# 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, 2022**

# 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

```
#ifndef LU_HPP
#define LU_HPP

#include "../matrix.hpp"

#include <algorithm>
#include <cmath>
#include <utility>

template<class T>
class lu_t {
private:
    using matrix = matrix_t<T>;
    using vec = std::vector<T>;
    using pii = std::pair<size_t, size_t>;

    const T EPS = 1e-6;

    matrix l;
    matrix u;
    T det;
    std::vector<pii> swaps;

    void decompose() {
        size_t n = u.rows();
        for (size_t i = 0; i < n; ++i) {
            size_t max_el_ind = i;
            for (size_t j = i + 1; j < n; ++j) {
                if (abs(u[j][i]) > abs(u[max_el_ind][i])) {
                    max_el_ind = j;
                }
            }
            if (max_el_ind != i) {
                pii perm = std::make_pair(i, max_el_ind);
                swaps.push_back(perm);
                u.swap_rows(i, max_el_ind);
                l.swap_rows(i, max_el_ind);
                l.swap_cols(i, max_el_ind);
            }
            for (size_t j = i + 1; j < n; ++j) {
                if (abs(u[i][i]) < EPS) {
                    continue;
                }
                T mu = u[j][i] / u[i][i];
                l[j][i] = mu;
                for (size_t k = 0; k < n; ++k) {
                    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    }
62public:
63    lu_t(const matrix & matr) {
64        if (matr.rows() != matr.cols()) {
65            throw std::invalid_argument("Matrix is not square");
66        }
67        l = matrix::identity(matr.rows());
68        u = matrix(matr);
69        decompose();
70    }
71
72    friend std::ostream & operator << (std::ostream & out, const lu_t<T> & lu) {
73        out << "Matrix L:\n" << lu.l << "Matrix U:\n" << lu.u;
74        return out;
75    }
76
77    T get_det() {
78        return det;
79    }
80
81    vec solve(vec b) {
82        int n = b.size();
83        do_swaps(b);
84        vec z(n);
85        for (int i = 0; i < n; ++i) {
86            T summary = b[i];
87            for (int j = 0; j < i; ++j) {
88                summary -= z[j] * l[i][j];
89            }
90            z[i] = summary;
91        }
92        vec x(n);
93        for (int i = n - 1; i >= 0; --i) {
94            if (abs(u[i][i]) < EPS) {
95                continue;
96            }
```

```
97              T summary = z[i];
98              for (int j = n - 1; j > i; --j) {
99                  summary -= x[j] * u[i][j];
100             }
101             x[i] = summary / u[i][i];
102         }
103         return x;
104     }
105
106     matrix inv_matrix() {
107         size_t n = l.rows();
108         matrix res(n);
109         for (size_t i = 0; i < n; ++i) {
110             vec b(n);
111             b[i] = T(1);
112             vec x = solve(b);
113             for (size_t j = 0; j < n; ++j) {
114                 res[j][i] = x[j];
115             }
116         }
117         return res;
118     }
119
120     ~lu_t() = default;
121 };
122
123 #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

```
#ifndef TRIDIAG_HPP
#define TRIDIAG_HPP

#include <exception>
#include <iostream>
#include <vector>

template<class T>
class tridiag_t {
private:
    using vec = std::vector<T>;

    const T EPS = 1e-6;

    int n;
    vec a;
    vec b;
    vec c;
public:
    tridiag_t(const int & _n) : n(_n), a(n), b(n), c(n) {}

    tridiag_t(const vec & _a, const vec & _b, const vec & _c) {
        if (!(_a.size() == _b.size() and _a.size() == _c.size())) {
            throw std::invalid_argument("Sizes of a, b, c are invalid");
        }
        n = _a.size();
        a = _a;
        b = _b;
        c = _c;
    }

    friend std::istream & operator >> (std::istream & in, tridiag_t<T> & tridiag) {
        in >> tridiag.b[0] >> tridiag.c[0];
        for (int i = 1; i < tridiag.n - 1; ++i) {
            in >> tridiag.a[i] >> tridiag.b[i] >> tridiag.c[i];
        }
        in >> tridiag.a.back() >> tridiag.b.back();
        return in;
    }

    vec solve(const vec & d) {
        int m = d.size();
        if (n != m) {
            throw std::invalid_argument("Size of vector d is invalid");
        }
        vec p(n);
        p[0] = -c[0] / b[0];
```

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

