

Московский авиационный институт  
(национальный исследовательский университет)

Факультет информационных технологий и прикладной  
математики

Кафедра вычислительной математики и программирования

Лабораторные работы по курсу «Численные методы»

Студент: М. А. Инютин  
Преподаватель: Д. Л. Ревизников  
Группа: М8О-307Б-19  
Дата:  
Оценка:  
Подпись:

Москва, 2021

# 1 Вычислительные методы линейной алгебры

## 1 LU-разложение матриц. Метод Гаусса

### 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.

### 1.2 Консоль

```
$ make
g++ -g -pedantic -std=c++17 -Wall -Wextra -Werror main.cpp -o solution
$ cat tests/3.in
4
-7 3 -4 7
8 -1 -7 6
9 9 3 -6
-7 -9 -8 -5
-126 29 27 34
$ ./solution <tests/3.in
Решение системы:
x1 = 8.000000
x2 = -9.000000
x3 = 2.000000
x4 = -5.000000
Определитель матрицы: 16500.000000
Обратная матрица:
-0.054545,0.054545,0.006061,-0.018182
0.086000,-0.016000,0.082667,0.002000
-0.059818,-0.050182,-0.058909,-0.073273
0.017273,0.032727,-0.063030,-0.060909
```

### 1.3 Исходный код

```
1  #ifndef LU_HPP
2  #define LU_HPP
3
4  #include <algorithm>
5  #include <cmath>
6  #include <utility>
7
8  #include "../matrix.hpp"
9
10 template <class T>
11 class lu_t {
12     private:
13         using matrix = matrix_t<T>;
14         using vec = std::vector<T>;
15         using pii = std::pair<size_t, size_t>;
16
17         const T EPS = 1e-6;
18
19         matrix l;
20         matrix u;
21         T det;
22         std::vector<pii> swaps;
23
24     void decompose() {
25         size_t n = u.rows();
26         for (size_t i = 0; i < n; ++i) {
27             size_t max_el_ind = i;
28             for (size_t j = i + 1; j < n; ++j) {
29                 if (abs(u[j][i]) > abs(u[max_el_ind][i])) {
30                     max_el_ind = j;
31                 }
32             }
33             if (max_el_ind != i) {
34                 pii perm = std::make_pair(i, max_el_ind);
35                 swaps.push_back(perm);
36                 u.swap_rows(i, max_el_ind);
37                 l.swap_rows(i, max_el_ind);
38                 l.swap_cols(i, max_el_ind);
39             }
40             for (size_t j = i + 1; j < n; ++j) {
41                 if (abs(u[i][i]) < EPS) {
42                     continue;
43                 }
44                 T mu = u[j][i] / u[i][i];
45                 l[j][i] = mu;
46                 for (size_t k = 0; k < n; ++k) {
47                     u[j][k] -= mu * u[i][k];
```

```

48         }
49     }
50 }
51 det = (swaps.size() & 1 ? -1 : 1);
52 for (size_t i = 0; i < n; ++i) {
53     det *= u[i][i];
54 }
55 }
56
57 void do_swaps(vec& x) {
58     for (pii elem : swaps) {
59         std::swap(x[elem.first], x[elem.second]);
60     }
61 }
62
63 public:
64     lu_t(const matrix& matr) {
65         if (matr.rows() != matr.cols()) {
66             throw std::invalid_argument("Matrix is not square");
67         }
68         l = matrix::identity(matr.rows());
69         u = matrix(matr);
70         decompose();
71     }
72
73     friend std::ostream& operator<<(std::ostream& out, const lu_t<T>& lu) {
74         out << "Matrix L:\n" << lu.l << "Matrix U:\n" << lu.u;
75         return out;
76     }
77
78     T get_det() { return det; }
79
80     vec solve(vec b) {
81         int n = b.size();
82         do_swaps(b);
83         vec z(n);
84         for (int i = 0; i < n; ++i) {
85             T summary = b[i];
86             for (int j = 0; j < i; ++j) {
87                 summary -= z[j] * l[i][j];
88             }
89             z[i] = summary;
90         }
91         vec x(n);
92         for (int i = n - 1; i >= 0; --i) {
93             if (abs(u[i][i]) < EPS) {
94                 continue;
95             }
96             T summary = z[i];

```

```

97         for (int j = n - 1; j > i; --j) {
98             summary -= x[j] * u[i][j];
99         }
100         x[i] = summary / u[i][i];
101     }
102     return x;
103 }
104
105 matrix inv_matrix() {
106     size_t n = l.rows();
107     matrix res(n);
108     for (size_t i = 0; i < n; ++i) {
109         vec b(n);
110         b[i] = T(1);
111         vec x = solve(b);
112         for (size_t j = 0; j < n; ++j) {
113             res[j][i] = x[j];
114         }
115     }
116     return res;
117 }
118
119 ~lu_t() = default;
120 };
121
122 #endif /* LU_HPP */

```

## 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.

### 2.2 Консоль

```
$ cat tests/3.in
5
-7 -6
6 12 0
-3 5 0
-9 21 8
-5 -6
-75 126 13 -40 -24
$ make
g++ -g -pedantic -std=c++17 -Wall -Wextra -Werror main.cpp -o solution
$ cat tests/3.in
5
-7 -6
6 12 0
-3 5 0
-9 21 8
-5 -6
-75 126 13 -40 -24
$ ./solution <tests/3.in
Решение системы:
x1 = 3.000000
x2 = 9.000000
x3 = 8.000000
x4 = 0.000000
x5 = 4.000000
```

## 2.3 Исходный код

```
1  #ifndef TRIDIAG_HPP
2  #define TRIDIAG_HPP
3
4  #include <exception>
5  #include <iostream>
6  #include <vector>
7
8  template <class T>
9  class tridiag_t {
10     private:
11         using vec = std::vector<T>;
12
13         const double EPS = 1e-9;
14
15         int n;
16         vec a;
17         vec b;
18         vec c;
19
20     public:
21         tridiag_t(const int& _n) : n(_n), a(n), b(n), c(n) {}
22
23         tridiag_t(const vec& _a, const vec& _b, const vec& _c) {
24             if (!(_a.size() == _b.size() and _a.size() == _c.size())) {
25                 throw std::invalid_argument("Sizes of a, b, c are invalid");
26             }
27             n = _a.size();
28             a = _a;
29             b = _b;
30             c = _c;
31         }
32
33         friend std::istream& operator>>(std::istream& in, tridiag_t<T>& tridiag) {
34             in >> tridiag.b[0] >> tridiag.c[0];
35             for (int i = 1; i < tridiag.n - 1; ++i) {
36                 in >> tridiag.a[i] >> tridiag.b[i] >> tridiag.c[i];
37             }
38             in >> tridiag.a.back() >> tridiag.b.back();
39             return in;
40         }
41
42         vec solve(const vec& d) {
43             int m = d.size();
44             if (n != m) {
45                 throw std::invalid_argument("Size of vector d is invalid");
46             }
47             vec p(n);
```

```

48     p[0] = -c[0] / b[0];
49     vec q(n);
50     q[0] = d[0] / b[0];
51     for (int i = 1; i < n; ++i) {
52         p[i] = -c[i] / (b[i] + a[i] * p[i - 1]);
53         q[i] = (d[i] - a[i] * q[i - 1]) / (b[i] + a[i] * p[i - 1]);
54     }
55     vec x(n);
56     x.back() = q.back();
57     for (int i = n - 2; i >= 0; --i) {
58         x[i] = p[i] * x[i + 1] + q[i];
59     }
60     return x;
61 }
62
63 ~tridiag_t() = default;
64 };
65
66 #endif /* TRIDIAG_HPP */

```



## 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.

### 3.2 Консоль

```
$ make
g++ -g -pedantic -std=c++17 -Wall -Wextra -Werror main.cpp -o solution
$ cat tests/3.in
4 0.000000001
28 9 -3 -7
-5 21 -5 -3
-8 1 -16 5
0 -2 5 8
-159 63 -45 24
$ ./solution <tests/3.in
Метод простых итераций
Решени получено за 53 итераций
Решение системы:
x1 = -6.000000
x2 = 3.000000
x3 = 6.000000
x4 = 0.000000
Метод Зейделя
Решени получено за 20 итераций
Решение системы:
x1 = -6.000000
x2 = 3.000000
x3 = 6.000000
x4 = 0.000000
```

