Numerary

Выпуск latest

сент. 20, 2020

Поиск корней

Numerary - это научная библиотека, которая содержит множество численных методов.

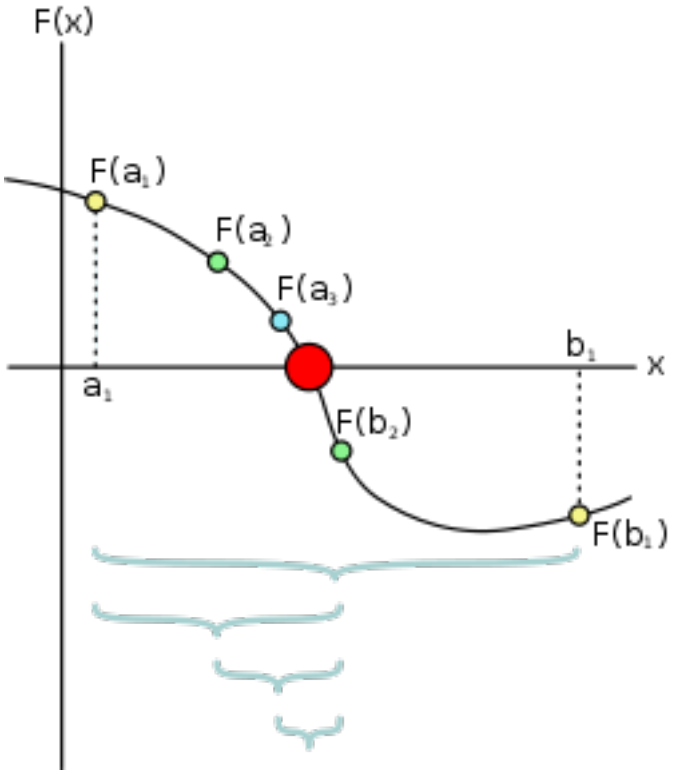
Глава 1

Содержание

1.1 Метод деления пополам

1.1.1 Обзор

В математике **метод деления пополам** - это метод поиска корней, который применяется к любым непрерывным функциям, для которых известны два значения с противоположными знаками. Метод состоит из многократного деления пополам интервала, определяемого этими значениями, и последующего выбора подинтервала, в котором функция меняет знак и, следовательно, должна содержать корень. Это очень простой и надежный метод, но он также относительно медленный. Из-за этого его часто используют для получения грубого приближения к решению, которое затем используется в качестве отправной точки для более быстро сходящихся методов. Этот метод также называют **методом деления интервала вдвое**, **методом двоичного поиска** или **методом дихотомии**.



1.1.2 Метод

Метод применим для численного решения уравнения $f(x)$: $math$:„ где f - непрерывная функция, определенная на интервале $[a, b]$, а $f(a)$ и $f(b)$ имеют противоположные знаки. . В этом случае говорят, что a и b заключают в скобки корень, поскольку по теореме о промежуточном значении непрерывная функция f должна иметь хотя бы один корень в интервале (a, b).

На каждом шаге метод делит интервал на две части, вычисляя среднюю точку $c = \frac{a+b}{2}$ интервала и значение функции $f(c)$ в этой точке. Если только c не является корнем (что очень маловероятно, но возможно), теперь есть только две возможности: либо $f(a)$ и $f(c)$ имеют противоположные знаки и скобки для корня, либо $f(c)$ и $f(b)$ иметь противоположные знаки и заключать в скобки корень. Метод выбирает подинтервал, который гарантированно является скобкой, в качестве нового интервала, который будет использоваться на следующем шаге. Таким образом, интервал, содержащий ноль f, уменьшается по ширине на 50% на каждом шаге. Процесс продолжается до тех пор, пока интервал не станет достаточно малым.

Явно, если $f(a)$ и $f(c)$ имеют противоположные знаки, тогда метод устанавливает c как новое значение для b, а если $f(b)$ и $f(c)$ имеют противоположные знаки, то метод устанавливает c как новое значение a. (Если $f(c) = 0$, то c может быть принято как решение, и процесс останавливается.) В обоих случаях новые $f(a)$ и $f(b)$ имеют противоположные знаки, поэтому метод применим к этому меньшему интервалу .

Итерационные задачи

1. Вычислите c, середину интервала, $c = \frac{a+b}{2}$.
2. Вычислить значение функции в средней точке, $f(c)$.

3. Если сходимость удовлетворительная (т. е. $c - a$ достаточно мало или $|f(c)|$ достаточно мало), верните c и прекратите повторение.
4. Изучите знак $f(c)$ и замените $(a, f(a))$ или $(b, f(b))$ на $(c, f(c))$ так, чтобы в новом интервале было пересечение нуля.

1.1.3 Использование

Представьте, что мы хотим найти корень следующей функции:

$$f(x) = \sin(x), x \in [-1, 1] \tag{1.1}$$

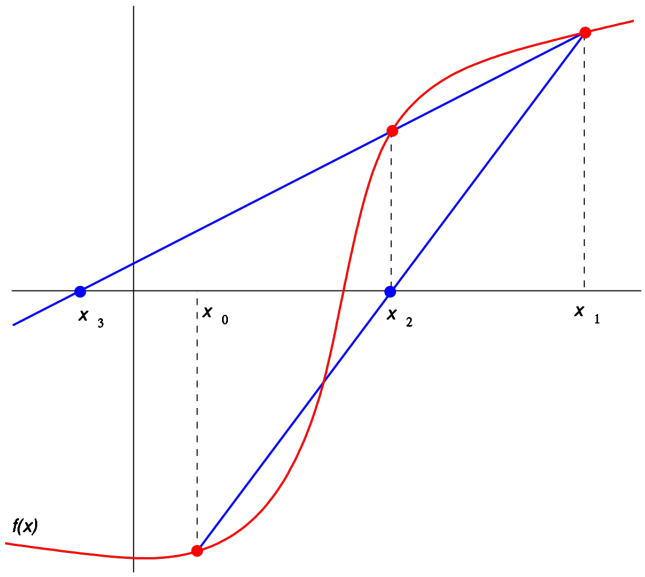
Тогда код будет выглядеть так:

```cpp
// example_root_bisection.cpp

#include <iostream>
#include "../src/numerary.hpp" // Numerary library

using namespace std;
using namespace numerary;

/* Function to found the root */
double f(double x) {
    return sin(x);
}

/* The main function */
int main() {

    const double eps = 1.e-9; // eps value for method (1.e-9 by default)
    double a = -1; // "a" value of segment [a, b]
    double b = 1; // "b" value of segment [a, b]
    double root;
    short int is_found;

    is_found = Numerary::root(f, a, b, &root, "bisection", eps);

    if (is_found == 1) {
        cout << "Root is in x = " << root << endl;
    } else {
        cout << "Method not allowed!" << endl;
    }

    return 0;
}
```

1.2 Метод хорд

1.2.1 Обзор

В численном анализе **метод секущих** - это алгоритм поиска корней, который использует последовательность корней секущих линий для лучшего приближения корня функции f. Метод секущих можно рассматривать как конечно-разностную аппроксимацию метода Ньютона. Однако этот метод был разработан независимо от метода Ньютона и предшествовал ему более чем на 3000 лет.



