

**Московский авиационный институт
(национальный исследовательский университет)**

**Факультет информационных технологий и прикладной
математики**

Кафедра вычислительной математики и программирования

Лабораторные работы по курсу «Численные методы»

Студент: В. В. Бирюков
Преподаватель: Д. Л. Ревизников
Группа: М8О-307Б-19
Дата:
Оценка:
Подпись:

Москва, 2022

1 Численные методы линейной алгебры

1 LU-разложение матриц

1.1 Постановка задачи

Реализовать алгоритм LU - разложения матриц (с выбором главного элемента) в виде программы. Используя разработанное программное обеспечение, решить систему линейных алгебраических уравнений (СЛАУ). Для матрицы СЛАУ вычислить определитель и обратную матрицу.

Вариант:

$$\begin{cases} 2x_1 + 7x_2 - 8x_3 + 6x_4 = -39 \\ 4x_1 + 4x_2 - 7x_3 - 7x_4 = 41 \\ -x_1 - 4x_2 + 6x_3 + 3x_4 = 4 \\ 9x_1 - 7x_2 - 2x_3 - 8x_4 = 113 \end{cases}$$

1.2 Результаты работы

```
$ cat tests/1/2.txt
```

```
4
```

```
2 7 -8 6
```

```
4 4 0 -7
```

```
-1 -3 6 3
```

```
9 -7 -2 -8
```

```
-39 41 4 113
```

```
$ ./lab1_1 <tests/1/2.txt
```

Решение системы:

```
8.000   -3.000    2.000   -3.000
```

Обратная матрица системы:

```
0.102    0.072    0.161    0.074
```

```
0.040    0.111    0.035   -0.054
```

```
-0.004    0.087    0.155   -0.021
```

```
0.081   -0.038    0.112    0.011
```

Определитель матрицы системы:

```
-4924.000
```

1.3 Исходный код

```
1  #pragma once
2
3  #include <vector>
4  #include <utility>
5  #include <stdexcept>
6
7  #include "matrix.hpp"
8  #include "vector.hpp"
9
10 template <class T>
11 struct LUP {
12     Matrix<T> L;
13     Matrix<T> U;
14     std::vector<std::pair<size_t, size_t>> P;
15
16     LUP(size_t n): L(n), U(n), P() { }
17
18     LUP(const Matrix<T>& matrix): L(matrix.Size()), U(matrix.Size()), P() {
19         matrix.Decompose(L, U, P);
20     }
21
22     void Assign(const Matrix<T>& matrix) {
23         if (L.Size() != matrix.Size()) {
24             L = Matrix<T>(matrix.Size());
25             U = Matrix<T>(matrix.Size());
26         }
27         matrix.Decompose(L, U, P);
28     }
29
30     Matrix<T> Compose() {
31         return L * U;
32     }
33
34     T Det() {
35         T res = 1;
36         for (size_t i = 0; i < U.Size(); ++i) {
37             res *= U[i][i];
38         }
39         if (P.size() & 1) {
40             res = -res;
41         }
42         return res;
43     }
44
45     Vector<T> Solve(Vector<T> b) {
46         if (b.Size() != L.Size()) {
47             throw std::runtime_error("Dimension mismatch");
```

```

48     }
49     for (const std::pair<size_t, size_t>& p: P) {
50         std::swap(b[p.first], b[p.second]);
51     }
52
53     Vector<T> x(L.Size()), z(L.Size());
54     for (size_t i = 0; i < b.Size(); ++i) {
55         z[i] = b[i];
56         for (size_t j = 0; j < i; ++j) {
57             z[i] -= L[i][j] * z[j];
58         }
59     }
60
61     for (int i = b.Size()-1; i >= 0; --i) {
62         x[i] = z[i];
63         for (int j = b.Size()-1; j > i; --j) {
64             x[i] -= U[i][j] * x[j];
65         }
66         x[i] /= U[i][i];
67     }
68
69     return x;
70 }
71
72 Matrix<T> Invert() {
73     Vector<T> e(L.Size());
74     Matrix<T> inv(L.Size());
75
76     for (size_t i = 0; i < L.Size(); ++i) {
77         e[i] = 1;
78         Vector<T> b = Solve(e);
79         for (size_t j = 0; j < L.Size(); ++j) {
80             inv[j][i] = b[j];
81         }
82         e[i] = 0;
83     }
84
85     return inv;
86 }
87 };

```

2 Метод прогонки

2.1 Постановка задачи

Реализовать метод прогонки в виде программы, задавая в качестве входных данных ненулевые элементы матрицы системы и вектор правых частей. Используя разработанное программное обеспечение, решить СЛАУ с трехдиагональной матрицей.

Вариант:

$$\begin{cases} 10x_1 + 5x_2 = -120 \\ 3x_1 + 10x_2 - 2x_3 = -91 \\ 2x_2 - 9x_3 - 5x_4 = 5 \\ 5x_3 + 16x_4 - 4x_5 = -74 \\ -8x_4 + 16x_5 = -56 \end{cases}$$

2.2 Результаты работы

```
$ cat tests/2/2.txt
```

```
5
```

```
10 5
```

```
3 10 -2
```

```
2 -9 -5
```

```
5 16 -4
```

```
-8 16
```

```
-120 -91 5 -74 -56
```

```
$ ./lab1_2 <tests/2/2.txt
```

Решение системы:

```
-9.000    -6.000    2.000   -7.000   -7.000
```

2.3 Исходный код

```
1  #pragma once
2
3  #include <vector>
4  #include <iostream>
5  #include <iomanip>
6  #include <stdexcept>
7
8  #include "vector.hpp"
9
10 template <class T>
11 class TDMatrix {
12 private:
13     size_t _size;
14
15 public:
16     std::vector<T> a, b, c;
17
18     TDMatrix(size_t n) : _size(n), a(n), b(n), c(n) { }
19
20     TDMatrix(std::vector<T> a, std::vector<T> b, std::vector<T> c) : _size(a.size()), a(
        a), b(b), c(c) {
21         if (a.size() != b.size() || b.size() != c.size()) {
22             throw std::runtime_error("Incompatible arrays");
23         }
24     }
25
26     size_t Size() const {
27         return _size;
28     }
29
30     Vector<T> Solve(const Vector<T>& vec) {
31         if (vec.Size() != _size) {
32             throw std::runtime_error("Dimension mismatch");
33         }
34
35         Vector<T> p(_size-1), q(_size-1), x(_size);
36         p[0] = - c[0] / b[0];
37         q[0] = vec[0] / b[0];
38
39         for (size_t i = 1; i < _size-1; ++i) {
40             p[i] = - c[i] / (b[i] + a[i] * p[i-1]);
41             q[i] = (vec[i] - a[i] * q[i-1]) / (b[i] + a[i] * p[i-1]);
42         }
43
44         x[_size-1] = (vec[_size-1] - a[_size-1] * q[_size-2]) / (b[_size-1] + a[_size-1] *
            p[_size-2]);
45         for (int i = _size - 2; i >= 0; --i) {
```

```

46     x[i] = p[i] * x[i+1] + q[i];
47 }
48
49     return x;
50 }
51
52 template <class U>
53 friend std::ostream& operator<<(std::ostream&, const TDMatrix<U>&);
54 template <class U>
55 friend std::istream& operator>>(std::istream&, TDMatrix<U>&);
56 };
57
58 template <class T>
59 std::ostream& operator<<(std::ostream& os, const TDMatrix<T>& matrix) {
60     os << matrix.b[0] << ' ' << matrix.c[0] << '\n';
61     for (size_t i = 1; i < matrix.Size()-1; ++i) {
62         os << std::setw(8) << matrix.a[i] << ' ';
63         os << std::setw(8) << matrix.b[i] << ' ';
64         os << std::setw(8) << matrix.c[i] << '\n';
65     }
66     os << matrix.a.back() << ' ' << matrix.b.back() << '\n';
67     return os;
68 }
69
70 template <class T>
71 std::istream& operator>>(std::istream& is, TDMatrix<T>& matrix) {
72     is >> matrix.b[0] >> matrix.c[0];
73     for (size_t i = 1; i < matrix.Size()-1; ++i) {
74         is >> matrix.a[i] >> matrix.b[i] >> matrix.c[i];
75     }
76     is >> matrix.a.back() >> matrix.b.back();
77     return is;
78 }

```

3 Итерационные методы решения СЛАУ

3.1 Постановка задачи

Реализовать метод простых итераций и метод Зейделя в виде программ, задавая в качестве входных данных матрицу системы, вектор правых частей и точность вычислений. Используя разработанное программное обеспечение, решить СЛАУ. Проанализировать количество итераций, необходимое для достижения заданной точности.

