

KỸ THUẬT LẬP TRÌNH

Giảng viên hướng dẫn: Nguyễn Thị Thanh Huyền

Nhóm sinh viên: nhóm 1 lớp 116444

Nguyễn Minh Hiếu 20185349

Phạm Minh Đức 20185337

Đào Như Quỳnh 20185472



I	Mở đầu	3
II	Cơ sở lý thuyết	4
1	Nội dung phương pháp	4
2	Sự hội tụ của phương pháp	4
3	Sai số	6
III	Thuật toán	7
1	Thuật toán lặp Newton	7
2	Thuật toán cho điều kiện	7
2.1	Thuật toán Horner	7
2.2	Brute-force Search và Gradient Descent	8
2.3	Xét điều kiện bằng tìm nghiệm đạo hàm	8
IV	Thiết kế chương trình	10
1	Cấu trúc dữ liệu	10
2	Lặp Newton	11
3	Tìm miền chứa nghiệm	11
4	Tìm điều kiện bằng Brute-force Search và Gradient Descent	12
5	Tìm điều kiện bằng đệ quy	14
6	Các chương trình phụ	15
6.1	Chương trình hiển thị số thập phân	15
6.2	Chương trình thu hẹp khoảng phân li	16
V	Kiểm tra và đánh giá	17
1	Kiểm tra chương trình	17
2	Đánh giá kết quả	19
VI	Tổng kết	21

Mở đầu

Trong lĩnh vực Toán ứng dụng, việc giải phương trình vô cùng quan trọng và có nhiều ứng dụng, nhất là đa thức. Tuy nhiên đối với đa thức bậc cao thì tìm nghiệm đúng vô cùng khó khăn. Từ đó việc tìm ra cách để tìm nghiệm gần đúng của một phương trình đã được rất nhiều nhà khoa học nghiên cứu và tìm hiểu.

Một trong những phương pháp tìm nghiệm gần đúng đó chính là phương pháp lặp Newton - Raphson, đặt tên theo Isaac Newton và Joseph Raphson, cho phép tìm nghiệm xấp xỉ gần đúng của một hàm số mà chúng ta đã được tiếp cận trong học phần Giải tích số.

Ứng dụng những kiến thức đã học trong bộ môn Kỹ Thuật Lập Trình do cô Nguyễn Thị Thanh Huyền giảng dạy, chúng em đã xây dựng lên một chương trình dựa vào thuật toán Newton - Raphson để giải tìm nghiệm gần đúng của phương trình đa thức bậc n . Trong bài báo cáo này chúng em xin trình bày chi tiết về quá trình thiết kế thành một chương trình cụ thể, thuật toán cũng như mã nguồn.

Chúng em xin cảm ơn cô Huyền vì những kiến thức cô đã truyền tải tới chúng em thông qua học phần này. Bài báo cáo của chúng em tuy đã hoàn thiện nhưng cũng không thể tránh khỏi những sai lầm, thiếu sót, chúng em rất mong nhận được sự góp ý từ phía cô để có thể rút kinh nghiệm cũng như hoàn thiện bài của mình một cách tốt nhất.

1 Nội dung phương pháp

Phương pháp Newton-Raphson

Giả thiết (a, b) là khoảng phân li nghiệm, f liên tục và có đạo hàm liên tục đến cấp hai, f' và f'' không đổi dấu, quá trình lặp

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}$$

được gọi là lặp theo phương pháp tiếp tuyến (phương pháp Newton-Raphson). Khi đó, nếu chọn x_0 sao cho $f(x_0)f''(x_0) > 0$, dãy $\{x_n\}$ hội tụ về nghiệm của phương trình $f(x) = 0$

Ý tưởng của phương pháp là xuất phát từ một điểm x_0 gần với nghiệm đúng x^* trên khoảng đang xét, ta xấp xỉ hàm số bằng tiếp tuyến tương ứng tại điểm đang xét để xác định dãy lặp $\{x_n\}$ hội tụ về nghiệm đúng x^* . Tại điểm $(x_0, f(x_0))$ ban đầu, xác định tiếp tuyến

$$y = f'(x_0)(x - x_0) + f(x_0)$$

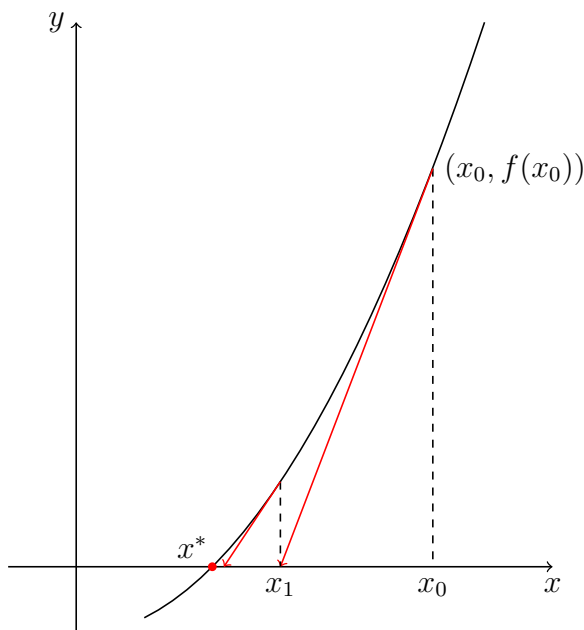
tiếp tuyến này cắt trục hoành tại điểm $(x_1, 0)$, khi đó ta có

$$0 = f'(x_0)(x_1 - x_0) + f(x_0)$$

từ đó suy ra $x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$.