1.2.2 Метод

Метод секанса определяется рекуррентным соотношением

$$x_n = x_{n-1} - f\left(x_{n-1}\right) \frac{x_{n-1} - x_{n-2}}{f\left(x_{n-1}\right) - f\left(x_{n-2}\right)} = \frac{x_{n-2} f\left(x_{n-1}\right) - x_{n-1} f\left(x_{n-2}\right)}{f\left(x_{n-1}\right) - f\left(x_{n-2}\right)}. \tag{1.2}$$

Как видно из рекуррентного соотношения, метод секущей требует двух начальных значений x_0 и x_1, которые в идеале следует выбирать так, чтобы они лежали близко к корню.

1.2.3 Вывод метода

Начиная с начальных значений x_0 и x_1, мы строим линию через точки $(x0, f(x0))$ and $(x1, f(x1))$, как показано на рисунке выше. В форме наклон-пересечение уравнение этой прямой имеет вид

$$y = \frac{f\left(x_1\right) - f\left(x_0\right)}{x_1 - x_0} \left(x - x_1\right) + f\left(x_1\right). \tag{1.3}$$

Корень этой линейной функции, то есть значение x такое, что $y = 0$, равно

$$x = x_1 - f\left(x_1\right) \frac{x_1 - x_0}{f\left(x_1\right) - f\left(x_0\right)}. \tag{1.4}$$

Затем мы используем это новое значение x как x_2 и повторяем процесс, используя x_1 и x_2 вместо x_0 и x_1. Мы продолжаем этот процесс, решая для x_3, x_4 и т. Д., Пока не достигнем достаточно высокого уровня точности (достаточно малая разница между x_n и $x_n - 1$):

$$\begin{aligned} x_2 &= x_1 - f\left(x_1\right) \frac{x_1 - x_0}{f\left(x_1\right) - f\left(x_0\right)} \\ x_3 &= x_2 - f\left(x_2\right) \frac{x_2 - x_1}{f\left(x_2\right) - f\left(x_1\right)} \\ &\vdots \\ x_n &= x_{n-1} - f\left(x_{n-1}\right) \frac{x_{n-1} - x_{n-2}}{f\left(x_{n-1}\right) - f\left(x_{n-2}\right)}. \end{aligned} \tag{1.5}$$

1.2.4 Использование

Представьте, что мы хотим минимизировать следующую функцию:

$$f(x) = \sin x, x \in [-1, 1] \tag{1.6}$$

Тогда код будет выглядеть так:

```
// example_root_secant.cpp

#include <iostream>
#include "../src/numerary.hpp" // Numerary library

using namespace std;
using namespace numerary;

/* Function to found the root */
double f(double x) {
    return sin(x);
}

/* The main function */
int main() {

    const double eps = 1.e-9; // eps value for method (1.e-9 by default)
    double a = -1; // "a" value of segment [a, b]
    double b = 1; // "b" value of segment [a, b]
    double root;
    short int is_found;

    is_found = Numerary::root(f, a, b, &root, "secant", eps);

    if (is_found == 1) {
        cout << "Root is in x = " << root << endl;
    } else {
        cout << "Method not allowed!" << endl;
    }

    return 0;
}
```

1.3 Интегральное приближение - правило Симпсона

1.3.1 Определение

Предположим, что $f(x)$ определена на интервале $[a, b]$. Тогда правило Симпсона на всем интервале приближает определенный интеграл от $f(x)$ на интервале по формуле

$$\int_a^b f(x)dx \approx \frac{b-a}{6}\left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right) \tag{1.7}$$

Идея состоит в том, что если $f(x) = 1, x, x^2$, эта формула является точным равенством. Итак, правило Симпсона дает правильный интеграл от любой квадратичной функции. В общем, правило Симпсона приближает $f(x)$ параболой через точки на графике $f(x)$ с x-координатами $a, \frac{a+b}{2}, b$.

Правило Симпсона обычно применяется путем разбиения интервала на N подинтервалов одинакового размера, где N - четное число, и аппроксимации интеграла по каждой паре смежных подынтервалов с использованием приведенной выше оценки.

То есть пусть $x_0 = a, x_1 = a + \frac{b-a}{N}, x_2 = a + 2\frac{b-x}{N}, \ldots, x_{N-1} = a + (N-1)\frac{b-a}{N}, x_N = b$. Тогда

$$\int_a^{x_2} f(x)dx \approx \frac{b-a}{3N}\left(f(a) + 4f\left(x_1\right) + f\left(x_2\right)\right) \tag{1.8}$$

$$\int_{x_2}^{x_4} f(x)dx \approx \frac{b-a}{3N}\left(f(x_2) + 4f\left(x_3\right) + f\left(x_4\right)\right) \tag{1.9}$$

и так далее. Сложение дает

$$\int_a^b f(x)dx \approx \frac{b-a}{3N}\left(f(a) + 4f\left(x_1\right) + 2f\left(x_2\right) + 4f\left(x_3\right) + 2f\left(x_4\right) + \cdots + 4f\left(x_{N-1}\right) + f(b)\right). \tag{1.10}$$

1.3.2 Использование

Представьте, что мы хотим интегрировать следующее выражение:

$$\int_0^1 (5x^3 + 2\cos x)dx. \tag{1.11}$$

Тогда код будет выглядеть так:

```
// example_integral_simpson.cpp

#include <iostream>
#include "../src/numerary.hpp"

using namespace std;
using namespace numerary;

/* Function to be integrated */
double f(double x) {
    return 5*pow(x, 3) + 2*cos(x);
}

/* The main function */
int main() {

    const double from = 0; // Lower bound of integral
    const double to = 1; // Upper bound of integral
    const string method = "simpson"; // Numerical method we will use for integration ("simpson" by␣
↪default)
    const double eps = 1.e-9; // eps value for integration (1.e-9 by default)

    double *I = Numerary::integrate(f, from, to, method, eps);

    cout << "ans = " << I[0] << endl; // Value of calculated integral
    cout << "err = " << I[1] << endl; // Error of calculated integral value

    return 0;
}
```

1.4 Метод деления пополам

1.4.1 Использование

Представьте, что мы хотим минимизировать следующую функцию:

$$f(x) = 2x^2 - 5x + 3, x \in [0, 2] \tag{1.12}$$

Тогда код будет выглядеть так:

```
// example_minimum_bisection.cpp

#include <iostream>
#include "../src/numerary.hpp" // Numerary library

using namespace std;
using namespace numerary;

/* Function to found local minimum */
double f(double x) {
    return 2*x*x - 5*x + 3;
}

/* The main function */
int main() {

    const double eps = 1.e-9; // eps value for method (1.e-9 by default)
    double a = 0; // "a" value of segment [a, b]
    double b = 2; // "b" value of segment [a, b]
    double minimum;
    short int is_found;

    is_found = Numerary::minimum(f, a, b, &minimum, "bisection", eps);

    if (is_found == 1) {
        cout << "Minimum is in x = " << minimum << endl;
    } else {
        cout << "Method not allowed!" << endl;
    }

    return 0;
}
```

