Numerary

Выпуск 0.0.1

сент. 20, 2020

Root finding

Numerary is Scientific Library that contains many numerical methods.

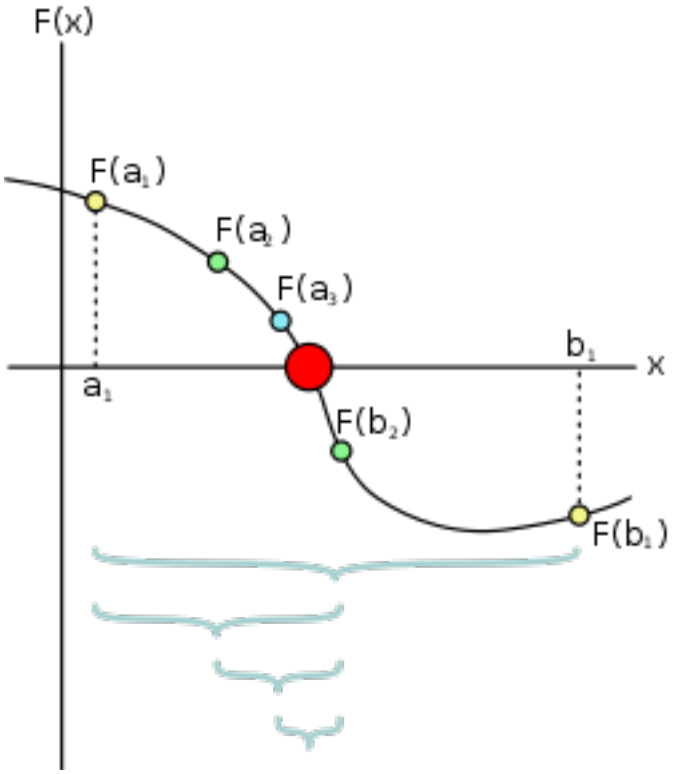
Глава 1

Contents

1.1 Bisection Method

1.1.1 Overview

In mathematics, the **bisection method** is a root-finding method that applies to any continuous functions for which one knows two values with opposite signs. The method consists of repeatedly bisecting the interval defined by these values and then selecting the subinterval in which the function changes sign, and therefore must contain a root. It is a very simple and robust method, but it is also relatively slow. Because of this, it is often used to obtain a rough approximation to a solution which is then used as a starting point for more rapidly converging methods. The method is also called the **interval halving method**, the **binary search method**, or the **dichotomy method**.



1.1.2 The Method

The method is applicable for numerically solving the equation $f(x) = 0$ for the real variable x, where f is a continuous function defined on an interval $[a, b]$ and where $f(a)$ and $f(b)$ have opposite signs. In this case a and b are said to bracket a root since, by the intermediate value theorem, the continuous function f must have at least one root in the interval (a, b).

At each step the method divides the interval in two by computing the midpoint $c = \frac{a+b}{2}$ of the interval and the value of the function $f(c)$ at that point. Unless c is itself a root (which is very unlikely, but possible) there are now only two possibilities: either $f(a)$ and $f(c)$ have opposite signs and bracket a root, or $f(c)$ and $f(b)$ have opposite signs and bracket a root. The method selects the subinterval that is guaranteed to be a bracket as the new interval to be used in the next step. In this way an interval that contains a zero of f is reduced in width by *50%* at each step. The process is continued until the interval is sufficiently small.

Explicitly, if $f(a)$ and $f(c)$ have opposite signs, then the method sets c as the new value for b, and if $f(b)$ and $f(c)$ have opposite signs then the method sets c as the new a. (If $f(c) = 0$ then c may be taken as the solution and the process stops.) In both cases, the new $f(a)$ and $f(b)$ have opposite signs, so the method is applicable to this smaller interval.

Iteration Tasks

1. Calculate c, the midpoint of the interval, $c = \frac{a+b}{2}$.
2. Calculate the function value at the midpoint, $f(c)$.
3. If convergence is satisfactory (that is, $c - a$ is sufficiently small, or $|f(c)|$ is sufficiently small), return c and stop iterating.

4. Examine the sign of $f(c)$ and replace either $(a, f(a))$ or $(b, f(b))$ with $(c, f(c))$ so that there is a zero crossing within the new interval.