Tiếp tục lặp lại như vậy ta sẽ có công thức lặp:

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}$$



2 Sự hội tụ của phương pháp

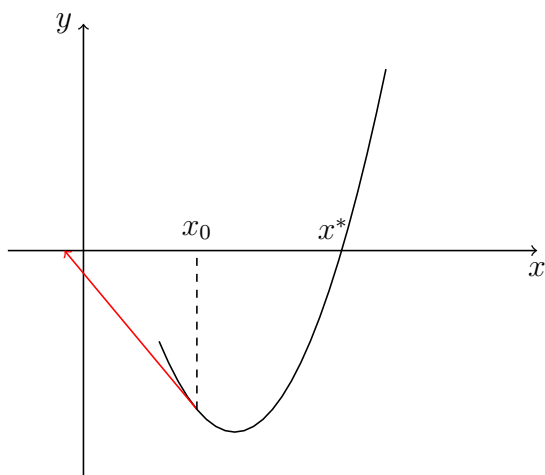
Phương pháp tiếp tuyến hội tụ tới nghiệm đúng trên khoảng cách ly nghiệm với điều kiện f' , f'' không đổi dấu và chọn xấp xỉ đầu là điểm Fourier

- Điều kiện f' không đổi dấu: Đảm bảo hội tụ về nghiệm trong khoảng đang xét và không gặp trường hợp chia cho 0
- Điều kiện f'' không đổi dấu: Đảm bảo không gặp trường hợp lặp vô hạn

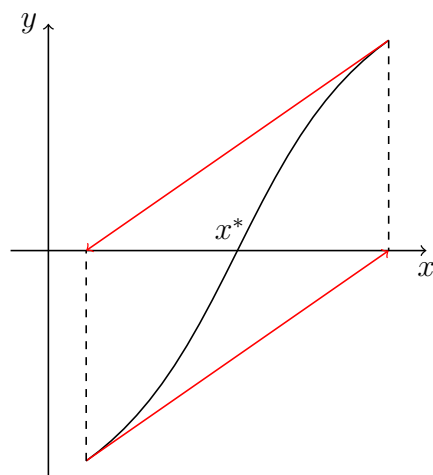
- Điều kiện xấp xỉ ban đầu được chọn là điểm Fourier, tức là thỏa mãn

$$f(x_0)f''(x_0) > 0$$

đảm bảo dãy $\{x_n\}$ hội tụ đơn điệu về nghiệm đúng.

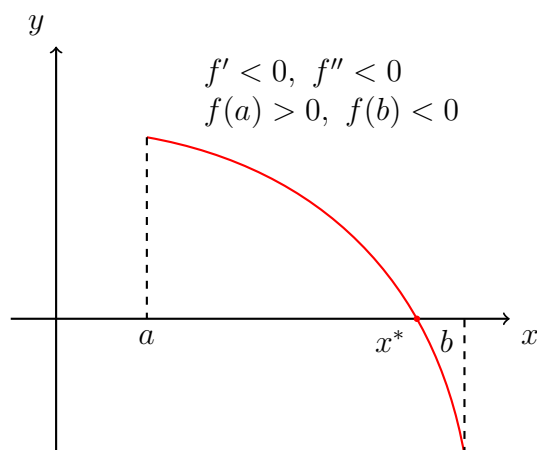
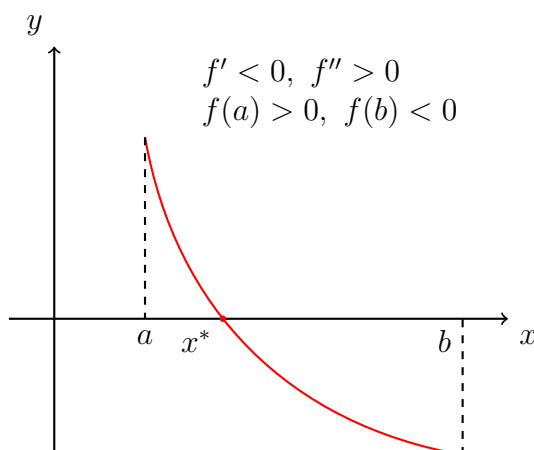
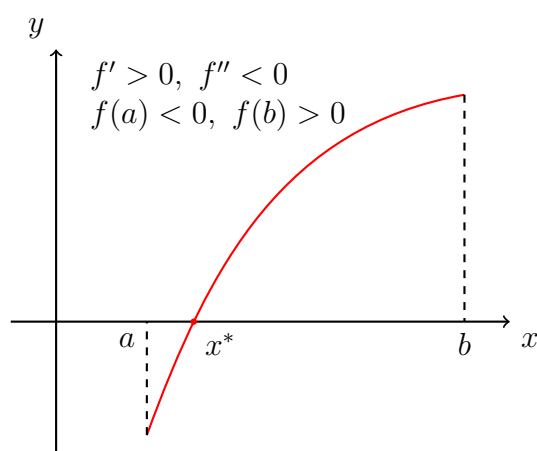
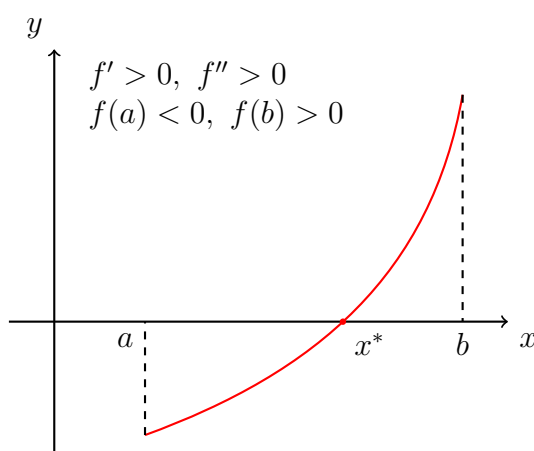