1.5 Метод золотого сечения

1.5.1 Использование

Представьте, что мы хотим минимизировать следующую функцию:

$$f(x) = x^2 + \sin(3x), x \in [-1, 1] \tag{1.13}$$

Тогда код будет выглядеть так:

```
// example_minimum_golden_ratio.cpp

#include <iostream>
#include "../src/numerary.hpp" // Numerary library

using namespace std;
using namespace numerary;

/* Function to found local minimum */
double f(double x) {
    return x*x + sin(3*x);
}

/* The main function */
int main() {

    const double eps = 1.e-9; // eps value for method (1.e-9 by default)
    double a = -1; // "a" value of segment [a, b]
    double b = 1; // "b" value of segment [a, b]
    double minimum;
    short int is_found;

    is_found = Numerary::minimum(f, a, b, &minimum, "golden_ratio", eps);

    if (is_found == 1) {
        cout << "Minimum is in x = " << minimum << endl;
    } else {
        cout << "Method not allowed!" << endl;
    }

    return 0;
}
```

1.6 Метод деления пополам

1.6.1 Использование

Представьте, что мы хотим максимизировать следующую функцию:

$$f(x) = \sin x, x \in [-2, 2] \tag{1.14}$$

Тогда код будет выглядеть так:

```
// example_maximum_bisection.cpp

#include <iostream>
#include "../src/numerary.hpp" // Numerary library

using namespace std;
using namespace numerary;

/* Function to found local maximum */
double f(double x) {
    return sin(x);
```

(continues on next page)

(продолжение с предыдущей страницы)

```
}

/* The main function */
int main() {

    const double eps = 1.e-9; // eps value for method (1.e-9 by default)
    double a = -2; // "a" value of segment [a, b]
    double b = 2; // "b" value of segment [a, b]
    double maximum;
    short int is_found;

    is_found = Numerary::maximum(f, a, b, &maximum, "bisection", eps);

    if (is_found == 1) {
        cout << "Maximum is in x = " << maximum << endl;
    } else {
        cout << "Method not allowed!" << endl;
    }

    return 0;
}
```

1.7 Метод золотого сечения

1.7.1 Использование

Представьте, что мы хотим максимизировать следующую функцию:

$$f(x) = \frac{1}{1+x^2}, x \in [-2, 2] \tag{1.15}$$

Тогда код будет выглядеть так:

```
// example_maximum_golden_ratio.cpp

#include <iostream>
#include "../src/numerary.hpp" // Numerary library

using namespace std;
using namespace numerary;

/* Function to found local maximum */
double f(double x) {
    return 1.0 / (1.0 + x*x);
}

/* The main function */
int main() {

    const double eps = 1.e-9; // eps value for method (1.e-9 by default)
    double a = -2; // "a" value of segment [a, b]
    double b = 2; // "b" value of segment [a, b]
    double maximum;
    short int is_found;
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    is_found = Numerary::maximum(f, a, b, &maximum, "golden_ratio", eps);

    if (is_found == 1) {
        cout << "Maximum is in x = " << maximum << endl;
    } else {
        cout << "Method not allowed!" << endl;
    }

    return 0;
}
```

1.8 Метод исключения Гаусса

1.8.1 Алгоритм

Метод Гаусса - классический метод решения системы линейных алгебраических уравнений (СЛАУ). Рассмотрим систему линейных уравнений с действительными постоянными коэффициентами

$$\begin{cases} a_{11} \cdot x_1 + a_{12} \cdot x_2 + & \cdots & + a_{1n} \cdot x_n = y_1 \\ a_{21} \cdot x_1 + a_{22} \cdot x_2 + & \cdots & + a_{2n} \cdot x_n = y_2 \\ & \cdots & \\ a_{n1} \cdot x_1 + a_{n2} \cdot x_2 + & \cdots & + a_{nn} \cdot x_n = y_n \end{cases}$$

или в матричной форме

$$Ax = y,$$

где

$$A = \begin{pmatrix} a_{11} & \dots & a_{1n} \\ & \dots & \\ a_{n1} & \dots & a_{nn} \end{pmatrix}, \quad x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}, \quad y = \begin{pmatrix} y_1 \\ \vdots \\ y_m \end{pmatrix}.$$

Метод Гаусса решения системы линейных уравнений включает в себя 2 стадии:

- последовательное (прямое) исключение;
- обратная подстановка.

Последовательное исключение

Исключения Гаусса основаны на идее последовательного исключения переменных по одной до тех пор, пока не останется только одно уравнение с одной переменной в левой части. Затем это уравнение решается относительно единственной переменной. Таким образом, систему уравнений приводят к треугольной (ступенчатой) форме. Для этого среди элементов первого столбца матрицы выбирают ненулевой (а чаще максимальный) элемент и перемещают его на крайнее верхнее положение перестановкой строк. Затем нормируют все уравнения, разделив его на коэффициент ai1, где i– номер столбца.

$$\begin{cases} x_1 + \frac{a_{12}}{a_{11}} \cdot x_2 + & \cdots & + \frac{a_{1n}}{a_{11}} \cdot x_n = \frac{y_1}{a_{11}} \\ x_1 + \frac{a_{22}}{a_{21}} \cdot x_2 + & \cdots & + \frac{a_{2n}}{a_{21}} \cdot x_n = \frac{y_2}{a_{21}} \\ & \cdots & \\ x_1 + \frac{a_{n2}}{a_{n1}} \cdot x_2 + & \cdots & + \frac{a_{nn}}{a_{n1}} \cdot x_n = \frac{y_n}{a_{n1}} \end{cases}$$

Затем вычитают получившуюся после перестановки первую строку из остальных строк:

$$\begin{cases} x_1 & + & \frac{a_{12}}{a_{11}} \cdot x_2 & + \cdots + & \frac{a_{1n}}{a_{11}} \cdot x_n & = & \frac{y_1}{a_{11}} \\ 0 & + & \left(\frac{a_{22}}{a_{21}} - \frac{a_{12}}{a_{11}}\right) \cdot x_2 & + \cdots + & \left(\frac{a_{2n}}{a_{21}} - \frac{a_{1n}}{a_{11}}\right) \cdot x_n & = & \left(\frac{y_2}{a_{21}} - \frac{y_1}{a_{11}}\right) \\ & & & \cdots & & & \\ 0 & + & \left(\frac{a_{n2}}{a_{n1}} - \frac{a_{12}}{a_{11}}\right) \cdot x_2 & + \cdots + & \left(\frac{a_{nn}}{a_{n1}} - \frac{a_{1n}}{a_{11}}\right) \cdot x_n & = & \left(\frac{y_n}{a_{n1}} - \frac{y_1}{a_{11}}\right) \end{cases}$$

Получают новую систему уравнений, в которой заменены соответствующие коэффициенты.

$$\begin{cases} x_1 & + & a'_{12} \cdot x_2 & + \cdots + & a'_{1n} \cdot x_n & = & y'_1 \\ 0 & + & a'_{22} \cdot x_2 & + \cdots + & a'_{2n} \cdot x_n & = & y'_2 \\ & & & \cdots & & & \\ 0 & + & a'_{n2} \cdot x_2 & + \cdots + & a'_{nn} \cdot x_n & = & y'_n \end{cases}$$

После того, как указанные преобразования были совершены, первую строку и первый столбец мысленно вычёркивают и продолжают указанный процесс для всех последующих уравнений пока не останется уравнение с одной неизвестной:

$$\begin{cases} x_1 & + & a'_{12} \cdot x_2 & + & a'_{13} \cdot x_3 & + \cdots + & a'_{1n} \cdot x_n & = & y'_1 \\ 0 & + & x_2 & + & a''_{23} \cdot x_3 & + \cdots + & a''_{2n} \cdot x_n & = & y''_2 \\ 0 & + & 0 & + & x_3 & + \cdots + & a'''_{3n} \cdot x_n & = & y'''_3 \\ & & & & & \cdots & & & \\ 0 & + & 0 & + & 0 & + \cdots + & x_n & = & y_n^{n\prime} \end{cases}$$

