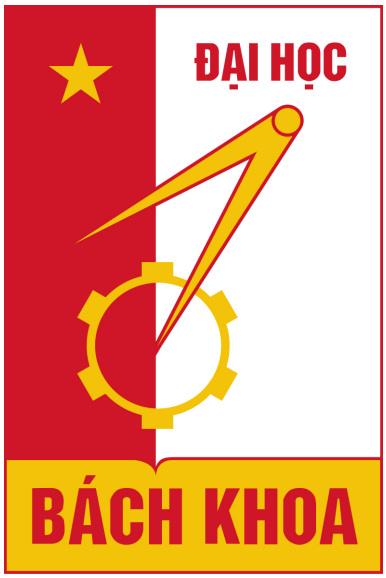
TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN TOÁN ỨNG DỤNG VÀ TIN HỌC

o0o



TÌM NGHIỆM CỦA PHƯƠNG TRÌNH ĐA THỨC ÁP DỤNG PHƯƠNG PHÁP TIẾP TUYẾN

NHÓM 33

Giáo viên hướng dẫn: TS. Nguyễn Thị Thanh Huyền

Sinh viên thực hiện : Trần Đại Dương - 20195863
Nguyễn Văn Thanh Tùng - 20195940

Lớp : Toán tin 02 - K64

Hà Nội - 2021

Mục lục

Chương 1

Khai báo chung

Mở đầu chương trình, nhóm khai báo các thư viện dùng trong chương trình bao gồm:

- stdio.h
- stdlib.h
- math.h

Tiếp đến là định nghĩa kiểu dữ liệu cho đa thức đặt tên là **Polynomial** bao gồm 2 trường:

- degree (kiểu int): lưu bậc của đa thức
- coefficient (kiểu double*): lưu các hệ số của đa thức từ bậc cao xuống bậc thấp

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

// Polynomial abstract data type
typedef struct polynomial {
    int degree;
    double* coefficient;
} Polynomial;
```

Sau đó là khai báo các biến toàn cục dùng chung cho cả chương trình bao gồm:

- rootIntervals (con trỏ double*): mảng kiểu double lưu các khoảng nghiệm thực của phương trình
- output (con trỏ FILE): trỏ đến file kết quả của chương trình

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

// Polynomial abstract data type
typedef struct polynomial {
    int degree;
    double* coefficient;
} Polynomial;
```

Tiếp đến là khai báo khuôn mẫu hàm, chi tiết các hàm sẽ được trình bày ở chương 3.

```c
/* ------------ FUNCTION'S PROTOTYPE ------------ */
// Function to display menu
void displayMenu();

// Function to input the polynomial
Polynomial initPolynomial();

// Function to display interval
void displayInterval(int size);

// Function to get the value of polynomial at point x
double getValue(Polynomial polynomial, double x);

// Function to compute the derivative of polynomial at point x
double computeDerivative(Polynomial polynomial, double x);

// Function to get the first derivative function of a polynomial
Polynomial getDerivativeFunction(Polynomial polynomial);

// Function to find range of real roots
double* findRangeOfRealRoots(Polynomial polynomial);

// Function to find upper bound of range of real roots
double findUpperBound(Polynomial polynomial);

// Function to find lower bound of range of real roots
double findLowerBound(Polynomial polynomial);
```

```
46  // Function to shrink an interval containing root
47  double* shrinkIntervalContainingRoot(Polynomial polynomial, double* interval);
48
49  // Function to define error function 1
50  double firstErrorFunction(Polynomial polynomial, double* interval, double x);
51
52  // Function to define error function 2
53  double secondErrorFunction(Polynomial polynomial, double *interval, double cur, double prev);
54
55  // Function to find a root with constant number of iteration
56  double findRootWithConstantIteration(Polynomial polynomial, double *interval);
57
58  // Function to find a root with given error e and use the first error function
59  double findRootWithGivenErrorUseFirstErrorFunc(Polynomial polynomial, double *interval);
60
61  // Function to find a root with given error e and use the second error function
62  double findRootWithGivenErrorUseSecondErrorFunc(Polynomial polynomial, double *interval);
63
64  // Function to find a root with given error e satisfying |Xn - Xn-1| <= e
65  double findRootWithGivenError(Polynomial polynomial, double *interval);
66
67  // Function to check if an interval contains a real root
68  int hasRealRoot(Polynomial polynomial, double start, double end);
69
70  // Gradient decent function
71  double gradientDescent(Polynomial polynomial, double guess, double upperBound);
72
73  // find all interval containing roots
74  int findAllIntervalContainingRoots(Polynomial polynomial);
75
76  // Function to display shrinked interval
77  void displayShrinkedInterval(double* interval, int size);
```

Chương 2

Chương trình chính

2.1 Mở file

Chương trình chính bắt đầu bằng việc kết nối con trỏ file với file **output.txt** nằm trong cùng thư mục với file code chính, và cho phép ghi tiếp vào file.
Tiếp đó kiểm tra xem có kết nối được với file không. Nếu thất bại, thông báo kết nối file thất bại và kết thúc chương trình. Nếu thành công, thông báo file đã được mở và tiếp tục chương trình.

```
79  /* ------------ MAIN FUNCTION ------------ */
80  int main() {
81
82      // Create and open output file in the current directory
83      output = fopen("output.txt", "a");
84      if (!output) {
85          printf("Cannot open this file\n");
86          return 1;
87      }
88      else {
89          printf("File is opened\n");
90      }
```

2.2 Khai báo các biến

Trong hàm main chứa các biến sau:

- **polynomial** (Kiểu Polynomial - đã được định nghĩa từ phần trước): lưu trữ đa thức
- **option** (kiểu int): lưu lựa chọn của người dùng, dựa vào đó để điều khiển chương trình

- **quit** (kiểu int): chỉ nhận giá trị 1 (đúng) hoặc 0(sai), điều khiển việc người dùng muốn tiếp tục hay kết thúc chương trình. Giá trị khởi đầu là 0 - tức người dùng chưa muốn thoát.
- **size** (kiểu int): lưu độ dài của mảng rootInterval(chứa khoảng nghiệm của phương trình) đã được khai báo là biến toàn cục.
- **shrinkedInterval** (con trỏ double*): lưu khoảng nghiệm sau khi đã được rút gọn thỏa mãn |a - b| $<$ 0.5 và được cấp phát bộ nhớ lưu được hai giá trị kiểu double.
- **root1, root2, root3, root4** (kiểu double): lưu các nghiệm tìm ở các yêu cầu 3, 4 và 5 trong đề bài.

```
92      // Polynomial initialization
93      Polynomial polynomial;
94
95      // Main flow
96      int option;
97      int quit = 0;
98      int size; // size of interval containing all roots
99      double* shrinkedInterval = (double*)malloc(sizeof(double) * 2);
100     double root1, root2, root3, root4; // for storing the roots found
```

2.3 Thao tác với người dùng

Phần chính trong hàm main sẽ là việc yêu cầu người dùng nhập vào một lựa chọn, xử lí yêu cầu của người dùng cho tới khi người dùng muốn kết thúc chương trình.

Tạo một vòng lặp **while**, lặp với điều kiện biến **quit** khác 1 (!quit).

Việc đầu tiên trong vòng lặp là gọi hàm **displayMenu()** có nhiệm vụ hiển thị các chức năng của chương trình cho người dùng chọn. Cụ thể, khi gọi hàm **displayMenu()** sẽ hiển thị như sau:

```
 ============================MENU==============================
|1. Enter the polynomial
|2. Find all intervals containing root of the polynomial
|3. Find interval containing root [a, b] of the polynomial such that |a - b| <= 0.5
|4. Find the root of the polynomial given a particular number of iteration in the interval(a, b) and the two error functions
|5. Find the root of the polynomial given a particular error in the interval(a, b) and the two error functions
|6. Find the root Xn of the polynomial in the interval (a, b) such that |Xn - Xn-1| <= e (e is entered by user)
|0. Exit the program
Your option:
```

Tiếp theo, yêu cầu người dùng nhập vào lựa chọn của mình, lưu yêu cầu của người dùng vào biến **option**:

```
105         // prompt user for option
106         scanf("%d", &option);
```

Sau khi đã có yêu cầu của người dùng, tiến hành thực hiện theo yêu cầu đó dùng câu lệnh điều kiện **switch-case**, cụ thể nếu người dùng tiến hành:

- Nhập 1:
  - Xóa màn hình hiển thị menu
  - gọi hàm **initPolynomial()**, yêu cầu người dùng nhập dữ liệu của đa thức và lưu vào biến **polynomial**
  - đặt lệnh **break** ở cuối.

  Code chi tiết:

```
108         switch (option) {
109             case 1:
110                 system("clear");
111                 polynomial = initPolynomial();
112                 break;
```

  Chương trình khi người dùng nhập 1 và đa thức bậc 3, các hệ số lần lượt là 1, 3, -5 và 1:

```
PROBLEMS   OUTPUT   TERMINAL   DEBUG CONSOLE

Enter the degree of the polynomial: 3
Enter the coefficients of the polynomial: 1 3 -5 1
```

- Nhập 2:
  - Xóa màn hình hiển thị menu
  - Gọi hàm **findAllIntervalContainingRoots** tìm các khoảng chứa nghiệm thực của phương trình và lưu vào mảng global **rootInterval**, và lưu số phần tử trong mảng **rootInterval** vào biến **size**.
  - Gọi hàm **displayInterval** để hiển thị kết quả cho người dùng và lưu kết quả vào file **output.txt**. Nếu size <= 1 thì hiển thị phương trình vô nghiệm. Ngược lại thì hiển thị ra các khoảng chứa nghiệm.
  - Nếu size <= 1 thì gán biến **quit = 1** để thoát chương trình. Ngược lại thì gán **quit = 0** để tiếp tục.
  - đặt lệnh **break** ở cuối.