Trường hợp f' đổi dấu



Trường hợp f'' đổi dấu

Như vậy, các khoảng được gọi là thỏa mãn nếu là một trong 4 trường hợp sau



3 Sai số

Công thức sai số

Xét trên (a, b) thỏa mãn điều kiện thực hiện phương pháp, giả sử x^* là nghiệm đúng của phương trình $f(x) = 0$, nghiệm x_n là nghiệm gần đúng sau n lần lặp. Khi đó với $m_1 = \min_{x \in (a, b)} |f'(x)|$, ta có

$$|x_n - x^*| \leq \frac{|f(x_n)|}{m_1}$$

Từ công thức trên, ta suy ra công thức sai số xấp xỉ hai lần liên tiếp

$$|x_n - x^*| \leq \frac{M_2}{2m_1} |x_n - x_{n-1}|^2$$

với $m_1 = \min_{x \in (a, b)} |f'(x)|$ và $M_2 = \max_{x \in (a, b)} |f''(x)|$

Như vậy, phương pháp tiếp tuyến cho tốc độ hội tụ bậc 2. Khi đó, nếu x_n đúng đến m số chữ số thập phân thì x_{n+1} sẽ đúng đến khoảng $2m$ chữ số thập phân.

Thuật toán

1 Thuật toán lặp Newton

Thuật toán lặp Newton rất đơn giản, xét trên khoảng (a, b) thỏa mãn điều kiện thuật toán

Bước 1. Tính $e = \sqrt{\frac{2m_1\varepsilon}{M_2}}$

Bước 2. Chọn điểm Fourier $x_0 = a$ nếu $f(a)f''(a) > 0$ hoặc $x_0 = b$ nếu $f(b)f''(b) > 0$

Bước 3. Thực hiện lặp $x_k = x_{k-1} - \frac{f(x_{k-1})}{f'(x_{k-1})}$

Bước 4. Nếu $|x_k - x_{k-1}| \leq e$ thì chuyển đến bước 5, ngược lại chuyển đến bước 3.

Bước 5. In kết quả

2 Thuật toán cho điều kiện

Điều kiện để thực hiện phương pháp tiếp tuyến là $[a, b]$ là khoảng phân li nghiệm. Công việc cần làm là tìm ra các khoảng này, đồng thời tính được hai giá trị m_1 và M_2

2.1 Thuật toán Horner

Thuật toán dùng để tìm khoảng chứa nghiệm của đa thức. Ta có định lí

Định lí

Cho đa thức bậc n hệ số thực $f(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$ với $a_n \neq 0$. Miền chứa nghiệm của đa thức là

$$\left(-1 - \frac{\max |a_i|}{|a_n|}, 1 + \frac{\max |a_i|}{|a_n|} \right) \quad i = \overline{0, n}$$

Như vậy dựa vào định lí ta tìm được miền chứa nghiệm của phương trình. Nếu dùng thuật toán này để chia khoảng vừa tìm được thành nhiều phần, và xét hai đầu các khoảng được chia có trái dấu không, thì khi các nghiệm gần nhau có thể khoảng đó chưa phải khoảng phân ly. Do đó cách này không hề hiệu quả, ta phải dùng thêm các giải thuật khác.

Dễ thấy, đối với đa thức, khoảng phân li nghiệm là khoảng không chứa cực trị và điểm uốn. Vậy, ta đi tìm cực trị và điểm uốn của phương trình, có hai cách thực hiện:

- Duyệt tìm các cực đại cực tiểu địa phương (cực trị), là các điểm nhỏ hơn hoặc lớn hơn lân cận của nó.
- Tìm nghiệm của $f'(x)$, $f''(x)$.

2.2 Brute-force Search và Gradient Descent

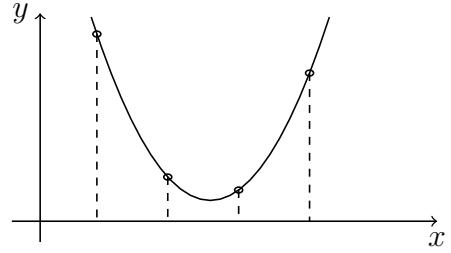
Sử dụng thuật toán này để tìm các điểm cực tiểu và cực đại.

Bước 1. Chia đoạn đang xét thành $m+1$ khoảng bằng nhau bởi m điểm u_1, u_2, \dots, u_m

Bước 2. Cực trị xấp xỉ là u_i nếu

- $f(u_i) \leq f(u_{i-1})$ và $f(u_i) \leq f(u_{i+1})$ (cực tiểu) hoặc $f(u_i) \geq f(u_{i-1})$ và $f(u_i) \geq f(u_{i+1})$ (cực đại)
- $|f'(u_i)| \leq \varepsilon$

Tuy nhiên, đối với thuật toán này việc tìm kiếm là khó khăn với miền lớn, khi đó khoảng chia là không đủ nhỏ để sai số là phù hợp (bị bỏ qua điểm cực trị). Tương tự như vậy, nếu m không đủ lớn ($m < 1000$). Nhưng nếu chia đủ nhỏ thì khối lượng tính toán là lớn.



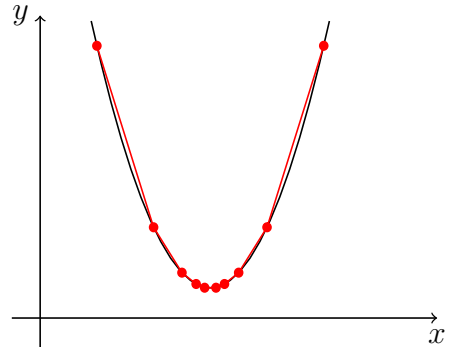
Thuật toán Gradient Descent là thuật toán tối ưu hóa việc tìm cực đại và cực tiểu bằng công thức lặp

$$x_k = x_{k-1} - \theta f'(x_{k-1})$$

Khi đó, dãy $\{x_k\}$ hội tụ đến cực trị khi learning rate θ đủ nhỏ. Ở đây f' là gradient.