Обратная подстановка

Обратная подстановка предполагает подстановку полученного на предыдущем шаге значения переменной xn в предыдущие уравнения:

$$\begin{aligned} x_{n-1} &= y_{n-1}^{(n-1)'} - a_{(n-1)n}^{(n-1)\prime} \cdot x_n \\ x_{n-2} + a_{(n-2)(n-1)}^{(n-2)\prime} \cdot x_{n-1} &= y_{n-2}^{(n-2)'} - a_{(n-2)n}^{(n-2)\prime} \cdot x_n \\ & \cdots \\ x_2 + a''_{23} \cdot x_3 + \cdots + a''_{2(n-1)} \cdot x_{n-1} &= y''_2 - a''_{2n} \cdot x_n \\ x_1 + a'_{12} \cdot x_2 + a'_{13} \cdot x_3 + \cdots + a'_{1(n-1)} \cdot x_{n-1} &= y'_1 - a'_{1n} \cdot x_n \end{aligned}$$

Эта процедура повторяется для всех оставшихся решений:

$$\begin{aligned} x_{n-2} &= \left(y_{n-2}^{(n-2)'} - a_{(n-2)n}^{(n-2)'} \cdot x_n\right) - a_{(n-2)(n-1)}^{(n-2)\prime} \cdot x_{n-1} \\ & \cdots \\ x_2 + a''_{23} \cdot x_3 + \cdots &= (y''_2 - a''_{2n} \cdot x_n) - a''_{2(n-1)} \cdot x_{n-1} \\ x_1 + a'_{12} \cdot x_2 + a'_{13} \cdot x_3 + \cdots &= (y'_1 - a'_{1n} \cdot x_n) - a'_{1(n-1)} \cdot x_{n-1} \end{aligned}$$

1.8.2 Использование

Представьте, что мы хотим решить следующую линейную систему уравнений:

$$\begin{pmatrix} 2 & 1 \\ -1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 5 \\ 2 \end{pmatrix}.$$

Тогда код будет выглядеть так:

```cpp
// example_gauss_elimination.cpp

#include <iostream>
#include "../src/numerary.hpp" // Numerary library

using namespace std;
using namespace numerary;

/* The main function */
int main() {

    double **a = new double*[2];
    double *y = new double[2];
    double *x = new double[2];
    short int is_solved;

    for (int i = 0; i < 2; i ++)
        a[i] = new double[2];


    // Initialize matrix A
    a[0][0] = 2;
    a[0][1] = 1;

    a[1][0] = -1;
    a[1][1] = 1;

    // Initialize vector y
    y[0] = 5;
    y[1] = 2;

    is_solved = Numerary::linear_systems_of_equations(a, y, x, 2, "gauss");

    if (is_solved == 1) {
        cout << "System solved!" << endl;
        cout << "x = (" << x[0] << ", " << x[1] << ")" << endl;
    } else {
        cout << "Method is not allowed!";
    }

    for (int i = 0; i < 2; i++) delete[] a[i];

    delete[] a;
    delete[] x;
    delete[] y;

    return 0;
}
```

1.9 Метод Ньютона

1.9.1 Обзор

Метод Ньютона - один из самых популярных численных методов, и Бёрден и Фейрес даже называют его самым мощным методом, который используется для решения уравнения $f(x) = 0$. Этот метод

основан на разложении в ряд Тейлора функции $f(x)$ относительно точки x_1:

$$f(x) = f(x_1) + (x - x_1) f'(x_1) + \frac{1}{2!}(x - x_1)^2 f''(x_1) + \cdots \tag{1.16}$$

где f, а также его производные первого и второго порядка, f' и f' вычисляются в x_1. Если мы возьмем первые два члена разложения в ряд Тейлора, мы получим:

$$f(x) \approx f(x_1) + (x - x_1) f'(x_1). \tag{1.17}$$

Затем мы устанавливаем предыдущее выражение равным нулю (т.е. $f(x) = 0$), чтобы найти корень уравнения, которое дает нам:

$$f(x_1) + (x - x_1) f'(x_1) = 0. \tag{1.18}$$

Переставляя предыдущее выражение, мы получаем следующее приближение к корню, что дает нам:

$$x = x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} \tag{1.19}$$

Таким образом, обобщая предыдущее выражение, мы получаем итерационный метод Ньютона:

$$x_i = x_{i-1} - \frac{f(x_{i-1})}{f'(x_{i-1})}, i \in \mathbb{N} \tag{1.20}$$

где $x_i \to \bar{x}$ (при $i \to \infty$), а x - приближение к корню функции f (x).

Примечание: Поскольку итерации начинают иметь одинаковые повторяющиеся значения, т.е. как $x_i = x_{i+1} = \bar{x}$, это указывает на то, что $f(x)$ сходится к $\bar{x}$. Таким образом, x_i - это корень функции $f(x)$.

1.9.2 Метод

Шаг 1:

Пусть $\mathbf{x}^{(0)} = \left(x_1^{(0)}, x_2^{(0)}, \ldots, x_n^{(0)}\right)$ - заданный начальный вектор.

Шаг 2:

Вычислите $J\left(\mathbf{x}^{(0)}\right)$ и $\mathbf{F}\left(\mathbf{x}^{(0)}\right)$.

Шаг 3:

Теперь нам нужно вычислить вектор $\mathbf{y}^{(0)}$, где

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \tag{1.21}$$

Чтобы найти $\mathbf{y}^{(0)}$, мы решаем линейную систему :math:‘Jleft(mathbf{x}^{(0)}right) mathbf{y}^{(0)}=-mathbf{F}left(mathbf{x}^{(0)}right)‘используя метод исключения Гаусса.

Примечание: преобразовав систему на *шаге 3*, мы получим $\mathbf{y}^{(0)} = -J\left(\mathbf{x}^{(0)}\right)^{-1} \mathbf{F}\left(\mathbf{x}^{(0)}\right)$. Значение этого состоит в том, что, поскольку math:*mathbf{y}^{(0)}=-Jleft(mathbf{x}^{(0)}right)^{-1} mathbf{F}left(mathbf{x}^{(0)}right)*, мы можем заменить $-J\left(\mathbf{x}^{(0)}\right)^{-1} \mathbf{F}\left(\mathbf{x}^{(0)}\right)$ в наша итерационная формула с $\mathbf{y}^{(0)}$. Этот результат даст, что

$$\mathbf{x}^{(k)} = \mathbf{x}^{(k-1)} - J\left(\mathbf{x}^{(k-1)}\right)^{-1} \mathbf{F}\left(\mathbf{x}^{(k-1)}\right) = \mathbf{x}^{(k-1)} - \mathbf{y}^{(k-1)} \tag{1.22}$$

Шаг 4:

Как только $\mathbf{y}^{(0)}$ найден, мы можем приступить к завершению первой итерации, решив для $\mathbf{x}^{(1)}$. Таким образом, используя результат *шага 3*, мы имеем

$$\mathbf{x}^{(1)} = \mathbf{x}^{(0)} + \mathbf{y}^{(0)} = \begin{bmatrix} x_1^{(0)} \\ x_2^{(0)} \\ \vdots \\ x_n^{(0)} \end{bmatrix} + \begin{bmatrix} y_1^{(0)} \\ y_2^{(0)} \\ \vdots \\ y_n^{(0)} \end{bmatrix} \tag{1.23}$$

Шаг 5:

После того, как мы вычислили $\mathbf{x}^{(1)}$, мы повторяем процесс снова, пока $\mathbf{x}^{(k)}$ не сходится к $\bar{x}$. Это указывает на то, что мы достигли решения $\mathbf{F}(\mathbf{x}) = \mathbf{0}$, где $\bar{x}$ - решение системы.