  Code chi tiết:

```
113         case 2:
114             system("clear");
115             size = findAllIntervalContainingRoots(polynomial);
116             displayInterval(size);
117             quit = (size < 1) ? 1 : 0;
118             break;
```

Chương trình khi người dùng nhập 1 và đa thức bậc 3, các hệ số lần lượt là 1, 3, -5 và 1:

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

===== ALL INTERVALS CONTAINING ROOTS =====
[-6.000000, -2.632993]
[-2.632993, 0.632993]
[0.632993, 3.236068]
```

- Nhập 3:
  - Xóa màn hình hiển thị menu
  - gán dữ liệu cho mảng **shrinkedInterval** (mảng lưu khoảng nghiệm của phương trình thỏa mãn |a - b| <= 0.5 với hai phần tử đầu tiên của mảng **rootInterval**.
  - Gọi hàm **shrinkIntervalContainingRoot** để thu hẹp khoảng nghiệm và lưu lại vào **shrinkedInterval**.
  - Goị hàm **displayShrinkedInterval** để hiển thị mảng vừa thu hẹp và ghi vào file.
  - đặt lệnh **break** ở cuối.

Code chi tiết:

```
case 3:
    system("clear");
    shrinkedInterval[0] = rootIntervals[0];
    shrinkedInterval[1] = rootIntervals[1];
    shrinkedInterval = shrinkIntervalContainingRoot(polynomial, shrinkedInterval);
    displayShrinkedInterval(shrinkedInterval, 2);
    break;
```

Chương trình khi người dùng nhập 1 và đa thức bậc 3, các hệ số lần lượt là 1, 3, -5 và 1:

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

===== SHRINKED INTERVAL =====
[-4.316497, -3.895621]
```

- Nhập 4:
  - Xóa màn hình hiển thị menu
  - Gọi hàm **findRootWithConstantIteration** yêu cầu người dùng nhập vào số lần lặp, tìm nghiệm của phương trình trong khoảng **shrinkedInterval** đã tìm được ở phần trước, lưu nghiệm vào biến **root1**. Và đánh giá sai số theo hai công thức:

$$|x_n - x^*| \le \frac{|f(x_n)|}{m_1} \qquad (1)$$

$$|x_n - x^*| \le \frac{M_2}{2m_1}|x_n - x_{n-1}|^2 \qquad (2)$$

Ở đó $m_1 = min_{x\in[a,b]}|f'(x)|$; $M_2 = max_{x\in[a,b]}|f''(x)|$

  - đặt lệnh **break** ở cuối.

Code chi tiết:

```
case 4:
    system("clear");
    root1 = findRootWithConstantIteration(polynomial, shrinkedInterval);
    break;
```

Chương trình khi người dùng nhập 1 và đa thức bậc 3, các hệ số lần lượt là 1, 3, -5 và 1 với số lần lặp là 10:

```
PROBLEMS   OUTPUT   TERMINAL   DEBUG CONSOLE

Enter number of iteration: 10
x0 = -4.11
===== ROOT FOUND BY CONSTANT NUMBER OF ITERATION =====
x = -4.236068

First error: 0.00000000000000062133
Second error: 0.00000000000000000000
```

- Nhập 5:
  - Xóa màn hình hiển thị menu
  - Gọi hàm **findRootWithGivenErrorUseFirstErrorFunc** và hàm **findRootWithGivenErrorUseSecondErrorFunc**, lưu kết quả lần lượt vào biến**root1** và **root2**. Hai hàm trên lần lượt yêu cầu người dùng nhập vào sai số và tìm nghiệm của phương trình trong khoảng **shrinkedInterval** đã tìm được ở phần trước
  - đặt lệnh **break** ở cuối.

Code chi tiết:

```
case 5:
    system("clear");
    root2 = findRootWithGivenErrorUseFirstErrorFunc(polynomial, shrinkedInterval);
    root3 = findRootWithGivenErrorUseSecondErrorFunc(polynomial, shrinkedInterval);
    break;
```

Chương trình khi người dùng nhập 1 và đa thức bậc 3, các hệ số lần lượt là 1, 3, -5 và 1 với cả hai sai số là 0.000001:

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

Enter given error: 0.000001
===== ROOT FOUND BY FIRST ERROR FUNCTION =====
x = -4.236068

Enter given error: 0.000001
===== ROOT FOUND BY SECOND ERROR FUNCTION =====
x = -4.236068
```

- Nhập 6:
  - Xóa màn hình hiển thị menu
  - Gọi hàm **findRootWithConstantIteration** yêu cầu người dùng nhập vào sai số e, tìm nghiệm của phương trình trong khoảng **shrinkedInterval** đã tìm được ở phần trước, lặp đến khi |Xn - Xn-1| <= e, lưu nghiệm vào biến **root4**.
  - đặt lệnh **break** ở cuối.

Code chi tiết:

```
case 6:
    system("clear");
    root4 = findRootWithGivenError(polynomial, shrinkedInterval);
    break;
```

Chương trình khi người dùng nhập 1 và đa thức bậc 3, các hệ số lần lượt là 1, 3, -5 và 1 vớ sai số là 0.000001:

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

Enter given error: 0.0000001
===== ROOT FOUND BY GIVEN ERROR =====
x = -4.236068
```

- Nhập 0:

- Xóa màn hình hiển thị menu
- Gán biến **quit = 0** để thoát khỏi vòng lặp
- đặt lệnh **break** ở cuối.

Code chi tiết:

```
139             case 0:
140                 quit = 1;
141                 break;
```

2.4 Kết thúc chương trình

Giải phóng bộ nhớ đã cấp phát cho các mảng **rootIntervals** và **rootIntervals**

Đóng file **output.txt**

```
145     // free
146     free(rootIntervals);
147     free(shrinkedInterval);
148
149     // close file
150     fclose(output);
151     return 0;
152 }
```

Chương 3

Các hàm thành phần

3.1 Hàm hiển thị menu

Gọi hàm **displayMenu()** để hiển thị menu cho người dùng, trong đó:

- 1. Nhập dữ liệu cho đa thức
- 2-6. Tương ứng với các yêu cầu 1-5 trong đề bài
- 0. Thoát chương trình

```
/* ------------ FUNCTION'S DETAILS ------------ */
// Function to display menu
void displayMenu()
{
    printf("\t\t\t  ============================MENU==============================\n");
    printf("\t\t\t |1. Enter the polynomial\n");
    printf("\t\t\t |2. Find all intervals containing root of the polynomial\n");
    printf("\t\t\t |3. Find interval containing root [a, b] of the polynomial such that |a - b| <= 0.5\n");
    printf("\t\t\t |4. Find the root of the polynomial given a particular number of iteration in the interval(a, b) and the two error functions\n");
    printf("\t\t\t |5. Find the root of the polynomial given a particular error in the interval(a, b) and the two error functions\n");
    printf("\t\t\t |6. Find the root Xn of the polynomial in the interval (a, b) such that |Xn - Xn-1| <= e (e is entered by user)\n");
    printf("\t\t\t |0. Exit the program\n");
    printf("\t\t\t Your option: ");
}
```

3.2 Hàm khởi tạo đa thức

Gọi hàm **initPolynomial()** để yêu cầu người dùng nhập dữ liệu cho đa thức
Hàm **initPolynomial()** không có tham số đầu vào và trả về kiểu dữ liệu **Polynomial**

- Viết dòng "===== INPUT POLYNOMIAL ======" vào file **output.txt**
- Khai báo biến **polynomial** lưu dữ liệu đa thức
- Yêu cầu người dùng nhập vào bậc của đa thức. Nếu người dùng nhập vào số âm, yêu cầu nhập lại

- Viết bậc của đa thức ra file output
- sau khi đã có bậc của đa thức, cấp phát dữ liệu để lưu trữ hệ số của đa thức. Yêu cầu người dùng nhập hệ số từ bậc cao đến thấp và với mỗi hệ số người dùng nhập, ghi dữ liệu ra file output.
- trả về đa thức vừa nhập và lưu vào biến **polynomial** trong hàm main.

Code chi tiết:

```
// Function for inputing the polynomial
Polynomial initPolynomial() {
    // write to file
    fprintf(output, " ===== INPUT POLYNOMIAL ====== \n");

    // declaration
    Polynomial polynomial;
    int i; // loop variable

    // prompt user for degree of polynomial
    do {
        printf("Enter the degree of the polynomial: ");
        scanf("%d", &polynomial.degree);
        if (polynomial.degree < 0) {
            printf("The degree of a polynomial must be greater or equal to 0!\n");
        }
    } while (polynomial.degree < 0);

    // write degree to the file
    fprintf(output, "Degree: %d\n", polynomial.degree);

    // allocate memory for storing polynomial's coefficients
    polynomial.coefficient = (double* ) malloc((polynomial.degree + 1) * sizeof(double));

    // prompt user for coefficients of polynomial
    printf("Enter the coefficients of the polynomial: ");
    fprintf(output, "Coefficients: ");
    for (i = 0; i <= polynomial.degree; i++) {
        scanf("%lf", &polynomial.coefficient[i]);
        fprintf(output, "%7.4lf", polynomial.coefficient[i]); // write the coefficient to output file
    }
    fprintf(output, "\n\n");

    return polynomial;
}
```

3.3 Hàm tính giá trị của đa thức tại một điểm

Gọi hàm **getValue(Polynomial polynomial, double x)** để yêu cầu người dùng nhập dữ liệu cho đa thức

Hàm **getValue(Polynomial polynomial, double x)** nhận vào hai tham số là đa thức cần tình (polynomial:Polynomial) và điểm cần tính giá trị (x:double) và trả về kiểu dữ liệu **double**

- Khởi tạo giá trị cần tính **value = 0**
- Lặp qua từng hệ số của đa thức, với mỗi hệ số, áp dụng công thức đạo hàm đa thức, thay x vào tính kết quả rồi cộng dồn kết quả vào biến**value**
- trả về giá trị của **value** sau khi đã lặp qua hết các hệ số.