### 3.3 Исходный код

```
1  #ifndef ITERATION_HPP
2  #define ITERATION_HPP
3
4  #include <cmath>
5
6  #include "../matrix.hpp"
7
8  class iter_solver {
9  private:
10     using matrix_t = matrix<double>;
11     using vec_t = std::vector<double>;
12
13     matrix_t a;
14     size_t n;
15     double eps;
16
17     static constexpr double INF = 1e18;
18
19 public:
20     int iter_count;
21
22     iter_solver(const matrix_t& _a, double _eps = 1e-6) {
23         if (_a.rows() != _a.cols()) {
24             throw std::invalid_argument("Matrix is not square");
25         }
26         a = matrix_t(_a);
27         n = a.rows();
28         eps = _eps;
29     }
30
31     static double norm(const matrix_t& m) {
32         double res = -INF;
33         for (size_t i = 0; i < m.rows(); ++i) {
34             double s = 0;
35             for (double elem : m[i]) {
36                 s += std::abs(elem);
37             }
38             res = std::max(res, s);
39         }
40         return res;
41     }
42
43     static double norm(const vec_t& v) {
44         double res = -INF;
45         for (double elem : v) {
46             res = std::max(res, std::abs(elem));
47         }
48     }
49 }
```

```

48     return res;
49 }
50
51 std::pair<matrix, vec> precalc_ab(const vec& b, matrix& alpha, vec& beta) {
52     for (size_t i = 0; i < n; ++i) {
53         beta[i] = b[i] / a[i][i];
54         for (size_t j = 0; j < n; ++j) {
55             if (i != j) {
56                 alpha[i][j] = -a[i][j] / a[i][i];
57             }
58         }
59     }
60     return std::make_pair(alpha, beta);
61 }
62
63 vec solve_simple(const vec& b) {
64     matrix alpha(n);
65     vec beta(n);
66     precalc_ab(b, alpha, beta);
67     double eps_coef = 1.0;
68     if (norm(alpha) - 1.0 < eps) {
69         eps_coef = norm(alpha) / (1.0 - norm(alpha));
70     }
71     double eps_k = 1.0;
72     vec x(beta);
73     iter_count = 0;
74     while (eps_k > eps) {
75         vec x_k = beta + alpha * x;
76         eps_k = eps_coef * norm(x_k - x);
77         x = x_k;
78         ++iter_count;
79     }
80     return x;
81 }
82
83 vec zeidel(const vec& x, const matrix& alpha, const vec& beta) {
84     vec x_k(beta);
85     for (size_t i = 0; i < n; ++i) {
86         for (size_t j = 0; j < i; ++j) {
87             x_k[i] += x_k[j] * alpha[i][j];
88         }
89         for (size_t j = i; j < n; ++j) {
90             x_k[i] += x[j] * alpha[i][j];
91         }
92     }
93     return x_k;
94 }
95
96 vec solve_zeidel(const vec& b) {

```

```

97     matrix alpha(n);
98     vec beta(n);
99     precalc_ab(b, alpha, beta);
100    matrix c(n);
101    for (size_t i = 0; i < n; ++i) {
102        for (size_t j = i; j < n; ++j) {
103            c[i][j] = alpha[i][j];
104        }
105    }
106    double eps_coef = 1.0;
107    if (norm(alpha) - 1.0 < eps) {
108        eps_coef = norm(c) / (1.0 - norm(alpha));
109    }
110    double eps_k = 1.0;
111    vec x(beta);
112    iter_count = 0;
113    while (eps_k > eps) {
114        vec x_k = zeidel(x, alpha, beta);
115        eps_k = eps_coef * norm(x_k - x);
116        x = x_k;
117        ++iter_count;
118    }
119    return x;
120 }
121
122 ~iter_solver() = default;
123 };
124
125 #endif /* ITERATION_HPP */

```

## 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.

### 4.2 Консоль

```
$ make
g++ -g -pedantic -std=c++17 -Wall -Wextra -Werror main.cpp -o solution
$ cat tests/3.in
3 0.000001
-7 -6 8
-6 3 -7
8 -7 4
$ ./solution <tests/3.in
Собственные значения:
l_1 = -11.607818
l_2 = 15.020412
l_3 = -3.412593
Собственные векторы:
0.905671,-0.412378,0.098514
0.190483,0.603339,0.774402
-0.378784,-0.682588,0.624977
Решение получено за 7 итераций
```

### 4.3 Исходный код

```
1  #ifndef ROTATION_HPP
2  #define ROTATION_HPP
3
4  #include <cmath>
5
6  #include "../matrix.hpp"
7
8  class rotation {
9  private:
10     using matrix = matrix_t<double>;
11     using vec = std::vector<double>;
12
13     static constexpr double GLOBAL_EPS = 1e-9;
14
15     size_t n;
16     matrix a;
17     double eps;
18     matrix v;
19
20     static double norm(const matrix& m) {
21         double res = 0;
22         for (size_t i = 0; i < m.rows(); ++i) {
23             for (size_t j = 0; j < m.cols(); ++j) {
24                 if (i == j) {
25                     continue;
26                 }
27                 res += m[i][j] * m[i][j];
28             }
29         }
30         return std::sqrt(res);
31     }
32
33     double calc_phi(size_t i, size_t j) {
34         if (std::abs(a[i][i] - a[j][j]) < GLOBAL_EPS) {
35             return std::atan2(1.0, 1.0);
36         } else {
37             return 0.5 * std::atan2(2 * a[i][j], a[i][i] - a[j][j]);
38         }
39     }
40
41     matrix create_rotation(size_t i, size_t j, double phi) {
42         matrix u = matrix::identity(n);
43         u[i][i] = std::cos(phi);
44         u[i][j] = -std::sin(phi);
45         u[j][i] = std::sin(phi);
46         u[j][j] = std::cos(phi);
47         return u;
```

```

48     }
49
50     void build() {
51         iter_count = 0;
52         while (norm(a) > eps) {
53             ++iter_count;
54             size_t i = 0, j = 1;
55             for (size_t ii = 0; ii < n; ++ii) {
56                 for (size_t jj = 0; jj < n; ++jj) {
57                     if (ii == jj) {
58                         continue;
59                     }
60                     if (std::abs(a[ii][jj]) > std::abs(a[i][j])) {
61                         i = ii;
62                         j = jj;
63                     }
64                 }
65             }
66             double phi = calc_phi(i, j);
67             matrix u = create_rotation(i, j, phi);
68             v = v * u;
69             a = u.t() * a * u;
70         }
71     }
72
73 public:
74     int iter_count;
75
76     rotation(const matrix& _a, double _eps) {
77         if (_a.rows() != _a.cols()) {
78             throw std::invalid_argument("Matrix is not square");
79         }
80         a = matrix(_a);
81         n = a.rows();
82         eps = _eps;
83         v = matrix::identity(n);
84         build();
85     };
86
87     matrix get_eigen_vectors() { return v; }
88
89     vec get_eigen_values() {
90         vec res(n);
91         for (size_t i = 0; i < n; ++i) {
92             res[i] = a[i][i];
93         }
94         return res;
95     }
96

```

```
97 | ~rotation() = default;  
98 | };  
99 |  
100 | #endif /* ROTATION_HPP */
```



## 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.

### 5.2 Консоль

```
$ make
g++ -g -pedantic -std=c++17 -Wall -Wextra -Werror main.cpp -o solution
$ cat tests/3.in
3 0.000001
-1 4 -4
2 -5 0
-8 -2 0
$ ./solution <tests/3.in
Решение получено за 32 итераций
Собственные значения:
l_1 = -7.547969
l_2 = 5.664787
l_3 = -4.116817
```