Примечание: когда набор векторов сходится, норма $\left\|\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\right\| = 0$. Это означает, что

$$\left\|\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\right\| = \sqrt{\left(x_1^{(k)} - x_1^{(k-1)}\right)^2 + \cdots + \left(x_n^{(k)} - x_n^{(k-1)}\right)^2} = 0 \tag{1.24}$$

1.9.3 Использование

представьте, что мы хотим решить следующую нелинейную систему уравнений:

$$\begin{cases} f(x,y) = x^2 + y^2 - 5 \\ g(x,y) = y - 3x + 5 \end{cases} \tag{1.25}$$

тогда код будет выглядеть так:

```
// example_newton.cpp

#include <iostream>
#include "../src/numerary.hpp" // Numerary library

using namespace std;
using namespace numerary;

/* System to solve */
void system(double *x, double *fv, int n) {
    fv[0] = x[0]*x[0] + x[1]*x[1] - 5;
    fv[1] = x[1] - 3*x[0] + 5;
}

/* The main function */
int main() {

    const int maxiter = 100; // Maximum interations for set method (100 by default)
    const double eps = 1.e-7; // eps value for method (1.e-7 by default)
    double *x = new double[2], *fv = new double[2];
    short int is_solved;

    // Initial point
    x[0] = 1; x[1] = 2;
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    is_solved = Numerary::nonlinear_systems_of_equations(system, x, fv, 2, "newton", eps, maxiter);

    if (is_solved == 1) {
        cout << "Solved!" << endl;
        cout << "x = " << x[0] << endl;
        cout << "y = " << x[1] << endl;
    } else {
        cout << "Method not allowed!";
    }

    delete[] x;
    delete[] fv;

    return 0;
}
```

1.10 Метод Дорманда-Принса

1.10.1 Определение

Одноэтапный расчет в методе Дорманда-Принса выполняется следующим образом.

$$
\begin{aligned}
k_1 &= hf\left(t_k, y_k\right), \\
k_2 &= hf\left(t_k + \frac{1}{5}h, y_k + \frac{1}{5}k_1\right), \\
k_3 &= hf\left(t_k + \frac{3}{10}h, y_k + \frac{3}{40}k_1 + \frac{9}{40}k_2\right), \\
k_4 &= hf\left(t_k + \frac{4}{5}h, y_k + \frac{44}{45}k_1 - \frac{56}{15}k_2 + \frac{32}{9}k_3\right), \\
k_5 &= hf\left(t_k + \frac{8}{9}h, y_k + \frac{19372}{6561}k_1 - \frac{25360}{2187}k_2 + \frac{64448}{6561}k_3 - \frac{212}{729}k_4\right), \\
k_6 &= hf\left(t_k + h, y_k + \frac{9017}{3168}k_1 - \frac{355}{33}k_2 - \frac{46732}{5247}k_3 + \frac{49}{176}k_4 - \frac{5103}{18656}k_5\right), \\
k_7 &= hf\left(t_k + h, y_k + \frac{35}{384}k_1 + \frac{500}{1113}k_3 + \frac{125}{192}k_4 - \frac{2187}{6784}k_5 + \frac{11}{84}k_6\right).
\end{aligned}
\tag{1.26}
$$

Тогда значение следующего шага y_{k+1} вычисляется как

$$
y_{k+1} = y_k + \frac{35}{384}k_1 + \frac{500}{1113}k_3 + \frac{125}{192}k_4 - \frac{2187}{6784}k_5 + \frac{11}{84}k_6. \tag{1.27}
$$

Это вычисление методом Рунге-Кутты порядка 4. Мы должны знать, что мы не используем k_2, хотя оно используется для вычисления k_3 и так далее.

Далее мы вычислим значение следующего шага z_{k+1} методом Рунге-Кутты порядка 5 как

$$
z_{k+1} = y_k + \frac{5179}{57600}k_1 + \frac{7571}{16695}k_3 + \frac{393}{640}k_4 - \frac{92097}{339200}k_5 + \frac{187}{2100}k_6 + \frac{1}{40}k_7 \tag{1.28}
$$

Рассчитываем разницу двух следующих значений $|z_{k+1} - y_{k+1}|$.

$$
|z_{k+1} - y_{k+1}| = \left|\frac{71}{57600}k_1 - \frac{71}{16695}k_3 + \frac{71}{1920}k_4 - \frac{17253}{339200}k_5 + \frac{22}{525}k_6 - \frac{1}{40}k_7\right| \tag{1.29}
$$

Это считается ошибкой в y_{k+1}. Мы вычисляем оптимальный интервал времени h_{opt} как,

$$s = \left(\frac{\varepsilon h}{2 \left| z_{k+1} - y_{k+1} \right|} \right)^{\frac{1}{5}}, h_{opt} = sh, \tag{1.30}$$

где h в правой части - старый интервал времени. В практическом программировании этот новый h_{opt} будет использоваться на следующем этапе расчета, хотя автор считает, что его также следует использовать в текущих расчетах, когда он очень мал, например, наполовину или меньше.

1.10.2 Использование

Представьте, что мы хотим минимизировать следующее дифференциальное уравнение:

$$y' = 3\frac{y}{x} + x^3 + x, y(1) = 3 \tag{1.31}$$

Тогда код будет выглядеть так:

```
// example_dorpi.cpp

#include <iostream>
#include "../src/numerary.hpp" // Numerary library

using namespace std;
using namespace numerary;

/* Equation to solve */
double equation(double x, double y) {
    return 3.0*y/x + x*x*x + x;
}

/* The main function */
int main() {

    double *y = new double[2];
    double x0, x, h;
    short int is_solved;

    // Initial point
    x0 = 1; y[0] = 3;

    // Point where we want calculate y(x)
    x = 2.0;

    // Step size
    h = 0.01;

    is_solved = Numerary::ordinary_differential_equations(equation, y, x0, h, x, "dorpi_4_5");

    if (is_solved == 0) {
        cout << "Solved!" << endl;
        cout << "y(" << x << ") = " << y[1] << endl;
    } else {
        cout << "Method not allowed!" << endl;
    }

    delete[] y;
```

(continues on next page)