1.1.3 Usage

Imagine that we want to find the root of the following function:

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

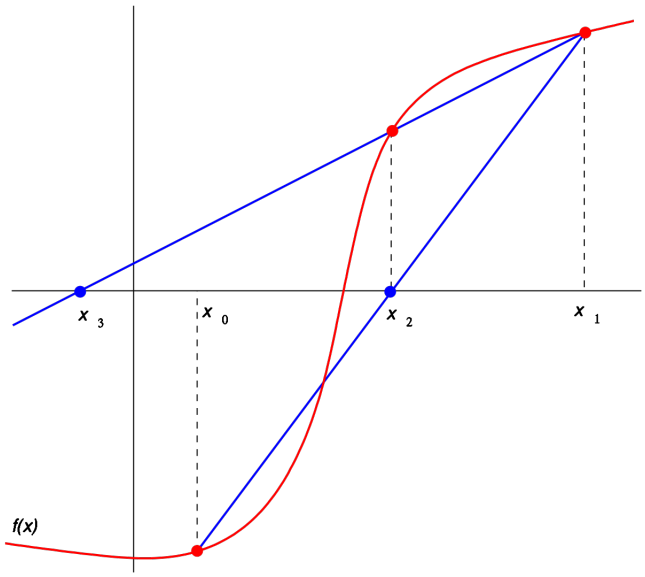
Then the code will look like this:

```
// 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 Secant Method

1.2.1 Overview

In numerical analysis, the **secant method** is a root-finding algorithm that uses a succession of roots of secant lines to better approximate a root of a function f. The secant method can be thought of as a finite-difference approximation of Newton's method. However, the method was developed independently of Newton's method and predates it by over 3000 years.



1.2.2 The Method

The secant method is defined by the recurrence relation

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

As can be seen from the recurrence relation, the secant method requires two initial values, x_0 and x_1, which should ideally be chosen to lie close to the root.

1.2.3 Derivation Of The Method

Starting with initial values x_0 and x_1, we construct a line through the points $(x0, f(x0))$ and $(x1, f(x1))$, as shown in the picture above. In slope–intercept form, the equation of this line is

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

The root of this linear function, that is the value of x such that $y = 0$ is

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

We then use this new value of x as x_2 and repeat the process, using x_1 and x_2 instead of x_0 and x_1. We continue this process, solving for x_3, x_4, etc., until we reach a sufficiently high level of precision (a sufficiently small difference between x_n and $x_n - 1$):

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

1.2.4 Usage

Imagine that we want to minimize the following function:

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

Then the code will look like this:

```
// 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 Integral Approximation - Simpson's Rule

1.3.1 Definition

Suppose $f(x)$ is defined on the interval $[a, b]$. Then Simpson's rule on the entire interval approximates the definite integral of $f(x)$ on the interval by the formula

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

The idea is that if $f(x) = 1, x, x^2$, this formula is an exact equality. So Simpson's rule gives the correct integral of any quadratic function. In general, Simpson's rule approximates $f(x)$ by a parabola through the points on the graph of $f(x)$ with x-coordinates $a, \frac{a+b}{2}, b$.

Simpson's rule is usually applied by breaking the interval into N equal-sized subintervals, where N is an even number, and approximating the integral over each pair of adjacent subintervals using the above estimate.

That is, let $x_0 = a, x_1 = a + \frac{b-a}{N}, x_2 = a + 2\frac{b-x}{N}, \ldots, x_{N-1} = a + (N-1)\frac{b-a}{N}, x_N = b$. Then

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

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

and so on. Adding these up gives

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

1.3.2 Usage

Imagine that we want to integrate the following expression:

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

Then the code will look like this:

```
// 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 Bisection Method

1.4.1 Usage

Imagine that we want to minimize the following function:

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

Then the code will look like this:

```
// 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 Golden Ratio Method

1.5.1 Usage

Imagine that we want to minimize the following function:

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

Then the code will look like this:

```cpp
// 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 Bisection Method

1.6.1 Usage

Imagine that we want to maximize the following function:

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

Then the code will look like this:

```cpp
// 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 Golden Ratio Method

1.7.1 Usage

Imagine that we want to maximize the following function:

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

Then the code will look like this:

```
// 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 Gauss Elimination Method

1.8.1 Algorithm

The Gauss method is a classical method for solving a system of linear algebraic equations (SLAE). Consider a system of linear equations with real constant coefficients

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

or in matrix form

$$Ax = y,$$

where

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

The Gauss method of solving a system of linear equations includes 2 stages:

- sequential (direct) exception;
- reverse substitution.

Sequential exception

Gauss exceptions are based on the idea of successive exceptions variables one at a time until only one equation remains with one variable on the left side. Then this equation is solved with respect to a single variable. Thus, the system of equations lead to a triangular (step) shape. For this, among the elements the first column of the matrix is selected nonzero (and most often maximum) element and move it to its highest position by rearranging lines. Then all the equations are normalized, dividing it by the coefficient ai1, where i is the column number.

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

Then the first line obtained after the permutation is subtracted from the remaining lines:

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

A new system of equations is obtained, in which the corresponding coefficients are replaced.

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

After the indicated transformations have been completed, the first row and the first column are mentally deleted and continue the specified process for all subsequent equations until an equation with one unknown:

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

Reverse substitution

Reverse substitution involves the substitution of the value of x_n obtained in the previous step into the previous equations:

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

This procedure is repeated for all remaining solutions:

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

1.8.2 Usage

Imagine that we want to solve following linear system of equations:

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

Then the code will look like this:

```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 Newton's Method

1.9.1 Overview

Newton's method is one of the most popular numerical methods, and is even referred by *Burden* and *Faires* as the most powerful method that is used to solve for the equation $f(x) = 0$. This method originates

from the *Taylor's series* expansion of the function $f(x)$ about the point $x1$:

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

where f, and its first and second order derivatives, f' and f'' are calculated at x_1. If we take the first two terms of the Taylor's series expansion we have:

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

We then set previous expression to zero (i.e $f(x) = 0$) to find the root of the equation which gives us:

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

Rearranging the previous expression we obtain the next approximation to the root, giving us:

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

Thus generalizing previous expression we obtain Newton's iterative method:

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

where $x_i \to \bar{x}$ (as $i \to \infty$), and x is the approximation to a root of the function $f(x)$.

Note: As the iterations begin to have the same repeated values i.e. as $x_i = x_{i+1} = \bar{x}$ this is an indication that $f(x)$ converges to $\bar{x}$. Thus x_i is the root of the function $f(x)$.

1.9.2 The Method

Step 1:

Let $\mathbf{x}^{(0)} = \left(x_1^{(0)}, x_2^{(0)}, \ldots, x_n^{(0)}\right)$ be a given initial vector.

Step 2:

Calculate $J\left(\mathbf{x}^{(0)}\right)$ and $\mathbf{F}\left(\mathbf{x}^{(0)}\right)$.

Step 3:

We now have to calculate the vector $\mathbf{y}^{(0)}$, where

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

In order to find $\mathbf{y}^{(0)}$, we solve the linear system $J\left(\mathbf{x}^{(0)}\right)\mathbf{y}^{(0)} = -\mathbf{F}\left(\mathbf{x}^{(0)}\right)$, using Gaussian Elimination.

Note: Rearranging the system in *Step 3*, we get that $\mathbf{y}^{(0)} = -J\left(\mathbf{x}^{(0)}\right)^{-1}\mathbf{F}\left(\mathbf{x}^{(0)}\right)$. The significance of this is that, since $\mathbf{y}^{(0)} = -J\left(\mathbf{x}^{(0)}\right)^{-1}\mathbf{F}\left(\mathbf{x}^{(0)}\right)$, we can replace $-J\left(\mathbf{x}^{(0)}\right)^{-1}\mathbf{F}\left(\mathbf{x}^{(0)}\right)$ in our iterative formula with $\mathbf{y}^{(0)}$. This result will yield that

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

Step 4:

Once $\mathbf{y}^{(0)}$ is found, we can now proceed to finish the first iteration by solving for $\mathbf{x}^{(1)}$. Thus using the result from *Step 3*, we have that

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

Step 5:

Once we have calculated $\mathbf{x}^{(1)}$, we repeat the process again, until $\mathbf{x}^{(k)}$ converges to $\bar{x}$. This indicates we have reached the solution to $\mathbf{F}(\mathbf{x}) = \mathbf{0}$, where $\bar{x}$ is the solution to the system.