### 5.3 Исходный код

```
1 | #ifndef QR_ALGO_HPP
2 | #define QR_ALGO_HPP
3 |
4 | #include <cmath>
5 | #include <complex>
6 |
7 | #include "../matrix.hpp"
8 |
9 | class qr_algo {
10 |     private:
11 |         using matrix_t = matrix_t<double>;
12 |         using vec = std::vector<double>;
13 |         using complex = std::complex<double>;
14 |         using pcc = std::pair<complex, complex>;
15 |         using vec_complex = std::vector<complex>;
16 |
17 |         static constexpr double INF = 1e18;
18 |         static constexpr complex COMPLEX_INF = complex(INF, INF);
19 |
20 |         size_t n;
21 |         matrix a;
22 |         double eps;
23 |         vec_complex eigen;
24 |
25 |         double vtv(const vec& v) {
26 |             double res = 0;
27 |             for (double elem : v) {
28 |                 res += elem * elem;
29 |             }
30 |             return res;
31 |         }
32 |
33 |         double norm(const vec& v) { return std::sqrt(vtv(v)); }
34 |
35 |         matrix vvt(const vec& b) {
36 |             size_t n_b = b.size();
37 |             matrix res(n_b);
38 |             for (size_t i = 0; i < n_b; ++i) {
39 |                 for (size_t j = 0; j < n_b; ++j) {
40 |                     res[i][j] = b[i] * b[j];
41 |                 }
42 |             }
43 |             return res;
44 |         }
45 |
46 |         double sign(double x) {
47 |             if (x < eps) {
```

```

48         return -1.0;
49     } else if (x > eps) {
50         return 1.0;
51     } else {
52         return 0.0;
53     }
54 }
55
56 matrix householder(const vec& b, int id) {
57     vec v(b);
58     v[id] += sign(b[id]) * norm(b);
59     return matrix::identity(n) - (2.0 / vtv(v)) * vvt(v);
60 }
61
62 pcc solve_sq(double a11, double a12, double a21, double a22) {
63     double a = 1.0;
64     double b = -(a11 + a22);
65     double c = a11 * a22 - a12 * a21;
66     double d_sq = b * b - 4.0 * a * c;
67     if (d_sq > eps) {
68         complex bad(NAN, NAN);
69         return std::make_pair(bad, bad);
70     }
71     complex d(0.0, std::sqrt(-d_sq));
72     complex x1 = (-b + d) / (2.0 * a);
73     complex x2 = (-b - d) / (2.0 * a);
74     return std::make_pair(x1, x2);
75 }
76
77 bool check_diag() {
78     for (size_t i = 0; i < n; ++i) {
79         double col_sum = 0;
80         for (size_t j = i + 2; j < n; ++j) {
81             col_sum += a[j][i] * a[j][i];
82         }
83         double norm = std::sqrt(col_sum);
84         if (!(norm < eps)) {
85             return false;
86         }
87     }
88     return true;
89 }
90
91 void calc_eigen() {
92     for (size_t i = 0; i < n; ++i) {
93         if (i < n - 1 and !(abs(a[i + 1][i]) < eps)) {
94             auto [l1, l2] = solve_sq(a[i][i], a[i][i + 1], a[i + 1][i],
95                                     a[i + 1][i + 1]);
96             if (std::isnan(l1.real())) {

```

```

97         eigen[i] = COMPLEX_INF;
98         ++i;
99         eigen[i] = COMPLEX_INF;
100         continue;
101     }
102     eigen[i] = l1;
103     eigen[++i] = l2;
104 } else {
105     eigen[i] = a[i][i];
106 }
107 }
108 }
109
110 bool check_eps() {
111     if (!check_diag()) {
112         return false;
113     }
114     vec_complex prev_eigen(eigen);
115     calc_eigen();
116     for (size_t i = 0; i < n; ++i) {
117         bool bad = (std::norm(eigen[i] - COMPLEX_INF) < eps);
118         if (bad) {
119             return false;
120         }
121         double delta = std::norm(eigen[i] - prev_eigen[i]);
122         if (delta > eps) {
123             return false;
124         }
125     }
126     return true;
127 }
128
129 void build() {
130     iter_count = 0;
131     while (!check_eps()) {
132         ++iter_count;
133         matrix q = matrix::identity(n);
134         matrix r(a);
135         for (size_t i = 0; i < n - 1; ++i) {
136             vec b(n);
137             for (size_t j = i; j < n; ++j) {
138                 b[j] = r[j][i];
139             }
140             matrix h = householder(b, i);
141             q = q * h;
142             r = h * r;
143         }
144         a = r * q;
145     }

```

```

146     }
147
148 public:
149     int iter_count;
150
151     qr_algo(const matrix& _a, double _eps) {
152         if (_a.rows() != _a.cols()) {
153             throw std::invalid_argument("Matrix is not square");
154         }
155         n = _a.rows();
156         a = matrix(_a);
157         eps = _eps;
158         eigen.resize(n, COMPLEX_INF);
159         build();
160     };
161
162     vec_complex get_eigen_values() {
163         calc_eigen();
164         return eigen;
165     }
166
167     ~qr_algo() = default;
168 };
169
170 #endif /* QR_ALGO_HPP */

```

## 2 Численные методы решения 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.

#### 1.2 Консоль

```
$ make
g++ -g -pedantic -std=c++17 -Wall -Wextra -Werror main.cpp -o solution
$ cat tests/2.in
0.5 1 0.000001
$ ./solution <tests/2.in
x_0 = 0.774356940
Решение методом простой итерации получено за 9 итераций
x_0 = 0.774356593
Решение методом Ньютона получена за 5 итераций
$ cat tests/3.in
0.5 1 0.0000000001
$ ./solution <tests/3.in
x_0 = 0.774356594
Решение методом простой итерации получено за 14 итераций
x_0 = 0.774356593
Решение методом Ньютона получена за 5 итераций
```

### 1.3 Исходный код

```
1 | #ifndef SOLVER_HPP
2 | #define SOLVER_HPP
3 |
4 | #include <cmath>
5 |
6 | int iter_count = 0;
7 |
8 | double f(double x) { return std::sin(x) - 2.0 * x * x + 0.5; }
9 |
10 | double f_s(double x) { return std::cos(x) - 4.0 * x; }
11 |
12 | double f_ss(double x) { return -std::sin(x) - 4.0; }
13 |
14 | double phi(double x) { return std::sqrt(0.5 * std::sin(x) + 0.25); }
15 |
16 | double phi_s(double x) { return std::cos(x) / (4.0 * phi(x)); }
17 |
18 | double iter_solve(double l, double r, double eps) {
19 |     iter_count = 0;
20 |     double x_k = r;
21 |     double dx = 1.0;
22 |     double q = std::max(std::abs(phi_s(l)), std::abs(phi_s(r)));
23 |     double eps_coef = q / (1.0 - q);
24 |     do {
25 |         double x_k1 = phi(x_k);
26 |         dx = eps_coef * std::abs(x_k1 - x_k);
27 |         ++iter_count;
28 |         x_k = x_k1;
29 |     } while (dx > eps);
30 |     return x_k;
31 | }
32 |
33 | double newton_solve(double l, double r, double eps) {
34 |     double x0 = l;
35 |     if (!(f(x0) * f_ss(x0) > eps)) {
36 |         x0 = r;
37 |     }
38 |     iter_count = 0;
39 |     double x_k = x0;
40 |     double dx = 1.0;
41 |     do {
42 |         double x_k1 = x_k - f(x_k) / f_s(x_k);
43 |         dx = std::abs(x_k1 - x_k);
44 |         ++iter_count;
45 |         x_k = x_k1;
46 |     } while (dx > eps);
47 |     return x_k;
```

```
48 || }  
49 ||  
50 || #endif /* SOLVER_HPP */
```



## 2 Решение нелинейных систем 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.

### 2.2 Консоль

```
$ make
g++ -g -pedantic -std=c++17 -Wall -Wextra -Werror main.cpp -o solution
$ cat tests/2.in
0 1
1 2
0.000001
$ ./solution <tests/2.in
x_0 = 0.832187878,y0 = 1.739406197
Решение методом простой итерации получено за 77 итераций
x_0 = 0.832187922,y0 = 1.739406179
Решение методом Ньютона получена за 4 итераций
$ cat tests/3.in
0 1
1 2
0.000000001
$ ./solution <tests/3.in
x_0 = 0.832187922,y0 = 1.739406179
Решение методом простой итерации получено за 111 итераций
x_0 = 0.832187922,y0 = 1.739406179
Решение методом Ньютона получена за 4 итераций
```