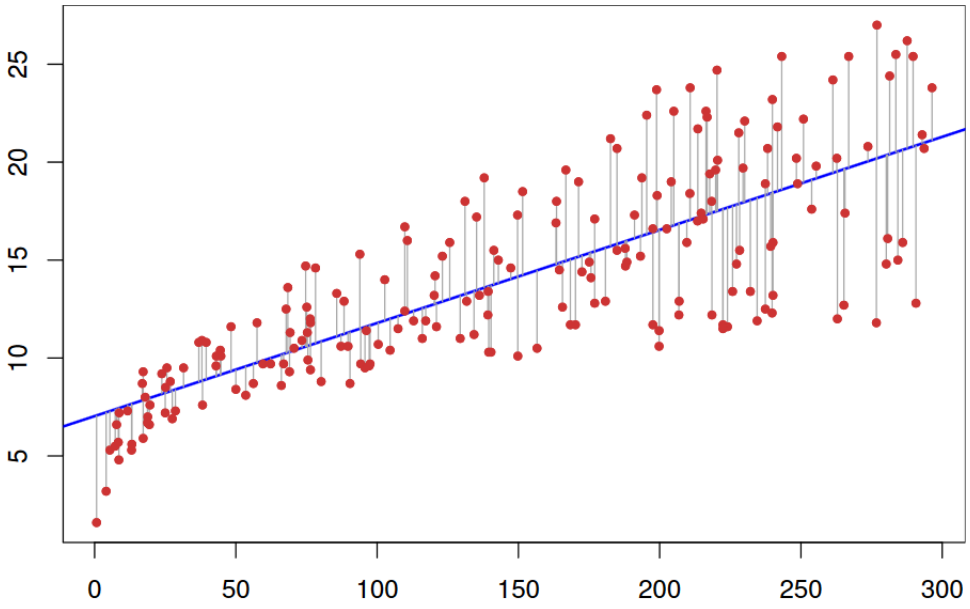
(продолжение с предыдущей страницы)

```
    return 0;
}
```

1.11 Линейная регрессия

1.11.1 Введение

В статистике линейная регрессия - это линейный подход к моделированию взаимосвязи между скалярным откликом (или зависимой переменной) и одной или несколькими независимыми переменными (или независимыми переменными). Случай одной независимой переменной называется простой линейной регрессией. Для нескольких независимых переменных процесс называется множественной линейной регрессией. Этот термин отличается от многомерной линейной регрессии, в которой прогнозируются несколько коррелированных зависимых переменных, а не одна скалярная переменная.



1.11.2 Модель простой линейной регрессии

Простейшая детерминированная математическая связь между двумя переменными x и y - это линейная зависимость: $y = \beta_0 + \beta_1 x$.

Цель этого раздела - разработать эквивалентную *линейную вероятностную модель*.

Если две (случайные) переменные вероятностно связаны, то для фиксированного значения x существует неопределенность в значении второй переменной.

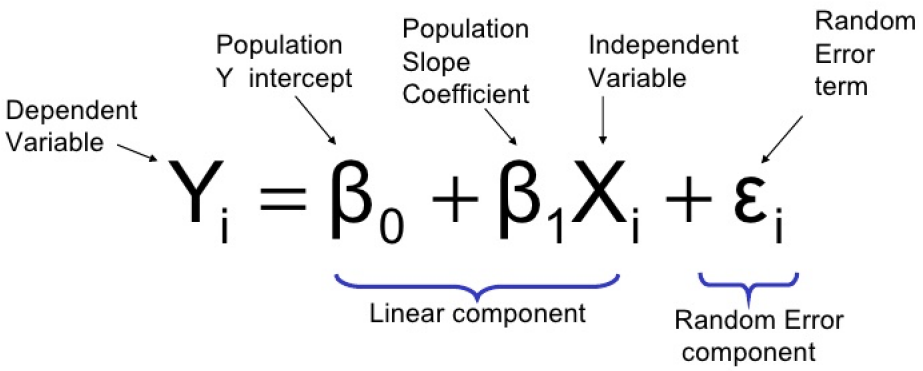
Итак, мы предполагаем $Y = \beta_0 + \beta_1 x + \varepsilon$, где ε - случайная величина.

Две переменные связаны линейно «в среднем», если для фиксированного x фактическое значение Y отличается от его ожидаемого значения на случайную величину (т.е. имеется случайная ошибка).

1.11.3 Линейная вероятностная модель

Определение: (Модель простой линейной регрессии)

Существуют параметры β_0, β_1 и σ^2, такие, что для любого фиксированного значения независимой переменной x зависимая переменная является случайной величиной, связанной с x через уравнение модели



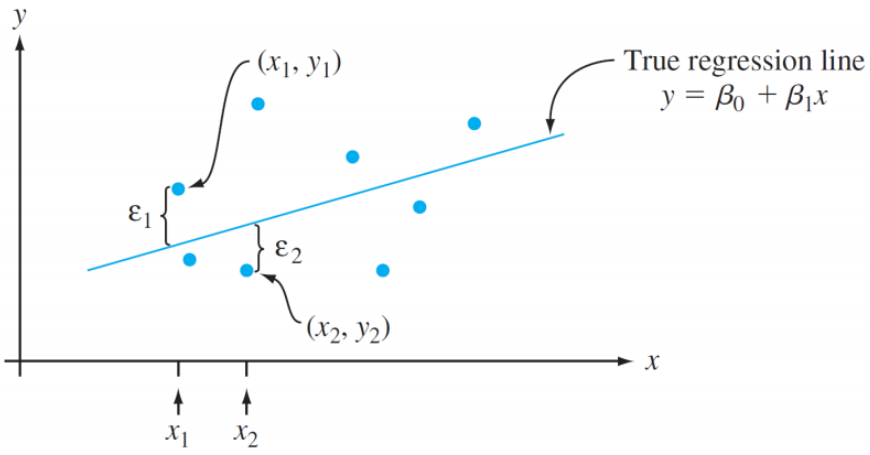
Величина ε в уравнении модели является «ошибкой» - случайной величиной, симметрично распределенной с

$$E(\varepsilon) = 0 \text{ and } V(\varepsilon) = \sigma_\varepsilon^2 = \sigma^2 \tag{1.32}$$

(пока никаких предположений о распределении ε не делалось)

- $\boldsymbol{X}$: независимая, предикторная или объясняющая переменная (обычно известная).
- $\boldsymbol{Y}$: зависимая переменная или переменная ответа. При фиксированном x, Y будет случайной величиной.
- $\boldsymbol{\varepsilon}$: случайное отклонение или случайная ошибка. При фиксированном x, ε будет случайной величиной.
- $\boldsymbol{\beta_0}$: среднее значение Y, когда x равно нулю (пересечение истинной линии регрессии)
- ожидаемое (среднее) изменение Y, связанное с увеличением на 1 единицу значения x. (наклон истинной линии регрессии)

Точки $(x_1, y_1), \ldots, (x_n, y_n)$, полученные в результате n независимых наблюдений, затем будут разбросаны вокруг истинной линии регрессии:



1.11.4 Оценка параметров модели

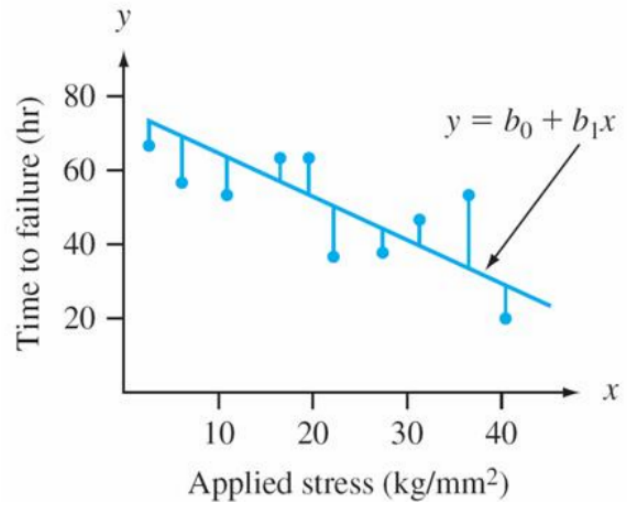
Значения β_0, β_1 и σ почти никогда не будут известны исследователю.

