Assignment 1

Write a program in C to solve a root of the equation $x^3 - 4x + 1 = 0$ using Bisection Method.

Algorithm

```
1. Start

2. Define function f(x)

3. Input
        a. Lower and Upper guesses x0 and x1
        b. tolerable error e

4. If f(x0)*f(x1) > 0
        print "Incorrect initial guesses"
   End If

5. Do
        x2 = (x0+x1)/2

        If f(x0)*f(x2) < 0
                x1 = x2
        Else
                x0 = x2
        End If

   while abs(f(x2) > e

6. Print root as x2

7. Stop
```

Source Code: Bisection.c

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

#define E 0.00001

float f(float x) {
    return x*x*x - 4*x ;
}

float bisection(float a, float b) {
    if(f(a) * f(b) < 0) {
        float mid = a;
        while(fabs(b-a) >= E) {
            mid = (a+b)/2;
            if(f(mid) == 0) return mid;
            else if(f(a) * f(mid) < 0) b = mid;
            else a = mid;
        }
        return mid;

    } else {
        printf("Invalid initial guess.\n");
        return -1.0;
    }
}

int main()
{
    float a, b;
    printf("Enter the value of a (lower bound)\n");
    scanf("%f", &a);

    printf("Enter the value of b (upper bound)\n");
    scanf("%f", &b);

    float root = bisection(a, b);
    printf("Root = %.5f\n", root);

    return 0;
}
```

Output

```
Enter the value of a (lower bound)
1
Enter the value of b (upper bound)
2
Root = 1.86080
```

Assignment 2

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

Algorithm

1. Start
2. Define function f(x)

3. Input
    a. Lower and Upper guesses x0 and x1
    b. tolerable error e

4. If f(x0)*f(x1) > 0
    print "Incorrect initial guesses"
  End If

5. Do
    x2 = x0 - ((x0-x1) * f(x0))/(f(x0) - f(x1))

    If f(x0)*f(x2) < 0
        x1 = x2
    Else
        x0 = x2
    End If

  While abs(f(x2) > e

6. Print root as x2

7. Stop

Source Code: RegulaFalsi.c

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

double f(double x) {
   // Define the function whose roots we want to find
   return x*x*x - 4*x + 1;
}

double regular_falsi(double a, double b, double tol) {
   // Implements the Regular Falsi method to find a root of f(x) between a and b tol is the the desired accuracy

   double fa = f(a);
   double fb = f(b);
   double c, fc;
   int iter = 0;

   do {
      // Calculate the next approximation of the root
      c = a - fa * (b - a) / (fb - fa);
      fc = f(c);
```

```
// Check if we have found a root
        if (fabs(fc) < tol) {
            printf("Root found at x = %f\n", c);
            return c;
        }

        // Update the interval [a, b]
        if (fc * fa < 0) {
            b = c;
            fb = fc;
        } else {
            a = c;
            fa = fc;
        }

        iter++;
    } while (iter < 100000); // set a very large number of iterations as a fallback

    printf("Regular Falsi method failed to converge within the maximum number of iterations\n");
    return NAN;
}

int main() {
    double a, b, tol;

    printf("Enter the value of a (lower bound)\n");
    scanf("%lf", &a);

    printf("Enter the value of b (upper bound)\n");
    scanf("%lf", &b);

    printf("Enter the value of tolerance.\n");
    scanf("%lf", &tol);

    regular_falsi(a, b, tol);

    return 0;
}
```

Output

```
Enter the value of a (lower bound)
1
Enter the value of b (upper bound)
2
Enter the value of tolerance
0.000001

Root found at x = 1.860806
```