Code chi tiết:

```
// Function to get the value of polynomial at point x
double getValue(Polynomial polynomial, double x) {
    // declaration
    double value = 0;
    int i; // loop variable

    // get value at point x
    for (i = 0; i <= polynomial.degree; i++) {
        value += polynomial.coefficient[i] * pow(x, (double)(polynomial.degree-i));
    }

    // return
    return value;
}
```

3.4 Hàm tính đạo hàm của hàm số

Gọi hàm **getDerivativeFunction(Polynomial polynomial)** để lưu đạo hàm cấp một của đa thức

Hàm **getDerivativeFunction(Polynomial polynomial)** nhận vào một tham số là đa thức cần tình (polynomial:Polynomial) và trả về kiểu dữ liệu **Polynomial**

- Khai báo biến **df** lưu đạo hàm của đa thức
- Lưu bậc của đạo hàm bằng bậc của đa thức - 1
- Cấp phát bộ nhớ cho hệ số của đạo hàm sau khi đã có được bậc của đạo hàm
- Tính toán hệ số của đạo hàm và lưu giá trị

- Trả về đa thức là đạo hàm vừa tính được

Code chi tiết:

```
// Function to get the first derivative function of a polynomial
Polynomial getDerivativeFunction(Polynomial polynomial) {
    // Variable declaration
    Polynomial df;
    int i; // loop variable

    // get the derivative function
    df.degree = polynomial.degree - 1; // degree of derivative function
    df.coefficient = (double*) malloc (df.degree * sizeof(double));
    for (i = 0; i <= df.degree; i++) {
        df.coefficient[i] = (polynomial.degree-i) * polynomial.coefficient[i];
    }

    // return
    return df;
}
```

3.5 Hàm tìm tất cả các khoảng nghiệm của đa thức

Gọi hàm **findAllIntervalContainingRoots(Polynomial polynomial)** để tìm ra các khoảng chưa nghiệm thực của đa thức

Hàm **findAllIntervalContainingRoots(Polynomial polynomial)** nhận vào một tham số là đa thức cần tình (polynomial:Polynomial), lưu giá trị đầu mút của các khoảng nghiệm vào mảng **rootInterval** và trả về kiểu dữ liệu **int** là độ dài của mảng **rootInterval**

- Cấp phát bộ nhớ cho mảng **rootInterval** có thể chứa được một giá trị kiểu double, gán cho biến size(lưu độ dài của mảng **rootInterval** bằng 1)
- Tìm cận trên và cận dưới của khoảng chứa nghiệm bằng việc gọi hai hàm **findUpperBound** và **findLowerBound** và đưa cận dưới là giá trị đầu tiên trong mảng **rootInterval**
- Khởi tạo biến **step** là bước nhảy sử dụng để tìm cực trị
- Khởi tạo biến **i** là giá trị khởi đầu để tìm cực trị
- Khởi tạo biến **sign** để kiểm soát dấu của đạo hàm tại điểm i
- Tính đạo hàm của hàm số đã cho, lưu vào biến **df**
- lặp cho tới khi i vượt quá cận trên
  - Gọi hàm **gradientDescent** để tìm điểm cực trị, lưu giá trị tìm được vào **i**

  - Kiểm tra giá trị vừa tìm được và giá trị ngay trước đó trong mảng **rootInterval** có chứa nghiệm thực hay không bằng cách gọi hàm **hasRealRoot**. Nếu có tồn tại thì cấp phát thêm một ô nhớ cho mảng **rootInterval** và tăng size lên 1 đơn vị, điểm cực trị i vào cuối mảng **rootInterval**
  - tính dấu của đạo hàm tại giá trị i vừa tìm được
  - tằng giá trị **i** một lượng bằng **step** cho tới khi đạo hàm đổi dấu, hoặc kết thúc quá trình nếu **i** vượt quá cận trên của khoảng chứa nghiệm
- Trả về giá trị của biến **size**

Code chi tiết:

```
// find all interval containing roots
int findAllIntervalContainingRoots(Polynomial polynomial)
{
    // variables
    rootIntervals = (double*)malloc(sizeof(double) * 1);
    double upperBound = findUpperBound(polynomial);
    double lowerBound = findLowerBound(polynomial);
    int size = 1;
    rootIntervals[0] = lowerBound;
    double step = 1e-3;
    double i = lowerBound;
    int sign;

    // find derivative function
    Polynomial df = getDerivativeFunction(polynomial);

    // find extrema
    while (i < upperBound) {
        i = gradientDescent(df, i, upperBound);
        if (hasRealRoot(polynomial, rootIntervals[size-1], i)) {
            rootIntervals = (double*)realloc(rootIntervals, sizeof(double) * (size + 1));
            rootIntervals[size++] = i;
        }
        sign = (getValue(df, i) > 0) ? 1 : -1;
        do {
            i += step;
            if (i >= upperBound) {
                break;
            }
        } while ( (getValue(df, i) * sign > 0) && (fabs(getValue(df, i) > getValue(df, i + step))));
    }
    return size;
}
```

3.5.1 Hàm tìm cận trên của miền chứa nghiệm

Gọi hàm **findUpperBound(Polynomial polynomial)** để tìm cận trên của khoảng chứa nghiệm thực của đa thức

Hàm **findUpperBound(Polynomial polynomial)** nhận vào một tham số là đa thức cần tình (polynomial:Polynomial) trả về kiểu dữ liệu **double** là cận trên của khoảng chứa nghiệm

- Khởi tạo biến **k** là biến lưu chỉ số giá trị âm đầu tiên trong các hệ số của đa thức với giá trị ban đầu là -1.
- Khai báo biến B là giá trị tuyệt đối lớn nhất trong các hệ số âm của đa thức
- Khai báo biến A là giá trị tuyệt đối lớn nhất trong các hệ số bắt đầu từ chỉ số 1 của đa thức
- Xét hai trường hợp:
  - Nếu hệ số đầu tiên của đa thức dương, tính theo công thức:

    $1 + \frac{A}{|a_0|}$

    * Lặp qua các hệ số của đa thức để tìm giá trị hệ số có giá trị âm đầu tiên.
    * Kiểm tra nếu k == -1, tức không có hệ số âm, trả về giá trị 0
    * Nếu k != -1, lặp qua các hệ số của đa thức một lần nữa để tìm B, rồi tính cận trên theo công thức trên.
  - Nếu hệ số đầu tiên của đa thức âm, tính theo công thức:

    $1 + \sqrt[k]{\frac{B}{a_0}}$

    * Lặp qua các hệ số của đa thức, từ hệ số có chỉ số 1, tìm A.
    * tính cận trên theo công thức trên

Code chi tiết:

```
double findUpperBound(Polynomial polynomial) {
    // variable's declaration
    int i; // loop variable
    int k = -1; // index of the first negative coefficient
    double B; // maximum value of the absolute value among negative coefficient      You, 6 ho
    int A;    // maximum of absolute value of coefficients except the first coefficient

    if (polynomial.coefficient[0] > 0) {
        // find the first negative coefficient
        for (i = 0; i <= polynomial.degree; i++) {
            if (polynomial.coefficient[i] < 0) {
                k = i;
                break;
            }
        }
        // if there is no negative coefficient => upperbound = 0
        if (k == -1) {
            return 0;
        }
        // otherwise find B
        else {
            B = fabs(polynomial.coefficient[k]);
            for (i = k; i <= polynomial.degree; i++) {
                if (polynomial.coefficient[i] < 0 &&  B < fabs(polynomial.coefficient[i])) {
                    B = fabs(polynomial.coefficient[i]);
                }
            }
            return 1 + pow(fabs(B / polynomial.coefficient[0]), 1.0 / k);
        }
    }
    else {
        // find maximum of absolute value of coefficients except the first coefficient
        A = fabs(polynomial.coefficient[1]);
        for (i = 2; i < polynomial.degree; i++) {
            if (A < fabs(polynomial.coefficient[i])) {
                A = fabs(polynomial.coefficient[i]);
            }
        }

        return 1 + A / fabs(polynomial.coefficient[0]);
    }
}
```

3.5.2 Hàm tìm cận dưới của miền chứa nghiệm

Gọi hàm **findLowerBound(Polynomial polynomial)** để tìm cận dưới của khoảng chứa nghiệm thực của đa thức

Hàm **findLowerBound(Polynomial polynomial)** nhận vào một tham số là đa thức cần tình (polynomial:Polynomial) trả về kiểu dữ liệu **double** là cận dưới của khoảng chứa nghiệm

- Khai báo **polynomialTemp** biến tạm thời để lưu giá trị của đa thức, ta sẽ thao tác trên biến tạm thay vì đa thức ban đầu.
- copy dữ liệu của biến **polynomial** sang biến **polynomialTemp**, Thay x = -x, tính lại hệ số của biến **polynomialTemp**
- Gọi hàm **findUpperBound(polynomialTemp)** rồi nhân giá trị tính được với (-1) để tìm ra cận dưới
- Trả lại giá trị vừa tìm được

Code chi tiết:

```
// Function to find lower bound of range of real roots
double findLowerBound(Polynomial polynomial) {
    // variable's declaration
    int i;
    Polynomial polynomialTemp;
    polynomialTemp.coefficient = (double*)malloc(sizeof(double) * (polynomial.degree + 1));
    polynomialTemp.degree = polynomial.degree;

    // Find P(-x)
    if (polynomial.degree % 2 == 0) {
        for (i = 0; i < polynomial.degree; i++) {
            polynomialTemp.coefficient[i] = (i % 2 == 0) ? polynomial.coefficient[i] : -polynomial.coefficient[i];
        }
    }
    else {
        for (i = 0; i < polynomial.degree; i++){
            polynomialTemp.coefficient[i] = (i % 2 != 0) ? polynomial.coefficient[i] : -polynomial.coefficient[i];
        }
    }
    polynomialTemp.coefficient[polynomialTemp.degree] = polynomial.coefficient[polynomial.degree];

    // return lower bound
    return (-1) * findUpperBound(polynomialTemp);
}
```

3.5.3 Hàm kiểm tra một khoảng có chứa nghiệm thực hay không

Gọi hàm **hasRealRoot(Polynomial polynomial, double start, double end)** để tìm cận dưới của khoảng chứa nghiệm thực của đa thức

Hàm **hasRealRoot(Polynomial polynomial, double start, double end)** nhận vào ba tham số là đa thức cần tình (polynomial:Polynomial), hai đầu mút start và end (kiểu double) trả về kiểu dữ liệu **int** có thể coi là giá trị boolean cho biết khoảng [start, end] có chứa nghiệm thực của đa thức polynomial hay không.

