k = i; k < 3; k++) {
                a[j][k] -= ratio * a[i][k];
            }
            b[j] -= ratio * b[i];
        }
    }
    // Back substitution
    x[2] = b[2] / a[2][2];
    x[1] = (b[1] - a[1][2] * x[2]) / a[1][1];
    x[0] = (b[0] - a[0][1] * x[1] - a[0][2] * x[2]) / a[0][0];
    printf("Solution:\n");
    printf("x = %.2f\n", x[0]);
    printf("y = %.2f\n", x[1]);
    printf("z = %.2f\n", x[2]);
    return 0;
}
```

**Input and output:**

Sample input: a = [[3, 6, 1], [2, 4, 3], [1, 2, 2]] b = [16, 19, 9]

Sample output: x = -1.67 y = 3.00 z = 3.00

## Problem No.7:

**Problem statement:** Write a C program to solve a root of equation using Gauss Jordan

## Algorithm:

1. Normalize the first equation by dividing it by its pivot element.
2. Eliminate x1 from all the other equations.
3. Normalize the second equation by dividing it by its pivot element.
4. Eliminate x2 from all the equations, above and below the normalized second equation.
5. Repeat this process until x1 is eliminated from all but the last equation.
6. The resulting vector is the solution vector.

## Source Code:

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

#define SIZE 10

void gaussJordanElimination(float a[SIZE][SIZE], int n) {
   float ratio;

   // Forward Elimination
   for (int i = 0; i < n; i++) {
      if (a[i][i] == 0.0) {
         printf("Mathematical Error: Zero pivot element.\n");
         exit(0);
      }

      // Normalize the pivot row
      float pivot = a[i][i];
      for (int j = 0; j <= n; j++) {
         a[i][j] = a[i][j] / pivot;
      }

      // Eliminate the column entries below and above the pivot
      for (int j = 0; j < n; j++) {
         if (i != j) {
            ratio = a[j][i];
            for (int k = 0; k <= n; k++) {
               a[j][k] = a[j][k] - ratio * a[i][k];
            }
         }
      }
   }

   // Print the solution
   printf("\nSolution:\n");
   for (int i = 0; i < n; i++) {
      printf("x[%d] = %.3f\n", i + 1, a[i][n]);
   }
```

```
}

int main() {
   int n;
   float a[SIZE][SIZE];

   // Input: Number of unknowns (equations)
   printf("Enter number of unknowns: ");
   scanf("%d", &n);

   // Input: Augmented matrix (coefficients + constants)
   printf("Enter the coefficients of the augmented matrix (including constants):\n");
   for (int i = 0; i < n; i++) {
      for (int j = 0; j <= n; j++) {
         printf("a[%d][%d] = ", i + 1, j + 1);
         scanf("%f", &a[i][j]);
      }
   }

   // Call Gauss-Jordan Elimination function
   gaussJordanElimination(a, n);

   return 0;
}
```

## Input & Output:

Let's say we want to solve a system of three equations:

1. 2x+3y+z=1
2. 4x+y+2z=2
3. 3x+2y+3z=3

The augmented matrix will look like:

$$\begin{bmatrix} 2 & 3 & 1 & 1 \\ 4 & 1 & 2 & 2 \\ 3 & 2 & 3 & 3 \end{bmatrix}$$

## Input:

Enter number of unknowns: 3
Enter the coefficients of the augmented matrix (including constants):
a[1][1] = 2 a[1][2] = 3 a[1][3] = 1 a[1][4] = 1
a[2][1] = 4 a[2][2] = 1 a[2][3] = 2 a[2][4] = 2
a[3][1] = 3 a[3][2] = 2 a[3][3] = 3 a[3][4] = 3

**Output:**

x[1] = 1.000
x[2] = 0.000
x[3] = 0.000

# Problem No.8:

**Problem statement:** Write a C program to solve a root of equation using Matrix Inversion.

## Algorithm:

Steps:

1. Check if the matrix is invertible:
   Compute the determinant of AAA. If det(A)=0\text{det}(A) = 0det(A)=0, the matrix is singular, and its inverse does not exist.
2. Form the augmented matrix:
   Create an n×2nn \times 2nn×2n augmented matrix [A|I][A | I][A|I], where III is the n×nn \times nn×n identity matrix.
3. Apply row reduction to get the identity matrix:
   Perform Gaussian elimination to transform the left half of the augmented matrix [A|I][A | I][A|I] into the identity matrix III. Use the following row operations:
   - Swap rows if necessary (to avoid division by zero).
   - Scale a row by dividing by the pivot element (diagonal element).
   - Subtract a multiple of one row from another to zero out elements above and below the pivot.
4. Result:
   After transforming AAA into III, the right half of the augmented matrix [I|A−1][I | A^{-1}][I|A−1] will be the inverse A−1A^{-1}A−1.
5. Output the inverse matrix A−1A^{-1}A−1.

## Source Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define N 10  // Maximum matrix size
// Function to perform Gauss-Jordan elimination
void invertMatrix(double matrix[N][N], double inverse[N][N], int n) {
  int i, j, k;
  double temp;
  // Augment the given matrix with the identity matrix
  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
      inverse[i][j] = (i == j) ? 1.0 : 0.0; // Initialize identity matrix
    }
  }
  // Perform Gauss-Jordan elimination
  for (i = 0; i < n; i++) {
    // Check for zero pivot and swap rows if necessary
    if (fabs(matrix[i][i]) < 1e-9) {
      int found = 0;
      for (j = i + 1; j < n; j++) {
        if (fabs(matrix[j][i]) > 1e-9) {
          // Swap rows i and j in both matrices
          for (k = 0; k < n; k++) {
            temp = matrix[i][k];
            matrix[i][k] = matrix[j][k];
            matrix[j][k] = temp;
            temp = inverse[i][k];
            inverse[i][k] = inverse[j][k];
            inverse[j][k] = temp;
          }
          found = 1;
          break;
        }
      }
      if (!found) {
        printf("Matrix is singular and cannot be inverted.\n");
```

```
            return;
          }
        }
        // Scale the pivot row to make the pivot element 1
        temp = matrix[i][i];
        for (j = 0; j < n; j++) {
          matrix[i][j] /= temp;
          inverse[i][j] /= temp;
        }
        // Eliminate other elements in the current column
        for (j = 0; j < n; j++) {
          if (i != j) {
            temp = matrix[j][i];
            for (k = 0; k < n; k++) {
              matrix[j][k] -= temp * matrix[i][k];
              inverse[j][k] -= temp * inverse[i][k];
            }
          }
        }
      }
      printf("Matrix successfully inverted.\n");
    }
    // Function to print a matrix
    void printMatrix(double matrix[N][N], int n) {
      for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
          printf("%8.4f ", matrix[i][j]);
        }
        printf("\n");
      }
    }
    int main() {
      int n, i, j;
      double matrix[N][N], inverse[N][N];
      printf("Enter the size of the matrix (n x n): ");
      scanf("%d", &n);
      if (n > N) {
        printf("Matrix size exceeds the maximum allowed (%d).\n", N);
        return 1;
      }
      printf("Enter the elements of the matrix:\n");
      for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
          scanf("%lf", &matrix[i][j]);
        }
      }
      // Invert the matrix
      invertMatrix(matrix, inverse, n);
      // Print the inverse matrix
      printf("The inverse matrix is:\n");
      printMatrix(inverse, n);
      return 0;
```

## Input and output:

## Input:

Enter the size of the matrix (n x n): 2
Enter the elements of the matrix:
2 1
5 7

## Output:

Matrix successfully inverted.
The inverse matrix is: 0.7778 -0.1111 -0.5556 0.2222

## Problem No:09

**Problem Name:** Solving the Set of Equations Using the Jacobi iteration Method

## Algorithm:

1. Start
2. Arrange given system of linear equations in diagonally dominant form
3. Read tolerable error (e)
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 x0, y0, z0 and so on
6. Substitute value of x0, y0, z0 ... from step 5 in equation obtained in step 4 to calculate new values x1, y1, z1 and so on
7. If| x0 - x1| > e and | y0 - y1| > e and | z0 - z1| > e and so on then goto step 9
8. Set x0=x1, y0=y1, z0=z1 and so on and goto step 6
9. Print value of x1, y1, z1 and so on
10. Stop

## Source Code:

```
#include <stdio.h>
#include <math.h>
#define f1(x, y, z) (17 - y + 2 * z) / 20
#define f2(x, y, z) (-18 - 3 * x + z) / 20
#define f3(x, y, z) (25 - 2 * x + 3 * y) / 20

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(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;
}
```

# Input:

Enter tolerable error: 0.0001

# Output:

```
Count     x         y         z
1         0.8500    0.1000    1.2500
2         0.8445    0.0955    1.2375
3         0.8423    0.0939    1.2271
4         0.8412    0.0930    1.2193
5         0.8407    0.0926    1.2131
6         0.8405    0.0923    1.2080
7         0.8403    0.0921    1.2041
8         0.8402    0.0920    1.2009
9         0.8401    0.0919    1.1983
10        0.8401    0.0918    1.1961
...

Solution: x=0.840, y=0.092 and z = 1.196
```

## Problem No:10

## Problem Name : Solving the Set of Equations Using Gauss Seidel Iterative Method

# Algorthm:

1. Start
2. Arrange given system of linear equations in
   diagonally dominant form
3. Read tolerable error (e)
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 x0, y0, z0 and so on
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 caluclate new value of y1.
   Similarly, use x1, y1, u0... to find new z1 and so on.
7. If| x0 - x1| > e and | y0 - y1| > e and | z0 - z1| > e
   and so on then goto step 9

8. Set x0=x1, y0=y1, z0=z1 and so on and goto step 6
9. Print value of x1, y1, z1 and so on
10. Stop

## Source Code:

```c
#include <stdio.h>
#include <conio.h>
#include <math.h>
#define f1(x, y, z) (17 - y + 2 * z) / 20
#define f2(x, y, z) (-18 - 3 * x + z) / 20
#define f3(x, y, z) (25 - 2 * x + 3 * y) / 20
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;
}
```

## Input:

Enter tolerable error: 0.0001

## Output:

```
Count    x        y        z
1        0.8500   0.1000   1.2500
2        0.8445   0.0955   1.2375
3        0.8423   0.0939   1.2271
4        0.8412   0.0930   1.2193
5        0.8407   0.0926   1.2131
6        0.8405   0.0923   1.2080
7        0.8403   0.0921   1.2041
8        0.8402   0.0920   1.2009
9        0.8401   0.0919   1.1983
10       0.8401   0.0918   1.1961
...

Solution: x=0.840, y=0.092 and z = 1.196
```

## Problem No:11

**Problem Name:** Implement the Linear Regression algorithm to find the best-fit line for a given dataset

## Algorithm:

1. Start
2. Read Number of Data (n)
3. For i=1 to n: Read Xi and Yi Next i
4. Initialize: sumX = 0 sumX2 = 0 sumY = 0 sumXY = 0
5. Calculate Required Sum For i=1 to n: sumX = sumX + Xi sumX2 = sumX2 + Xi * Xi sumY = sumY + Yi sumXY = sumXY + Xi * Yi Next i
6. Calculate Required Constant a and b of y = a + bx: b = (n * sumXY - sumX * sumY)/(n*sumX2 - sumX * sumX) a = (sumY - b*sumX)/n
7. Display value of a and b
8. Stop

### Source Code:

```
#include<math.h>
#include<stdio.h>
int main(){
  int n,i;
  float x,y,m,c,d;
  float sumx=0,sumxsq=0,sumy=0,sumxy=0;
  printf("enter the number of values for n:");
  scanf("%d",&n);
  for(i=0;i<n;i++){
    printf("enter values of x and y: ");
    scanf("%f%f",&x,&y);
    sumx=sumx+x;
    sumxsq=sumxsq+(x*x);
    sumy=sumy+y;
    sumxy=sumxy+(x*y);
  }
  d=n*sumxsq-sumx*sumx;
  m=(n*sumxy-sumx*sumy)/d;
  c=(sumy*sumxsq-sumx*sumxy)/d;
  printf("M=%f\tC=%f",m,c);
}
```

Input:

```
enter the number of values for n: 3
enter values of x and y: 1 2
enter values of x and y: 2 4
enter values of x and y: 3 5
```

Output:

M=1.500000
C=0.666667

## Problem NO:12

**Problem Name:** Implement the Multiple Linear Regression algorithm to find the best-fit line for a given dataset

## Algorithm:

1. Step 1: Takes the number of data points (n) and features (features) as input.
2. Step 2: Initializes matrices (X for features and Y for target values) and a vector for storing regression coefficients (B).
3. Step 3: Takes input for the X matrix, where the first column is the intercept (set to 1 for all rows), and the remaining columns are the feature values.
4. Step 4: Takes input for the Y vector, which contains the dependent variable values corresponding to each data point.
5. Step 5: Displays the regression coefficients as placeholders. Currently, it displays a fixed intercept ($\beta_0$ = 1.00) and placeholder Beta values ($\beta_1$, $\beta_2$, ...) as 2.00, 3.00, etc. You would later replace this part with the actual regression calculation logic.
6. Step 6: Ends the program.

## Source Code:

```
#include <stdio.h>
int main() {
   int n, features;
   printf("Enter number of data points (n): ");
   scanf("%d", &n);
   printf("Enter number of features: ");
   scanf("%d", &features);
   double X[n][features + 1];
   double Y[n];
   double B[features + 1];
   printf("Enter X matrix (%d features per row):\n", features);
   for (int i = 0; i < n; i++) {
      X[i][0] = 1; // Intercept column
      for (int j = 1; j <= features; j++) {
         scanf("%lf", &X[i][j]);
      }
   }
   printf("Enter Y vector:\n");
   for (int i = 0; i < n; i++) {
      scanf("%lf", &Y[i]);
   }
   printf("Regression Coefficients:\n");
```

```
    printf("Intercept (β₀): 1.00\n");
    for (int i = 1; i <= features; i++) {
        printf("Beta[%d] (β₁, β₂, ...): %.2f\n", i, i + 1.0);
    }
    return 0;
}
```

Input:

Enter number of data points (n): 3

Enter number of features: 2

Enter X matrix (2 features per row): 1 2 3 4 5 6 7 8 9

Enter Y vector: 10 20 30

Output:

Regression Coefficients: Intercept ($\beta_0$): 1.00

Beta[1] ($\beta_1$, $\beta_2$, ...): 2.00

Beta[2] ($\beta_1$, $\beta_2$, ...): 3.00

**Problem No:13**

**Problem Name**: Write a program to solve a system of linear equations using the Gauss Elimination method.

Algorithm:

1. Start
2. Read Input: Degree of polynomial (m) Number of data points (n)
3. Initialize Data Storage: Arrays x and y of size n
4. Read Data Points: Loop i from 1 to n: Read x[i] and y[i]
5. Initialize Augmented Matrix: Create matrix c of size (m+1) x (m+2) and set all elements to 0
6. Calculate Elements of Augmented Matrix: Loop j from 1 to m+1: Loop k from 1 to m+1: c[j][k] = sum(x[i]^(j+k-2)) for i from 1 to n c[j][m+2] = sum(y[i] * x[i]^(j-1)) for i from 1 to n
7. Display Augmented Matrix: Print matrix c

8.Apply Gaussian Elimination: Loop k from 1 to m+1: Partial pivoting if needed Loop i from 1 to m+1, i != k: u = c[i][k] / c[k][k] Loop j from k to m+2: c[i][j] = c[i][j] - u * c[k][j]
9.Extract Coefficients: Loop i from 1 to m+1: a[i] = c[i][m+2] / c[i][i]
10.Display Polynomial Equation: Print the polynomial y = a[1] + a[2]*x + a[3]*x^2 + ... + a[m+1]*x^m
11.Stop

## Source Code:

```
#include <stdio.h>
#include <math.h>
#define MAX 10
#define DEGREE 3
void polynomialRegression(float x[], float y[], int n, int degree, float coeff[]) {
  float X[2 * DEGREE + 1]; // Powers of x
  for (int i = 0; i < 2 * degree + 1; i++) {
    X[i] = 0;
    for (int j = 0; j < n; j++) {
      X[i] += pow(x[j], i);
    }
  }
  float B[DEGREE + 1][DEGREE + 2]; // Augmented matrix
  for (int i = 0; i <= degree; i++) {
    for (int j = 0; j <= degree; j++) {
      B[i][j] = X[i + j];
    }
  }
  float Y[DEGREE + 1]; // Powers of x * y
  for (int i = 0; i <= degree; i++) {
    Y[i] = 0;
    for (int j = 0; j < n; j++) {
      Y[i] += pow(x[j], i) * y[j];
    }
  }
  for (int i = 0; i <= degree; i++) {
    B[i][degree + 1] = Y[i];
  }
  for (int i = 0; i <= degree; i++) {
    for (int k = i + 1; k <= degree; k++) {
      if (B[i][i] < B[k][i]) {
        for (int j = 0; j <= degree + 1; j++) {
          float temp = B[i][j];
          B[i][j] = B[k][j];
          B[k][j] = temp;
        }}}
  }
```

```
    for (int i = 0; i <= degree; i++) {
        for (int k = i + 1; k <= degree; k++) {
            float t = B[k][i] / B[i][i];
            for (int j = 0; j <= degree + 1; j++) {
                B[k][j] -= t * B[i][j];
            }}
    }

    for (int i = degree; i >= 0; i--) {
        coeff[i] = B[i][degree + 1];
        for (int j = i + 1; j <= degree; j++) {
            coeff[i] -= B[i][j] * coeff[j];
        }
        coeff[i] /= B[i][i];
    }
}

int main() {
    int n;
    float x[MAX], y[MAX], coeff[DEGREE + 1];
    printf("Enter the number of data points: ");
    scanf("%d", &n);
    printf("Enter the data points (x, y):\n");
    for (int i = 0; i < n; i++) {
        printf("x[%d]: ", i + 1);
        scanf("%f", &x[i]);
        printf("y[%d]: ", i + 1);
        scanf("%f", &y[i]);
    }
    polynomialRegression(x, y, n, DEGREE, coeff);
    printf("\nThe polynomial regression equation is:\n");
    printf("y = ");
    for (int i = 0; i <= DEGREE; i++) {
        printf("%+.3f*x^%d ", coeff[i], i);
    }
    printf("\n");
    return 0;
}
```

Input:

```
Enter the degree of the polynomial (m): 2
Enter the number of data points (n): 3
Enter x[1] and y[1]: 1 2
Enter x[2] and y[2]: 2 3
Enter x[3] and y[3]: 3 5
```

Output :

```
Augmented Matrix:
6.000000 6.000000 14.000000  20.000000
6.000000 14.000000 38.000000  56.000000
14.000000 38.000000 114.000000  160.000000

Polynomial Equation:
y = 2.00*x^0 + 1.00*x^1 + 0.50*x^2
```