## 2.3 Исходный код

```
1 | #ifndef SYSTEM_SOLVER_HPP
2 | #define SYSTEM_SOLVER_HPP
3 |
4 | #include "../lab1_1/lu.hpp"
5 |
6 | int iter_count = 0;
7 |
8 | const double a = 1;
9 |
10 | double phi1(double x1, double x2) {
11 |     (void)x1;
12 |     return a + std::cos(x2);
13 | }
14 |
15 | double phi1_s(double x1, double x2) {
16 |     (void)x1;
17 |     return -std::sin(x2);
18 | }
19 |
20 | double phi2(double x1, double x2) {
21 |     (void)x2;
22 |     return a + std::sin(x1);
23 | }
24 |
25 | double phi2_s(double x1, double x2) {
26 |     (void)x2;
27 |     return std::cos(x1);
28 | }
29 |
30 | double phi(double x1, double x2) { return phi1_s(x1, x2) * phi2_s(x1, x2); }
31 |
32 | using pdd = std::pair<double, double>;
33 |
34 | pdd iter_solve(double l1, double r1, double l2, double r2, double eps) {
35 |     iter_count = 0;
36 |     double x_1_k = r1;
37 |     double x_2_k = r2;
38 |     double q = -1;
39 |     q = std::max(q, std::abs(phi(l1, r1)));
40 |     q = std::max(q, std::abs(phi(l1, r2)));
41 |     q = std::max(q, std::abs(phi(l2, r1)));
42 |     q = std::max(q, std::abs(phi(l2, r2)));
43 |     double eps_coef = q / (1 - q);
44 |     double dx = 1;
45 |     do {
46 |         double x_1_k1 = phi1(x_1_k, x_2_k);
47 |         double x_2_k1 = phi2(x_1_k, x_2_k);
```

```

48         dx = eps_coef * (std::abs(x_1_k1 - x_1_k) + std::abs(x_2_k1 - x_2_k));
49         ++iter_count;
50         x_1_k = x_1_k1;
51         x_2_k = x_2_k1;
52     } while (dx > eps);
53     return std::make_pair(x_1_k, x_2_k);
54 }
55
56 using matrix = matrix_t<double>;
57 using lu = lu_t<double>;
58 using vec = std::vector<double>;
59
60 double f1(double x1, double x2) { return x1 - std::cos(x2) - a; }
61
62 double f2(double x1, double x2) { return x2 - std::sin(x1) - a; }
63
64 matrix j(double x1, double x2) {
65     matrix res(2);
66     res[0][0] = 1.0;
67     res[0][1] = std::sin(x2);
68     res[1][0] = -std::cos(x1);
69     res[1][1] = 1.0;
70     return res;
71 }
72
73 double norm(const vec& v) {
74     double res = 0;
75     for (double elem : v) {
76         res = std::max(res, std::abs(elem));
77     }
78     return res;
79 }
80
81 pdd newton_solve(double x1_0, double x2_0, double eps) {
82     iter_count = 0;
83     vec x_k = {x1_0, x2_0};
84     double dx = 1;
85     do {
86         double x1 = x_k[0];
87         double x2 = x_k[1];
88         lu jacobi(j(x1, x2));
89         vec f_k = {f1(x1, x2), f2(x1, x2)};
90         vec delta_x = jacobi.solve(f_k);
91         vec x_k1 = x_k - delta_x;
92         dx = norm(x_k1 - x_k);
93         ++iter_count;
94         x_k = x_k1;
95     } while (dx > eps);
96     return std::make_pair(x_k[0], x_k[1]);

```

```
97 || }  
98 ||  
99 || #endif /* SYSTEM_SOLVER_HPP */
```

## 3 Методы приближения функций

### 1 Полиномиальная интерполяция

#### 1.1 Постановка задачи

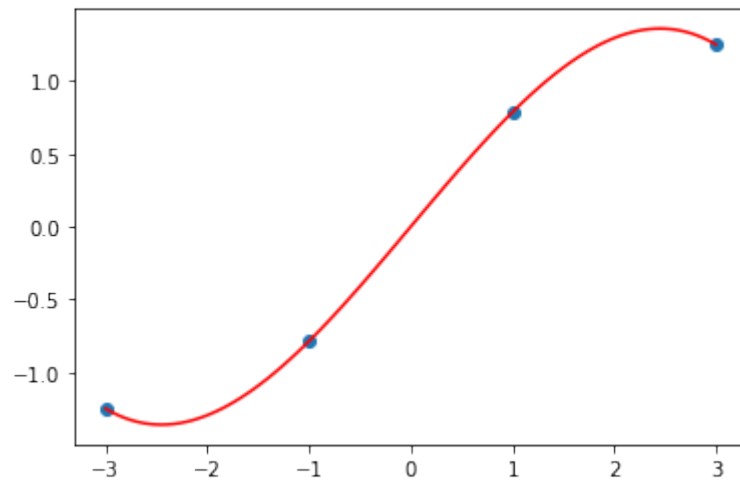
Используя таблицу значений  $Y_i$  функции  $y = f(x)$ , вычисленных в точках  $X_i$ ,  $i = 0, \dots, 3$  построить интерполяционные многочлены Лагранжа и Ньютона, проходящие через точки  $\{X_i, Y_i\}$ . Вычислить значение погрешности интерполяции в точке  $X^*$ .

#### 1.2 Консоль

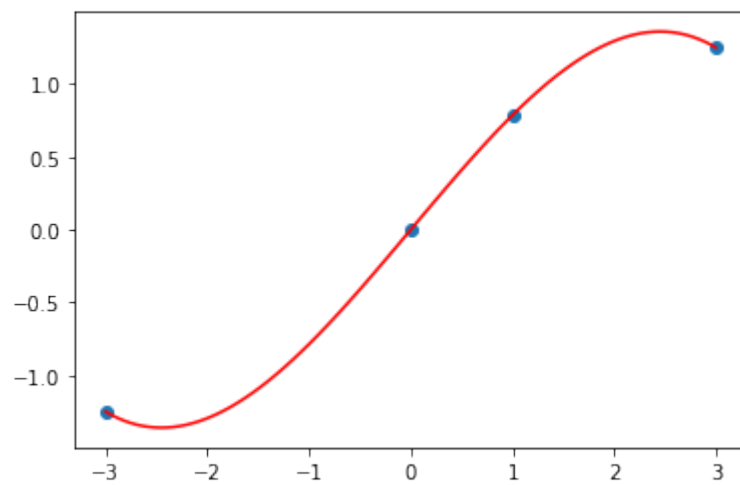
```
$ make
g++ -g -pedantic -std=c++17 -Wall -Wextra -Werror main.cpp -o solution
$ cat tests/1.in
4
-3 -1 1 3
-0.5
$ ./solution <tests/1.in
Интерполяционный многочлен Лагранжа: 0.831529 * x -0.0461312 * x ^ 3
Погрешность в точке X*: 0.0536493
Интерполяционный многочлен Ньютона: 0.831529 * x -0.0461312 * x ^ 3
Погрешность в точке X*: 0.0536493
$ cat tests/2.in
4
-3 0 1 3
-0.5
$ ./solution <tests/2.in
Интерполяционный многочлен Лагранжа: 0.831529 * x -0.0461312 * x ^ 3
Погрешность в точке X*: 0.0536493
Интерполяционный многочлен Ньютона: 0.831529 * x -0.0461312 * x ^ 3
Погрешность в точке X*: 0.0536493
```

### 1.3 Результат

Первый набор точек



Второй набор точек



## 1.4 Исходный код

```
1  #ifndef INTERPOLATOR_HPP
2  #define INTERPOLATOR_HPP
3
4  #include "../polynom.hpp"
5
6  using vec = std::vector<double>;
7
8  class inter_lagrange {
9      vec x;
10     vec y;
11     size_t n;
12
13 public:
14     inter_lagrange(const vec& _x, const vec& _y) : x(_x), y(_y), n(x.size()){};
15
16     polynom operator()() {
17         polynom res(vec({0}));
18         for (size_t i = 0; i < n; ++i) {
19             polynom li(vec({1}));
20             for (size_t j = 0; j < n; ++j) {
21                 if (i == j) {
22                     continue;
23                 }
24                 polynom xij(vec({-x[j], 1}));
25                 li = li * xij;
26                 li = li / (x[i] - x[j]);
27             }
28             res = res + y[i] * li;
29         }
30         return res;
31     }
32 };
33
34 class inter_newton {
35 private:
36     using vvd = std::vector<std::vector<double> >;
37     using vvb = std::vector<std::vector<bool> >;
38
39     vec x;
40     vec y;
41     size_t n;
42
43     vvd memo;
44     vvb calc;
45
46     double f(int l, int r) {
47         if (calc[l][r]) {
```

```

48         return memo[l][r];
49     }
50     calc[l][r] = true;
51     double res;
52     if (l + 1 == r) {
53         res = (y[l] - y[r]) / (x[l] - x[r]);
54     } else {
55         res = (f(l, r - 1) - f(l + 1, r)) / (x[l] - x[r]);
56     }
57     return memo[l][r] = res;
58 }
59
60 public:
61     inter_newton(const vec& _x, const vec& _y) : x(_x), y(_y), n(x.size()) {
62         memo.resize(n, std::vector<double>(n));
63         calc.resize(n, std::vector<bool>(n));
64     };
65
66     polynom operator()() {
67         polynom res(vec({y[0]}));
68         polynom li(vec({-x[0], 1}));
69         int r = 0;
70         for (size_t i = 1; i < n; ++i) {
71             res = res + f(0, ++r) * li;
72             li = li * polynom(vec({-x[i], 1}));
73         }
74         return res;
75     }
76 };
77
78 #endif /* INTERPOLATOR_HPP */

```



## 2 Сплайн-интерполяция

### 2.1 Постановка задачи

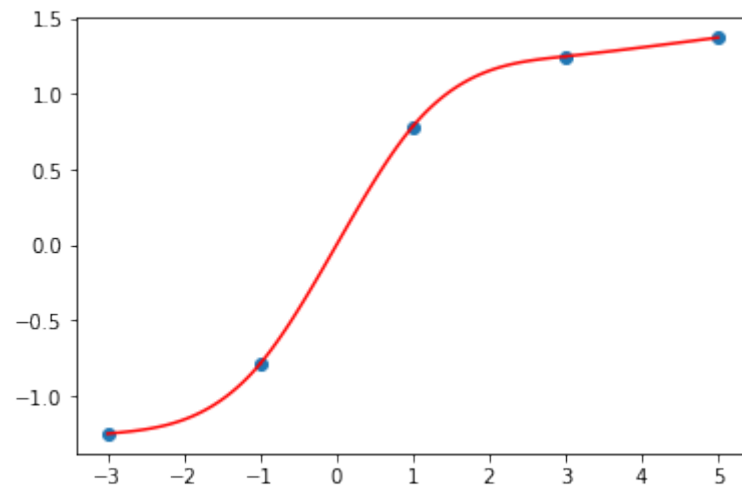
Построить кубический сплайн для функции, заданной в узлах интерполяции, предполагая, что сплайн имеет нулевую кривизну при  $x = x_0$  и  $x = x_4$ . Вычислить значение функции в точке  $x = X^*$ .