Tuy nhiên, ta cần xét khoảng và tìm giá trị lớn nhất nhỏ nhất trên cả đoạn. Do đó, kết hợp Brute-force Search và Gradient Descent, khắc phục được một phần nhược điểm khi chia khoảng. Như vậy thay vì chia thành các khoảng bằng nhau, chọn u_{i+1} thỏa mãn

$$u_{i+1} = u_i + \theta |f'(u_i)|$$



và chọn $\theta = \frac{b-a}{1000}$ sẽ cho chương trình tối ưu hơn.

Giải thuật này sẽ tốn ít nhất 1000 vòng lặp để xét khoảng thỏa mãn điều kiện và tìm giá trị lớn nhất nhỏ nhất.

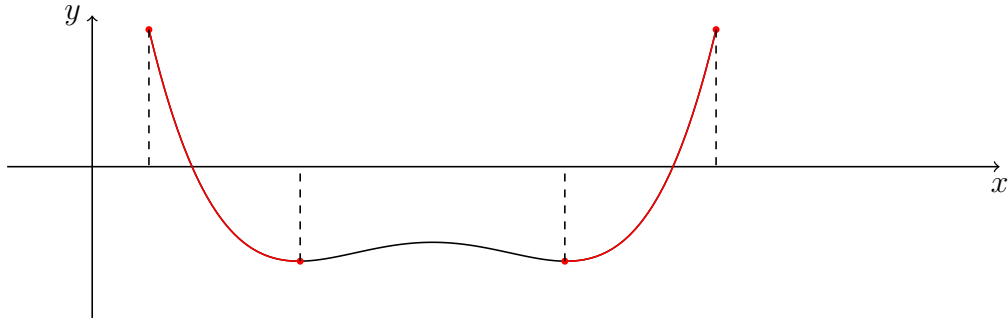
Như vậy, với giải thuật như trên, đa thức bậc n có tối đa n nghiệm, sẽ có tối đa n khoảng phân li nghiệm cần xét. Giả sử mỗi khoảng phân li cần không quá ω lần lặp để hội tụ đến nghiệm, tổng thời gian thực hiện chương trình

$$T \approx (2000 + \omega)n$$

Độ phức tạp thuật toán là $O(n\omega)$. Tuy nhiên bài toán xử lý một khối lượng tính toán cực kì lớn, điều này sẽ giải thích kĩ hơn ở phần sau.

2.3 Xét điều kiện bằng tìm nghiệm đạo hàm

Ngoài cách duyệt tìm cực trị, ta hoàn toàn có thể tìm nghiệm của các đạo hàm, từ đó tìm được khoảng thỏa mãn là khoảng không chứa nghiệm của $f'(x) = 0$ và $f''(x) = 0$, đồng thời hai đầu khoảng trái dấu.



Nhờ vào tính chất đặc biệt của đa thức $f^{(n)}(x) = 0 \forall n \in \mathbb{N}$, ta đưa bài toán trở thành đệ quy tìm nghiệm của đạo hàm.

Đa thức bậc n có tối đa n nghiệm, sẽ có tối đa n khoảng phân li nghiệm cần xét. Giả sử mỗi khoảng phân li cần không quá ω lần lặp để hội tụ đến nghiệm, tổng thời gian thực hiện chương trình

$$T = [n + (n - 1) + \dots + 1]\omega = \frac{n(n + 1)}{2}\omega$$

Độ phức tạp thuật toán là $O(n^2\omega)$ nhưng sẽ xử lý khối lượng tính toán nhỏ hơn nhiều vì phương pháp Newton hội tụ nhanh và chỉ cần thực hiện tìm nghiệm.

1 Cấu trúc dữ liệu

Để lưu trữ một đa thức hay đạo hàm của nó, ta cần lưu hệ số và số mũ. Như vậy cần một kiểu dữ liệu dạng mảng hoặc danh sách, trong đó khả dụng là

- **Array:** Mảng là một cấu trúc dữ liệu khá hiệu quả, cho phép truy cập phần tử nhanh, dễ dàng tính toán. Tuy nhiên kích thước có hạn nên sẽ không phù hợp với đa thức hợp bởi rất nhiều đơn thức.
- **Linked list:** Danh sách móc nối là một cấu trúc dữ liệu ổn định trong việc thêm bớt. Tuy nhiên, cấu trúc này không phù hợp với bài toán vì để lưu một tập hợp dữ liệu n phần tử cần $2n$ ô nhớ (một để dữ liệu, một để chứa con trỏ tới ô khác), thực tế bài toán sẽ cần đến $3n$ ô nhớ. Hơn nữa, trong bài toán cần tính giá trị hàm số tại một điểm rất nhiều, do đó cần duyệt danh sách nhiều lần. Khi đó tốc độ của danh sách móc nối là chậm.
- **Tree:** Cấu trúc cây sẽ không phù hợp với bài toán vì mục tiêu bài toán không phải là thêm bớt phần tử hay duyệt tìm đơn thức.