Note: When a set of vectors converges, the norm $\left\|\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\right\| = 0$. This means that

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

1.9.3 Usage

imagine that we want to solve the following nonlinear system of equations:

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

then the code will look like this:

```
// 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;

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

(continues on next page)

(продолжение с предыдущей страницы)

```
    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 Dormand-Prince Method

1.10.1 Definition

The one step calculation in the Dormand-Prince method is done as the following.

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

Then the next step value y_{k+1} is calculated as

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

This is a calculation by Runge-Kutta method of order 4. We have to be aware that we do not use k_2, though it is used to calculate k_3 and so on.

Next, we will calculate the next step value z_{k+1} by Runge-Kutta method of order 5 as

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

We calculate the difference of the two next values $|z_{k+1} - y_{k+1}|$.

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

This is considered as the error in y_{k+1}. We calculate the optimal time interval h_{opt} as,

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

where h in the right side is the old time interval. In practical programming, this new h_{opt} will be used in the next step of the calculation, though the author thinks it should be also used in the present calculation when it is very small, half or smaller for example.

1.10.2 Usage

Imagine that we want to minimize the following differential equation:

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

Then the code will look like this:

```
// 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)

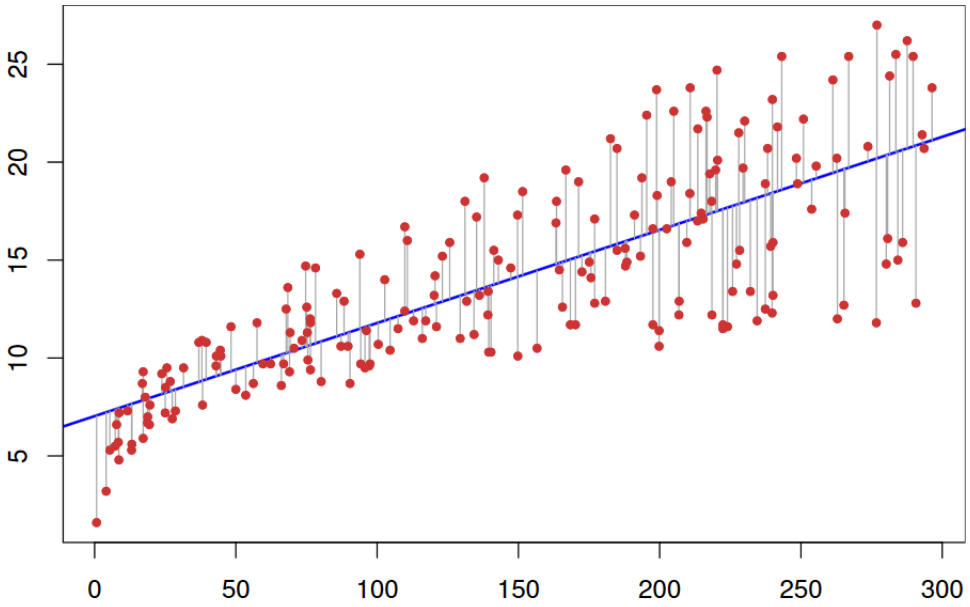
(продолжение с предыдущей страницы)