Вариант:

$$\begin{cases} 24x_1 + 2x_2 + 4x_3 - 9x_4 = -9 \\ -6x_1 - 27x_2 - 8x_3 - 6x_4 = -76 \\ -4x_1 + 8x_2 + 19x_3 + 6x_4 = -79 \\ 4x_1 + 5x_2 - 3x_3 - 13x_4 = -70 \end{cases}$$

3.2 Результаты работы

```
$ cat tests/3/2.txt
```

```
4
```

```
24 2 4 -9
```

```
-6 -27 -8 -6
```

```
-4 8 19 6
```

```
4 5 -3 -13
```

```
-9 -76 -79 -70
```

```
0.0001
```

```
$ ./lab1_3 <tests/3/2.txt
```

```
Решение методом Якоби:
```

```
3.999999 1.999999 -6.999998 8.999998
```

```
Количество итераций: 20
```

```
Решение методом Зейделя:
```

```
4.000000 2.000000 -7.000000 9.000000
```

```
Количество итераций: 10
```


3.3 Исходный код

```
1  #pragma once
2
3  #include <cmath>
4  #include <stdexcept>
5
6  #include "matrix.hpp"
7  #include "vector.hpp"
8
9  template <class T>
10 size_t IterationMethod(const Matrix<T>& A, const Vector<T>& b, const Matrix<T>& M,
    Vector<T>& x, double eps) {
11     if ((A.Size() != b.Size()) || (b.Size() != M.Size()) || (M.Size() != x.Size())) {
12         throw std::runtime_error("Dimension mismatch");
13     }
14     Matrix<T> alpha = Matrix<T>::Identity(A.Size()) + M * A;
15     Vector<T> beta = -(M * b);
16     x = beta;
17
18     T alpha_norm = alpha.Norm();
19     size_t count = 0;
20     double eps_k;
21     Vector<T> next_x(x.Size());
22
23     do {
24         next_x = alpha * x + beta;
25         if (alpha_norm < 1) {
26             eps_k = alpha_norm / (1.0 - alpha_norm) * (next_x - x).Norm();
27         } else {
28             eps_k = (next_x - x).Norm();
29         }
30         std::swap(next_x, x);
31         ++count;
32     } while (eps_k > eps);
33
34     return count;
35 }
36
37 template <class T>
38 size_t JacobiMethod(const Matrix<T>& A, const Vector<T>& b, Vector<T>& x, double eps)
    {
39     Matrix<T> M(A.Size());
40     for (size_t i = 0; i < A.Size(); ++i) {
41         M[i][i] = -1.0 / A[i][i];
42     }
43     return IterationMethod(A, b, M, x, eps);
44 }
45
```

```

46 template <class T>
47 size_t SeidelMethod(const Matrix<T>& A, const Vector<T>& b, const Matrix<T>& M, Vector
    <T>& x, double eps) {
48     if ((A.Size() != b.Size()) || (b.Size() != M.Size()) || (M.Size() != x.Size())) {
49         throw std::runtime_error("Dimension mismatch");
50     }
51     Matrix<T> alpha = Matrix<T>::Identity(A.Size()) + M * A;
52     Vector<T> beta = -(M * b);
53     x = beta;
54
55     Matrix<T> alpha2 = alpha;
56     for (size_t i = 0; i < alpha2.Size(); ++i) {
57         for (size_t j = 0; j < i; ++j) {
58             alpha2[i][j] = 0;
59         }
60     }
61
62     T alpha_norm = alpha.Norm();
63     T alpha2_norm = alpha2.Norm();
64     size_t count = 0;
65     double eps_k;
66     Vector<T> next_x(x.Size());
67
68     do {
69         for (size_t i = 0; i < x.Size(); ++i) {
70             next_x[i] = beta[i];
71             for (size_t j = 0; j < i; ++j) {
72                 next_x[i] += alpha[i][j] * next_x[j];
73             }
74             for (size_t j = i; j < x.Size(); ++j) {
75                 next_x[i] += alpha[i][j] * x[j];
76             }
77         }
78
79         if (alpha_norm < 1) {
80             eps_k = alpha2_norm / (1.0 - alpha_norm) * (next_x - x).Norm();
81         } else {
82             eps_k = (next_x - x).Norm();
83         }
84         std::swap(next_x, x);
85         ++count;
86     } while (eps_k > eps);
87
88     return count;
89 }
90
91 template <class T>
92 size_t JacobiSeidelMethod(const Matrix<T>& A, const Vector<T>& b, Vector<T>& x, double
    eps) {

```

```

93 | Matrix<T> M(A.Size());
94 | for (size_t i = 0; i < A.Size(); ++i) {
95 |     M[i][i] = -1.0 / A[i][i];
96 | }
97 | return SeidelMethod(A, b, M, x, eps);
98 | }

```

4 Метод вращений

4.1 Постановка задачи

Реализовать метод вращений в виде программы, задавая в качестве входных данных матрицу и точность вычислений. Используя разработанное программное обеспечение, найти собственные значения и собственные векторы симметрических матриц. Проанализировать зависимость погрешности вычислений от числа итераций.

Вариант:

$$\begin{pmatrix} -9 & 7 & 5 \\ 7 & 8 & 9 \\ 5 & 9 & 8 \end{pmatrix}$$

4.2 Результаты работы

```
$ cat tests/4/2.txt
```

```
3
```

```
-9 7 5
```

```
7 8 9
```

```
5 9 8
```

```
0.0001
```

```
$ ./lab1_4 <tests/4/2.txt
```

Собственные значения:

```
19.532   -0.836  -11.696
```

Матрица собственных векторов:

```
0.286   -0.115   0.951
```

```
0.691   -0.663  -0.288
```

```
0.664    0.740  -0.110
```

Количество итераций: 6

4.3 Исходный код

```
1 | #pragma once
2 |
3 | #include <cmath>
4 |
5 | #include "matrix.hpp"
6 |
7 | template <class T>
8 | size_t RotationMethod(Matrix<T> A, double eps, std::vector<T>& values, Matrix<T>& U) {
9 |     values.assign(A.Size(), 0);
10 |    U = Matrix<T>::Identity(A.Size());
11 |
12 |    size_t k, l;
13 |    T max;
14 |    Matrix<T> transform(A.Size());
15 |    double angle, eps_k;
16 |    size_t count = 0;
17 |    do {
18 |        max = 0;
19 |        for (size_t i = 1; i < A.Size(); ++i) {
20 |            for (size_t j = 0; j < i; ++j) {
21 |                if (std::abs(A[i][j]) > max) {
22 |                    max = std::abs(A[i][j]);
23 |                    l = i;
24 |                    k = j;
25 |                }
26 |            }
27 |        }
28 |
29 |        angle = 0.5 * std::atan2(2 * A[k][l], A[k][k] - A[l][l]);
30 |        transform = Matrix<T>::Rotation(A.Size(), k, l, angle);
31 |
32 |        A = transform.Transpose() * A * transform;
33 |        U = U * transform;
34 |
35 |        eps_k = 0;
36 |        for (size_t i = 1; i < A.Size(); ++i) {
37 |            for (size_t j = 0; j < i; ++j) {
38 |                eps_k += A[i][j] * A[i][j];
39 |            }
40 |        }
41 |        eps_k = std::sqrt(eps_k);
42 |
43 |        ++count;
44 |    } while (eps_k > eps);
45 |
46 |    for (size_t i = 0; i < A.Size(); ++i) {
47 |        values[i] = A[i][i];
```

```
48 || }  
49 ||  
50 || return count;  
51 || }
```

5 QR алгоритм

5.1 Постановка задачи

Реализовать алгоритм QR – разложения матриц в виде программы. На его основе разработать программу, реализующую QR – алгоритм решения полной проблемы собственных значений произвольных матриц, задавая в качестве входных данных матрицу и точность вычислений. С использованием разработанного программного обеспечения найти собственные значения матрицы.

Вариант:

$$\begin{pmatrix} -6 & -4 & 0 \\ -7 & 6 & -7 \\ -2 & -6 & -7 \end{pmatrix}$$

5.2 Результаты работы

```
$ cat tests/5/2.txt
```

```
3
```

```
-6 -4 0
```

```
-7 6 -7
```

```
-2 -6 -7
```

```
0.00001
```

```
$ ./lab1_5 <tests/5/2.txt
```

```
Собственные значения:
```

```
-11.3395 10.0120 -5.6725
```

```
Количество итераций: 112
```