Như vậy ta sẽ dùng cấu trúc mảng, và bài toán yêu cầu độ chính xác cao nên ta sẽ dùng kiểu dữ liệu `double`. Để thuận tiện, tạo một kiểu dữ liệu riêng là kiểu dữ liệu đa thức

```
1 struct PolynomialFunction {
2     double coeff[100]; // Mảng chứa hệ số của đơn thức bậc i
3     int deg; // Bậc của đa thức
4 } func;
5 typedef struct PolynomialFunction polynomial;
```

Khi đó, dễ dàng tính được đạo hàm của $f(x)$ bằng

```
1 polynomial diff(polynomial func) {
2     polynomial dfunc;
3     int i;
4     dfunc.deg = func.deg - 1; // Bậc của đa thức f'
5     for (i = 0; i <= dfunc.deg; i++) {
6         dfunc.coeff[i] = func.coeff[i + 1] * (i + 1);
7     }
8     return dfunc;
9 }
```

và hàm tính giá trị một hàm số tại một điểm

```
1 double calc(polynomial func, double x) {
2     double value = 0;
```

```

3     int i;
4     for (i = 0; i <= func.deg; i++) {
5         value += func.coeff[i] * pow(x, i);
6     }
7     return value;
8 }

```

2 Lặp Newton

Xây dựng chương trình lặp phương pháp Newton rất đơn giản, ta chỉ cần thực hiện lặp trên khoảng thỏa mãn cho đến khi đạt đủ sai số

```

1 // Chọn điểm Fourier
2 if (sign(calc(func, a)) == sign(calc(ddfunc, a))) {
3     x = a;
4 } else {
5     x = b;
6 }
7 // Thực hiện lặp Newton với err là sai số
8 do {
9     s = x;
10    x = x - calc(func, x) / calc(dfunc, x);
11    k++;
12 } while (fabs(x - s) >= err);

```

Vấn đề là ta phải tìm được miền nghiệm và khoảng phân ly nghiệm thì mới thực hiện lặp được.

3 Tìm miền chứa nghiệm

Áp dụng tìm miền chứa nghiệm bằng định lí nêu trên

```

1 void rootsRange(polynomial func, double a, double b, double
   range[]) {
2     int i;
3     double max_coeff = fabs(func.coeff[0]);
4     // Tìm hệ số có môđun lớn nhất
5     for (i = 1; i <= func.deg; i++) {
6         if (max_coeff < fabs(func.coeff[i])) {
7             max_coeff = fabs(func.coeff[i]);
8         }
9     }
10    // Nếu miền tìm được ngoài khoảng [a,b] yêu cầu thì thay đổi
11    range[0] = -1 - max_coeff / fabs(func.coeff[func.deg]);
12    if (a > range[0]) {
13        range[0] = a;
14    }

```

```

15     range[1] = 1 + max_coeff / fabs(func.coeff[func.deg]);
16     if (b < range[1]) {
17         range[1] = b;
18     }
19 }

```

Khi đó ta sẽ tìm miền chứa nghiệm trên $(-\infty, +\infty)$ bằng
`rootsRange(func, -INF, INF, D);`

4 Tìm điều kiện bằng Brute-force Search và Gradient Descent

Cốt lõi của thuật toán chính là duyệt tìm giá trị nhỏ nhất lớn nhất trên đoạn và trả về điểm có giá trị đó

```

1  double min(polynomial func, double a, double b) {
2      double x0 = a, e = 1e-3, xmin = a, alpha = (b - a) / 10000;
3      polynomial dfunc = diff(func);
4      do {
5          // Bước nhảy để vượt qua điểm cực trị để tìm điểm tiếp
           theo
6          x0 = x0 + e;
7          // Dừng tìm kiếm khi x0 là xấp xỉ cực trị, tức là
            $f'(x_0) \leq e$ 
8          while ((fabs(calc(dfunc, x0)) > e) && (x0 <= b)) {
9              // Step size/ learning rate là alpha, gradient là
                $f'(x_0)$ 
10             x0 = x0 + alpha * fabs(calc(dfunc, x0));
11         }
12         if (x0 > b) break;
13         else
14             // Min là giá trị nhỏ nhất trong các cực trị tìm được hoặc
               là hai đầu mút (nếu không có cực trị)
15             if (calc(func, x0) < calc(func, xmin)) {
16                 xmin = x0;
17             }
18     } while (true);
19     if (calc(func, xmin) < calc(func, b)) return xmin;
20     else return b;
21 }

```

Làm tương tự khi tìm max. Khi đó sai số sẽ được tính bằng

```

1  //  $m_1 = \min |f'(x)|$ 
2  if (calc(dfunc, a) < 0) {
3      m1 = calc(dfunc, max(dfunc, a, b));
4  }
5  else {

```

```

6     m1 = calc(dfunc, min(dfunc, a, b));
7 }
8 //  $M_2 = \max |f''(x)|$ 
9 if (calc(ddfunc, a) < 0) {
10     m2 = calc(ddfunc, min(ddfunc, a, b));
11 }
12 else {
13     m2 = calc(ddfunc, max(ddfunc, a, b));
14 }
15 // Sai số là  $c = \sqrt{\frac{2m_1 e}{M_2}}$ 
16 c = sqrt(fabs(2.0 * m1 * e / m2));

```

Sau đó thực hiện chia miền chứa nghiệm thành nhiều phần, và xét xem các phần này có phải khoảng thỏa mãn hay không. Ở đây chia làm n phần với đa thức bậc n

```

1 rootsRange(func, -INF, INF, D);
2 printf("\nKhoang chua nghiem [%lf, %lf]", D[0], D[1]);
3 int n = func.deg;
4 double range[n];
5 // Lưu các khoảng chia
6 for (i = 0; i <= n; i++) {
7     range[i] = D[0] + i * (D[1] - D[0]) / n;
8 }
9 for (i = 0; i < n; i++) {
10     // Hai đầu phải trái dấu
11     if (sign(calc(func, range[i])) != sign(calc(func, range[i + 1]))) {
12         // Tìm đoạn mà  $f'$  không đổi dấu
13         a = min(func, range[i], range[i + 1]);
14         b = max(func, range[i], range[i + 1]);
15         // Xét điều kiện  $f''$  không đổi dấu
16         if (sign(calc(diff(diff(func)), a)) ==
17             sign(calc(diff(diff(func)), b))) {
18             if (a > b) {
19                 sol(func, b, a, err);
20             } else {
21                 sol(func, a, b, err);
22             }
23         }
24     }
25 }

```

Tuy nhiên với cách chia như vậy, các khoảng chia sẽ có thể chứa nhiều hơn 1 nghiệm, ta sẽ chia nhỏ hơn (chia thành n^3 khoảng), nhưng đồng thời sẽ cần tính toán nhiều hơn. Như vậy với cách làm này, ta sẽ không xử lý được các phương trình có nghiệm rất gần nhau.