```cpp
#ifndef ITERATION_HPP
#define ITERATION_HPP

#include <cmath>
#include "../matrix.hpp"

class iter_solver {
private:
    using matrix = matrix_t<double>;
    using vec = std::vector<double>;

    matrix a;
    size_t n;
    double eps;

    static constexpr double INF = 1e18;
public:
    int iter_count;

    iter_solver(const matrix & _a, double _eps = 1e-6) {
        if (_a.rows() != _a.cols()) {
            throw std::invalid_argument("Matrix is not square");
        }
        a = matrix(_a);
        n = a.rows();
        eps = _eps;
    }

    static double norm(const matrix & m) {
        double res = -INF;
        for (size_t i = 0; i < m.rows(); ++i) {
            double s = 0;
            for (double elem : m[i]) {
                s += std::abs(elem);
            }
            res = std::max(res, s);
        }
        return res;
    }

    static double norm(const vec & v) {
        double res = -INF;
        for (double elem : v) {
            res = std::max(res, std::abs(elem));
        }
        return res;
    }
```

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

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

```
#ifndef ROTATION_HPP
#define ROTATION_HPP

#include <cmath>
#include "../matrix.hpp"

class rotation {
private:
    using matrix = matrix_t<double>;
    using vec = std::vector<double>;

    static constexpr double GLOBAL_EPS = 1e-9;

    size_t n;
    matrix a;
    double eps;
    matrix v;

    static double norm(const matrix & m) {
        double res = 0;
        for (size_t i = 0; i < m.rows(); ++i) {
            for (size_t j = 0; j < m.cols(); ++j) {
                if (i == j) {
                    continue;
                }
                res += m[i][j] * m[i][j];
            }
        }
        return std::sqrt(res);
    }

    double calc_phi(size_t i, size_t j) {
        if (std::abs(a[i][i] - a[j][j]) < GLOBAL_EPS) {
            return std::atan2(1.0, 1.0);
        } else {
            return 0.5 * std::atan2(2 * a[i][j], a[i][i] - a[j][j]);
        }
    }

    matrix create_rotation(size_t i, size_t j, double phi) {
        matrix u = matrix::identity(n);
        u[i][i] = std::cos(phi);
        u[i][j] = -std::sin(phi);
        u[j][i] = std::sin(phi);
        u[j][j] = std::cos(phi);
        return u;
    }
```

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