5.3 Исходный код

```
1  #pragma once
2
3  #include <cmath>
4  #include <vector>
5  #include <complex>
6
7  #include "matrix.hpp"
8  #include "vector.hpp"
9
10 template <class T>
11 void QR_Decomposition(const Matrix<T>& A, Matrix<T>& Q, Matrix<T>& R) {
12     R = A;
13     Q = Matrix<T>::Identity(A.Size());
14     Vector<T> v(A.Size());
15     Matrix<T> H(A.Size());
16     for (size_t k = 0; k < A.Size() - 1; ++k) {
17         for (size_t i = 0; i < k; ++i) {
18             v[i] = 0;
19         }
20         v[k] = R[k][k] * R[k][k];
21         for (size_t i = k+1; i < A.Size(); ++i) {
22             v[i] = R[i][k];
23             v[k] += R[i][k] * R[i][k];
24         }
25         v[k] = std::sqrt(v[k]) * (R[k][k] >= 0 ? 1 : -1) + R[k][k];
26
27         H = Matrix<T>::Householder(A.Size(), v);
28         R = H * R;
29         Q = Q * H;
30     }
31 }
32
33 template <class T>
34 size_t QR_Eigenvalues(Matrix<T> A, double eps, std::vector<std::complex<T>>&
    eigenvalues) {
35     Matrix<T> Q(A.Size()), R(A.Size());
36     size_t iter_count = 0;
37     Vector<double> eps_1(A.Size()), eps_2(A.Size()), eps_3(A.Size());
38     eigenvalues.assign(A.Size(), std::complex<T>());
39     std::vector<std::complex<T>> prev_eigenvalues(A.Size(), 1e18);
40
41     do {
42         QR_Decomposition(A, Q, R);
43         A = R * Q;
44
45         // вычисление собственных значений
46         eigenvalues.swap(prev_eigenvalues);
```



```

47     for (size_t i = 0; i < A.Size(); ++i) {
48         if (i == A.Size()-1 || std::abs(A[i+1][i]) < eps) {
49             eigenvalues[i] = std::complex(A[i][i]);
50         } else {
51             T d = (A[i][i] + A[i+1][i+1]) * (A[i][i] + A[i+1][i+1]) - 4 * (A[i][i] * A[i
52                 +1][i+1] - A[i][i+1] * A[i+1][i]);
53             if (d > eps) {
54                 continue;
55             }
56             T re = (A[i][i] + A[i+1][i+1]) * 0.5;
57             T im = std::sqrt(std::abs(d)) * 0.5;
58             if (std::abs(im) < eps) {
59                 continue;
60             }
61             eigenvalues[i] = std::complex(re, im);
62             eigenvalues[i+1] = std::complex(re, -im);
63             ++i;
64         }
65     }
66     // вычисление погрешностей
67     for (size_t j = 0; j < A.Size(); ++j) {
68         eps_1[j] = 0;
69         for (size_t i = j+1; i < A.Size(); ++i) {
70             eps_1[j] += A[i][j] * A[i][j];
71         }
72         eps_1[j] = std::sqrt(eps_1[j]);
73
74         eps_2[j] = 0;
75         for (size_t i = j+2; i < A.Size(); ++i) {
76             eps_2[j] += A[i][j] * A[i][j];
77         }
78         eps_2[j] = std::sqrt(eps_2[j]);
79
80         eps_3[j] = std::abs(prev_eigenvalues[j]) - std::abs(eigenvalues[j]);
81     }
82
83     ++iter_count;
84 } while ((eps_1.Norm(2) > eps && eps_2.Norm(2) > eps) || eps_3.Norm() > eps);
85
86 return iter_count;
87 }

```

2 Решение нелинейных уравнений и систем нелинейных уравнений

1 Решение нелинейных уравнений

1.1 Постановка задачи

Реализовать методы простой итерации и Ньютона решения нелинейных уравнений в виде программ, задавая в качестве входных данных точность вычислений. С использованием разработанного программного обеспечения найти положительный корень нелинейного уравнения (начальное приближение определить графически). Проанализировать зависимость погрешности вычислений от количества итераций.

Вариант:

$$\ln x + 2 - x^2 = 0$$

1.2 Результаты работы

```
$ ./lab2_1
```

```
0.001
```

Метод простой итерации:

Корень: -0.58782084

Количество итераций: 49

Корень: 1.05723005

Количество итераций: 11

Метод Ньютона:

Корень: -0.58760883

Количество итераций: 6

Корень: 1.05710366

Количество итераций: 4

```
$ ./lab2_1
```

```
0.00000001
```

Метод простой итерации:

Корень: -0.58760883

Количество итераций: 127

Корень: 1.05710355

Количество итераций: 29

Метод Ньютона:

Корень: -0.58760883
Количество итераций: 7
Корень: 1.05710355
Количество итераций: 6

1.3 Исходный код

```
1 | #pragma once
2 |
3 | #include <functional>
4 | #include <cmath>
5 |
6 | template <class T>
7 | using interval_t = std::pair<T, T>;
8 |
9 | template <class T>
10 | int sign(T x) {
11 |     if (x > 0) {
12 |         return 1;
13 |     } else if (x < 0) {
14 |         return -1;
15 |     } else {
16 |         return 0;
17 |     }
18 | }
19 |
20 | template <class T>
21 | std::pair<T, size_t> IterationMethod(T x, const interval_t<T>& interval, const std::
    function<T(T)>& f, const std::function<T(T)>& df, double eps) {
22 |     T lambda = sign(df(x)) / std::max(std::abs(df(interval.first)), std::abs(df(interval
    .second)));
23 |     double q = std::max(std::abs(1 - lambda * df(interval.first)), std::abs(1 - lambda *
    df(interval.second)));
24 |     q = q / (1 - q);
25 |
26 |     double eps_k;
27 |     T next_x;
28 |     size_t iter_count = 0;
29 |     do {
30 |         next_x = x - lambda * f(x);
31 |         eps_k = q * std::abs(next_x - x);
32 |
33 |         std::swap(next_x, x);
34 |         ++iter_count;
35 |     } while (eps_k > eps);
36 |
37 |     return {x, iter_count};
38 | }
```

```

39
40 template <class T>
41 std::pair<T, size_t> NewtonMethod(T l, T r, const std::function<T(T)>& f, const std::
    function<T(T)>& df, const std::function<T(T)>& ddf, double eps) {
42     T x = l;
43     if (f(x) * ddf(x) < eps) {
44         x = r;
45     }
46     double eps_k;
47     T next_x;
48     size_t iter_count = 0;
49     do {
50         next_x = x - f(x) / df(x);
51         eps_k = std::abs(next_x - x);
52
53         std::swap(next_x, x);
54         ++iter_count;
55     } while (eps_k > eps);
56
57     return {x, iter_count};
58 }

```

2 Решение систем нелинейных уравнений

2.1 Постановка задачи

Реализовать методы простой итерации и Ньютона решения систем нелинейных уравнений в виде программного кода, задавая в качестве входных данных точность вычислений. С использованием разработанного программного обеспечения решить систему нелинейных уравнений (при наличии нескольких решений найти то из них, в котором значения неизвестных являются положительными); начальное приближение определить графически. Проанализировать зависимость погрешности вычислений от количества итераций.

Вариант:

$$\begin{cases} (x_1^2 + 9)x_2 - 27 = 0 \\ (x_1 - 1.5)^2 + (x_2 - 1.5)^2 - 9 = 0 \end{cases}$$

2.2 Результаты работы

```
$ ./lab2_2
```

```
0.001 1
```

```
Метод простой итерации:
```

```
Решение: -1.32480839 2.51043055
```

```
Количество итераций: 7
```

```
Решение: 4.44697542 0.93831944
```

```
Количество итераций: 9
```

```
Метод Ньютона:
```

```
Решение: -1.32469087 2.51050556
```

```
Количество итераций: 4
```

```
Решение: 4.44694707 0.93830348
```

```
Количество итераций: 4
```

```
$ ./lab2_2
```

```
0.00000001 1
```

```
Метод простой итерации:
```

```
Решение: -1.32469087 2.51050556
```

```
Количество итераций: 17
```

```
Решение: 4.44694707 0.93830348
```

```
Количество итераций: 20
```

```
Метод Ньютона:
```

Решение: -1.32469087 2.51050556

Количество итераций: 5

Решение: 4.44694707 0.93830348

Количество итераций: 5

2.3 Исходный код

```
1  #pragma once
2
3  #include <functional>
4  #include <vector>
5  #include <stdexcept>
6
7  #include "../linear/matrix.hpp"
8  #include "../linear/vector.hpp"
9  #include "../linear/lup.hpp"
10 #include "../function/optimization.hpp"
11
12 template <class T, class... Args>
13 class Matrix<std::function<T(Args...)>> {
14 private:
15     using function_t = std::function<T(Args...)>;
16
17     std::vector<std::vector<function_t>> _data;
18     size_t _size;
19
20     Matrix() : _data(), _size(0) { }
21
22 public:
23     Matrix(size_t n) : _data(n, std::vector<function_t>(n)), _size(n) { }
24
25     Matrix(std::initializer_list<std::vector<function_t>> list) : _data(list), _size(
        _data.size()) {
26         for (const std::vector<function_t>& row: _data) {
27             if (row.size() != _size) {
28                 throw std::runtime_error("Incorrect initializer list");
29             }
30         }
31     }
32
33     std::vector<function_t>& operator[](size_t index) {
34         return _data[index];
35     }
36
37     const std::vector<function_t>& operator[](size_t index) const {
38         return _data[index];
39     }
40 }
```

```

41 | size_t Size() const {
42 |     return _size;
43 | }
44 |
45 | Matrix<T> operator()(Args... args) const {
46 |     Matrix<T> res(_size);
47 |     for (size_t i = 0; i < _size; ++i) {
48 |         for (size_t j = 0; j < _size; ++j) {
49 |             res[i][j] = _data[i][j](args...);
50 |         }
51 |     }
52 |     return res;
53 | }
54 | };
55 |
56 | template <class T, class... Args>
57 | class Vector<std::function<T(Args...)>> {
58 | private:
59 |     using function_t = std::function<T(Args...)>;
60 |
61 |     std::vector<function_t> _data;
62 |     size_t _size;
63 |
64 |     Vector() : _data(), _size(0) { }
65 |
66 | public:
67 |     Vector(size_t n) : _data(n), _size(n) { }
68 |
69 |     Vector(std::initializer_list<function_t> list) : _data(list), _size(_data.size()) {
70 |     }
71 |
72 |     function_t& operator[](size_t index) {
73 |         return _data[index];
74 |     }
75 |
76 |     const function_t& operator[](size_t index) const {
77 |         return _data[index];
78 |     }
79 |
80 |     size_t Size() const {
81 |         return _size;
82 |     }
83 |
84 |     Vector<T> operator()(Args... args) const {
85 |         Vector<T> res(_size);
86 |         for (size_t i = 0; i < _size; ++i) {
87 |             res[i] = _data[i](args...);
88 |         }
89 |         return res;