5 Tìm điều kiện bằng đệ quy

Để lưu các nghiệm của đạo hàm tìm được, dùng một mảng 2 chiều kích cỡ $n \times (n+1)$ với hàng thứ i là nghiệm của đa thức bậc i (đạo hàm lần thứ $n - i$) và phần tử đầu tiên lưu số lượng nghiệm tìm được. Tuy nhiên, ta có thể cải tiến bằng cách xóa các dữ liệu không cần thiết, khi đó chỉ cần 2 mảng một chiều.

Đệ quy tìm nghiệm với điều kiện dừng là khi gặp đa thức bậc 1 và bậc 0, ta suy ra được tập nghiệm của nó

```
1 if (func.deg == 1) {
2     roots[1][0] = 1;
3     roots[1][1] = - func.coeff[0] / func.coeff[1];
4 }
5 else
6 if (func.deg == 0) {
7     roots[0][0] = 0;
8 }
```

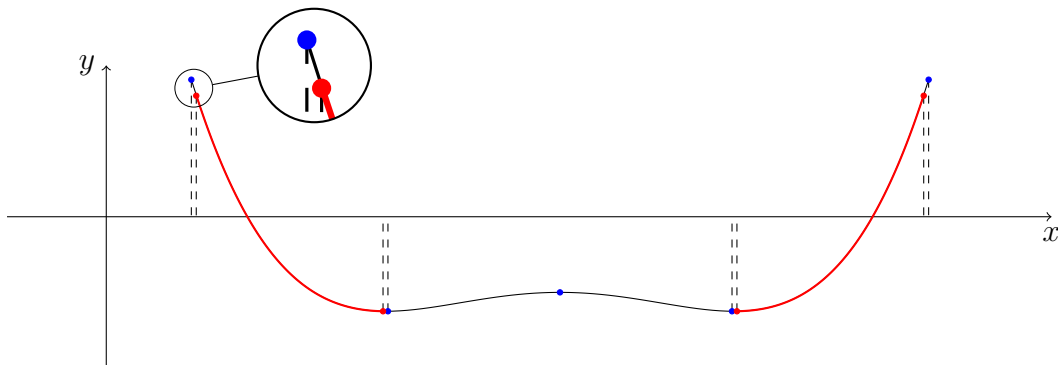
Đa thức bậc k sẽ cần đến nghiệm của đa thức bậc $k - 1$ và $k - 2$ vì các khoảng thỏa mãn là các khoảng không chứa những điểm là nghiệm của 2 đạo hàm gần nhất. Do đó khi tìm được những điểm này, ta ghép lại thành 1 mảng và xét duyệt trên mảng này

```
1 merge(roots[n - 1], roots[n - 2], sp);
2 merge(sp, ab, point);
3 // Với point là mảng chứa các điểm nghiệm của f' và f'' và hai đầu
4 miền nghiệm
5 for (i = 1; i < (int) point[0]; i++) {
6     // Xét khoảng nằm giữa 2 điểm (bỏ qua các điểm này), nếu nó
7     trái dấu ở hai đầu thì đây là khoảng cần tìm
8     if (sign(calc(func, point[i] + e / 2)) != sign(calc(func,
9         point[i + 1] - e / 2))) {
10         // Tìm nghiệm bằng lặp Newton khi đã có khoảng phân li
11         roots[n][index++] = nrsolve(func, point[i] + e / 2, point[i
12             + 1] - e / 2, e);
13     }
14     // Trường hợp nghiệm của f', f'' cũng là nghiệm của f
15     if (fabs(calc(func, point[i])) <= fabs(calc(diff(func),
16         point[i]) * e)) {
17         roots[n][index++] = point[i];
18     }
19 }
20 if (fabs(calc(func, point[(int) point[0]])) <=
21     fabs(calc(diff(func), point[(int) point[0]]) * e)) {
22     roots[n][index++] = point[(int) point[0]];
23 }
24 roots[n][0] = index - 1;
25 printf("\nRoot Set: ");
26 // Tìm được tập nghiệm của phương trình
```

```

21 for (i = 1; i <= (int) roots[n][0]; i++) {
22     printf("\n    x%d = %lf ", i, roots[n][i]);
23 }

```



Do bỏ qua các điểm là nghiệm của $f'(x) = 0$ và $f''(x) = 0$, nên ta phải xét thêm trường hợp các điểm đó cũng có thể là nghiệm của $f(x) = 0$.

Vì $\Delta f(x) = f'(x)\Delta x$, nên suy ra với x là xấp xỉ nghiệm sai số ε thì

$$|f(x)| = |f(x) - f(x^*)| < \varepsilon |f'(x)|$$

Như vậy cách làm trên sẽ cho đủ nghiệm của phương trình cần tính

6 Các chương trình phụ

6.1 Chương trình hiển thị số thập phân

Ta thực hiện ngắt phần nguyên và phần thập phân, sau đó chuyển sang dạng xâu kí tự với số lượng kí tự như ý muốn, rồi ghép các xâu lại với nhau

```

1 char * display(double var) {
2     int i;
3     long long intpart = (long long) var, decpart = (long
4         long)(fabs(var * pow(10, digits) - (double) intpart *
5             pow(10, digits)));
6     char istring[25], dstring[25];
7     lltoa(intpart, istring, 10);
8     lltoa(decpart, dstring, 10);
9     if ((var < 0) && (intpart == 0)) {
10         strcpy(dis, "-");
11         strcat(dis, istring);
12     } else {
13         strcpy(dis, istring);
14     }
15     strcat(dis, ".");
16     if (strlen(dstring) < digits) {
17         for (i = 0; i < digits - strlen(dstring); i++) {
18             strcat(dis, "0");
19         }
20     }
21 }