```
    return 0;
}
```

1.11 Linear Regression

1.11.1 Introduction

In statistics, linear regression is a linear approach to modeling the relationship between a scalar response (or dependent variable) and one or more explanatory variables (or independent variables). The case of one explanatory variable is called simple linear regression. For more than one explanatory variable, the process is called multiple linear regression. This term is distinct from multivariate linear regression, where multiple correlated dependent variables are predicted, rather than a single scalar variable.



1.11.2 The Simple Linear Regression Model

The simplest deterministic mathematical relationship between two variables x and y is a linear relationship: $y = \beta_0 + \beta_1 x$.

The objective of this section is to develop an equivalent *linear probabilistic model*.

If the two (random) variables are probabilistically related, then for a fixed value of x, there is uncertainty in the value of the second variable.

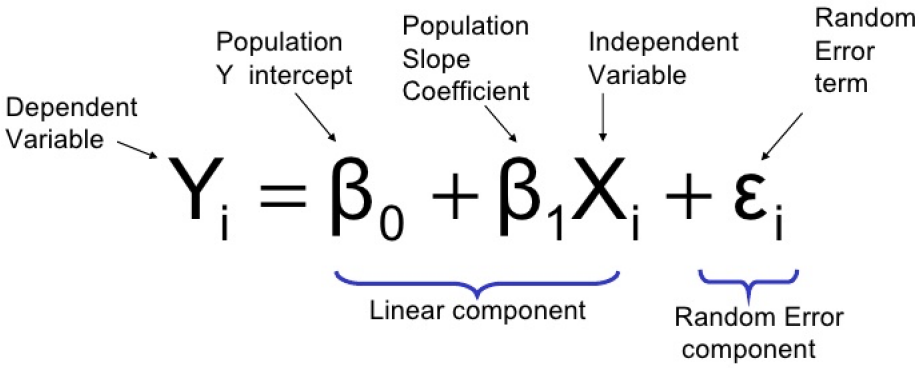
So we assume $Y = \beta_0 + \beta_1 x + \varepsilon$, where ε is a random variable.

Two variables are related linearly "on average" if for fixed x the actual value of Y differs from its expected value by a random amount (i.e. there is random error).

1.11.3 A Linear Probabilistic Model

Definition: (The Simple Linear Regression Model)

There are parameters β_0, β_1, and σ^2, such that for any fixed value of the independent variable x, the dependent variable is a random variable related to x through the model equation



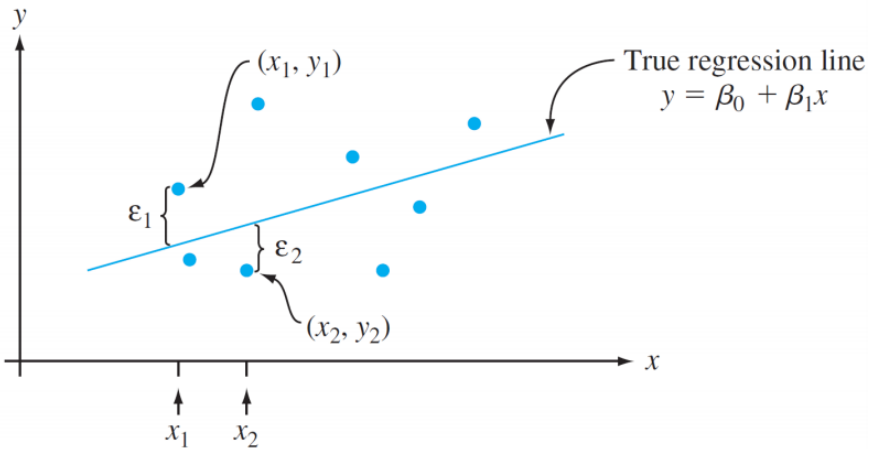
The quantity ε in the model equation is the "error" - a random variable, assumed to be symmetrically distributed with

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

(no assumption made about the distribution of ε, yet)

- $\boldsymbol{X}$: the independent, predictor, or explanatory variable (usually known).
- $\boldsymbol{Y}$: the dependent or response variable. For fixed x, Y will be random variable.
- $\boldsymbol{\varepsilon}$: the random deviation or random error term. For fixed x, ε will be random variable.
- $\boldsymbol{\beta_0}$: the average value of Y when x is zero (the intercept of the true regression line)
- $\boldsymbol{\beta_1}$: the expected (average) change in Y associated with a 1-unit increase in the value of x. (the slope of the true regression line)

The points $(x_1, y_1), \ldots, (x_n, y_n)$ resulting from n independent observations will then be scattered about the true regression line:



1.11.4 Estimating Model Parameters

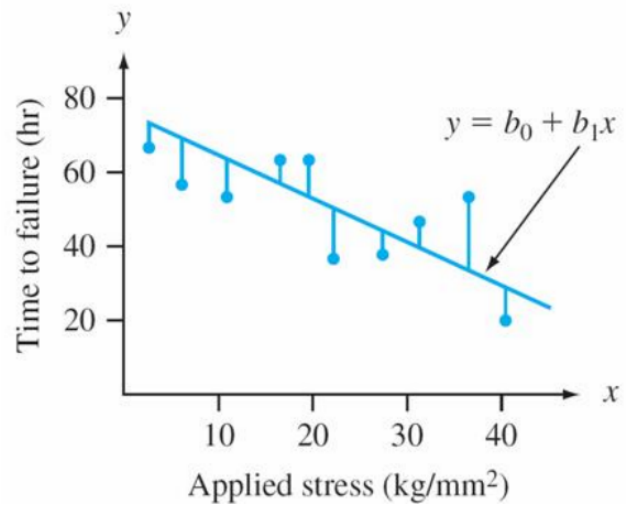
The values of β_0, $beta_1$, and $sigma$ will almost never be known to an investigator.

Instead, sample data consists of n observed pairs $(x_1, y_1), \ldots, (x_n, y_n)$ from which the model parameters and the true regression line itself can be estimated.

The data (pairs) are assumed to have been obtained independently of one another.

The "best fit" line is motivated by the principle of **least squares**, which can be traced back to the German mathematician **Gauss** (1777–1855):

A line provides the best fit to the data if the sum of the squared vertical distances (deviations) from the observed points to that line is as small as it can be.



The sum of *squared vertical deviations* from the points $(x_1, y_1), \ldots, (x_n, y_n)$

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

The point estimates of β_0 and β_1, denoted by and, are called the least squares estimates – they are those values that minimize $f(b_0, b_1)$.

The fitted **regression line** or **least squares** line is then the line whose equation is $y = \hat{\beta}_0 + \hat{\beta}_1 x$.

The minimizing values of b_0 and b_1 are found by taking partial derivatives of $f(b_0, b_1)$ with respect to both b_0 and b_1, equating them both to zero [analogously to $f'(b) = 0$ in univariate calculus], and solving the equations

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

The least squares estimate of the slope coefficient β_1 of the true regression line is

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

Shortcut formulas for the numerator and denominator of $\hat{\beta}_1$ are

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

The least squares estimate of the intercept b_0 of the true regression line is

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

1.11.5 Usage

Imagine that we have following points and we want to build a linear regression model:

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

Then the code will look like this:

```
// 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;
    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 Lagrange's Interpolation

1.12.1 Lagrange polynomial

In numerical analysis, Lagrange polynomials are used for polynomial interpolation. For a given set of points (x_j, y_j) with no two x_j values equal, the Lagrange polynomial is the polynomial of lowest degree that assumes

at each value x_j the corresponding value x_j, so that the functions coincide at each point.

1.12.2 Definition

Given a set of $k+1$ data points $(x_0, y_0), \ldots, (x_j, y_j), \ldots, (x_k, y_k)$ where no two x_j are the same, the **interpolation polynomial in the Lagrange form** is a linear combination $L(x) := \sum_{j=0}^{k} y_j \ell_j(x)$, of Lagrange basis polynomials

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

where $0 \le j \le k$. Note how, given the initial assumption that no two x_j are the same, $x_j - x_m \ne 0$, so this expression is always well-defined. The reason pairs $x_i = x_j$ with $y_i \ne y_j$ are not allowed is that no interpolation function L such that $y_i = L(x_i)$ would exist; a function can only get one value for each argumetn x_i. On the other hand, if also $y_i = y_j$, then those two points would actually be one single point.

For all $i \ne j$, $\ell(x)$ includes the term $(x - x_i)$ in the numerator, so the whole pruduct will be zero at $x = x_i$:

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

On the other hand,

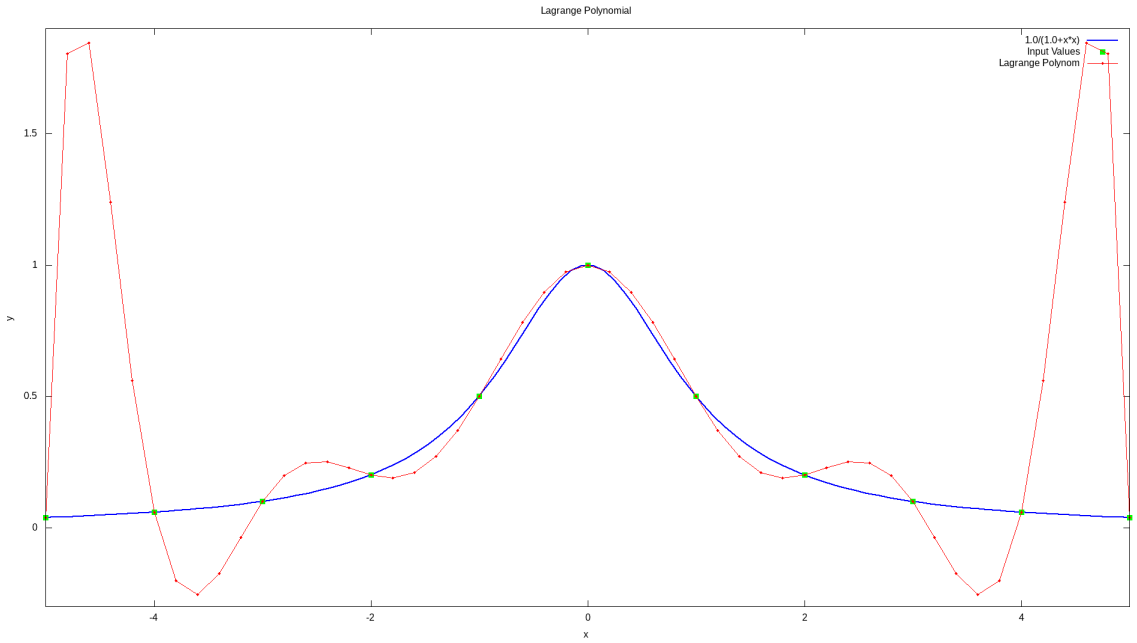
$$\ell_i(x_i) := \prod_{m \ne i} \frac{x_i - x_m}{x_i - x_m} = 1 \tag{1.40}$$

In other words, all basis polynomials are zero at $x = x_i$, except $\ell_i(x)$, for which it holds that $\ell_i(x_i) = 1$, because it lacks the $(x - x_i)$ term.

It follows that $\ell_i(x_i) = y_i$, so at each point x_i, $L(x_i) = y_i + 0 + 0 + \cdots + 0 = y_i$, showing that L interpolates the function exactly.

1.12.3 Runge's example

The function $f(x) = \frac{1}{1+x^2}$ cannot be interpolated accurately on $[5, 5]$ using a tenth-degree polynomial (dashed curve) with equally-spaced interpolation points. This example that illustrates the difficulty that one can generally expect with high-degree polynomial interpolation with equally-spaced points is known as *Runge's example*.



1.12.4 Usage

Imagine that we have following points and we want to build a Lagrange polynomial with this points:

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

Then the code will look like this:

```
// example_lagrange_polynomial.cpp

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

using namespace std;
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;
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    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 Numerical differentiation. Calculation of the first derivative.

1.13.1 Definition

By definition, the first derivative of a smooth function $f(x)$ at a point x is calculated as

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

When calculating the first derivative of the function $f(x)$ on a computer, we replace the infinitesimal $h \to \infty$ with a small but finite value h:

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

where $\mathrm{O}(h)$ is the derivative calculation error, which naturally depends on h. Previous formula is called a difference scheme for calculating the first derivative (more precisely, a right difference scheme or just a right difference). Similarly, maybe the left-hand difference scheme is written.

How to determine $\mathrm{O}(h)$? Expand the function $f(x)$ in a Taylor series at the point $x + h$:

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

whence it follows that in the first order of the expansion

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

By choosing a very small h, the round-off errors in computing on a computer can be comparable to or greater than h. Therefore, we are interested in an algorithm that gives lower error value for the same value of h.

Such an improved algorithm can be easily obtained by expanding the function $f(x)$ into a Taylor series at the points $x + h$ and $x - h$, then subtracting one result from the other, which gives

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

where the error in calculating the first derivative

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

This is the central difference scheme (central difference).

In principle, it is possible to follow the path of improving the accuracy of the method for calculating the first derivative and further. For example, considering the expansion of the function $f(x)$ in a Taylor series at the points $x + h$, $x + 2h$, $x - h$, and $x - 2h$, one can obtain a four-point scheme etc.

1.13.2 Usage

Imagine that we want to find the derivative of the following function:

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

Then the code will look like this:

```
// 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);

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

    return 0;
}
```

1.14 Incomplete Gamma Function

1.14.1 Definition

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

1.14.2 Usage

Imagine that we want to calculate the value of:

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

Then the code will look like this:

```
// 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;
}
```