```

```

89     }
90 };
91
92 template <class T>
93 using function_t = std::function<T(Vector<T>>>;
94
95 template <class T>
96 size_t IterationMethod(Vector<T>& x, const Vector<T>& a, const Vector<T>& b, const
    Vector<function_t<T>>& F, const Matrix<function_t<T>>& J, double eps) {
97     double eps_k;
98     Vector<T> next_x(x.Size());
99     size_t iter_count = 0;
100
101     Matrix<T> lambda = LUP(J(x)).Invert(), E = Matrix<T>::Identity(x.Size());
102     function_t<T> Jphi = [&E, &lambda, &J](Vector<T> x){ return (E - lambda * J(x)).Norm
        (); };
103
104     std::function<bool(Vector<T>>> region = [&a, &b](Vector<T> x) {
105         bool in = true;
106         for (size_t i = 0; i < x.Size(); ++i) {
107             if (x[i] < a[i] || x[i] > b[i]) {
108                 in = false;
109             }
110         }
111         return in;
112     };
113     std::vector<Vector<T>> simplex{a};
114     Vector<T> point = a;
115     for (size_t i = 0; i < a.Size(); ++i) {
116         point[i] = b[i];
117         simplex.push_back(point);
118         point[i] = a[i];
119     }
120
121     T q = Jphi(NelderMeadMethod<T, std::greater<T>>(Jphi, simplex, eps, region));
122     if (q > 1 - eps) {
123         throw std::runtime_error("Incorrect interval");
124     }
125     q = q / (1 - q);
126
127     do {
128         next_x = x - lambda * F(x);
129         eps_k = q * (next_x - x).Norm();
130
131         std::swap(next_x, x);
132         ++iter_count;
133     } while(eps_k > eps);
134
135     return iter_count;

```



```

136 }
137
138 template <class T>
139 size_t NewtonMethod(Vector<T>& x, const Vector<function_t<T>>& F, const Matrix<
    function_t<T>>& J, double eps, int k) {
140     double eps_k;
141     Vector<T> dx(x.Size());
142     size_t iter_count = 0;
143     LUP<T> lu(F.Size());
144     if (k == 0) {
145         lu.Assign(J(x));
146     }
147
148     do {
149         if (k > 0 && iter_count % k == 0) {
150             lu.Assign(J(x));
151         }
152
153         dx = lu.Solve(-F(x));
154         x += dx;
155         eps_k = dx.Norm();
156
157         ++iter_count;
158     } while (eps_k > eps);
159
160     return iter_count;
161 }

```

3 Приближение функций. Численные дифференцирование и интегрирование

1 Интерполяционные многочлены

1.1 Постановка задачи

Используя таблицу значений Y_i функции $y = f(x)$, вычисленных в точках X_i , $i = 0, \dots, 3$, построить интерполяционные многочлены Лагранжа и Ньютона, проходящие через точки $\{X_i, Y_i\}$. Вычислить значение погрешности интерполяции в точке X^* .

Вариант:

$$y = \cos(x), X^* = \frac{\pi}{4}$$

а). $X_i = 0, \frac{\pi}{6}, \frac{2\pi}{6}, \frac{3\pi}{6}$

б). $X_i = 0, \frac{\pi}{6}, \frac{5\pi}{12}, \frac{\pi}{2}$

1.2 Результаты работы

```
$ cat tests/1/1.txt
```

```
4
```

```
0 0.5235987755982988 1.0471975511965976 1.5707963267948966
0.7853981633974483
```

```
$ ./lab3_1 <tests/1/1.txt
```

Многочлен в форме Лагранжа:

```
0.1139 * x**3 -0.6021 * x**2 + 0.0282 * x**1 + 1.0000
```

Многочлен в форме Ньютона:

```
0.1139 * x**3 -0.6021 * x**2 + 0.0282 * x**1 + 1.0000
```

Погрешность интерполяции:

```
0.0012
```

```
$ cat tests/1/2.txt
```

```
4
```

```
0 0.5235987755982988 1.3089969389957472 1.5707963267948966
0.7853981633974483
```

```
$ ./lab3_1 <tests/1/2.txt
```

Многочлен в форме Лагранжа:

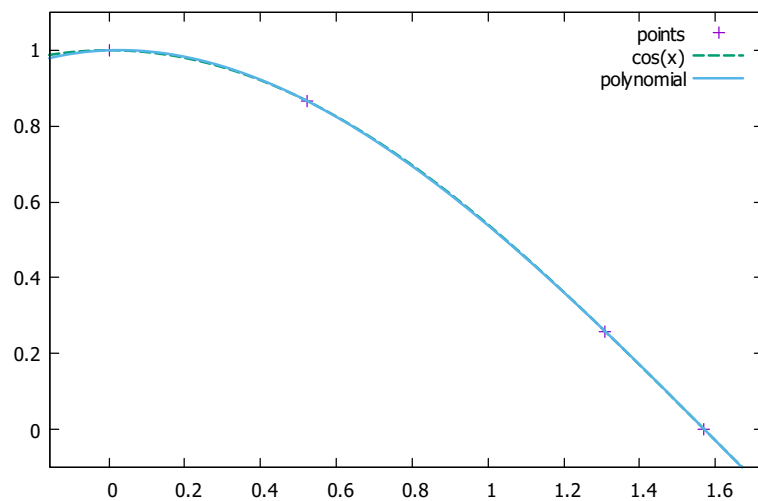
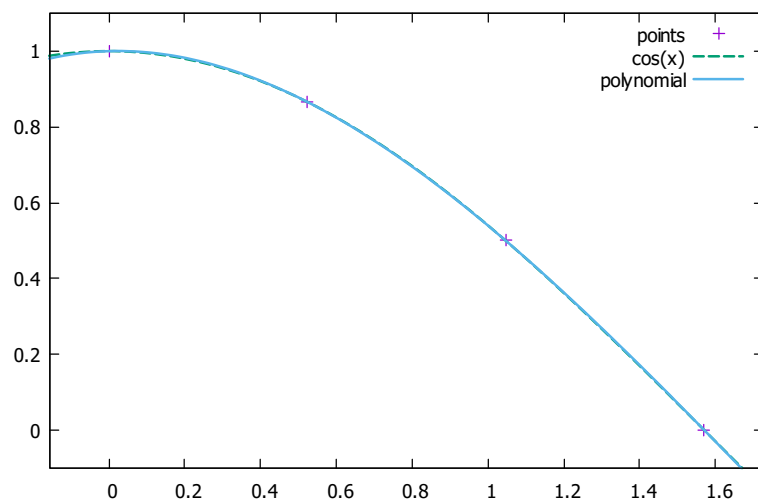
```
0.1206 * x**3 -0.6161 * x**2 + 0.0337 * x**1 + 1.0000
```

Многочлен в форме Ньютона:

$0.1206 * x^3 - 0.6161 * x^2 + 0.0337 * x + 1.0000$

Погрешность интерполяции:

0.0023



1.3 Исходный код

```
1  #pragma once
2
3  #include <vector>
4  #include <functional>
5
6  #include "polynomial.hpp"
7
8  template <class T>
9  Polynomial<T> LagrangePolynomial(const std::vector<T>& x, const std::vector<T>& y) {
10     if (x.size() != y.size()) {
11         throw std::runtime_error("Incompatible arrays");
12     }
13     Polynomial<T> p(x.size() - 1);
14
15     T div;
16     Polynomial<T> l;
17     for (size_t i = 0; i < x.size(); ++i) {
18         l.Assign(0, 1);
19         div = 1;
20
21         for (size_t j = 0; j < x.size(); ++j) {
22             if (j == i) {
23                 continue;
24             }
25             div *= x[i] - x[j];
26             l *= Polynomial<T>{-x[j], 1}; //  $x - x_j$ 
27         }
28         l *= y[i] / div;
29         p += l;
30     }
31
32     return p;
33 }
34
35 template <class T>
36 T DividedDifference(const std::vector<T>& x, const std::vector<T>& y) {
37     if (x.size() != y.size()) {
38         throw std::runtime_error("Incompatible arrays");
39     }
40
41     T res = 0, res_i;
42     for (size_t i = 0; i < x.size(); ++i) {
43         res_i = 1;
44         for (size_t j = 0; j < x.size(); ++j) {
45             if (i == j) {
46                 continue;
47             }
```

```

48     res_i *= x[i] - x[j];
49 }
50
51     res += y[i] / res_i;
52 }
53
54     return res;
55 }
56
57 template <class T>
58 Polynomial<T> NewtonPolynomial(const std::vector<T>& x, const std::vector<T>& y) {
59     if (x.size() != y.size()) {
60         throw std::runtime_error("Incompatible arrays");
61     }
62
63     Polynomial<T> p, p_i;
64     std::vector<T> x_i, y_i;
65     for (size_t i = 0; i < x.size(); ++i) {
66         x_i.push_back(x[i]);
67         y_i.push_back(y[i]);
68         p_i.Assign(0, DividedDifference(x_i, y_i));
69
70         for (size_t j = 0; j < i; ++j) {
71             p_i *= Polynomial<T>{-x[j], 1};
72         }
73
74         p += p_i;
75     }
76
77     return p;
78 }