Code chi tiết:

```
// Function to check if an interval contains a real root
int hasRealRoot(Polynomial polynomial, double start, double end)
{
    return getValue(polynomial, start) * getValue(polynomial, end) < 0;
}
```

3.5.4 Hàm tìm cực trị của đa thức

Gọi hàm **gradientDescent(Polynomial df, double guess, double upperBound)** để tìm cận dưới của khoảng chứa nghiệm thực của đa thức. Hàm này áp dụng thuật toán Gradient descent để tìm cực trị.

Hàm **gradientDescent(Polynomial df, double guess, double upperBound)** nhận vào ba tham số là đạo hàm của đa thức cần tình (df:Polynomial), giá trị dự đoán ban đầu (guess:double) và cận trên của khoảng chứa nghiệm (upperBound: double) trả về kiểu dữ liệu **double** là cực trị của đa thức cần tìm

- Khai báo biến **sign** để kiểm soát dấu của đạo hàm tại điểm dự đoán (guess)
- Khởi tạo biến **learningRate** để kiểm soát sự thay đổi của điểm dự đoán
- Khởi tạo biến **esp** là sai số mong muốn
- Xác định dấu của đạo hàm tại guess. Nếu giá trị đạo hàm tại guess bằng 0, trả về guess, cự trị được tìm thành công. Nếu giá trị đạo hàm tại guess âm, gán **sign = -1** và gán **sign = 1** nếu ngược lại
- lặp cho tới khi giá trị tại đạo hàm tại **guess** nhỏ hơn hoặc bằng sai số **esp**
- trong mỗi vòng lặp gán **guess = guess + sign * leaningRate * getValue(df, guess)**
- Thoát khỏi vòng lặp nếu guess vượt quá cận trên của miền chứa nghiệm.

Code chi tiết:

```
// Gradient decent function
double gradientDescent(Polynomial df, double guess, double upperBound)
{
    // variable's declaration
    int sign;
    double leaningRate = 0.001;
    double eps = 1e-10;

    // find extrema
    if (getValue(df, guess) == 0) {
        return guess;
    }
    else if (getValue(df, guess) < 0) {
        sign = -1;
    }
    else {
        sign = 1;
    }

    while (fabs(getValue(df, guess)) > eps) {
        guess = guess + sign * leaningRate * getValue(df, guess);
        if (guess > upperBound) {
            return upperBound;
        }
    }
    return guess;
}
```

3.5.5 Hàm hiển thị và lưu khoảng nghiệm

Gọi hàm **displayInterval(int size)** để hiển thị ra màn hình và lưu vào file output các khoảng chứa nghiệm của phương trình

Hàm **displayInterval(int size)** nhận vào một tham số là độ dài của mảng **rootIntervals** (size: int) và không trả về giá trị

- Lưu dòng ===== ALL INTERVALS CONTAINING ROOTS ===== vào file output.txt
- Kiểm tra phương trình có vô nghiệm hay không bằng cách xét:
  - Nếu size == 1, tức phương trình vô nghiệm, hiển thị thông báo ra màn hình và lưu vào trong file output.txt
  - Nếu ngược lại, tức phương trình có ít nhất một nghiệm, lặp qua các phần tử trong mảng **rootIntervals**, in ra màn hình và ghi vào file output.txt

Code chi tiết:

```c
// Function to display interval
void displayInterval(int size)
{
    // check if polynomial has a root
    fprintf(output, "===== ALL INTERVALS CONTAINING ROOTS =====\n");
    if (size == 1) {
        printf("The function has no root\n\n");
        fprintf(output, "The function has no root\n\n");
    }
    else {
        printf("===== ALL INTERVALS CONTAINING ROOTS =====\n");
        for (int i = 0; i < size - 1; i++) {
            printf("[%lf, %lf]\n", rootIntervals[i], rootIntervals[i+1]);
            fprintf(output, "[%lf, %lf]\n", rootIntervals[i], rootIntervals[i + 1]);
        }
        fprintf(output, "\n");
    }
}
```

3.6 Hàm thu hẹp khoảng nghiệm

Gọi hàm **shrinkIntervalContainingRoot(Polynomial polynomial, double *interval)** để thu hẹp khoảng chứa nghiệm của phương trình bằng phương pháp chia đôi

Hàm **shrinkIntervalContainingRoot(Polynomial polynomial, double *interval)** nhận vào hai tham số là đa thức cần tính (polynomial: Polynomial) và môt mảng gồm hai phần tử chứa hai đầu mút của khoảng chứa nghiệm (interval: double*)

- Khởi tạo biến tolerance = 0.5 để giới hạn độ dài của khoảng
- khởi tạo biến start và end lần lượt là hai giá trị thứ nhất và thứ hai trong mảng **interval**
- Khai báo biến mid là giá trị khởi đầu khi áp dụng phương pháp chia đôi để thu hẹp khoảng nghiệm
- Khai báo mảng **shrinkedInterval** để lưu khoảng sau khi được thu hẹp và cấp phát ô nhớ để mảng này chứa được hai giá trị kiểu double.
- Lặp cho tới khi độ dài của khoảng chứa nghiệm nhỏ hơn hoặc bằng **tolerance**, với mỗi vòng lặp, tình mid là giá trị trung bình của start và end.
  - Nếu giá trị của đa thức tại điểm **mid** cùng dấu với giá trị của đa thức tại điểm **start**, gán **start = mid**

  Nếu ngược lại, gán **end = mid**

- Đưa lần lượt giá trị của start và end vào mảng **shrinkedInterval**
- Trả về mảng **shrinkedInterval**

Code chi tiết:

```
// Function to shrink an interval containing root
double *shrinkIntervalContainingRoot(Polynomial polynomial, double *interval) {
    // declaration
    double tolerance = 0.5; // restrict the length of interval
    double start = interval[0]; // the begining of interval
    double end = interval[1]; // the end of the interval
    double mid; // midpoint of start and end
    double *shrinkedInterval = (double*) malloc (2 * sizeof(double));

        // shrink the interval until |start - end| <= tolerant
        while (fabs(start - end) > tolerance)
    {
        mid = (start + end) / 2;
        if (getValue(polynomial, mid) * getValue(polynomial, start) > 0) {
            start = mid;
        }
        else {
            end = mid;
        }
    }

    // store interval
    shrinkedInterval[0] = start, shrinkedInterval[1] = end;

    // return
    return shrinkedInterval;
}
```

3.6.1 Hàm hiển thị và lưu khoảng nghiệm thu hẹp

Gọi hàm **displayShrinkedInterval(double* interval, int size)** để hiển thị ra màn hình và lưu vào file output khoảng chứa nghiệm của phương trình sau khi đã được thu hẹp

Hàm **displayShrinkedInterval(double* interval, int size)** nhận vào hai tham số là khoảng cần hiển thị và độ dài của mảng (size: int) và không trả về giá trị

- Lưu dòng ===== SHRINKED INTERVAL ===== vào file output.txt
- Hiển thị ra màn hình và lưu vào file output.txt khoảng nghiệm vừa thu hẹp

Code chi tiết:

```c
// Function to display shrinked interval
void displayShrinkedInterval(double* interval, int size)
{
    fprintf(output, "===== SHRINKED INTERVAL =====\n");
    for (int i = 0; i < size - 1; i++)
    {
        printf("===== SHRINKED INTERVAL =====\n");
        printf("[%lf, %lf]\n", interval[i], interval[i + 1]);
        fprintf(output, "[%lf, %lf]\n", interval[i], interval[i + 1]);
    }
    fprintf(output, "\n");
}
```

3.7 Hàm tìm nghiệm với số lần lặp được nhập từ người dùng

Gọi hàm **findRootWithConstantIteration(Polynomial polynomial, double *interval)** để tìm nghiệm và đánh giá sai số theo cả hai công thức sai số.

Hàm **findRootWithConstantIteration(Polynomial polynomial, double *interval)** nhận vào hai tham số là đa thức cần tính (polynomial: Polynomial) và môt mảng gồm hai phần tử chứa hai đầu mút của khoảng chứa nghiệm (interval: double*)

- Khai báo biến n là lưu số lần lặp mà người dùng nhập vào
- khởi tạo biến start và end lần lượt là hai giá trị thứ nhất và thứ hai trong mảng **interval**
- Khai báo biến x0 là giá trị khởi đầu khi áp dụng phương pháp pháp tuyến để tìm nghiệm của phương trình
- Khai báo biến x1 là giá trị tiếp theo giá trị x0 khi áp dụng phương pháp pháp tuyến để tìm nghiệm của phương trình
- Khai báo hai biến firstError, secondError lần lượt để lưu giá trị của hàm sai số tại điểm đang xét
- Tìm đạo hàm của đa thức cần tìm nghiệm, lưu vào biến df
- Yêu cầu người dùng nhập vào số lần lặp, nếu người dùng nhập vào số âm hoặc bằng 0, yêu cầu nhập lại
- Lặp đúng với số lần người dùng yêu cầu, với mỗi vòng lặp:
  - Tính gía trị của đa thức tại điểm x0, lưu vào biến tạm y

  - Tính giá trị của đạo hàm tại điểm x0, lưu vào biến tạm dy
  - Tính x1 = x0 - y/dy
  - tính hai giá trị firstError, secondError bằng việc lần lượt gọi hàm **firstErrorFunction** và **secondErrorFunction**
  - Gán **x0 = x1**