### 2.2 Консоль

```
$ make
g++ -g -pedantic -std=c++17 -Wall -Wextra -Werror main.cpp -o solution
$ cat tests/2.in
5
-3.0 -1.0 1.0 3.0 5.0
-1.2490 -0.78540 0.78540 1.2490 1.3734
-0.5
$ ./solution <tests/2.in
Полученные сплайны:
i = 1,a = -1.2490,b = 0.0470,c = 0.0000,d = 0.0462
i = 2,a = -0.7854,b = 0.6014,c = 0.2772,d = -0.0926
i = 3,a = 0.7854,b = 0.5990,c = -0.2784,d = 0.0474
i = 4,a = 1.2490,b = 0.0542,c = 0.0060,d = -0.0010
```

Значение функции в точке  $x_0 = -0.5000, f(x_0) = -0.4270$

### 2.3 Результат



## 2.4 Исходный код

```
1  #ifndef CUBIC_SPLINE_HPP
2  #define CUBIC_SPLINE_HPP
3
4  #include "../lab1_2/tridiag.hpp"
5
6  class cubic_spline_t {
7      using vec = std::vector<double>;
8      using tridiag = tridiag_t<double>;
9      size_t n;
10     vec x;
11     vec y;
12     vec a, b, c, d;
13
14     void build_spline() {
15         vec h(n + 1);
16         h[0] = NAN;
17         for (size_t i = 1; i <= n; ++i) {
18             h[i] = x[i] - x[i - 1];
19         }
20         vec eq_a(n - 1);
21         vec eq_b(n - 1);
22         vec eq_c(n - 1);
23         vec eq_d(n - 1);
24         for (size_t i = 2; i <= n; ++i) {
25             eq_a[i - 2] = h[i - 1];
26             eq_b[i - 2] = 2.0 * (h[i - 1] + h[i]);
27             eq_c[i - 2] = h[i];
28             eq_d[i - 2] = 3.0 * ((y[i] - y[i - 1]) / h[i] -
29                               (y[i - 1] - y[i - 2]) / h[i - 1]);
30         }
31         eq_a[0] = 0.0;
32         eq_c.back() = 0.0;
33         // for (size_t i = 0; i < n - 1; ++i) {
34         //     printf("%lf %lf %lf %lf\n", eq_a[i], eq_b[i], eq_c[i], eq_d[i]);
35         // }
36         tridiag system_of_eq(eq_a, eq_b, eq_c);
37         vec c_solved = system_of_eq.solve(eq_d);
38         for (size_t i = 2; i <= n; ++i) {
39             c[i] = c_solved[i - 2];
40         }
41         for (size_t i = 1; i <= n; ++i) {
42             a[i] = y[i - 1];
43         }
44         for (size_t i = 1; i < n; ++i) {
45             b[i] =
46                 (y[i] - y[i - 1]) / h[i] - h[i] * (c[i + 1] + 2.0 * c[i]) / 3.0;
47             d[i] = (c[i + 1] - c[i]) / (3.0 * h[i]);
```

```

48     }
49     c[1] = 0.0;
50     b[n] = (y[n] - y[n - 1]) / h[n] - (2.0 / 3.0) * h[n] * c[n];
51     d[n] = -c[n] / (3.0 * h[n]);
52 }
53
54 public:
55     cubic_spline_t(const vec& _x, const vec& _y) {
56         if (_x.size() != _y.size()) {
57             throw std::invalid_argument("Sizes does not match");
58         }
59         x = _x;
60         y = _y;
61         n = x.size() - 1;
62         a.resize(n + 1);
63         b.resize(n + 1);
64         c.resize(n + 1);
65         d.resize(n + 1);
66         build_spline();
67     }
68
69     friend std::ostream& operator<<(std::ostream& out,
70                                     const cubic_spline_t& spline) {
71         for (size_t i = 1; i <= spline.n; ++i) {
72             out << "i = " << i << ", a = " << spline.a[i]
73                 << ", b = " << spline.b[i] << ", c = " << spline.c[i]
74                 << ", d = " << spline.d[i] << '\n';
75         }
76         return out;
77     }
78
79     double operator()(double x0) {
80         for (size_t i = 1; i <= n; ++i) {
81             if (x[i - 1] <= x0 and x0 <= x[i]) {
82                 double x1 = x0 - x[i - 1];
83                 double x2 = x1 * x1;
84                 double x3 = x2 * x1;
85                 return a[i] + b[i] * x1 + c[i] * x2 + d[i] * x3;
86             }
87         }
88         return NAN;
89     }
90 };
91
92 #endif /* CUBIC_SPLINE_HPP */

```

## 3 Метод наименьших квадратов

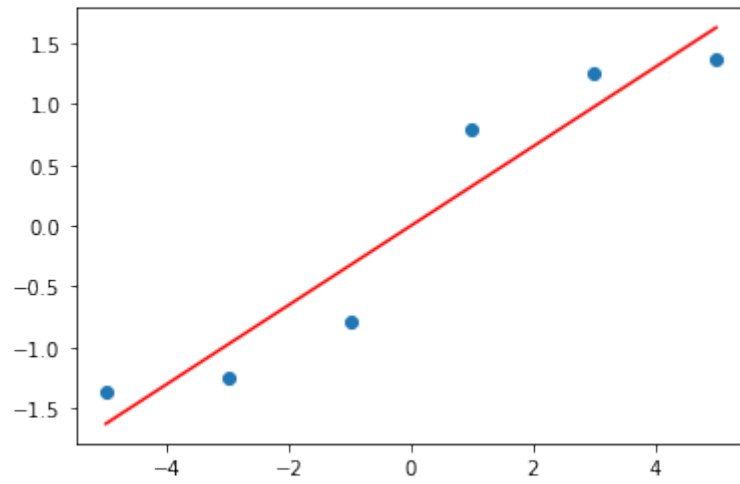
### 3.1 Постановка задачи

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.

### 3.2 Консоль

```
$ make
g++ -g -pedantic -std=c++17 -Wall -Wextra -Werror main.cpp -o solution
$ cat tests/2.in
6
-5.0 -3.0 -1.0 1.0 3.0 5.0
-1.3734 -1.249 -0.7854 0.7854 1.249 1.3734
$ ./solution <tests/2.in
Полученная функция первого порядка: 0.0000 0.3257
Значение суммы квадратов ошибок: 0.7007
Полученная функция второго порядка: 0.0000 0.3257 -0.0000
Значение суммы квадратов ошибок: 0.7007
```