Вместо этого выборочные данные состоят из n наблюдаемых пар $(x_1, y_1), \ldots, (x_n, y_n)$, по которым можно оценить параметры модели и саму истинную линию регрессии.

Предполагается, что данные (пары) были получены независимо друг от друга.

Линия «наилучшего соответствия» основана на принципе **наименьших квадратов**, который восходит к немецкому математику **Гауссу** (1777–1855):

Линия обеспечивает наилучшее соответствие данным, если сумма квадратов вертикальных расстояний (отклонений) от наблюдаемых точек до этой линии настолько мала, насколько это возможно.



Сумма квадратов вертикальных отклонений от точек $(x_1, y_1), \ldots, (x_n, y_n)$

$$f(b_0, b_1) = \sum_{i=1}^{n} [y_i - (b_0 + b_1 x_i)]^2 \tag{1.33}$$

Точечные оценки β_0 и β_1, обозначенные как и, называются оценками наименьших квадратов - это те значения, которые минимизируют $f(b_0, b_1)$.

Подгоняемая **линия регрессии** или линия **наименьших квадратов** - это линия, уравнение которой имеет вид $y = \hat{\beta}_0 + \hat{\beta}_1 x$.

Минимизирующие значения b_0 и b_1 находятся путем взятия частных производных от $f(b_0, b_1)$ как по b_0, так и по b_1, приравнивания их обоих к нулю [аналогично $f'(b) = 0$ в одномерном исчислении] и решения уравнения

$$\begin{aligned} \frac{\partial f(b_0,b_1)}{\partial b_0} &= \sum 2\left(y_i - b_0 - b_1 x_i\right)(-1) = 0 \\ \frac{\partial f(b_0,b_1)}{\partial b_1} &= \sum 2\left(y_i - b_0 - b_1 x_i\right)(-x_i) = 0 \end{aligned} \tag{1.34}$$

Оценка методом наименьших квадратов коэффициента наклона β_1 истинной линии регрессии равна

$$b_1 = \hat{\beta}_1 = \frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{\sum (x_i - \bar{x})^2} = \frac{S_{xy}}{S_{xx}} \tag{1.35}$$

Краткие формулы для числителя и знаменателя $\hat{\beta}_1$:

$$S_{xy} = \sum x_i y_i - \frac{(\sum x_i)(\sum y_i)}{n} \quad \text{and} \quad S_{xx} = \sum x_i^2 - \frac{(\sum x_i)^2}{n} \tag{1.36}$$

Оценка методом наименьших квадратов точки пересечения b_0 истинной линии регрессии равна

$$b_0 = \hat{\beta}_0 = \frac{\sum y_i - \hat{\beta}_1 \sum x_i}{n} = \bar{y} - \hat{\beta}_1 \bar{x} \tag{1.37}$$

1.11.5 Использование

Представьте, что у нас есть следующие точки, и мы хотим построить линейную регрессионную модель:

X	Y
1.0	1.0
2.0	2.0
3.0	1.3
4.0	3.75
5.0	2.25

Тогда код будет выглядеть так:

```
// example_linear_regression.cpp

#include <iostream>
#include "../src/numerary.hpp" // Numerary library

using namespace std;
using namespace numerary;

/* The main function */
int main() {

    const int N = 5; // Number of points
    double *X = new double[N], *Y = new double[N], *predicted_kc = new double[2];

    X[0] = 1.0; Y[0] = 1.0;
    X[1] = 2.0; Y[1] = 2.0;
    X[2] = 3.0; Y[2] = 1.3;
    X[3] = 4.0; Y[3] = 3.75;
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    X[4] = 5.0; Y[4] = 2.25;


    // Get predicted linear regression line
    predicted_kc = Numerary::linear_regression(X, Y, N);

    // Equation of regression line
    cout << "y = " << predicted_kc[0] << "*x + " << predicted_kc[1] << endl;

    // Reallocate memory
    delete[] X;
    delete[] Y;
    delete[] predicted_kc;

    return 0;
}
```

1.12 Интерполяция Лагранжа

1.12.1 Полином Лагранжа

В численном анализе полиномы Лагранжа используются для полиномиальной интерполяции. Для данного набора точек (x_j, y_j), в котором нет двух одинаковых значений x_j, многочлен Лагранжа является многочленом самой низкой степени, который принимает для каждого значения x_j соответствующее значение x_j, так что функции совпадают в каждой точке.

1.12.2 Определение

Для набора из $k+1$ точек данных $(x_0, y_0), \ldots, (x_j, y_j), \ldots, (x_k, y_k)$, где нет двух одинаковых x_j, интерполяционный многочлен в форме Лагранжа представляет собой линейную комбинацию $L(x) := \sum_{j=0}^{k} y_j \ell_j(x)$, базисных полиномов Лагранжа

$$\ell_j(x) := \prod_{\substack{0 \le m \le k \\ m \ne j}} \frac{x - x_m}{x_j - x_m} = \frac{(x - x_0)}{(x_j - x_0)} \cdots \frac{(x - x_{j-1})}{(x_j - x_{j-1})} \frac{(x - x_{j+1})}{(x_j - x_{j+1})} \cdots \frac{(x - x_k)}{(x_j - x_k)} \tag{1.38}$$

где $0 \le j \le k$. Обратите внимание на то, что при исходном предположении, что нет двух одинаковых x_j, $x_j - x_m \ne 0$, поэтому это выражение всегда хорошо определено. Причина, по которой пары $x_i = x_j$ с $y_i \ne y_j$ недопустимы, заключается в том, что не существует интерполяционной функции L, такой что $y_i = L(x_i)$; функция может получить только одно значение для каждого аргумента x_i. С другой стороны, если также $y_i = y_j$, то эти две точки фактически будут одной точкой.

Для всех $i \ne j$, $\ell(x)$ включает член $(x - x_i)$ в числителе, поэтому все произведение будет равно нулю при $x = x_i$

$$\ell_{j \ne i}(x_i) = \prod_{m \ne j} \frac{x_i - x_m}{x_j - x_m} = \frac{(x_i - x_0)}{(x_j - x_0)} \cdots \frac{(x_i - x_i)}{(x_j - x_i)} \cdots \frac{(x_i - x_k)}{(x_j - x_k)} = 0 \tag{1.39}$$