- Hiển thị ra màn hình và lưu vào file output.txt kết quả vừa tính:
  - Lưu dòng ===== ROOT FOUND BY CONSTANT NUMBER OF ITERATION ===== vào file output.txt, in dòng đó ra màn hình
  - In ra màn hình và lưu nghiệm x0 vào file output.txt
  - In ra màn hình và lưu vào file output.txt giá trị của firstError, secondError

Code chi tiết:

```
// Function to find a root with constant number of iteration
double findRootWithConstantIteration(Polynomial polynomial, double *interval) {
    // variable declaration
    double start = interval[0];
    double end = interval[1];
    int i; // loop variable
    int n; // number of iteration
    double x0 = (start + end) / 2;
    double x1; // next value for computation
    double firstError, secondError; // error for the first and second error function respectively

    // get the first order derivative of the polynomial
    Polynomial df = getDerivativeFunction(polynomial);

    // prompt user for number of iteration
    do {
        printf("Enter number of iteration: ");
        scanf("%d", &n);
        if (n <= 0) {
            printf("Number of iteration must be greater than 0!\n");
        }
    } while (n <= 0);
```

```
    // find root
    for (i = 0; i < n; i++) {
        double y = getValue(polynomial, x0);
        double dy = getValue(df, x0);

        // get next value
        x1 = x0 - y/dy;

        // get the first error (with first error function) and second error (with second error function)
        firstError = firstErrorFunction(polynomial, interval, x1);
        secondError = secondErrorFunction(polynomial, interval, x1, x0);

        // store current value
        x0 = x1;
    }

    // display result
    printf("===== ROOT FOUND BY CONSTANT NUMBER OF ITERATION =====\n");
    printf("x = %lf\n\n", x0);
    printf("First error: %.20lf\n", firstError);
    printf("Second error: %.20lf\n", secondError);
    fprintf(output, "===== ROOT FOUND BY CONSTANT NUMBER OF ITERATION =====\n");
    fprintf(output, "Number of iteration: %d\n", n);
    fprintf(output, "x = %lf\n", x0);
    fprintf(output, "First error: %.20lf\n", firstError);
    fprintf(output, "Second error: %.20lf\n\n", secondError);

    return x0;
}
```

Hàm tính sai số có hai công thức sau:

$$|x_n - x^*| \le \frac{|f(x_n)|}{m_1} \qquad (1)$$

$$|x_n - x^*| \le \frac{M_2}{2m_1}\left|x_n - x_{n-1}\right|^2 \qquad (2)$$

Ở đó $m_1 = min_{x\in[a,b]}|f'(x)|$; $M_2 = max_{x\in[a,b]}|f''(x)|$

3.7.1 Hàm tính sai số theo công thức 1

Gọi hàm **firstErrorFunction(Polynomial polynomial, double *interval, double x)** để tính sai số theo công thức (1)

Hàm **firstErrorFunction(Polynomial polynomial, double *interval, double x)** nhận vào ba tham số là đa thức cần tính (polynomial: Polynomial) và môt mảng gồm hai phần tử chứa hai đầu mút của khoảng chứa nghiệm (interval: double*) và điểm cần tình sai số (x: double)

- khởi tạo biến start và end lần lượt là hai giá trị thứ nhất và thứ hai trong mảng **interval**

- Khai báo biến min là giá trị nhỏ nhất của trị tuyệt đối của đạo hàm trong khoảng [start, end]
- Khai báo biến startValue, endValue để lưu lần lượt các giá trị đạo hàm tại start và end
- Tính đạo hàm của đa thức lưu vào biến df
- Tính startValue và endValue. Giữa hai giá trị đó, giá trị nào nhỏ hơn thì gán vào min
- Tính giá trị tuyệt đối của đa thức tại điểm x rồi chia cho min, trả về giá trị đó

Code chi tiết:

```
// Function to define error function 1
double firstErrorFunction(Polynomial polynomial, double *interval, double x) {
    // variable declaration
    double start = interval[0];
    double end = interval[1];
    double min; // the minimum value of |f'(x)| in [start, end]
    double startValue, endValue;

    // get the first order derivative function
    Polynomial df = getDerivativeFunction(polynomial);

    // find min
    startValue = fabs(getValue(df, start));
    endValue = fabs(getValue(df, end));
    min = (startValue < endValue) ? startValue : endValue;

    // compute and return the value of the error
    return fabs(getValue(polynomial, x)) / min;
}
```

3.7.2 Hàm tính sai số theo công thức 2

Gọi hàm **secondErrorFunction(Polynomial polynomial, double *interval, double cur, double prev)** để tính sai số theo công thức (2)

Hàm **secondErrorFunction(Polynomial polynomial, double *interval, double cur, double prev)** nhận vào bốn tham số là đa thức cần tính (polynomial: Polynomial) và môt mảng gồm hai phần tử chứa hai đầu mút của khoảng chứa nghiệm (interval: double*) và điểm cần tình sai số hiện tại (cur: double) và điểm đã tình sai số ngay trước đó (prev: double)

- khởi tạo biến start và end lần lượt là hai giá trị thứ nhất và thứ hai trong mảng **interval**

- Khai báo biến min là giá trị nhỏ nhất của trị tuyệt đối của đạo hàm cấp 1 trong khoảng [start, end]
- Khai báo biến max là giá trị lớn nhất của trị tuyệt đối của đạo hàm cấp2 trong khoảng [start, end]
- Tính đạo hàm cấp 1 của đa thức lưu vào biến df
- Tính đạo hàm cấp 2 của đa thức lưu vào biến ddf
- Tính giá trị của biến max và min rồi tính sai số theo công thức số (2) rồi trả về giá trị vừa tính được

Code chi tiết:

```
// Function to define error function 2
double secondErrorFunction(Polynomial polynomial, double *interval, double cur, double prev) {
    // variable declaration
    double start = interval[0];
    double end = interval[1];
    double min; // the minimum value of |f'(x)| in [start, end]
    double max; // the maximum value of |f''(x)| in [start, end]

    // get the first order derivative function
    Polynomial df = getDerivativeFunction(polynomial);
    Polynomial ddf = getDerivativeFunction(df);

    // find min
    double firstOrderStartValue = fabs(getValue(df, start));
    double firstOrderEndValue = fabs(getValue(df, end));
    min = (firstOrderStartValue < firstOrderEndValue) ? firstOrderStartValue : firstOrderEndValue;

    // find max
    double secondOrderStartValue = fabs(getValue(ddf, start));
    double secondOrderEndValue = fabs(getValue(ddf, end));
    max = (secondOrderStartValue > secondOrderEndValue) ? secondOrderStartValue : secondOrderEndValue;

    // compute and return the value of the error
    return (max * pow(fabs(cur - prev), 2.0)) / (2 * min);
}
```

3.8 Hàm tìm nghiệm với các hàm sai số 1

Gọi hàm **findRootWithGivenErrorUseFirstErrorFunc(Polynomial polynomial, double *interval)** để tìm nghiệm theo công thức sai số (1) và sai số nhập vào bởi người dùng.

Hàm **findRootWithGivenErrorUseFirstErrorFunc(Polynomial polynomial, double *interval)** nhận vào hai tham số là đa thức cần tính (polynomial: Polynomial) và môt mảng gồm hai phần tử chứa hai đầu mút của khoảng chứa nghiệm (interval: double*)

- Khai báo biến e là lưu sai số mà người dùng nhập vào
- khởi tạo biến start và end lần lượt là hai giá trị thứ nhất và thứ hai trong mảng **interval**
- khởi tạo biến x0 là giá trị khởi đầu khi áp dụng phương pháp pháp tuyến để tìm nghiệm của phương trình. Gán x0 là trung điểm của start và end.
- Khai báo biến error lưu giá trị của hàm sai số tại điểm đang xét theo công thức sai số 1
- Yêu cầu người dùng nhập vào sai s e, nếu người dùng nhập vào số âm hoặc bằng 0, yêu cầu nhập lại
- Tìm đạo hàm của đa thức cần tìm nghiệm, lưu vào biến df
- Lặp cho tới khi giá trị error nhỏ hơn hoặc bằng giá trị e, với mỗi vòng lặp:
  - Tính gía trị của đa thức tại điểm x0, lưu vào biến tạm y
  - Tính giá trị của đạo hàm tại điểm x0, lưu vào biến tạm dy
  - Tính x1 = x0 - y/dy
  - tính hai giá trị error bằng việc gọi hàm **firstErrorFunction(polynomial, interval, x0)**
- Hiển thị ra màn hình và lưu vào file output.txt kết quả vừa tính:
  - Lưu dòng ===== ROOT FOUND BY FIRST ERROR FUNCTION ===== vào file output.txt, in dòng đó ra màn hình
  - Lưu vào file output.txt giá trị của e
  - In ra màn hình và lưu nghiệm x0 vào file output.txt

Code chi tiết:

```c
// Function to find a root with given error e and use the first error function
double findRootWithGivenErrorUseFirstErrorFunc(Polynomial polynomial, double *interval) {
    // variable declaration
    double e; // given error
    double start = interval[0];
    double end = interval[1];
    int i; // loop variable
    double x0 = (start + end) / 2;
    double error;

    // prompt user for error
    do
    {
        printf("Enter given error: ");
        scanf("%lf", &e);
        if (e <= 0)
        {
            printf("The given error must be a small number greater than 0!\n");
        }
    } while (e <= 0);

    // get the first order derivative of the polynomial
    Polynomial firstOrderDerivative = getDerivativeFunction(polynomial);

    // compute error using first error function
    error = firstErrorFunction(polynomial, interval, x0);

    // find root
    while (error > e) {
        double y = getValue(polynomial, x0);
        double dy = getValue(firstOrderDerivative, x0);

        // get next value
        x0 = x0 - y / dy;

        // computer error
        error = firstErrorFunction(polynomial, interval, x0);
    }

    // display result
    printf("===== ROOT FOUND BY FIRST ERROR FUNCTION =====\n");
    printf("x = %lf\n\n", x0);
    fprintf(output, "===== ROOT FOUND BY FIRST ERROR FUNCTION =====\n");
    fprintf(output, "Error: %lf\n", e);
    fprintf(output, "x = %.15lf\n\n", x0);

    // return
    return x0;
}
```