### 3.3 Результат



### 3.4 Исходный код

```
1  #ifndef MINIMAL_SQUARE_HPP
2  #define MINIMAL_SQUARE_HPP
3
4  #include <functional>
5
6  #include "../lab1_1/lu.hpp"
7  #include "../polynom.hpp"
8
9  class minimal_square_t {
10     using vec = std::vector<double>;
11     using matrix = matrix_t<double>;
12     using lu = lu_t<double>;
13
14     using func = std::function<double(double)>;
15     using vf = std::vector<func>;
16
17     size_t n;
18     vec x;
19     vec y;
20     size_t m;
21     vec a;
22     vf phi;
23
24     void build() {
25         matrix lhs(n, m);
26         for (size_t i = 0; i < n; ++i) {
27             for (size_t j = 0; j < m; ++j) {
28                 lhs[i][j] = phi[j](x[i]);
29             }
30         }
31         matrix lhs_t = lhs.t();
32         lu lhs_lu(lhs_t * lhs);
33         vec rhs = lhs_t * y;
34         a = lhs_lu.solve(rhs);
35     }
36
37     double get(double x0) {
38         double res = 0.0;
39         for (size_t i = 0; i < m; ++i) {
40             res += a[i] * phi[i](x0);
41         }
42         return res;
43     }
44
45 public:
46     minimal_square_t(const vec& _x, const vec& _y, const vf& _phi) {
47         if (_x.size() != _y.size()) {
```

```

48         throw std::invalid_argument("Sizes does not match");
49     }
50     n = _x.size();
51     x = _x;
52     y = _y;
53     m = _phi.size();
54     a.resize(m);
55     phi = _phi;
56     build();
57 }
58
59 friend std::ostream& operator<<(std::ostream& out,
60                                const minimal_square_t& item) {
61     for (size_t i = 0; i < item.m; ++i) {
62         if (i) {
63             out << ' ';
64         }
65         out << item.a[i];
66     }
67     return out;
68 }
69
70 double mse() {
71     double res = 0;
72     for (size_t i = 0; i < n; ++i) {
73         res += std::pow(get(x[i]) - y[i], 2.0);
74     }
75     return res;
76 }
77
78 double operator()(double x0) { return get(x0); }
79 };
80
81 #endif /* MINIMAL_SQUARE_HPP */

```



## 4 Численное дифференцирование

### 4.1 Постановка задачи

Вычислить первую и вторую производную от таблично заданной функции  $y_i = f(x_i)$ ,  $i = 0, 1, 2, 3, 4$  в точке  $x = X^*$ .

### 4.2 Консоль

```
$ make
g++ -g -pedantic -std=c++17 -Wall -Wextra -Werror main.cpp -o solution
$ cat tests/2.in
5
0.0 0.5 1.0 1.5 2.0
0.0 0.97943 1.8415 2.4975 2.9093
1.0
$ ./solution <tests/2.in
Первая производная функции в точке x0 = 1.0000, f'(x0) = 1.5181
Вторая производная функции в точке x0 = 1.0000, f''(x0) = -0.8243
```

### 4.3 Исходный код

```
1  #ifndef TABLE_FUNCTION_HPP
2  #define TABLE_FUNCTION_HPP
3
4  #include <exception>
5  #include <vector>
6
7  const double EPS = 1e-9;
8
9  bool leq(double a, double b) { return (a < b) or (std::abs(b - a) < EPS); }
10
11 class table_function_t {
12     using vec = std::vector<double>;
13     size_t n;
14     vec x;
15     vec y;
16
17 public:
18     table_function_t(const vec& _x, const vec& _y) {
19         if (_x.size() != _y.size()) {
20             throw std::invalid_argument("Sizes does not match");
21         }
22         x = _x;
23         y = _y;
24         n = x.size();
25     }
26
27     double derivative1(double x0) {
28         for (size_t i = 0; i < n - 2; ++i) {
29             /* x in (x_i, x_{i+1}] */
30             if (x[i] < x0 and leq(x0, x[i + 1])) {
31                 double dydx1 = (y[i + 1] - y[i + 0]) / (x[i + 1] - x[i + 0]);
32                 double dydx2 = (y[i + 2] - y[i + 1]) / (x[i + 2] - x[i + 1]);
33                 double res = dydx1 + (dydx2 - dydx1) *
34                             (2.0 * x0 - x[i] - x[i + 1]) /
35                             (x[i + 2] - x[i]);
36                 return res;
37             }
38         }
39         return NAN;
40     }
41
42     double derivative2(double x0) {
43         for (size_t i = 0; i < n - 2; ++i) {
44             /* x in (x_i, x_{i+1}] */
45             if (x[i] < x0 and leq(x0, x[i + 1])) {
46                 double dydx1 = (y[i + 1] - y[i + 0]) / (x[i + 1] - x[i + 0]);
47                 double dydx2 = (y[i + 2] - y[i + 1]) / (x[i + 2] - x[i + 1]);
```

```

48 |         double res = 2.0 * (dydx2 - dydx1) / (x[i + 2] - x[i]);
49 |         return res;
50 |     }
51 | }
52 | return NAN;
53 | }
54 | };
55 |
56 | #endif /* TABLE_FUNCTION_HPP */

```

## 5 Численное интегрирование

### 5.1 Постановка задачи

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

### 5.2 Консоль

```
$ make
g++ -g -pedantic -std=c++17 -Wall -Wextra -Werror main.cpp -o solution
$ cat tests/1.in
0 2
0.5 0.25
$ ./solution <tests/1.in
Метод прямоугольников с шагом 0.5: 0.143739
Метод трапеций с шагом 0.5: 0.148748
Метод Симпсона с шагом 0.5: 0.145408

Метод прямоугольников с шагом 0.25: 0.144993
Метод трапеций с шагом 0.25: 0.146243
Метод Симпсона с шагом 0.25: 0.145409

Погрешность вычислений методом прямоугольников: 0.00167215
Погрешность вычислений методом трапеций: -0.0033396
Погрешность вычислений методом Симпсона: 1.56688e-06
```

### 5.3 Исходный код

```
1  #ifndef INTEGRATE_HPP
2  #define INTEGRATE_HPP
3
4  #include <cmath>
5
6  #include "../lab3_1/interpolator.hpp"
7
8  using func = double(double);
9
10 double integrate_rect(double l, double r, double h, func f) {
11     double x1 = l;
12     double x2 = l + h;
13     double res = 0;
14     while (x1 < r) {
15         res += h * f((x1 + x2) * 0.5);
16         x1 = x2;
17         x2 += h;
18     }
19     return res;
20 }
21
22 double integrate_trap(double l, double r, double h, func f) {
23     double x1 = l;
24     double x2 = l + h;
25     double res = 0;
26     while (x1 < r) {
27         res += h * (f(x1) + f(x2));
28         x1 = x2;
29         x2 += h;
30     }
31     return res * 0.5;
32 }
33
34 using vec = std::vector<double>;
35
36 double integrate_simp(double l, double r, double h, func f) {
37     double x1 = l;
38     double x2 = l + h;
39     double res = 0;
40     while (x1 < r) {
41         vec x = {x1, (x1 + x2) * 0.5, x2};
42         vec y = {f(x[0]), f(x[1]), f(x[2])};
43         inter_lagrange lagr(x, y);
44         res += lagr().integrate(x1, x2);
45         x1 = x2;
46         x2 += h;
47     }
48 }
```

```

48 |     return res;
49 | }
50 |
51 | inline double runge_romberg(double Fh, double Fkh, double k, double p) {
52 |     return (Fh - Fkh) / (std::pow(k, p) - 1.0);
53 | }
54 |
55 | #endif /* INTEGRATE_HPP */

```

## 4 Методы решения обыкновенных дифференциальных уравнений

### 1 Решение задачи Коши для ОДУ

#### 1.1 Постановка задачи

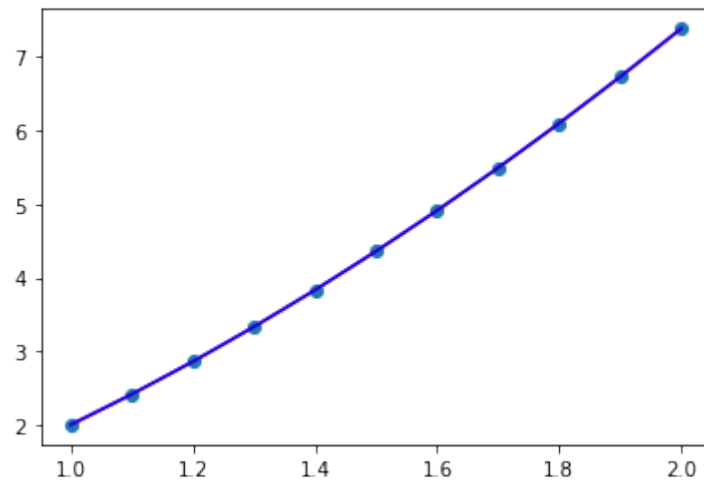
Реализовать методы Эйлера, Рунге-Кутты и Адамса 4-го порядка в виде программ, задавая в качестве входных данных шаг сетки  $h$ . С использованием разработанного программного обеспечения решить задачу Коши для ОДУ 2-го порядка на указанном отрезке. Оценить погрешность численного решения с использованием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