Assignment 3

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

Algorithm

1. Start.
2. Define a constant E with the value 0.00001.
3. Define a structure `Polynomial` with members `degree`, `coefficient`, and `next`.
4. Define a function `newNode` that takes `degree` and `coeff` as input and returns a new `Polynomial` node with the given values.
5. Define a function `insert` that takes `head`, `degree`, and `coeff` as input. It creates a new polynomial node using `newNode` and inserts it at the beginning of the linked list pointed to by `head`.
6. Define a function `takeInput` that takes the `degree` of the polynomial as input and builds the polynomial equation by repeatedly calling `insert` and taking coefficients as input from the user.
7. Define a function `derivative` that takes a polynomial equation as input and returns the derivative of the polynomial.
8. Define a function `fx` that takes `x` and a polynomial equation as input and evaluates the polynomial equation for the given value of `x`.
9. Define a function `newton_raphson` that takes `x`, a polynomial equation, and its derivative as input. It performs the Newton-Raphson iteration to find the approximate root of the polynomial equation and returns the value of the root.
10. Define a function `printPoly` that takes a polynomial equation as input and prints it in the desired format.
11. Define a function `f` that takes `x` as input and returns the value of the given function ($x^3 - 4x + 1$).
12. Define a function `df` that takes `x` as input and returns the derivative of the given function ($3x^2 - 4$).
13. Define a function `newton` that takes `x` as input and performs the Newton-Raphson iteration to find the approximate root of the given function. It returns the value of the root.
14. Start the `main` function.
15. Set `x` to 1.
16. Call the `newton` function with `x` as input and store the result in `root`.
17. Print the value of the approximate root with 5 decimal places.
18. Print the value of the function at the root.
19. Take input for the degree of the polynomial equation.
20. Call the `takeInput` function with the degree as input and store the result in `eqn`.
21. Call the `printPoly` function with `eqn` as input to display the polynomial equation.
22. Call the `derivative` function with `eqn` as input and store the result in `diff`.
23. Call the `printPoly` function with `diff` as input to display the derivative equation.
24. Set `x1` to 1.
25. Call the `newton_raphson` function with `x1`, `eqn`, and `diff` as input and store the result in `root1`.
26. Print the value of the approximate root with 5 decimal places.
27. Free the memory allocated for `eqn` and `diff`.
28. End.

Source Code: NewtonRaphson.c

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

#define E 0.00001

typedef struct  Polynomial
{
   int degree;
   float coefficient;
   struct Polynomial *next;
} Poly;

Poly *newNode(int degree, float coeff) {
   Poly *new_poly = malloc(sizeof(Poly));
   new_poly->degree = degree;
   new_poly->coefficient = coeff;
   new_poly->next = NULL;
   return new_poly;
}

void insert(Poly **head, int degree, float coeff) {
   Poly *new_poly = newNode(degree, coeff);
   if (!head || !(*head))
   {
      *head = new_poly;
   }
   else {
      new_poly->next = *head;
      *head = new_poly;
   }
}

// build polynomial equation
Poly* takeInput(int degree) {
   int size = degree + 1;

   Poly *eqn = NULL;

   for(int i = 0; i < degree+1; i++) {
      printf("\nEnter the coefficient of x^%d\n", i);
      float coeff;
      scanf("%f", &coeff);
      insert(&eqn, i, coeff);
   }

   return eqn;

}

// differentiate given polynomial
Poly* derivative(Poly *eqn) {
```

```
    Poly *diff = NULL;

    while(eqn) {
        if (eqn->degree > 0)
            insert(&diff, eqn->degree - 1, eqn->coefficient * eqn->degree);

        eqn = eqn->next;
    }

    return diff;
}

// sum the polynomial with given x value
float fx(float x, Poly *eqn) {
    float sum = 0.0;

    while(eqn) {
        sum += eqn->coefficient * pow(x, eqn->degree);
        eqn = eqn->next;
    }

    return sum;
}

float newton_raphson(float x, Poly *eqn, Poly* diff) {
    float h = fx(x, eqn) / fx(x, diff);

    printf("\n\tx\t|\tf(x)\t|\tf`(x)\t|\th");

    while(fabs(h) >= E) {

        printf("\n\t%.5f\t|\t%.5f\t|\t%.5f\t|\t%.5f\t", x, fx(x, eqn), fx(x, diff), h);

        h = fx(x, eqn) / fx(x, diff);
        x = x - h;
    }

    printf("\n\n");

    return x;