```

2 Интерполяция кубическими сплайнами

2.1 Постановка задачи

Построить кубический сплайн для функции, заданной в узлах интерполяции, предполагая, что сплайн имеет нулевую кривизну при $x = x_0$ и $x = x_4$. Вычислить значение функции в точке $x = X^*$

Вариант: $X^* = 1.5$

i	0	1	2	3	4
x_i	0.0	1.0	2.0	3.0	4.0
f_i	1.0	0.86603	0.5	0.0	-0.5

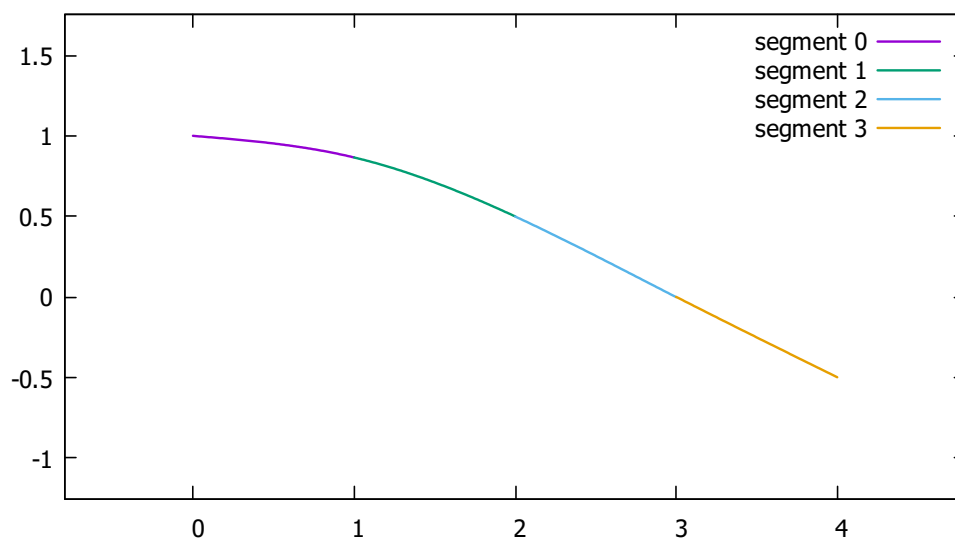
2.2 Результаты работы

```
$ ./lab3_2 <tests/2/2.txt
```

Значение в точке 1.5000: 0.7109

Сегменты кубического сплайна:

```
[0.0000:1.0000] -0.0526 * x**3 + 0.0000 * x**2 -0.0814 * x**1 + 1.0000  
[1.0000:2.0000] 0.0309 * x**3 -0.2504 * x**2 + 0.1691 * x**1 + 0.9165  
[2.0000:3.0000] 0.0271 * x**3 -0.2279 * x**2 + 0.1239 * x**1 + 0.9466  
[3.0000:4.0000] -0.0054 * x**3 + 0.0651 * x**2 -0.7550 * x**1 + 1.8255
```



2.3 Исходный код

```
1  #pragma once
2
3  #include <vector>
4  #include <iostream>
5
6  #include "polynomial.hpp"
7  #include "../linear/tridiagonal_matrix.hpp"
8  #include "../linear/vector.hpp"
9
10 template <class T>
11 class CubicSpline {
12 public:
13     std::vector<T> x;
14     std::vector<T> y;
15     std::vector<Polynomial<T>> segment;
16
17     CubicSpline(const std::vector<T>& x, const std::vector<T>& y) : x(x), y(y), segment(
18         x.size() - 1) {
19         if (x.size() != y.size()) {
20             throw std::runtime_error("Incompatible arrays");
21         }
22
23         std::vector<T> h(x.size() - 1);
24         for (size_t i = 1; i < x.size(); ++i) {
25             h[i-1] = x[i] - x[i-1];
26         }
27
28         size_t n = h.size();
29         TDMatrix<T> matrix(n - 1);
30         Vector<T> vec(n - 1);
31
32         matrix.b[0] = 2 * (h[0] + h[1]);
33         matrix.c[0] = h[1];
34         vec[0] = 3 * ((y[2] - y[1]) / h[1] - (y[1] - y[0]) / h[0]);
35
36         for (size_t i = 2; i < n - 1; ++i) {
37             matrix.a[i-1] = h[i-1];
38             matrix.b[i-1] = 2 * (h[i-1] + h[i]);
39             matrix.c[i-1] = h[i];
40             vec[i-1] = 3 * ((y[i+1] - y[i]) / h[i] - (y[i] - y[i-1]) / h[i-1]);
41         }
42
43         matrix.a[n-2] = h[n-2];
44         matrix.b[n-2] = 2 * (h[n-2] + h[n-1]);
45         vec[n-2] = 3 * ((y[n] - y[n-1]) / h[n-1] - (y[n-1] - y[n-2]) / h[n-2]);
46
47         std::vector<T> a(n), b(n), c = matrix.Solve(vec), d(n);
```

```

47     c.insert(c.begin(), 0);
48     for (size_t i = 0; i < n-1; ++i) {
49         a[i] = y[i];
50         b[i] = (y[i+1] - y[i]) / h[i] - h[i] * (c[i+1] + 2 * c[i]) / 3.;
51         d[i] = (c[i+1] - c[i]) / (3 * h[i]);
52     }
53     a[n-1] = y[n-1];
54     b[n-1] = (y[n] - y[n-1]) / h[n-1] - (2. / 3.) * h[n-1] * c[n-1];
55     d[n-1] = - c[n-1] / (3 * h[n-1]);
56
57     for (size_t i = 0; i < n; ++i) {
58         segment[i] = Polynomial<T>({a[i], b[i], c[i], d[i]}, x[i]);
59     }
60 }
61
62 size_t Size() const {
63     return segment.size();
64 }
65
66 const Polynomial<T>& operator[](size_t index) {
67     return segment[index];
68 }
69
70 T operator()(const T& value) {
71     if (value < x[0] || value > x.back()) {
72         throw std::runtime_error("Out of range");
73     }
74
75     for (size_t i = 0; i < x.size()-1; ++i) {
76         if (value >= x[i] && value <= x[i+1] ) {
77             return segment[i](value);
78         }
79     }
80
81     return 0; // just for silence the warning
82 }
83 };
84
85 template <class T>
86 std::ostream& operator<<(std::ostream& os, const CubicSpline<T>& spline) {
87     for (size_t i = 0; i < spline.x.size() - 1; ++i) {
88         os << '[' << spline.x[i] << ',' << spline.x[i+1] << "]" << spline.segment[i] << '\n';
89     }
90     return os;
91 }

```


3 Метод наименьших квадратов

3.1 Постановка задачи

Для таблично заданной функции путем решения нормальной системы МНК найти приближающие многочлены а) 1-ой и б) 2-ой степени. Для каждого из приближающих многочленов вычислить сумму квадратов ошибок. Построить графики приближаемой функции и приближающих многочленов.

Вариант:

i	0	1	2	3	4	5
x_i	-1.0	0.0	1.0	2.0	3.0	4.0
f_i	0.86603	1.0	0.86603	0.50	0.0	-0.5

3.2 Результаты работы

```
$ ./lab3_3 <tests/3/2.txt
```

Приближающий многочлен 1-ой степени:

$-0.2913 * x^{**1} + 0.8923$

Ошибка: 0.2708

Приближающий многочлен 2-ой степени:

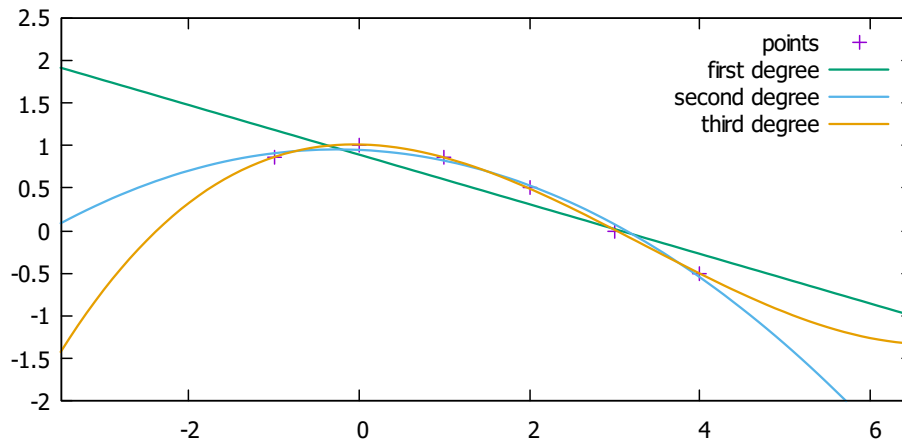
$-0.0827 * x^{**2} - 0.0431 * x^{**1} + 0.9475$

Ошибка: 0.0152

Приближающий многочлен 3-ой степени:

$0.0151 * x^{**3} - 0.1508 * x^{**2} - 0.0174 * x^{**1} + 1.0110$

Ошибка: 0.0003



3.3 Исходный код

```

1  #pragma once
2
3  #include <functional>
4  #include <vector>
5
6  #include "../linear/matrix.hpp"
7  #include "../linear/vector.hpp"
8  #include "../linear/lup.hpp"
9
10 template <class T>
11 std::vector<T> LSM(const std::vector<T>& x, const std::vector<T>& y, const std::vector
    <std::function<T(T)>>& basis) {
12     if (x.size() != y.size()) {
13         throw std::runtime_error("Incompatible arrays");
14     }
15
16     Matrix<T> Phi(x.size(), basis.size());
17     for (size_t i = 0; i < x.size(); ++i) {
18         for (size_t j = 0; j < basis.size(); ++j) {
19             Phi[i][j] = basis[j](x[i]);
20         }
21     }
22     Matrix<T> PhiT = Phi.Transpose();

```

```

23 | Matrix<T> G = PhiT * Phi;
24 | Vector<T> z = PhiT * Vector<T>(y);
25 |
26 | return LUP(G).Solve(z);
27 | }
28 |
29 | template <class T>
30 | T SquareError(const std::function<T(T)>& f, const std::vector<T>& x, const std::vector
    | <T>& y) {
31 |     T res = 0;
32 |     for (size_t i = 0; i < x.size(); ++i) {
33 |         res += (y[i] - f(x[i])) * (y[i] - f(x[i]));
34 |     }
35 |     return res;
36 | }

```

4 Численное дифференцирование

4.1 Постановка задачи

Вычислить первую и вторую производную от таблично заданной функции $y_i = f(x_i)$, $i = 0, \dots, 4$ в точке $x = X^*$.

Вариант: $X^* = 1.0$

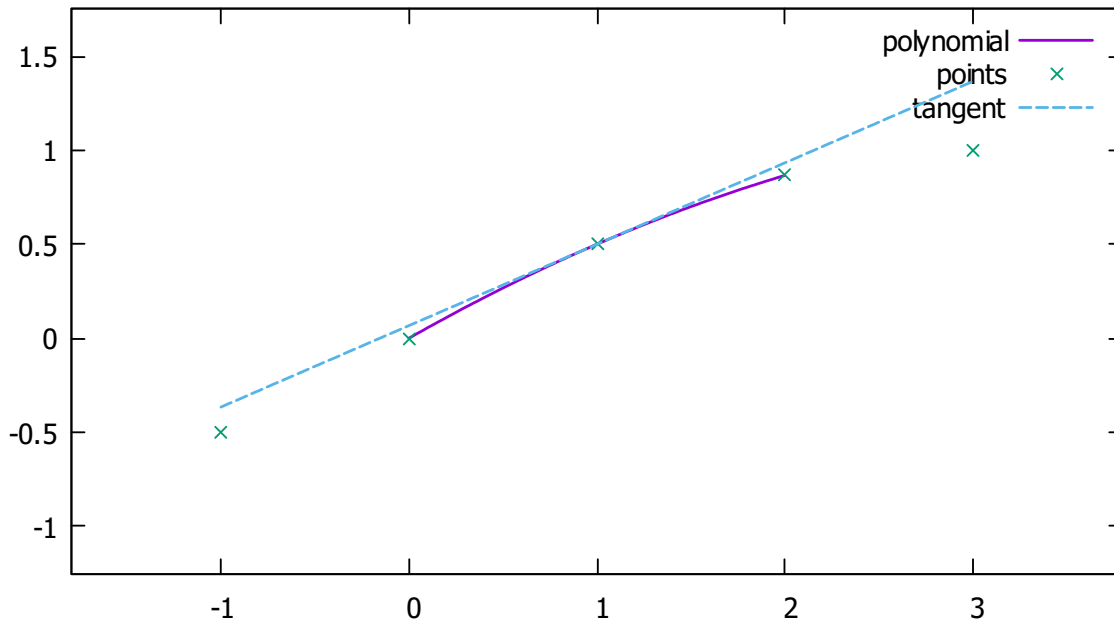
i	0	1	2	3	4
x_i	-1.0	0.0	1.0	2.0	3.0
f_i	-0.5	0.0	0.5	0.86603	1.0

4.2 Результаты работы

```
$ ./lab3_4 <tests/4/2.txt
```

Первая производная: 0.4330

Вторая производная: -0.1340



4.3 Исходный код

```
1  #pragma once
2
3  #include <vector>
4  #include <stdexcept>
5
6  template <class T>
7  T TableFirstDerivative(const std::vector<T>& x, const std::vector<T>& y, T value) {
8      if (x.size() != y.size()) {
9          throw std::runtime_error("Incompatible arrays");
10     }
11
12     if (x.size() < 3) {
13         throw std::runtime_error("Too few points");
14     }
15
16     if (value < x[0] || value > x.back()) {
17         throw std::runtime_error("Out of range");
18     }
19
20     size_t i = 0;
21     T min = value - x[0];
22     for (size_t j = 1; j < x.size(); ++j) {
23         if (min > std::abs(value - x[j])) {
24             min = std::abs(value - x[j]);
25             i = j - 1;
26         }
27     }
28
29     if (i == x.size() - 2) {
30         --i;
31     }
32
33     return (y[i+1] - y[i]) / (x[i+1] - x[i]) + ((y[i+2] - y[i+1]) / (x[i+2] - x[i+1]) -
34         (y[i+1] - y[i]) / (x[i+1] - x[i])) / (x[i+2] - x[i]) * (2 * value - x[i] - x[i
35         +1]));
36 }
37
38 template <class T>
39 T TableSecondDerivative(const std::vector<T>& x, const std::vector<T>& y, T value) {
40     if (x.size() != y.size()) {
41         throw std::runtime_error("Incompatible arrays");
42     }
43
44     if (x.size() < 3) {
45         throw std::runtime_error("Too few points");
46     }
47 }
```

```

46 | if (value < x[0] || value > x.back()) {
47 |     throw std::runtime_error("Out of range");
48 | }
49 |
50 | size_t i = 0;
51 | T min = value - x[0];
52 | for (size_t j = 1; j < x.size(); ++j) {
53 |     if (min > std::abs(value - x[j])) {
54 |         min = std::abs(value - x[j]);
55 |         i = j - 1;
56 |     }
57 | }
58 |
59 | if (i == x.size() - 2) {
60 |     --i;
61 | }
62 |
63 | return 2 * ((y[i+2] - y[i+1]) / (x[i+2] - x[i+1]) - (y[i+1] - y[i]) / (x[i+1] - x[i]
64 | )) / (x[i+2] - x[i]);

```

5 Численное интегрирование

5.1 Постановка задачи

Вычислить определенный интеграл $\int_{X_0}^{X_1} y dx$, методами прямоугольников, трапечий, Симпсона с шагами h_1, h_2 . Оценить погрешность вычислений, используя Метод Рунге-Ромберга:

Вариант:

$$y = \frac{x}{(3x + 4)^2}, \quad X_0 = 0, \quad X_1 = 4, \quad h_1 = 1.0, \quad h_2 = 0.5$$

5.2 Результаты работы

```
$ cat tests/5/1.txt
```

```
0 4 1
```

```
$ ./lab3_5 <tests/5/1.txt
```

```
Метод прямоугольников: 0.05816
```

```
Погрешность: 0.01816
```

```
Метод трапечий: 0.06597
```

```
Погрешность: 0.00345
```

```
Метод Симпсона: 0.06942
```

```
Погрешность: 0.00038
```

```
$ cat tests/5/2.txt
```

```
0 4 0.5
```

```
$ ./lab3_5 <tests/5/2.txt
```

```
Метод прямоугольников: 0.06550
```

```
Погрешность: 0.00734
```

```
Метод трапечий: 0.06941
```

```
Погрешность: 0.00114
```

```
Метод Симпсона: 0.07055
```

```
Погрешность: 0.00008
```

5.3 Исходный код

```
1 | #pragma once
2 |
3 | #include <functional>
4 | #include <cmath>
5 |
6 | #include "polynomial.hpp"
7 | #include "interpolation_polynomial.hpp"
8 |
9 | template <class T>
10 | T RectangleMethod(const std::function<T(T)>& f, T a, T b, T h) {
11 |     T res = 0;
12 |     int count = std::floor((b - a) / h);
13 |     T x = a;
14 |     for (int i = 0; i < count; ++i, x += h) {
15 |         res += f(x) * h;
16 |     }
17 |     return res;
18 | }
19 |
20 | template <class T>
21 | T TrapeziumMethod(const std::function<T(T)>& f, T a, T b, T h) {
22 |     T res = 0;
23 |     int count = std::floor((b - a) / h);
24 |     T prev_x = a, x = a + h;
25 |     for (int i = 0; i < count; ++i, prev_x = x, x += h) {
26 |         res += (f(prev_x) + f(x)) / 2 * h;
27 |     }
28 |     return res;
29 | }
30 |
31 | template <class T>
32 | T SimpsonMethod(const std::function<T(T)>& f, T a, T b, T h) {
33 |     T res = 0;
34 |     int count = std::floor((b - a) / h) / 2;
35 |     T x = a, next_x;
36 |     for (int i = 0; i < count; ++i, x = next_x) {
37 |         next_x = x + h + h;
38 |         Polynomial<T> p = LagrangePolynomial<T>({x, x+h, next_x}, {f(x), f(x+h), f(next_x)});
39 |         res += p.Integrate(x, next_x);
40 |     }
41 |     return res;
42 | }
43 |
44 | template <class T>
45 | T RungeRombergError(const std::function<T(std::function<T(T)>, T, T, T)>&
    IntegrationMethod, const std::function<T(T)>& f, T a, T b, T h, T r, int p) {
```



```

46 | T Ih = IntegrationMethod(f, a, b, h);
47 | T Irh = IntegrationMethod(f, a, b, r * h);
48 | return (Ih - Irh) / (std::pow(r, p) - 1);
49 | }
50 |
51 | template <class T>
52 | T RungeRombergError(const std::function<T(std::function<T(T)>, T, T, T)>&
    |     IntegrationMethod, const std::function<T(T)>& f, T Ih, T a, T b, T h, T r, int p)
    |     {
53 |     T Irh = IntegrationMethod(f, a, b, r * h);
54 |     return (Ih - Irh) / (std::pow(r, p) - 1);
55 | }