#### 1.2 Консоль

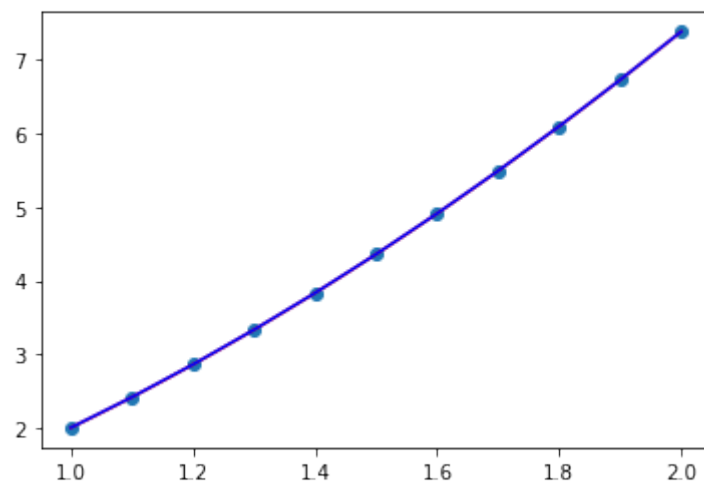
```
$ make
g++ -g -pedantic -std=c++17 -Wall -Wextra -Werror main.cpp -o solution
$ cat tests/1.in
1 2
2 4 0.1
$ ./solution <tests/1.in
Метод Эйлера:
x = [1.000000,1.100000,1.200000,1.300000,1.400000,1.500000,1.600000,1.700000,1.800000
y = [2.000000,2.414842,2.858788,3.331076,3.831064,4.358201,4.912010,5.492073,6.098021
Погрешность вычислений:
0.000006
Метод Рунге-Кутты:
x = [1.000000,1.100000,1.200000,1.300000,1.400000,1.500000,1.600000,1.700000,1.800000
y = [2.000000,2.414842,2.858788,3.331076,3.831064,4.358201,4.912010,5.492073,6.098021
Погрешность вычислений:
0.000000
Метод Адамса:
x = [1.000000,1.100000,1.200000,1.300000,1.400000,1.500000,1.600000,1.700000,1.800000
y = [2.000000,2.414842,2.858788,3.331076,3.831062,4.358197,4.912005,5.492067,6.098015
Погрешность вычислений:
0.000000
```

### 1.3 Результат

Метод Эйлера

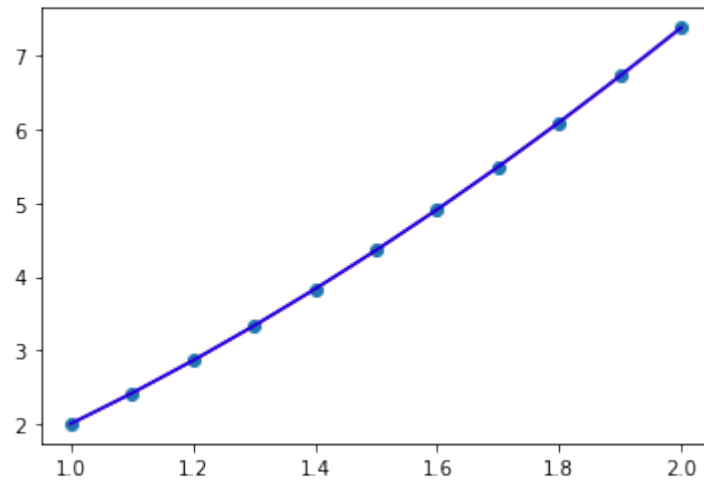


Метод Рунге-Кутты





Метод Адамса



## 1.4 Исходный код

```
1 | #ifndef SIMPLE_DESOLVE_HPP
2 | #define SIMPLE_DESOLVE_HPP
3 |
4 | #include <functional>
5 |
6 | #include "../de_utils.hpp"
7 |
8 | /* f(x, y, z) */
9 | using func = std::function<double(double, double, double)>;
10 | using vect = std::vector<tddd>;
11 | using vec = std::vector<double>;
12 |
13 | const double EPS = 1e-9;
14 |
15 | bool leq(double a, double b) { return (a < b) or (std::abs(b - a) < EPS); }
16 |
17 | class euler {
18 |     private:
19 |         double l, r;
20 |         func f, g;
21 |         double y0, z0;
22 |
23 |     public:
24 |         euler(const double _l, const double _r, const func _f, const func _g,
25 |             const double _y0, const double _z0)
26 |             : l(_l), r(_r), f(_f), g(_g), y0(_y0), z0(_z0) {}
27 |
28 |         vect solve(double h) {
29 |             vect res;
30 |             double xk = l;
31 |             double yk = y0;
32 |             double zk = z0;
33 |             res.push_back(std::make_tuple(xk, yk, zk));
34 |             while (leq(xk + h, r)) {
35 |                 double dy = h * f(xk, yk, zk);
36 |                 double dz = h * g(xk, yk, zk);
37 |                 xk += h;
38 |                 yk += dy;
39 |                 zk += dz;
40 |                 res.push_back(std::make_tuple(xk, yk, zk));
41 |             }
42 |             return res;
43 |         }
44 | };
45 |
46 | class runge {
47 |     private:
```

```

48     double l, r;
49     func f, g;
50     double y0, z0;
51
52 public:
53     runge(const double _l, const double _r, const func _f, const func _g,
54           const double _y0, const double _z0)
55         : l(_l), r(_r), f(_f), g(_g), y0(_y0), z0(_z0) {}
56
57     vect solve(double h) {
58         vect res;
59         double xk = l;
60         double yk = y0;
61         double zk = z0;
62         res.push_back(std::make_tuple(xk, yk, zk));
63         while (leq(xk + h, r)) {
64             double K1 = h * f(xk, yk, zk);
65             double L1 = h * g(xk, yk, zk);
66             double K2 = h * f(xk + 0.5 * h, yk + 0.5 * K1, zk + 0.5 * L1);
67             double L2 = h * g(xk + 0.5 * h, yk + 0.5 * K1, zk + 0.5 * L1);
68             double K3 = h * f(xk + 0.5 * h, yk + 0.5 * K2, zk + 0.5 * L2);
69             double L3 = h * g(xk + 0.5 * h, yk + 0.5 * K2, zk + 0.5 * L2);
70             double K4 = h * f(xk + h, yk + K3, zk + L3);
71             double L4 = h * g(xk + h, yk + K3, zk + L3);
72             double dy = (K1 + 2.0 * K2 + 2.0 * K3 + K4) / 6.0;
73             double dz = (L1 + 2.0 * L2 + 2.0 * L3 + L4) / 6.0;
74             xk += h;
75             yk += dy;
76             zk += dz;
77             res.push_back(std::make_tuple(xk, yk, zk));
78         }
79         return res;
80     }
81 };
82
83 class adams {
84 private:
85     double l, r;
86     func f, g;
87     double y0, z0;
88
89 public:
90     adams(const double _l, const double _r, const func _f, const func _g,
91           const double _y0, const double _z0)
92         : l(_l), r(_r), f(_f), g(_g), y0(_y0), z0(_z0) {}
93
94     double calc_tuple(func f, tddd xyz) {
95         return f(std::get<0>(xyz), std::get<1>(xyz), std::get<2>(xyz));
96     }

```

```

97
98 vect solve(double h) {
99     if (1 + 3.0 * h > r) {
100         throw std::invalid_argument("h is too big");
101     }
102     runge first_points(1, 1 + 3.0 * h, f, g, y0, z0);
103     vect res = first_points.solve(h);
104     size_t cnt = res.size();
105     double xk = std::get<0>(res.back());
106     double yk = std::get<1>(res.back());
107     double zk = std::get<2>(res.back());
108     while (leq(xk + h, r)) {
109         /* Predictor */
110         double dy = (h / 24.0) * (55.0 * calc_tuple(f, res[cnt - 1]) -
111                                   59.0 * calc_tuple(f, res[cnt - 2]) +
112                                   37.0 * calc_tuple(f, res[cnt - 3]) -
113                                   9.0 * calc_tuple(f, res[cnt - 4]));
114         double dz = (h / 24.0) * (55.0 * calc_tuple(g, res[cnt - 1]) -
115                                   59.0 * calc_tuple(g, res[cnt - 2]) +
116                                   37.0 * calc_tuple(g, res[cnt - 3]) -
117                                   9.0 * calc_tuple(g, res[cnt - 4]));
118         double xk1 = xk + h;
119         double yk1 = yk + dy;
120         double zk1 = zk + dz;
121         res.push_back(std::make_tuple(xk1, yk1, zk1));
122         ++cnt;
123         /* Corrector */
124         dy = (h / 24.0) * (9.0 * calc_tuple(f, res[cnt - 1]) +
125                           19.0 * calc_tuple(f, res[cnt - 2]) -
126                           5.0 * calc_tuple(f, res[cnt - 3]) +
127                           1.0 * calc_tuple(f, res[cnt - 4]));
128         dz = (h / 24.0) * (9.0 * calc_tuple(g, res[cnt - 1]) +
129                           19.0 * calc_tuple(g, res[cnt - 2]) -
130                           5.0 * calc_tuple(g, res[cnt - 3]) +
131                           1.0 * calc_tuple(g, res[cnt - 4]));
132         xk += h;
133         yk += dy;
134         zk += dz;
135         res.pop_back();
136         res.push_back(std::make_tuple(xk, yk, zk));
137     }
138     return res;
139 }
140 };
141
142 double runge_romberg(const vect& y_2h, const vect& y_h, double p) {
143     double coef = 1.0 / (std::pow(2, p) - 1.0);
144     double res = 0.0;
145     for (size_t i = 0; i < y_2h.size(); ++i) {