}

void printPoly(Poly *eqn) {
    printf("\nYour equation: ");

    while(eqn) {
        if (eqn->coefficient > 0)
        {
            printf(" + %.1fx^%d ", eqn->coefficient, eqn->degree);
        } else {
            printf(" %.1fx^%d ", eqn->coefficient, eqn->degree);
```

```
        }
        eqn = eqn->next;
    }

    printf("\n\n");
}



// given function
double f(double x) {
    return x*x*x - 4*x + 1;
}

// derivative of the function
double df(double x) {
    return 3*x*x - 4;
}

double newton(double x) {
    double h = f(x) / df(x);

    printf("\n\tx\t|\tf(x)\t|\tf`(x)\t|\th");

    while(fabs(h) >= E) {
        printf("\n\t%.5lf\t|\t%.5lf\t|\t%.5lf\t|\t%.5lf\t", x, f(x), df(x), h);

        h = f(x) / df(x);
        x = x - h;
    }

    printf("\n\n");

    return x;
}

int main() {
    double x = 1;
    double root = newton(x);
    printf("The value of approximate root upto 5 decimal places is: %0.5lf", root);
    printf("\nValue of the function at root %lf", f(root));


    printf("\n\nEnter the degree of the equation.\n");
    int degree;
    scanf("%d", &degree);
    Poly *eqn = takeInput(degree);
    printPoly(eqn);

    Poly *diff = derivative(eqn);
    printPoly(diff);

    float x1 = 1;
```

```
    float root1 = newton_raphson(x1, eqn, diff);
    printf("The value of approximate root upto 5 decimal places is: %0.5f", root1);

    free(eqn);
    free(diff);

    return 0;
}
```

Output:

```
Enter the degree of the equation.
3

Enter the coefficient of x^0
1

Enter the coefficient of x^1
-4

Enter the coefficient of x^2
0

Enter the coefficient of x^3
1



    x           |     f(x)        |     f`(x)       |     h
    1.00000 |     -2.00000 |     -1.00000 |     2.00000
    -1.00000|     4.00000  |     -1.00000 |     2.00000
    3.00000 |     16.00000|     23.00000|     -4.00000
    2.30435 |     4.01874  |     11.93006|     0.69565
    1.96749 |     0.74622  |     7.61304  |     0.33686
    1.86947 |     0.05577  |     6.48476  |     0.09802
    1.86087 |     0.00041  |     6.38852  |     0.00860
    1.86081 |     0.00000  |     6.38780  |     0.00006