3.9 Hàm tìm nghiệm với các hàm sai số 2

Việc tìm nghiệm với hàm sai số 2 tương tự với hàm sai số thứ nhất, chỉ khác thay vì tính giá trị biến error bằng hàm sai số 1 như trong phần trên, ta gọi hàm **secondErrorFunction(polynomial, interval, x0)** để tính **error**

Code chi tiết:

```
// Function to find a root with given error e and use the second error function
double findRootWithGivenErrorUseSecondErrorFunc(Polynomial polynomial, double *interval) {
    // variable declaration
    double e; // given error
    double start = interval[0];
    double end = interval[1];
    int i; // loop variable
    double x0 = (start + end) / 2;
    double x1; // next value
    double error; // error computed by second error function

    // prompt user for error
    do
    {
        printf("Enter given error: ");
        scanf("%lf", &e);
        if (e <= 0)
        {
            printf("The given error must be a small number greater than 0!\n");
        }
    } while (e <= 0);

    // get the first order derivative of the polynomial
    Polynomial firstOrderDerivative = getDerivativeFunction(polynomial);
```

```
    // find root
    i = 0;
    while (1)
    {
        double y = getValue(polynomial, x0);
        double dy = getValue(firstOrderDerivative, x0);

        // get next value
        x1 = x0 - y / dy;

        // computer error
        error = secondErrorFunction(polynomial, interval, x1, x0);

        // store current value
        x0 = x1;

        if (error < e) {
            break;
        }

        // counter
        i++;
    }

    // display result
    printf("===== ROOT FOUND BY SECOND ERROR FUNCTION =====\n");
    printf("x = %lf\n\n", x0);
    fprintf(output, "===== ROOT FOUND BY SECOND ERROR FUNCTION =====\n");
    fprintf(output, "Error: %lf\n", e);
    fprintf(output, "x = %.15lf\n\n", x0);

    // return
    return x0;
}
```

3.10 Hàm tìm nghiệm với sai số nhập vào từ người dùng

Gọi hàm **findRootWithGivenError(Polynomial polynomial, double *interval)** để tìm nghiệm theo công thức |Xn - Xn-1| <= e với e là sai số nhập vào bởi người dùng.

Hàm **findRootWithGivenError(Polynomial polynomial, double *interval)** nhận vào hai tham số là đa thức cần tính (polynomial: Polynomial) và môt mảng gồm hai phần tử chứa hai đầu mút của khoảng chứa nghiệm (interval: double*)

- Khai báo biến e là lưu sai số mà người dùng nhập vào
- Khởi tạo biến start và end lần lượt là hai giá trị thứ nhất và thứ hai trong mảng **interval**

- Khởi tạo biến x0 là giá trị khởi đầu khi áp dụng phương pháp pháp tuyến để tìm nghiệm của phương trình. Gán x0 là trung điểm của start và end.
- Khai báo biến x1 là giá trị tiếp theo khi áp dụng phương pháp pháp tuyến để tìm nghiệm của phương trình.
- Yêu cầu người dùng nhập vào sai s e, nếu người dùng nhập vào số âm hoặc bằng 0, yêu cầu nhập lại
- Tìm đạo hàm của đa thức cần tìm nghiệm, lưu vào biến df
- Tạo vòng lặp vô hạn, với mỗi vòng lặp:
  - Tính gía trị của đa thức tại điểm x0, lưu vào biến tạm y
  - Tính giá trị của đạo hàm tại điểm x0, lưu vào biến tạm dy
  - Tính x1 = x0 - y/dy
  - Nếu |x1 - x0| <= e thì kết thúc vòng lặp
  - gán **x0 = x1**
- Hiển thị ra màn hình và lưu vào file output.txt kết quả vừa tính:
  - Lưu dòng ===== ROOT FOUND BY GIVEN ERROR===== vào file output.txt, in dòng đó ra màn hình
  - Lưu vào file output.txt giá trị của e
  - In ra màn hình và lưu nghiệm x0 vào file output.txt

Code chi tiết:

```c
// Function to find a root with given error e satisfying |Xn - Xn-1| <= e
double findRootWithGivenError(Polynomial polynomial, double *interval) {
    // variable declaration
    double e; // given error
    double start = interval[0];
    double end = interval[1];
    int i; // loop variable
    double x0 = (start + end) / 2;
    double x1;    // next value

    // prompt user for error
    do
    {
        printf("Enter given error: ");
        scanf("%lf", &e);
        if (e <= 0)
        {
            printf("The given error must be a small number greater than 0!\n");
        }
    } while (e <= 0);

    // get the first order derivative of the polynomial
    Polynomial firstOrderDerivative = getDerivativeFunction(polynomial);


    // find root
    while (1)
    {
        double y = getValue(polynomial, x0);
        double dy = getValue(firstOrderDerivative, x0);

        // get next value
        x1 = x0 - y / dy;

        if (fabs(x1-x0) <= e) {
            break;
        }

        // store current value
        x0 = x1;
    }

    // display result
    printf("===== ROOT FOUND BY GIVEN ERROR =====\n");
    printf("x = %lf\n\n", x0);
    fprintf(output, "===== ROOT FOUND BY GIVEN ERROR =====\n");
    fprintf(output, "Error: %lf\n", e);
    fprintf(output, "x = %.15lf\n\n", x0);

    // return
    return x0;
}
```

Chương 4

Tổng kết

4.1 Toàn bộ chương trình

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

// Polynomial abstract data type
You, 9 hours ago | 1 author (You)
typedef struct polynomial {
    int degree;
    double* coefficient;
} Polynomial;

/* ------------ GLOBAL VARIABLES ------------ */
// The list to hold all the intervals containing roots of the equation
double* rootIntervals;

// The file for storing the output of the program
FILE *output;

// contraint for too small denominator when perform division
double CONSTRAINT =  1e-15;


/* ------------ FUNCTION'S PROTOTYPE ------------ */
// Function to display menu
void displayMenu();

// Function to input the polynomial
Polynomial initPolynomial();

// Function to display interval
void displayInterval(int size);

// Function to get the value of polynomial at point x
double getValue(Polynomial polynomial, double x);

// Function to get the first derivative function of a polynomial
Polynomial getDerivativeFunction(Polynomial polynomial);

// Function to find upper bound of range of real roots
double findUpperBound(Polynomial polynomial);
```

```
41  // Function to find lower bound of range of real roots
42  double findLowerBound(Polynomial polynomial);
43
44  // Function to shrink an interval containing root
45  double* shrinkIntervalContainingRoot(Polynomial polynomial, double* interval);
46
47  // Function to define error function 1
48  double firstErrorFunction(Polynomial polynomial, double* interval, double x);
49
50  // Function to define error function 2
51  double secondErrorFunction(Polynomial polynomial, double *interval, double cur, double prev);
52
53  // Function to find a root with constant number of iteration
54  double findRootWithConstantIteration(Polynomial polynomial, double *interval);
55
56  // Function to find a root with given error e and use the first error function
57  double findRootWithGivenErrorUseFirstErrorFunc(Polynomial polynomial, double *interval);
58
59  // Function to find a root with given error e and use the second error function
60  double findRootWithGivenErrorUseSecondErrorFunc(Polynomial polynomial, double *interval);
61
62  // Function to find a root with given error e satisfying |Xn - Xn-1| <= e
63  double findRootWithGivenError(Polynomial polynomial, double *interval);
64
65  // Function to check if an interval contains a real root
66  int hasRealRoot(Polynomial polynomial, double start, double end);
67
68  // Gradient decent function
69  double gradientDescent(Polynomial polynomial, double guess, double upperBound);
70
71  // find all interval containing roots
72  int findAllIntervalContainingRoots(Polynomial polynomial);
73
74  // Function to display shrinked interval
75  void displayShrinkedInterval(double* interval, int size);
76
77  /* ------------ MAIN FUNCTION ------------ */
78  int main() {
79
80      // Create and open output file in the current directory
81      output = fopen("output.txt", "a");
```

```c
/* ------------ MAIN FUNCTION ------------ */
int main() {

    // Create and open output file in the current directory
    output = fopen("output.txt", "a");
    if (!output) {
        printf("Cannot open this file\n");
        return 1;
    }
    else {
        printf("File is opened\n");
    }

    // Polynomial initialization
    Polynomial polynomial;

    // Main flow
    int option;
    int quit = 0;
    int size; // size of interval containing all roots
    double* shrinkedInterval = (double*)malloc(sizeof(double) * 2);
    double root1, root2, root3, root4; // for storing the roots found

    while (!quit) {
        displayMenu();

        // prompt user for option
        scanf("%d", &option);

        switch (option) {
            case 1:
                system("clear");
                polynomial = initPolynomial();
                break;
            case 2:
                system("clear");
                size = findAllIntervalContainingRoots(polynomial);
                displayInterval(size);
                quit = (size < 1) ? 1 : 0;
                break;
            case 3:
```

```
118                 system("clear");
119                 shrinkedInterval[0] = rootIntervals[0];
120                 shrinkedInterval[1] = rootIntervals[1];
121                 shrinkedInterval = shrinkIntervalContainingRoot(polynomial, shrinkedInterval);
122                 displayShrinkedInterval(shrinkedInterval, 2);
123                 break;
124             case 4:
125                 system("clear");
126                 root1 = findRootWithConstantIteration(polynomial, shrinkedInterval);
127                 break;
128             case 5:
129                 system("clear");
130                 root2 = findRootWithGivenErrorUseFirstErrorFunc(polynomial, shrinkedInterval);
131                 root3 = findRootWithGivenErrorUseSecondErrorFunc(polynomial, shrinkedInterval);
132                 break;
133             case 6:
134                 system("clear");
135                 root4 = findRootWithGivenError(polynomial, shrinkedInterval);
136                 break;
137             case 0:
138                 quit = 1;
139                 break;
140         }
141     }
142 
143     // free
144     free(rootIntervals);
145     free(shrinkedInterval);
146 
147     // close file
148     fclose(output);
149     return 0;
150 }
151 
152 /* ------------ FUNCTION'S DETAILS ------------ */
153 // Function to display menu
154 void displayMenu()
155 {
156     printf("\t\t\t  ============================MENU==============================\n");
157     printf("\t\t\t |1. Enter the polynomial\n");
158     printf("\t\t\t |2. Find all intervals containing root of the polynomial\n");
```