```

4 Численные методы решения обыкновенных дифференциальных уравнений

1 Решение задачи Коши для ОДУ второго порядка

1.1 Постановка задачи

Реализовать методы Эйлера, Рунге-Кутты и Адамса 4-го порядка в виде программ, задавая в качестве входных данных шаг сетки h . С использованием разработанного программного обеспечения решить задачу Коши для ОДУ 2-го порядка на указанном отрезке. Оценить погрешность численного решения с использованием метода Рунге – Ромберга и путем сравнения с точным решением.

Вариант:

$$y'' + y - 2 \cos x = 0$$

$$y(0) = 0$$

$$y'(0) = 1$$

$$x \in [0, 1], \quad h = 0.1$$

$$\text{Точное решение: } y = x \sin x + \cos x$$

1.2 Результаты работы

```
$ cat tests/1/1.txt
```

```
0 1
```

```
1
```

```
0
```

```
0.1
```

```
$ ./lab4_1 <tests/1/1.txt
```

Метод Эйлера:

```
1 1.005 1.01988 1.04418 1.07717 1.11783 1.16488 1.21681 1.27188 1.32817 1.38362
```

Погрешность: 0.00181351

Метод Рунге-Кутты:

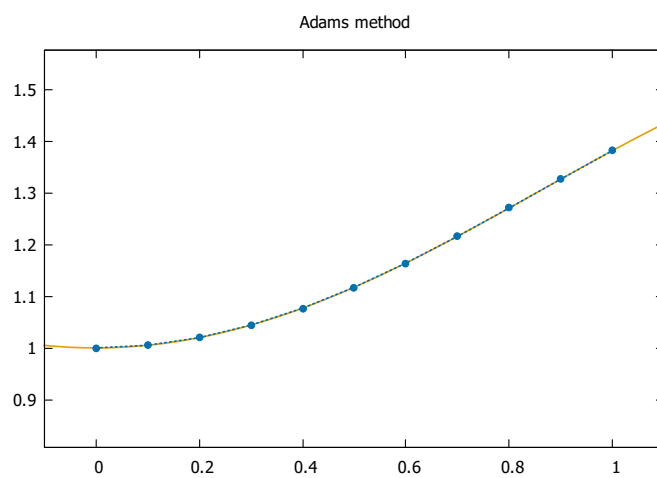
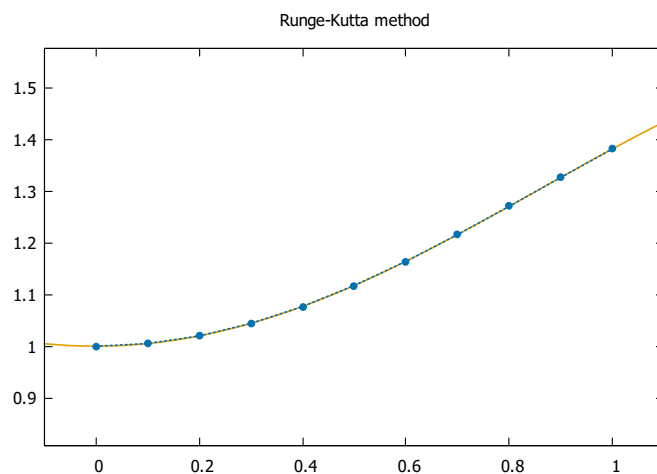
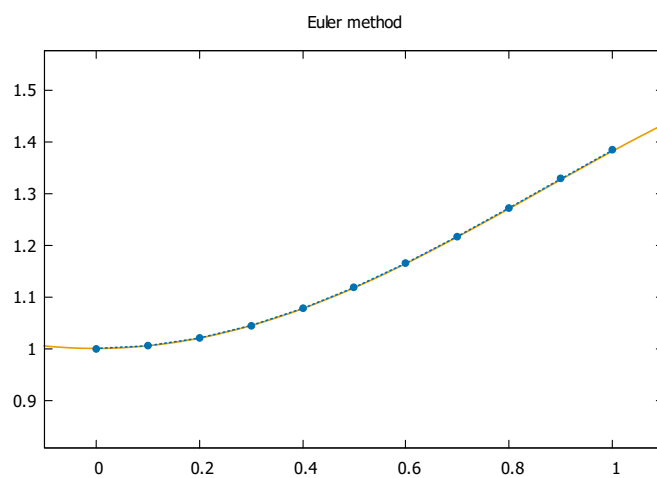
```
1 1.00499 1.0198 1.04399 1.07683 1.1173 1.16412 1.21579 1.27059 1.3266 1.38177
```

Погрешность: 1.11895e-06

Метод Адамса:

```
1 1.00499 1.0198 1.04399 1.07683 1.11729 1.16412 1.2158 1.27059 1.32661 1.38178
```

Погрешность: 1.60996e-06



1.3 Исходный код

```
1  #pragma once
2
3  #include <functional>
4  #include <vector>
5
6  template <class T>
7  using function_t = std::function<T(T, T, T)>;
8
9  const double EPS = 1e-6;
10
11 template <class T>
12 std::vector<std::vector<T>> EulerMethod(const function_t<T>& f, const function_t<T>& g
    , T y0, T z0, T start, T end, T h) {
13     std::vector<std::vector<T>> res(3, std::vector<T>());
14     T x = start, y = y0, z = z0;
15     res[0].push_back(x);
16     res[1].push_back(y);
17     res[2].push_back(z);
18
19     T pred_y, pred_z, next_x, next_y, next_z;
20     while (x + h < end || std::abs(end - x - h) < EPS) {
21         pred_y = y + h * f(x, y, z);
22         pred_z = z + h * g(x, y, z);
23
24         next_x = x + h;
25         next_y = y + h * (f(x, y, z) + f(next_x, pred_y, pred_z)) / 2;
26         next_z = z + h * (g(x, y, z) + g(next_x, pred_y, pred_z)) / 2;
27
28         x = next_x;
29         y = next_y;
30         z = next_z;
31
32         res[0].push_back(x);
33         res[1].push_back(y);
34         res[2].push_back(z);
35     }
36
37     return res;
38 }
39
40 template <class T>
41 std::vector<std::vector<T>> RungeKuttaMethod(const function_t<T>& f, const function_t<
    T>& g, T y0, T z0, T start, T end, T h) {
42     std::vector<std::vector<T>> res(3, std::vector<T>());
43     T x = start, y = y0, z = z0;
44     res[0].push_back(x);
45     res[1].push_back(y);
```

```

46     res[2].push_back(z);
47
48     T K1, K2, K3, K4, L1, L2, L3, L4;
49     while (x + h < end || std::abs(end - x - h) < EPS) {
50         K1 = h * f(x, y, z);
51         L1 = h * g(x, y, z);
52
53         K2 = h * f(x + h / 2, y + K1 / 2, z + L1 / 2);
54         L2 = h * g(x + h / 2, y + K1 / 2, z + L1 / 2);
55
56         K3 = h * f(x + h / 2, y + K2 / 2, z + L2 / 2);
57         L3 = h * g(x + h / 2, y + K2 / 2, z + L2 / 2);
58
59         K4 = h * f(x + h, y + K3, z + L3);
60         L4 = h * g(x + h, y + K3, z + L3);
61
62         x += h;
63         y += (K1 + 2 * K2 + 2 * K3 + K4) / 6;
64         z += (L1 + 2 * L2 + 2 * L3 + L4) / 6;
65
66         res[0].push_back(x);
67         res[1].push_back(y);
68         res[2].push_back(z);
69     }
70
71     return res;
72 }
73
74 template <class T>
75 std::vector<std::vector<T>> AdamsMethod(const function_t<T>& f, const function_t<T>& g
    , T y0, T z0, T start, T end, T h) {
76     std::vector<std::vector<T>> res = RungeKuttaMethod(f, g, y0, z0, start, start + 3 *
        h, h);
77     T x = res[0].back(), y = res[1].back(), z = res[2].back();
78
79     size_t k = 3;
80     T pred_y, pred_z, dy, dz;
81     while (x + h < end || std::abs(end - x - h) < EPS) {
82         pred_y = y + h * (55 * f(res[0][k], res[1][k], res[2][k]) -
83             59 * f(res[0][k-1], res[1][k-1], res[2][k-1]) +
84             37 * f(res[0][k-2], res[1][k-2], res[2][k-2]) -
85             9 * f(res[0][k-3], res[1][k-3], res[2][k-3])) / 24;
86
87         pred_z = z + h * (55 * g(res[0][k], res[1][k], res[2][k]) -
88             59 * g(res[0][k-1], res[1][k-1], res[2][k-1]) +
89             37 * g(res[0][k-2], res[1][k-2], res[2][k-2]) -
90             9 * g(res[0][k-3], res[1][k-3], res[2][k-3])) / 24;
91
92         dy = h * (9 * f(x + h, pred_y, pred_z) +