The value of approximate root upto 5 decimal places is: 1.86081
```

Assignment 4

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

$x_1 + x_2 + x_3 = 3$

$2x_1 + 3x_2 + x_3 = 6$

$x_1 - x_2 - x_3 = -3$

Algorithm

The algorithms consist of the following three major stages.
I. read the matrix a with the (n + 1)-th column having right hand side
vector.
II. reduce it to upper triangular form
III. use Backward substitution to get the solution.

Algorithm

READ MATRIX A (step 1-6)

1. read n
2. for i = 1 to n
3. for j = 1 to n+1
4. read a[i][j]
5. next j
6. next i

REDUCE TO UPPPER TRIANGULAR (steps 7-14)

7. for k = 1 to n-1
8. for i = k+1 to n
9. ratio = a[i][k]/a[k][k]
1O. for j = 1 to n+1
11. a[i][j] = a[i][j] - ratio * a[k][j]
12. next j
13. next i
14. next k

BACKWARD SUBSTITUTION (steps 15-22)

15. x[n] = a[n][n+1]/a[n][n]
16.for k = n-1 to 1 step - 1
17. x[k] = a[k][n+1]
18.for j = k+ 1 to n
19. x[k] = x[k] - a[k][j] * x[j]
20. next j
21. x[k] = x[k]/a[k][k]
22. next k

PRINT ANWSER

23.for i = 1 to n
24. print xi
25. next i
26. end

Source Code: GaussEliminationMethod.c

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

// takes a double value as input from the user
double getInput() {
   printf("Enter data: ");
   double input;
   scanf("%lf", &input);
   return input;
}

// fills the given matrix (requires external fill function pointer)
void fillMatrix(int size, double matrix[][size], double (*fillFunction)()) {
   for(int i = 0; i < size; i++)
      for(int j = 0; j < size; j++)
         matrix[i][j] = fillFunction();
}

// fills the given vector (requires external fill function pointer)
void fillVecotr(int size, double vector[size], double (*fillFunction)()) {
   for(int i = 0; i < size; i++)
      vector[i] = fillFunction();
}

// output the matrix in formatted form
void printMatrix(int size, double matrix[][size]) {
   for(int i = 0; i < size; i++) {
      putc('|', stdout);
      for(int j = 0; j < size; j++)
         printf(" %.2lf ", matrix[i][j]);
      puts("|");
   }
}

// output the vector in formatted form
void printVector(int size, double vector[size]) {
   putc('|', stdout);
   for(int i = 0; i < size; i++)
      printf(" %.2lf ", vector[i]);
   puts("|");
}

void swap(double *a, double *b) {
   double temp = *a;
   *a = *b;
   *b = temp;
}

// function for backward substitution
double* backwardSubstitute(int size, double matrix[][size], double vector[size]) {
   double *solutionVector = malloc(size * sizeof(double));
```

```
    for(int I = size-1; I >=0; i--) {
      double sum = 0.0;
      for(int j = i+1; j < size; j++) {
         sum += matrix[i][j] * solutionVector[j];
      }
      solutionVector[i] = (vector[i]-sum) / matrix[i][i];
   }
   return solutionVector;
}

// Gauss Elimination method
void solve(int size, double matrix[][size], double vector[size]) {
   for(int i = 0; i < size; i++) {
      int maxRow = i;
      double maxElement = matrix[maxRow][i];

      // loop for finding the max row
      for(int j = i+1; j < size; j++) {
         double currentElement = matrix[j][i];
         if(fabs(currentElement) > fabs(maxElement))
            maxRow = j;
      }

      // swap current row with max row
      for (int j = i; j < size; j++) {
         swap(&matrix[maxRow][j], &matrix[i][j]);
      }

      // swap vecotr row with the max row
      swap(&vector[i], &vector[maxRow]);

      // No solution when matrix is singular
      if(fabs(matrix[i][i]) <= 0.00001) {
         printf("No solution because Matrix is Singular.\n");
         exit(1);
      }

      // loop for generating the upper triangular matrix
      for (int j = i + 1; j < size; j++) {
         // pivoting start
         double alpha = matrix[j][i] / matrix[i][i];
         vector[j] -= alpha * vector[i];

         for (int k = i; k < size; k++) {
            matrix[j][k] -= alpha * matrix[i][k];
         }
         // pivoting end
      }
   }

   // print solution
   printVector(size, solutionVector);
   // free up space
```

```
    free(solutionVector);
}

// driver (main) function
int main() {
    printf("Enter the size: ");
    int size;
    scanf("%d", &size);

    double matrix[size][size];
    double vector[size];

    double getInput(); // function pointer

    puts("\nEnter data to fill matrix:");
    fillMatrix(size, matrix, getInput);
    printMatrix(size, matrix);

    puts("\nEnter data to fill vector:");
    fillVecotr(size, vector, getInput);
    printVector(size, vector);

    puts("\nSolution vectors is: ");
    solve(size, matrix, vector);

    return 0;
}
```

Output:

```
Enter the size: 3

Enter data to fill matrix:
Enter data: 1
Enter data: 1
Enter data: 1
Enter data: 2
Enter data: 3
Enter data: 1
Enter data: 1
Enter data: -1
Enter data: -1
| 1.00  1.00  1.00 |
| 2.00  3.00  1.00 |
| 1.00  -1.00  -1.00 |

Enter data to fill vector:
Enter data: 3
Enter data: 6
Enter data: -3
| 3.00  6.00  -3.00 |

Solution vectors is:
| 0.00  1.50  1.50 |
```