С другой стороны,

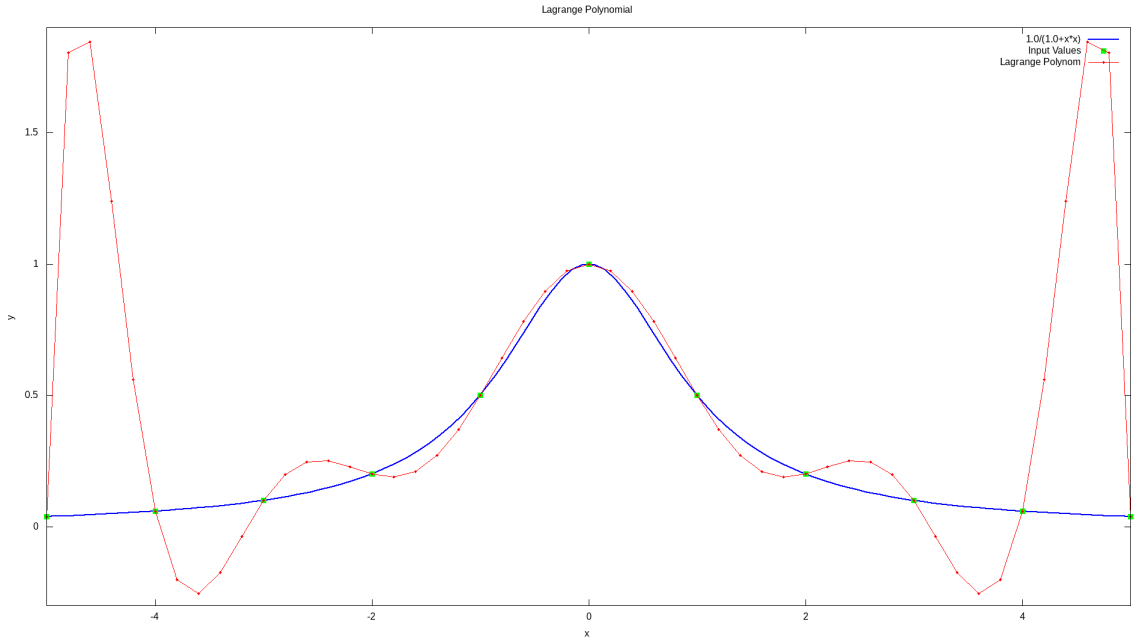
$$\ell_i(x_i) := \prod_{m \ne i} \frac{x_i - x_m}{x_i - x_m} = 1 \tag{1.40}$$

Другими словами, все базисные полиномы равны нулю при $x = x_i$, за исключением $\ell_i(x)$, для которого выполняется $\ell_i(x_i) = 1$, поскольку в нем отсутствует член $(x - x_i)$.

Отсюда следует, что $\ell_i(x_i) = y_i$, поэтому в каждой точке x_i $L(x_i) = y_i + 0 + 0 + \cdots + 0 = y_i$, показывая, что L точно интерполирует функцию.

1.12.3 Пример Рунге

Функция $f(x) = \frac{1}{1+x^2}$ не может быть точно интерполирована на $[5, 5]$ с использованием многочлена десятой степени (пунктирная кривая) с равноотстоящими точками интерполяции. Этот пример, иллюстрирующий сложность, которую обычно можно ожидать от полиномиальной интерполяции высокой степени с равноотстоящими точками, известен как *пример Рунге*.



1.12.4 Использование

Представьте, что у нас есть следующие точки, и мы хотим построить многочлен Лагранжа из этих точек:

X	Y
2.0	10.0
3.0	15.0
5.0	25.0
8.0	40.0
12.0	60.0

Тогда код будет выглядеть так:

```
// example_lagrange_polynomial.cpp

#include <iostream>
#include "../src/numerary.hpp" // Numerary library

using namespace std;
```

(continues on next page)

(продолжение с предыдущей страницы)

```
using namespace numerary;

/* The main function */
int main() {

    const int N = 5;
    double *X = new double[N], *Y = new double[N];
    double y, x;

    // Points to interpolate
    X[0] = 2.0; Y[0] = 10.0;
    X[1] = 3.0; Y[1] = 15.0;
    X[2] = 5.0; Y[2] = 25.0;
    X[3] = 8.0; Y[3] = 40.0;
    X[4] = 12.0; Y[4] = 60.0;

    // Point where we want to get value of Lagrange Polynomial
    x = 7.0;

    y = Numerary::lagrange_polynomial(X, Y, x, N);

    cout << "y(" << x << ") = " << y << endl;

    delete[] X;
    delete[] Y;

    return 0;
}
```

1.13 Численное дифференцирование. Расчет первой производной.

1.13.1 Определение

По определению первая производная гладкой функции f вточке x вычисляется как

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}. \tag{1.41}$$

При вычислении первой производной функции $f(x)$ на компьютере мы заменяем бесконечно малую $h \to \infty$:

$$f'(x) = \frac{f(x+h) - f(x)}{h} + \mathrm{O}(h), \tag{1.42}$$

где $\mathrm{O}(h)$ - это ошибка вычисления производной, которая, естественно, зависит от h. Предыдущая формула называется разностной схемой для вычисления первой производной (точнее, правильной разностной схемой или просто правильной разностной). Точно так же, может быть, написана левая разностная схема.

Как определить $\mathrm{O}(h)$? Разложим функцию $f(x)$ в ряд Тейлора в точке $x + h$:

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(x) + \ldots, \tag{1.43}$$

откуда следует, что в первом порядке разложения

$$\mathrm{O}(h) = -\frac{h}{2} f''(x) + \ldots. \tag{1.44}$$

Выбирая очень маленький h, ошибки округления при вычислениях на компьютере могут быть сопоставимы с h или больше. Поэтому нас интересует алгоритм, дающий меньшее значение ошибки при том же значении h.

Такой улучшенный алгоритм может быть легко получен путем разложения функции $f(x)$ в ряд Тейлора в точках $x + h$ и $x - h$, а затем вычитания одного результата из другого, что дает

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + \mathrm{O}\left(h^2\right), \tag{1.45}$$

где погрешность вычисления первой производной

$$\mathrm{O}\left(h^2\right) = -\frac{h^2}{6} f'''(x) + \ldots.$$

Это центральная разностная схема (центральная разностная схема).

В принципе, можно пойти по пути повышения точности метода вычисления первой производной и далее. Например, рассматривая разложение функции f (x) в ряд Тейлора в точках $x + h$, $x + 2h$, $x - h$ и $x - 2h$, можно получить четырехточечную схему и т. д.

1.13.2 Использование

Представьте, что мы хотим найти производную следующей функции:

$$f(x) = \sin(x) \tag{1.46}$$

Тогда код будет выглядеть так:

```
// example_first_order_derivative_h.cpp

#include <iostream>
#include "../src/numerary.hpp" // Numerary library

using namespace std;
using namespace numerary;

/* Functiion to derive */
double f(double x) {
    return sin(x);
}

/* The main function */
int main() {

    const short int order = 1;
    double x, dy_dx;

    // Point where we want get value of derivative function
    x = M_PI;

    dy_dx = Numerary::differentiate(f, order, x);
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    cout << "dy/dx (" << x << ") = " << dy_dx << endl;

    return 0;
}
```

1.14 Неполная гамма-функция

1.14.1 Определение

$$\gamma(s, x) = \int_0^x t^{s-1} \mathrm{e}^{-t} \mathrm{d}t \tag{1.47}$$

1.14.2 Использование

Представьте, что мы хотим вычислить значение:

$$\gamma(2, 1) \tag{1.48}$$

Тогда код будет выглядеть так:

```
// example_incomplete_gamma_function.cpp

#include <iostream>
#include "../src/numerary.hpp" // Numerary library

using namespace std;
using namespace numerary;

/* The main function */
int main() {

    double value;

    value = Numerary::incgamma(2, 1);

    cout << "IncGamma(2, 1) = " << value << endl;

    return 0;
}
```