```
97
98      ~rotation() = default;
99  };
100
101 #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

```cpp
#ifndef QR_ALGO_HPP
#define QR_ALGO_HPP

#include <cmath>
#include <complex>
#include "../matrix.hpp"

class qr_algo {
private:
    using matrix = matrix_t<double>;
    using vec = std::vector<double>;
    using complex = std::complex<double>;
    using pcc = std::pair<complex, complex>;
    using vec_complex = std::vector<complex>;

    static constexpr double INF = 1e18;
    static constexpr complex COMPLEX_INF = complex(INF, INF);

    size_t n;
    matrix a;
    double eps;
    vec_complex eigen;

    double vtv(const vec & v) {
        double res = 0;
        for (double elem : v) {
            res += elem * elem;
        }
        return res;
    }

    double norm(const vec & v) {
        return std::sqrt(vtv(v));
    }

    matrix vvt(const vec & b) {
        size_t n_b = b.size();
        matrix res(n_b);
        for (size_t i = 0; i < n_b; ++i) {
            for (size_t j = 0; j < n_b; ++j) {
                res[i][j] = b[i] * b[j];
            }
        }
        return res;
    }

    double sign(double x) {
```

```
48          if (x < eps) {
49              return -1.0;
50          } else if (x > eps) {
51              return 1.0;
52          } else {
53              return 0.0;
54          }
55      }
56
57      matrix householder(const vec & b, int id) {
58          vec v(b);
59          v[id] += sign(b[id]) * norm(b);
60          return matrix::identity(n) - (2.0 / vtv(v)) * vvt(v);
61      }
62
63      pcc solve_sq(double a11, double a12, double a21, double a22) {
64          double a = 1.0;
65          double b = -(a11 + a22);
66          double c = a11 * a22 - a12 * a21;
67          double d_sq = b * b - 4.0 * a * c;
68          if (d_sq > eps) {
69              complex bad(NAN, NAN);
70              return std::make_pair(bad, bad);
71          }
72          complex d(0.0, std::sqrt(-d_sq));
73          complex x1 = (-b + d) / (2.0 * a);
74          complex x2 = (-b - d) / (2.0 * a);
75          return std::make_pair(x1, x2);
76      }
77
78      bool check_diag() {
79          for (size_t i = 0; i < n; ++i) {
80              double col_sum = 0;
81              for (size_t j = i + 2; j < n; ++j) {
82                  col_sum += a[j][i] * a[j][i];
83              }
84              double norm = std::sqrt(col_sum);
85              if (!(norm < eps)) {
86                  return false;
87              }
88          }
89          return true;
90      }
91
92      void calc_eigen() {
93          for (size_t i = 0; i < n; ++i) {
94              if (i < n - 1 and !(abs(a[i + 1][i]) < eps)) {
95                  auto [l1, l2] = solve_sq(a[i][i], a[i][i + 1], a[i + 1][i], 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 0.000000001
$ ./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

```cpp
#ifndef SOLVER_HPP
#define SOLVER_HPP

#include <cmath>

int iter_count = 0;

double f(double x) {
    return std::sin(x) - 2.0 * x * x + 0.5;
}

double f_s(double x) {
    return std::cos(x) - 4.0 * x;
}

double f_ss(double x) {
    return -std::sin(x) - 4.0;
}

double phi(double x) {
    return std::sqrt(0.5 * std::sin(x) + 0.25);
}

double phi_s(double x) {
    return std::cos(x) / (4.0 * phi(x));
}

double iter_solve(double l, double r, double eps) {
    iter_count = 0;
    double x_k = r;
    double dx = 1.0;
    double q = std::max(std::abs(phi_s(l)), std::abs(phi_s(r)));
    double eps_coef = q / (1.0 - q);
    do {
        double x_k1 = phi(x_k);
        dx = eps_coef * std::abs(x_k1 - x_k);
        ++iter_count;
        x_k = x_k1;
    } while (dx > eps);
    return x_k;
}

double newton_solve(double l, double r, double eps) {
    double x0 = l;
    if (!(f(x0) * f_ss(x0) > eps)) {
        x0 = r;
    }
```