```
159     printf("\t\t\t |3. Find interval containing root [a, b] of the polynomial such that |a - b| <
160     printf("\t\t\t |4. Find the root of the polynomial given a particular number of iteration in
161     printf("\t\t\t |5. Find the root of the polynomial given a particular error in the interval(a
162     printf("\t\t\t |6. Find the root Xn of the polynomial in the interval (a, b) such that |Xn -
163     printf("\t\t\t |0. Exit the program\n");
164     printf("\t\t\t Your option: ");
165 }
166
167 // Function for inputing the polynomial
168 Polynomial initPolynomial() {
169     // write to file
170     fprintf(output, " ===== INPUT POLYNOMIAL ====== \n");
171
172     // declaration
173     Polynomial polynomial;
174     int i; // loop variable
175
176     // prompt user for degree of polynomial
177     do {
178         printf("Enter the degree of the polynomial: ");
179         scanf("%d", &polynomial.degree);
180         if (polynomial.degree < 0) {
181             printf("The degree of a polynomial must be greater or equal to 0!\n");
182         }
183     } while (polynomial.degree < 0);
184
185     // write degree to the file
186     fprintf(output, "Degree: %d\n", polynomial.degree);
187
188     // allocate memory for storing polynomial's coefficients
189     polynomial.coefficient = (double* ) malloc((polynomial.degree + 1) * sizeof(double));
190
191     // prompt user for coefficients of polynomial
192     printf("Enter the coefficients of the polynomial: ");
193     fprintf(output, "Coefficients: ");
194     for (i = 0; i <= polynomial.degree; i++) {
195         scanf("%lf", &polynomial.coefficient[i]);
196         fprintf(output, "%7.4lf", polynomial.coefficient[i]); // write the coefficient to output
197     }
198     fprintf(output, "\n\n");
199
```

```
200     return polynomial;
201 }
202
203 // Function to get the value of polynomial at point x
204 double getValue(Polynomial polynomial, double x) {
205     // declaration
206     double value = 0;
207     int i; // loop variable
208
209     // get value at point x
210     for (i = 0; i <= polynomial.degree; i++) {
211         value += polynomial.coefficient[i] * pow(x, (double)(polynomial.degree-i));
212     }
213
214     // return
215     return value;
216 }
217
218 // Function to get the first derivative function of a polynomial
219 Polynomial getDerivativeFunction(Polynomial polynomial) {
220     // Variable declaration
221     Polynomial df;
222     int i; // loop variable
223
224     // get the derivative function
225     df.degree = polynomial.degree - 1; // degree of derivative function
226     df.coefficient = (double*) malloc (df.degree * sizeof(double));
227     for (i = 0; i <= df.degree; i++) {
228         df.coefficient[i] = (polynomial.degree-i) * polynomial.coefficient[i];
229     }
230
231     // return
232     return df;
233 }
234
235 // Function to shrink an interval containing root
236 double *shrinkIntervalContainingRoot(Polynomial polynomial, double *interval) {
237     // declaration
238     double tolerance = 0.5; // restrict the length of interval
239     double start = interval[0]; // the begining of interval
240     double end = interval[1]; // the end of the interval
```

```
241     double mid; // midpoint of start and end
242     double *shrinkedInterval = (double*) malloc (2 * sizeof(double));
243 
244         // shrink the interval until |start - end| <= tolerant
245         while (fabs(start - end) > tolerance)
246     {
247         mid = (start + end) / 2;
248         if (getValue(polynomial, mid) * getValue(polynomial, start) > 0) {
249             start = mid;
250         }
251         else {
252             end = mid;
253         }
254     }
255 
256     // store interval
257     shrinkedInterval[0] = start, shrinkedInterval[1] = end;
258 
259     // return
260     return shrinkedInterval;
261 }
262 
263 // Function to define error function 1
264 double firstErrorFunction(Polynomial polynomial, double *interval, double x) {
265     // variable declaration
266     double start = interval[0];
267     double end = interval[1];
268     double min; // the minimum value of |f'(x)| in [start, end]
269     double startValue, endValue;
270 
271     // get the first order derivative function
272     Polynomial df = getDerivativeFunction(polynomial);
273 
274     // find min
275     startValue = fabs(getValue(df, start));
276     endValue = fabs(getValue(df, end));
277     min = (startValue < endValue) ? startValue : endValue;
278 
279     // compute and return the value of the error
280     return fabs(getValue(polynomial, x)) / min;
281 }
```

```
283  // Function to define error function 2
284  double secondErrorFunction(Polynomial polynomial, double *interval, double cur, double prev) {
285      // variable declaration
286      double start = interval[0];
287      double end = interval[1];
288      double min; // the minimum value of |f'(x)| in [start, end]
289      double max; // the maximum value of |f''(x)| in [start, end]
290
291      // get the first order derivative function
292      Polynomial df = getDerivativeFunction(polynomial);
293      Polynomial ddf = getDerivativeFunction(df);
294
295      // find min
296      double firstOrderStartValue = fabs(getValue(df, start));
297      double firstOrderEndValue = fabs(getValue(df, end));
298      min = (firstOrderStartValue < firstOrderEndValue) ? firstOrderStartValue : firstOrderEndValu
299
300      // find max
301      double secondOrderStartValue = fabs(getValue(ddf, start));
302      double secondOrderEndValue = fabs(getValue(ddf, end));
303      max = (secondOrderStartValue > secondOrderEndValue) ? secondOrderStartValue : secondOrderEnd
304
305      // compute and return the value of the error
306      return (max * pow(fabs(cur - prev), 2.0)) / (2 * min);
307  }
308
309  // Function to find a root with constant number of iteration
310  double findRootWithConstantIteration(Polynomial polynomial, double *interval) {
311      // variable declaration
312      double start = interval[0];
313      double end = interval[1];
314      int i; // loop variable
315      int n; // number of iteration
316      double x0 = (start + end) / 2;
317      double x1; // next value for computation
318      double firstError, secondError; // error for the first and second error function respectivel
319
320      // get the first order derivative of the polynomial
321      Polynomial df = getDerivativeFunction(polynomial);
322
323      // prompt user for number of iteration
```

```
323     // prompt user for number of iteration
324     do {
325         printf("Enter number of iteration: ");
326         scanf("%d", &n);
327         if (n <= 0) {
328             printf("Number of iteration must be greater than 0!\n");
329         }
330     } while (n <= 0);
331 
332     // find root
333     for (i = 0; i < n; i++) {
334         double y = getValue(polynomial, x0);
335         double dy = getValue(df, x0);
336 
337         // get next value
338         x1 = x0 - y/dy;
339 
340         // get the first error (with first error function) and second error (with second error fu
341         firstError = firstErrorFunction(polynomial, interval, x1);
342         secondError = secondErrorFunction(polynomial, interval, x1, x0);
343 
344         // store current value
345         x0 = x1;
346     }
347 
348     // display result
349     printf("===== ROOT FOUND BY CONSTANT NUMBER OF ITERATION =====\n");
350     printf("x = %lf\n\n", x0);
351     printf("First error: %.20lf\n", firstError);
352     printf("Second error: %.20lf\n", secondError);
353     fprintf(output, "===== ROOT FOUND BY CONSTANT NUMBER OF ITERATION =====\n");
354     fprintf(output, "Number of iteration: %d\n", n);
355     fprintf(output, "x = %lf\n", x0);
356     fprintf(output, "First error: %.20lf\n", firstError);
357     fprintf(output, "Second error: %.20lf\n\n", secondError);
358 
359     return x0;
360 }
361 
362 // Function to find a root with given error e and use the first error function
363 double findRootWithGivenErrorUseFirstErrorFunc(Polynomial polynomial, double *interval) {
```

```c
double findRootWithGivenErrorUseFirstErrorFunc(Polynomial polynomial, double *interval) {
    // variable declaration
    double e; // given error
    double start = interval[0];
    double end = interval[1];
    int i; // loop variable
    double x0 = (start + end) / 2;
    double error;

    // prompt user for error
    do
    {
        printf("Enter given error: ");
        scanf("%lf", &e);
        if (e <= 0)
        {
            printf("The given error must be a small number greater than 0!\n");
        }
    } while (e <= 0);

    // get the first order derivative of the polynomial
    Polynomial firstOrderDerivative = getDerivativeFunction(polynomial);

    // compute error using first error function
    error = firstErrorFunction(polynomial, interval, x0);

    // find root
    while (error > e) {
        double y = getValue(polynomial, x0);
        double dy = getValue(firstOrderDerivative, x0);

        // get next value
        x0 = x0 - y / dy;

        // computer error
        error = firstErrorFunction(polynomial, interval, x0);
    }

    // display result
    printf("===== ROOT FOUND BY FIRST ERROR FUNCTION =====\n");
    printf("x = %lf\n\n", x0);
```

```
403     printf("x = %lf\n\n", x0);
404     fprintf(output, "===== ROOT FOUND BY FIRST ERROR FUNCTION =====\n");
405     fprintf(output, "Error: %lf\n", e);
406     fprintf(output, "x = %.15lf\n\n", x0);
407
408     // return
409     return x0;
410 }
411
412 // Function to find a root with given error e and use the second error function
413 double findRootWithGivenErrorUseSecondErrorFunc(Polynomial polynomial, double *interval) {
414     // variable declaration
415     double e; // given error
416     double start = interval[0];
417     double end = interval[1];
418     int i; // loop variable
419     double x0 = (start + end) / 2;
420     double x1; // next value
421     double error; // error computed by second error function
422
423     // prompt user for error
424     do
425     {
426         printf("Enter given error: ");
427         scanf("%lf", &e);
428         if (e <= 0)
429         {
430             printf("The given error must be a small number greater than 0!\n");
431         }
432     } while (e <= 0);
433
434     // get the first order derivative of the polynomial
435     Polynomial firstOrderDerivative = getDerivativeFunction(polynomial);
436
437     // find root
438     i = 0;
439     while (1)
440     {
441         double y = getValue(polynomial, x0);
442         double dy = getValue(firstOrderDerivative, x0);
443
```