```

```

93         19 * f(res[0][k], res[1][k], res[2][k]) -
94         5 * f(res[0][k-1], res[1][k-1], res[2][k-1]) +
95             f(res[0][k-2], res[1][k-2], res[2][k-2])) / 24;
96
97     dz = h * (9 * g(x + h, pred_y, pred_z) +
98         19 * g(res[0][k], res[1][k], res[2][k]) -
99         5 * g(res[0][k-1], res[1][k-1], res[2][k-1]) +
100             g(res[0][k-2], res[1][k-2], res[2][k-2])) / 24;
101
102     x += h;
103     y += dy;
104     z += dz;
105     k += 1;
106
107     res[0].push_back(x);
108     res[1].push_back(y);
109     res[2].push_back(z);
110 }
111
112 return res;
113 }
114
115 template <class T>
116 T RungeRombergError(const std::vector<T>& uh, const std::vector<T>& urh, int p) {
117     double r = 2;
118     double coeff = 1. / (std::pow(r, p) - 1);
119     T err = 0;
120     for (size_t i = 0; i < urh.size(); ++i) {
121         err = std::max(err, std::abs(uh[2 * i] - urh[i]) * coeff);
122     }
123     return err;
124 }

```

2 Решение краевой задачи для ОДУ второго порядка

2.1 Постановка задачи

Реализовать метод стрельбы и конечно-разностный метод решения краевой задачи для ОДУ в виде программ. С использованием разработанного программного обеспечения решить краевую задачу для обыкновенного дифференциального уравнения 2-го порядка на указанном отрезке. Оценить погрешность численного решения с использованием метода Рунге – Ромберга и путем сравнения с точным решением.

Вариант:

$$xy'' + 2y' - xy = 0$$

$$y(1) = e^{-1}$$

$$y(2) = 0.5e^{-2}$$

$$\text{Точное решение: } y = \frac{e^{-x}}{x}$$

2.2 Результаты работы

0.1

Метод стрельбы:

0.367879 0.302612 0.250998 0.209643 0.176143 0.148756 0.126187 0.107462 0.0918336
0.0787208 0.0676676

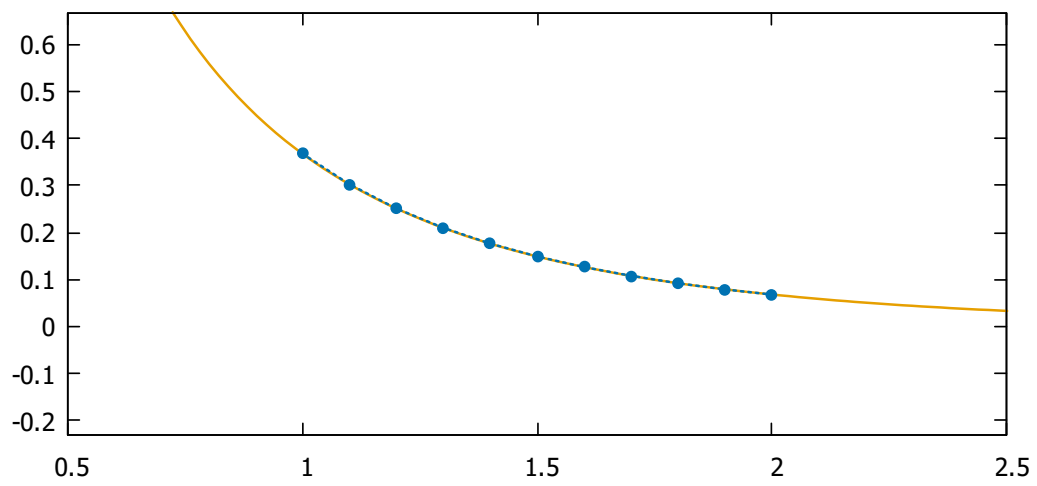
Погрешность: 1.69614e-05

Метод конечных разностей:

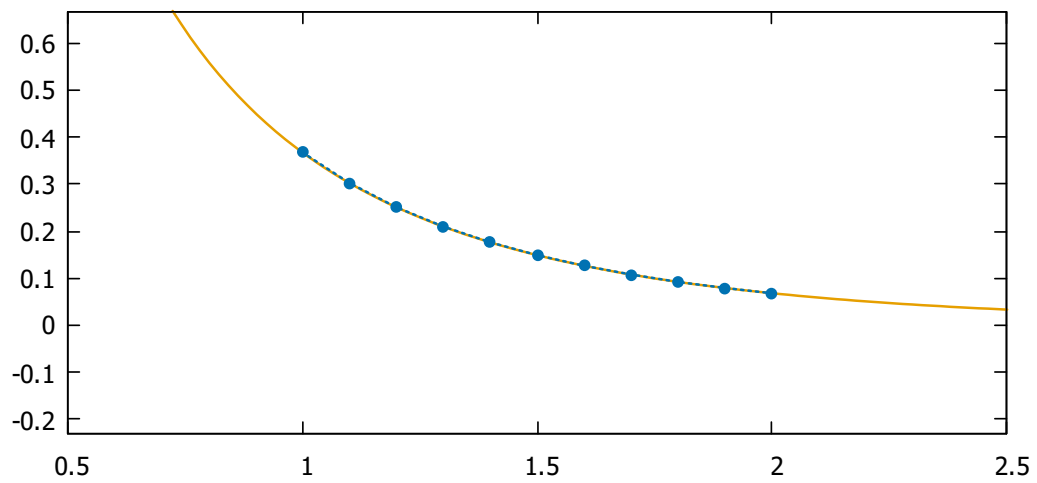
0.367879 0.302618 0.251008 0.209655 0.176156 0.148768 0.126198 0.107471 0.0918396
0.0787239 0.0676676

Погрешность: 1.51834e-05

Shooting method



Finite difference method



2.3 Исходный код

```
1 | #pragma once
2 |
3 | #include <functional>
4 | #include <vector>
5 |
6 | #include "cauchy_problem.hpp"
7 | #include "../linear/tridiagonal_matrix.hpp"
8 |
9 | template <class T>
10 | std::vector<std::vector<T>> ShootingMethod(const function_t<T>& f, const function_t<T>
    >& g, T y1, T y2, T start, T end, T h) {
11 |     std::function<T(T)> Phi = [y2](T y){ return y - y2; };
12 |
13 |     T mu0 = 1, mu1 = 2, phi0, phi1, dphi;
14 |     std::vector<std::vector<T>> solution = RungeKuttaMethod(f, g, y1, mu0, start, end, h
    );
15 |     phi0 = Phi(solution[1].back());
16 |     solution = RungeKuttaMethod(f, g, y1, mu1, start, end, h);
17 |     phi1 = Phi(solution[1].back());
18 |
19 |     while (std::abs(phi1) > EPS) {
20 |         dphi = (phi1 - phi0) / (mu1 - mu0);
21 |
22 |         mu0 = mu1;
23 |         mu1 -= phi1 / dphi;
24 |
25 |         phi0 = phi1;
26 |         solution = RungeKuttaMethod(f, g, y1, mu1, start, end, h);
27 |         phi1 = Phi(solution[1].back());
28 |     }
29 |
30 |     return solution;
31 | }
32 |
33 | template <class T>
34 | std::vector<std::vector<T>> FiniteDifferenceMethod(const std::function<T(T)>& p, const
    std::function<T(T)>& q, const std::function<T(T)>& r, T alpha1, T beta1, T gamma1
    , T alpha2, T beta2, T gamma2, T start, T end, T h) {
35 |     std::vector<std::vector<T>> res(2, std::vector<T>());
36 |     T x = start;
37 |     res[0].push_back(x);
38 |
39 |     std::vector<T> a, b, c, d;
40 |
41 |     a.push_back(0);
42 |     b.push_back(-alpha1 / h + beta1);
43 |     c.push_back(alpha1 / h);
```

```

44 | d.push_back(gamma1);
45 | while (x + 2 * h < end || std::abs(end - x - 2 * h) < EPS) {
46 |     x += h;
47 |     res[0].push_back(x);
48 |     a.push_back(1 / (h * h) - p(x) / (2 * h));
49 |     b.push_back(- 2 / (h * h) + q(x));
50 |     c.push_back(1 / (h * h) + p(x) / (2 * h));
51 |     d.push_back(-r(x));
52 | }
53 | x += h;
54 | res[0].push_back(x);
55 | a.push_back(-alpha2 / h);
56 | b.push_back(alpha2 / h + beta2);
57 | c.push_back(0);
58 | d.push_back(gamma2);
59 |
60 | TDMatrix<T> matrix(a, b, c);
61 | res[1] = matrix.Solve(d);
62 | return res;
63 | }

```