```

```

146 |         res = std::max(res, coef * std::abs(std::get<1>(y_2h[i]) -
147 |                                           std::get<1>(y_h[2 * i])));
148 |     }
149 |     return res;
150 | }
151 |
152 | #endif /* SIMPLE_DESOLVE_HPP */

```

## 2 Решение краевых задач

### 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.

### 2.2 Консоль

```
$ make
```

```
g++ -g -pedantic -std=c++17 -Wall -Wextra -Werror main.cpp -o solution
```

```
$ cat tests/1.in
```

```
0.1 0.0001
```

```
$ ./solution <tests/1.in
```

Метод стрельбы:

```
x = [1.000000,1.100000,1.200000,1.300000,1.400000,1.500000,1.600000,1.700000,1.800000
```

```
y = [3.082723,3.898207,4.896683,6.111204,7.580066,9.347569,11.464879,13.991012,16.993
```

Погрешность вычислений:

```
0.142434
```

Конечно-разностный метод:

```
x = [1.000000,1.100000,1.200000,1.300000,1.400000,1.500000,1.600000,1.700000,1.800000
```

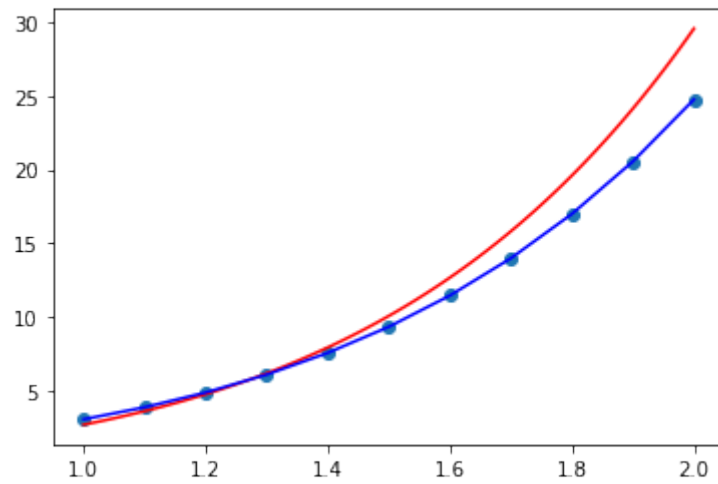
```
y = [1.171219,1.986704,3.035416,4.365472,6.033397,8.105462,10.659212,13.785218,17.589
```

Погрешность вычислений:

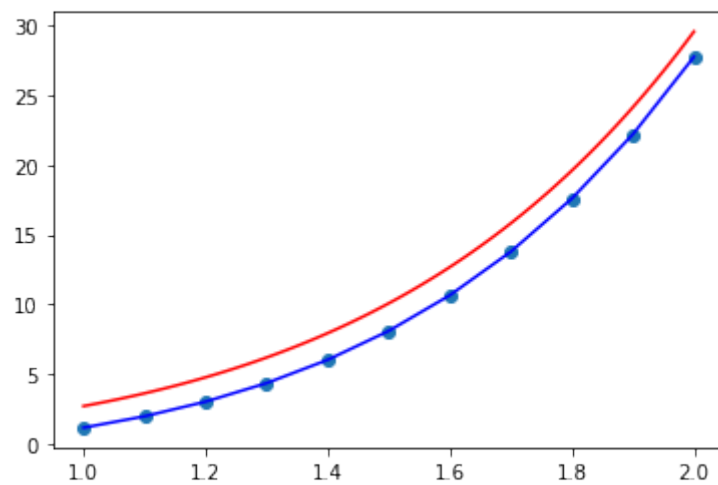
```
0.342752
```

## 2.3 Результат

Метод стрельбы



Конечно-разностный метод



## 2.4 Исходный код

```
1  #ifndef BOUNDARY_SOLVER_HPP
2  #define BOUNDARY_SOLVER_HPP
3
4  #include <cmath>
5
6  #include "../lab1_2/tridiag.hpp"
7  #include "../lab4_1/simple_desolve.hpp"
8
9  class shooting {
10 private:
11     double a, b;
12     func f, g;
13     double alpha, beta, y0;
14     double delta, gamma, y1;
15
16 public:
17     shooting(const double _a, const double _b, const func _f, const func _g,
18             const double _alpha, const double _beta, const double _y0,
19             const double _delta, const double _gamma, const double _y1)
20     : a(_a),
21       b(_b),
22       f(_f),
23       g(_g),
24       alpha(_alpha),
25       beta(_beta),
26       y0(_y0),
27       delta(_delta),
28       gamma(_gamma),
29       y1(_y1) {}
30
31     double get_start_cond(double eta) { return (y0 - alpha * eta) / beta; }
32
33     double get_eta_next(double eta_prev, double eta, const vect sol_prev,
34                        const vect sol) {
35         double yb_prev = std::get<1>(sol_prev.back());
36         double zb_prev = std::get<2>(sol_prev.back());
37         double phi_prev = delta * yb_prev + gamma * zb_prev - y1;
38         double yb = std::get<1>(sol.back());
39         double zb = std::get<2>(sol.back());
40         double phi = delta * yb + gamma * zb - y1;
41         return eta - (eta - eta_prev) / (phi - phi_prev) * phi;
42     }
43
44     vect solve(double h, double eps) {
45         double eta_prev = 1.0;
46         double eta = 0.8;
47         while (1) {
```



```

48     double runge_z0_prev = get_start_cond(eta_prev);
49     euler de_solver_prev(a, b, f, g, eta_prev, runge_z0_prev);
50     vect sol_prev = de_solver_prev.solve(h);
51
52     double runge_z0 = get_start_cond(eta);
53     euler de_solver(a, b, f, g, eta, runge_z0);
54     vect sol = de_solver.solve(h);
55
56     double eta_next = get_eta_next(eta_prev, eta, sol_prev, sol);
57     if (std::abs(eta_next - eta) < eps) {
58         return sol;
59     } else {
60         eta_prev = eta;
61         eta = eta_next;
62     }
63 }
64 }
65 };
66
67 class fin_dif {
68 private:
69     using fx = std::function<double(double)>;
70     using tridiag = tridiag_t<double>;
71
72     double a, b;
73     fx p, q, f;
74     double alpha, beta, y0;
75     double delta, gamma, y1;
76
77 public:
78     fin_dif(const double _a, const double _b, const fx _p, const fx _q,
79             const fx _f, const double _alpha, const double _beta,
80             const double _y0, const double _delta, const double _gamma,
81             const double _y1)
82     : a(_a),
83       b(_b),
84       p(_p),
85       q(_q),
86       f(_f),
87       alpha(_alpha),
88       beta(_beta),
89       y0(_y0),
90       delta(_delta),
91       gamma(_gamma),
92       y1(_y1) {}
93
94     vect solve(double h) {
95         size_t n = (b - a) / h;
96         vec xk(n + 1);

```

```

97     for (size_t i = 0; i <= n; ++i) {
98         xk[i] = a + h * i;
99     }
100     vec a(n + 1);
101     vec b(n + 1);
102     vec c(n + 1);
103     vec d(n + 1);
104     b[0] = h * alpha - beta;
105     c[0] = beta;
106     d[0] = h * y0;
107     a.back() = -gamma;
108     b.back() = h * delta + gamma;
109     d.back() = h * y1;
110     for (size_t i = 1; i < n; ++i) {
111         a[i] = 1.0 - p(xk[i]) * h * 0.5;
112         b[i] = -2.0 + h * h * q(xk[i]);
113         c[i] = 1.0 + p(xk[i]) * h * 0.5;
114         d[i] = h * h * f(xk[i]);
115     }
116     tridiag sys_eq(a, b, c);
117     vec yk = sys_eq.solve(d);
118     vect res;
119     for (size_t i = 0; i <= n; ++i) {
120         res.push_back(std::make_tuple(xk[i], yk[i], NAN));
121     }
122     return res;
123 }
124 };
125
126 #endif /* BOUNDARY_SOLVER_HPP */

```