```
444         // get next value
445         x1 = x0 - y / dy;
446 
447         // computer error
448         error = secondErrorFunction(polynomial, interval, x1, x0);
449 
450         // store current value      You, 9 hours ago • project ver1
451         x0 = x1;
452 
453         if (error < e) {
454             break;
455         }
456 
457         // counter
458         i++;
459     }
460 
461     // display result
462     printf("===== ROOT FOUND BY SECOND ERROR FUNCTION =====\n");
463     printf("x = %lf\n\n", x0);
464     fprintf(output, "===== ROOT FOUND BY SECOND ERROR FUNCTION =====\n");
465     fprintf(output, "Error: %lf\n", e);
466     fprintf(output, "x = %.15lf\n\n", x0);
467 
468     // return
469     return x0;
470 }
471 
472 // Function to find a root with given error e satisfying |Xn - Xn-1| <= e
473 double findRootWithGivenError(Polynomial polynomial, double *interval) {
474     // variable declaration
475     double e; // given error
476     double start = interval[0];
477     double end = interval[1];
478     int i; // loop variable
479     double x0 = (start + end) / 2;
480     double x1;    // next value
481 
482     // prompt user for error
483     do
484     {
```

```
483     do
484     {
485         printf("Enter given error: ");
486         scanf("%lf", &e);
487         if (e <= 0)
488         {
489             printf("The given error must be a small number greater than 0!\n");
490         }
491     } while (e <= 0);
492 
493     // get the first order derivative of the polynomial
494     Polynomial firstOrderDerivative = getDerivativeFunction(polynomial);
495 
496     // find root
497     while (1)
498     {
499         double y = getValue(polynomial, x0);
500         double dy = getValue(firstOrderDerivative, x0);
501 
502         // get next value
503         x1 = x0 - y / dy;
504 
505         if (fabs(x1-x0) <= e) {
506             break;
507         }
508 
509         // store current value
510         x0 = x1;
511     }
512 
513     // display result
514     printf("===== ROOT FOUND BY GIVEN ERROR =====\n");
515     printf("x = %lf\n\n", x0);
516     fprintf(output, "===== ROOT FOUND BY GIVEN ERROR =====\n");
517     fprintf(output, "Error: %lf\n", e);
518     fprintf(output, "x = %.15lf\n\n", x0);
519 
520     // return
521     return x0;
522 }
523 
```

```
523
524 // Function to check if an interval contains a real root
525 int hasRealRoot(Polynomial polynomial, double start, double end)
526 {
527     return getValue(polynomial, start) * getValue(polynomial, end) < 0;
528 }
529
530 // Function to find upper bound of range of real roots
531 double findUpperBound(Polynomial polynomial) {
532     // variable's declaration
533     int i; // loop variable
534     int k = -1; // index of the first negative coefficient
535     double B; // maximum value of the absolute value among negative coefficient
536     int A;    // maximum of absolute value of coefficients except the first coefficient
537
538     if (polynomial.coefficient[0] > 0) {
539         // find the first negative coefficient
540         for (i = 0; i <= polynomial.degree; i++) {
541             if (polynomial.coefficient[i] < 0) {
542                 k = i;
543                 break;
544             }
545         }
546         // if there is no negative coefficient => upperbound = 0
547         if (k == -1) {
548             return 0;
549         }
550         // otherwise find B
551         else {
552             B = fabs(polynomial.coefficient[k]);
553             for (i = k; i <= polynomial.degree; i++) {
554                 if (polynomial.coefficient[i] < 0 &&  B < fabs(polynomial.coefficient[i])) {
555                     B = fabs(polynomial.coefficient[i]);
556                 }
557             }
558             return 1 + pow(fabs(B / polynomial.coefficient[0]), 1.0 / k);
559         }
560     }
561     else {
562         // find maximum of absolute value of coefficients except the first coefficient
563         A = fabs(polynomial.coefficient[1]);
```

```
560         }
561         else {
562             // find maximum of absolute value of coefficients except the first coefficient
563             A = fabs(polynomial.coefficient[1]);
564             for (i = 2; i < polynomial.degree; i++) {
565                 if (A < fabs(polynomial.coefficient[i])) {
566                     A = fabs(polynomial.coefficient[i]);
567                 }
568             }
569 
570             return 1 + A / fabs(polynomial.coefficient[0]);
571         }
572     }
573 
574     // Function to find lower bound of range of real roots
575     double findLowerBound(Polynomial polynomial) {
576         // variable's declaration
577         int i;
578         Polynomial polynomialTemp;
579         polynomialTemp.coefficient = (double*)malloc(sizeof(double) * (polynomial.degree + 1));
580         polynomialTemp.degree = polynomial.degree;
581 
582         // Find P(-x)
583         if (polynomial.degree % 2 == 0) {
584             for (i = 0; i < polynomial.degree; i++) {
585                 polynomialTemp.coefficient[i] = (i % 2 == 0) ? polynomial.coefficient[i] : -polynomia
586             }
587         }
588         else {
589             for (i = 0; i < polynomial.degree; i++){
590                 polynomialTemp.coefficient[i] = (i % 2 != 0) ? polynomial.coefficient[i] : -polynomia
591             }
592         }
593         polynomialTemp.coefficient[polynomialTemp.degree] = polynomial.coefficient[polynomial.degree]
594 
595         // return lower bound
596         return (-1) * findUpperBound(polynomialTemp);
597     }
598 
599     // Gradient decent function
600     double gradientDescent(Polynomial df, double guess, double upperBound)
```

```
599  // Gradient decent function
600  double gradientDescent(Polynomial df, double guess, double upperBound)
601  {
602      // variable's declaration
603      int sign;
604      double leaningRate = 0.001;
605      double eps = 1e-10;
606
607      // find extrema
608      if (getValue(df, guess) == 0) {
609          return guess;
610      }
611      else if (getValue(df, guess) < 0) {
612          sign = -1;
613      }
614      else {
615          sign = 1;
616      }
617
618      while (fabs(getValue(df, guess)) > eps) {
619          guess = guess + sign * leaningRate * getValue(df, guess);
620          if (guess > upperBound) {
621              return upperBound;
622          }
623      }
624      return guess;
625  }
626
627  // find all interval containing roots
628  int findAllIntervalContainingRoots(Polynomial polynomial)
629  {
630      // variables
631      rootIntervals = (double*)malloc(sizeof(double) * 1);
632      double upperBound = findUpperBound(polynomial);
633      double lowerBound = findLowerBound(polynomial);
634      int size = 1;
635      rootIntervals[0] = lowerBound;
636      double step = 1e-3;
637      double i = lowerBound;
638      int sign;
639
```

```
640     // find derivative function
641     Polynomial df = getDerivativeFunction(polynomial);
642 
643     // find extrema
644     while (i < upperBound) {
645         i = gradientDescent(df, i, upperBound);
646         if (hasRealRoot(polynomial, rootIntervals[size-1], i)) {
647             rootIntervals = (double*)realloc(rootIntervals, sizeof(double) * (size + 1));
648             rootIntervals[size++] = i;
649         }
650         sign = (getValue(df, i) > 0) ? 1 : -1;
651         do {
652             i += step;
653             if (i >= upperBound) {
654                 break;
655             }
656         } while ( (getValue(df, i) * sign > 0) && (fabs(getValue(df, i) > getValue(df, i + step)
657     }
658     return size;
659 }
660 
661 // Function to display interval
662 void displayInterval(int size)
663 {
664     // check if polynomial has a root
665     fprintf(output, "===== ALL INTERVALS CONTAINING ROOTS =====\n");
666     if (size == 1) {
667         printf("The function has no root\n\n");
668         fprintf(output, "The function has no root\n\n");
669     }
670     else {
671         printf("===== ALL INTERVALS CONTAINING ROOTS =====\n");
672         for (int i = 0; i < size - 1; i++) {
673             printf("[%lf, %lf]\n", rootIntervals[i], rootIntervals[i+1]);
674             fprintf(output, "[%lf, %lf]\n", rootIntervals[i], rootIntervals[i + 1]);
675         }
676         fprintf(output, "\n");
677     }
678 }

680 // Function to display shrinked interval
681 void displayShrinkedInterval(double* interval, int size)
682 {
683     fprintf(output, "===== SHRINKED INTERVAL =====\n");
684     for (int i = 0; i < size - 1; i++)
685     {
686         printf("===== SHRINKED INTERVAL =====\n");
687         printf("[%lf, %lf]\n", interval[i], interval[i + 1]);
688         fprintf(output, "[%lf, %lf]\n", interval[i], interval[i + 1]);
689     }
690     fprintf(output, "\n");
691 }
```

4.2 File lưu kết quả

```
===== INPUT POLYNOMIAL ======
Degree: 3
Coefficients:  1.0000 3.0000-5.0000 1.0000

===== ALL INTERVALS CONTAINING ROOTS =====
[-6.000000, -2.632993]
[-2.632993, 0.632993]
[0.632993, 3.236068]

===== SHRINKED INTERVAL =====
[-4.316497, -3.895621]

===== ROOT FOUND BY CONSTANT NUMBER OF ITERATION =====
Number of iteration: 10
x = -4.236068
First error: 0.00000000000000062133
Second error: 0.00000000000000000000

===== ROOT FOUND BY FIRST ERROR FUNCTION =====
Error: 0.000001
x = -4.236067977738450

===== ROOT FOUND BY SECOND ERROR FUNCTION =====
Error: 0.000001
x = -4.236067977738450

===== ROOT FOUND BY GIVEN ERROR =====
Error: 0.000001
x = -4.236067977738450

```