```



```

18     }
19     strcat(dis, dstring);
20     return disp;
21 }

```

6.2 Chương trình thu hẹp khoảng phân li

Sử dụng phương pháp chia đôi: thực hiện chia đôi khoảng cho đến khi độ dài khoảng đạt mong muốn

```

1 void bisection(polynomial func, double a, double b, double
   distance, double range[]) {
2     int i;
3     double temp;
4     while (b - a > distance) {
5         temp = (b + a) / 2;
6         if (sign(calc(func, temp)) == sign(calc(func, b))) {
7             b = temp;
8         } else {
9             a = temp;
10        }
11    }
12    range[0] = a;
13    range[1] = b;
14 }

```

Kiểm tra và đánh giá

1 Kiểm tra chương trình

Trong phần này, sử dụng hai chương trình là

- **polynomial1.cpp** Chương trình minh họa thuật toán sử dụng Brute-force Search và Gradient Descent
- **polynomial2.cpp** Chương trình minh họa thuật toán sử dụng đệ quy

Đây là chương trình để minh họa so sánh tốc độ và xử lý dữ liệu của hai thuật toán, không phải chương trình hoàn thiện. Chương trình hoàn thiện là **polynomial.cpp**, được phát triển từ thuật toán thứ hai.

Ví dụ 1

Giải phương trình $x^4 - x^2 - 1 = 0$ với sai số 10^{-3}

Cả hai chương trình cho kết quả là giống nhau
Chương trình 1:

```
Ham so: -1.000000x^0 0.000000x^1 -1.000000x^2 0.000000x^3 1.000000x^4
Nhap sai so khac 0: 1e-3

Khoang chua nghiem [-2.000000, 2.000000]

Giai phuong trinh tren [-1.312500, -1.250000]
Error = 0.023855
Lan thu 1:
x = -1.274349136619504
Lan thu 2:
x = -1.272027919964315
Vay so lan lap: 2
x = -1.272028
Giai phuong trinh tren [1.250000, 1.312500]
Error = 0.023855
Lan thu 1:
x = 1.274349136619504
Lan thu 2:
x = 1.272027919964315
Vay so lan lap: 2
x = 1.272028
```

Chương trình 2:

```

Execute current function: -1.000000x^0 0.000000x^1 -1.000000x^2 0.000000x^3 1.000000x^4
Range: -2.000000 -0.707107 -0.408248 -0.000000 0.408248 0.707107 2.000000
Division range from -1.999500 to -0.707607
  Iteration 1:
  x = -1.606820
  Iteration 2:
  x = -1.376326
  Iteration 3:
  x = -1.285913
  Iteration 4:
  x = -1.272308
  Iteration 5:
  x = -1.272020
Total iterations: 5
x = -1.272020
Division range from 0.707607 to 1.999500
  Iteration 1:
  x = 1.606820
  Iteration 2:
  x = 1.376326
  Iteration 3:
  x = 1.285913
  Iteration 4:
  x = 1.272308
  Iteration 5:
  x = 1.272020
Total iterations: 5
x = 1.272020
Root Set:
x1 = -1.272020
x2 = 1.272020

```

Ví dụ 2

Giải phương trình $1000x^3 + 3000x^2 + 2999x - 999 = 0$ với sai số 10^{-3}

Phương trình có 3 nghiệm gần xấp xỉ nhau, do đó chương trình thứ nhất không thể cho kết quả vì khoảng chia là quá lớn so với khoảng cách các nghiệm. Kết quả chạy chương trình thứ hai

```

D:\dev\C\work\polynomial\polynomial1.exe
Iteration 13:
x = -1.031623
Iteration 14:
x = -1.031629
Iteration 15:
x = -1.031623
Total iterations: 15
x = -1.031623
Division range from -0.981222 to 3.999500
Error = 0.000002
  Iteration 1:
  x = 2.333044
  Iteration 2:
  x = 1.222096
  Iteration 3:
  x = 0.481498
  Iteration 4:
  x = -0.012185
  Iteration 5:
  x = -0.341232
  Iteration 6:
  x = -0.560483
  Iteration 7:
  x = -0.706483
  Iteration 8:
  x = -0.803562
  Iteration 9:
  x = -0.867900
  Iteration 10:
  x = -0.910218
  Iteration 11:
  x = -0.937564
  Iteration 12:
  x = -0.954484
  Iteration 13:
  x = -0.963837
  Iteration 14:
  x = -0.967644
  Iteration 15:
  x = -0.968353
  Iteration 16:
  x = -0.968377
Total iterations: 16
x = -0.968377
Root Set:
x1 = -1.031623
x2 = -1.000000
x3 = -0.968377

```

Ví dụ 3

Giải phương trình $x^{15} + x^{14} - x + 0,25 = 0$ sai số 10^{-3}

Chương trình thứ nhất cho kết quả

```
D:\devC\Cwork\polynomial\polynomial2.exe
Ham so: 0.250000x^0 -1.000000x^1 0.000000x^2 0.000000x^3 0.000000x^4 0.000000x^5 0.000000x^6 0.00
0000x^7 0.000000x^8 0.000000x^9 0.000000x^10 0.000000x^11 0.000000x^12 0.000000x^13 1.000000x^14
1.000000x^15
Nhap sai so khac 0: 1e-3

Khoang chua nghiem [-2.000000, 2.000000]

Giai phuong trinh tren [-1.165630, -1.164444]
Error = 0.011044
Lan thu 1:
x = -1.165624785567822
Vay so lan lap: 1
x = -1.165625
Giai phuong trinh tren [0.249481, 0.250667]
Error = 11.768860
Lan thu 1:
x = 0.250000004654767
Vay so lan lap: 1
x = 0.250000
Giai phuong trinh tren [0.927407, 0.928593]
Error = 0.011060
Lan thu 1:
x = 0.928075652477951
Vay so lan lap: 1
x = 0.928076
```

Chương trình thứ hai chạy ra kết quả sai do ban đầu chỉ cho bộ nhớ để giải phương trình bậc 10.

2 Đánh giá kết quả

Ta so sánh và đánh giá hai chương trình:

Chương trình sử dụng Brute-force Search và Gradient Descent

- Độ phức tạp thuật toán là $O(n\omega)$ với n là bậc của đa thức, ω là tốc độ hội tụ của phương pháp.
- Bộ nhớ: sử dụng ít hơn so với cách thứ 2
- Khối lượng tính toán: rất lớn. Ta thấy số lần gọi hàm tính giá trị lên đến hơn 17,000 lần. Trong ví dụ 1:


Profile Analysis

Function name	% time	Cumul. se...	Self secs	Calls	Self ts/call	Total ts/call
calc(PolynomialFunction, double)	0.00	0.00	0.00	17168	0.00	0.00
sign(double)	0.00	0.00	0.00	136	0.00	0.00
diff(PolynomialFunction)	0.00	0.00	0.00	20	0.00	0.00
max(PolynomialFunction, double, ...)	0.00	0.00	0.00	5	0.00	0.00
min(PolynomialFunction, double, ...)	0.00	0.00	0.00	3	0.00	0.00
sol(PolynomialFunction, double, d...	0.00	0.00	0.00	2	0.00	0.00
rootsRange(PolynomialFunction, ...)	0.00	0.00	0.00	1	0.00	0.00

- Ưu điểm: tiết kiệm bộ nhớ
- Nhược điểm: Khối lượng tính toán lớn, không giải được các phương trình có nghiệm sát nhau.

Chương trình sử dụng đệ quy

- Độ phức tạp thuật toán là $O(n^2\omega)$ với n là bậc của đa thức, ω là tốc độ hội tụ của phương pháp.
- Bộ nhớ: sử dụng nhiều hơn do phải lưu nghiệm và đệ quy
- Khối lượng tính toán: nhỏ hơn rất nhiều, ví dụ 1 cho kết quả

 Profile Analysis

Flat output	Call graph	Profiling Options					
Function name	% time	Cumul. se...	Self secs	Calls	Self ts/call	Total ts/call	
calc(PolynomialFunction, double)	0.00	0.00	0.00	174	0.00	0.00	
sign(double)	0.00	0.00	0.00	36	0.00	0.00	
diff(PolynomialFunction)	0.00	0.00	0.00	30	0.00	0.00	
merge(double*, double*, double*)	0.00	0.00	0.00	6	0.00	0.00	
solve(PolynomialFunction, double...	0.00	0.00	0.00	6	0.00	0.00	
rootsRange(PolynomialFunction, ...	0.00	0.00	0.00	1	0.00	0.00	
nrsolve(PolynomialFunction, doub...	0.00	0.00	0.00	1	0.00	0.00	

- Ưu điểm: giải phương trình ổn định
- Nhược điểm: Tốn bộ nhớ và phức tạp đối với các phương trình bậc cao do đạo hàm nhiều lần

Như vậy, tận dụng ưu điểm của phương pháp lặp Newton-Rapson là hội tụ nhanh, ta hoàn toàn có thể tối ưu được chương trình giải phương trình đa thức.

Tổng kết

Qua quá trình nghiên cứu về phương pháp tiếp tuyến để thiết kế chương trình và viết chương trình giải gần đúng phương trình đa thức, chúng em xin tổng kết lại như sau

- Phương pháp Newton là một trong những phương pháp rất quan trọng trong việc giải gần đúng nghiệm của phương trình $f(x) = 0$. Nó có những ưu điểm nổi bật hơn so với việc sử dụng các phương pháp chia đôi, dây cung hay lặp đơn. Mặc dù việc tính toán của phương pháp Newton phức tạp hơn các phương pháp trên do phải tính đạo hàm tại mỗi bước, nhưng với tốc độ hội tụ bậc hai, số lần lặp sẽ giảm đi đáng kể.
- Tuy nhiên không phải lúc nào chúng ta cũng có thể sử dụng phương pháp này. Khi sử dụng phương pháp Newton chúng ta cần hết sức lưu ý đến việc kiểm tra những điều kiện để phương pháp hội tụ. Mặt khác, khi tất cả các điều kiện đầu vào đều thỏa mãn, chúng ta cũng cần cân nhắc, có cái nhìn rõ ràng và thực tế nhất đối với từng bài toán và từng phương pháp để có được lời giải tối ưu nhất.
- Hơn nữa việc sử dụng cấu trúc dữ liệu và sử dụng các thuật toán để viết chương trình cũng rất quan trọng. Mỗi cấu trúc dữ liệu đều có ưu nhược điểm riêng trong việc lưu trữ và sử dụng. Và chúng ta nên tìm cách cải tiến trên nền tảng thuật toán cũ để bài toán được tối ưu hơn, đáp ứng tối đa nhu cầu người dùng.

Kiến thức chúng em còn nhiều hạn chế, nhưng với sự dẫn dắt tận tình và tâm huyết của cô Nguyễn Thị Thanh Huyền chúng em mới có thể hoàn thành được bài tập lớn trên tinh thần đoàn kết làm việc nhóm. Một lần nữa, chúng em xin chân thành cảm ơn và mong nhận được sự đóng góp của cô để chúng em được dần hoàn thiện hơn về kiến thức cũng như phong cách lập trình của chúng em sau này.