```
48      iter_count = 0;
49      double x_k = x0;
50      double dx = 1.0;
51      do {
52          double x_k1 = x_k - f(x_k) / f_s(x_k);
53          dx = std::abs(x_k1 - x_k);
54          ++iter_count;
55          x_k = x_k1;
56      } while (dx > eps);
57      return x_k;
58  }
59
60  #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

```
#ifndef SYSTEM_SOLVER_HPP
#define SYSTEM_SOLVER_HPP

#include "../lab1_1/lu.hpp"

int iter_count = 0;

const double a = 1;

double phi1(double x1, double x2) {
    (void)x1;
    return a + std::cos(x2);
}

double phi1_s(double x1, double x2) {
    (void)x1;
    return -std::sin(x2);
}

double phi2(double x1, double x2) {
    (void)x2;
    return a + std::sin(x1);
}

double phi2_s(double x1, double x2) {
    (void)x2;
    return std::cos(x1);
}

double phi(double x1, double x2) {
    return phi1_s(x1, x2) * phi2_s(x1, x2);
}

using pdd = std::pair<double, double>;

pdd iter_solve(double l1, double r1, double l2, double r2, double eps) {
    iter_count = 0;
    double x_1_k = r1;
    double x_2_k = r2;
    double q = -1;
    q = std::max(q, std::abs(phi(l1, r1)));
    q = std::max(q, std::abs(phi(l1, r2)));
    q = std::max(q, std::abs(phi(l2, r1)));
    q = std::max(q, std::abs(phi(l2, r2)));
    double eps_coef = q / (1 - q);
    double dx = 1;
    do {
```

```
48          double x_1_k1 = phi1(x_1_k, x_2_k);
49          double x_2_k1 = phi2(x_1_k, x_2_k);
50          dx = eps_coef * (std::abs(x_1_k1 - x_1_k) + std::abs(x_2_k1 - x_2_k));
51          ++iter_count;
52          x_1_k = x_1_k1;
53          x_2_k = x_2_k1;
54      } while (dx > eps);
55      return std::make_pair(x_1_k, x_2_k);
56  }
57
58  using matrix = matrix_t<double>;
59  using lu = lu_t<double>;
60  using vec = std::vector<double>;
61
62  double f1(double x1, double x2) {
63      return x1 - std::cos(x2) - a;
64  }
65
66  double f2(double x1, double x2) {
67      return x2 - std::sin(x1) - a;
68  }
69
70  matrix j(double x1, double x2) {
71      matrix res(2);
72      res[0][0] = 1.0;
73      res[0][1] = std::sin(x2);
74      res[1][0] = -std::cos(x1);
75      res[1][1] = 1.0;
76      return res;
77  }
78
79  double norm(const vec & v) {
80      double res = 0;
81      for (double elem : v) {
82          res = std::max(res, std::abs(elem));
83      }
84      return res;
85  }
86
87  pdd newton_solve(double x1_0, double x2_0, double eps) {
88      iter_count = 0;
89      vec x_k = {x1_0, x2_0};
90      double dx = 1;
91      do {
92          double x1 = x_k[0];
93          double x2 = x_k[1];
94          lu jacobi(j(x1, x2));
95          vec f_k = {f1(x1, x2), f2(x1, x2)};
96          vec delta_x = jacobi.solve(f_k);
```

```
97          vec x_k1 = x_k - delta_x;
98          dx = norm(x_k1 - x_k);
99          ++iter_count;
100         x_k = x_k1;
101     } while (dx > eps);
102     return std::make_pair(x_k[0], x_k[1]);
103 }
104 
105 #endif /* SYSTEM_SOLVER_HPP */
```

# 3 Методы приближения функций

## 1 Полиномиальная интерполяция

### 1.1 Постановка задачи

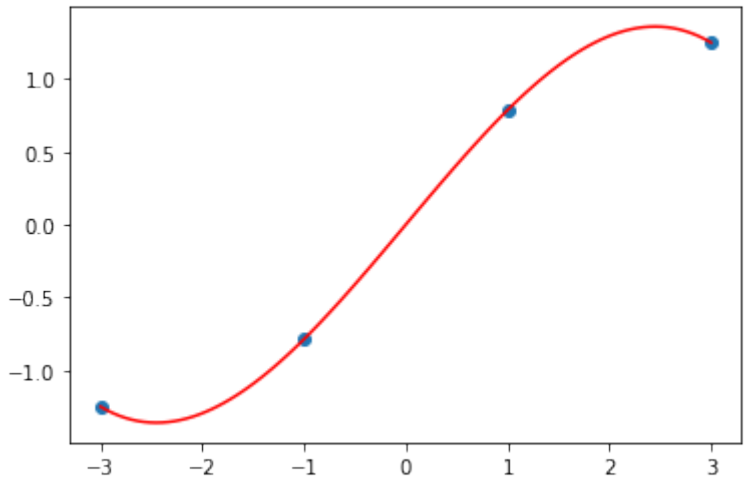
Используя таблицу значений $Y_i$ функции $y = f(x)$, вычисленных в точках $X_i$, $i = 0, \ldots, 3$ построить интерполяционные многочлены Лагранжа и Ньютона, проходящие через точки $\{X_i, Y_i\}$. Вычислить значение погрешности интерполяции в точке $X^*$.

### 1.2 Консоль

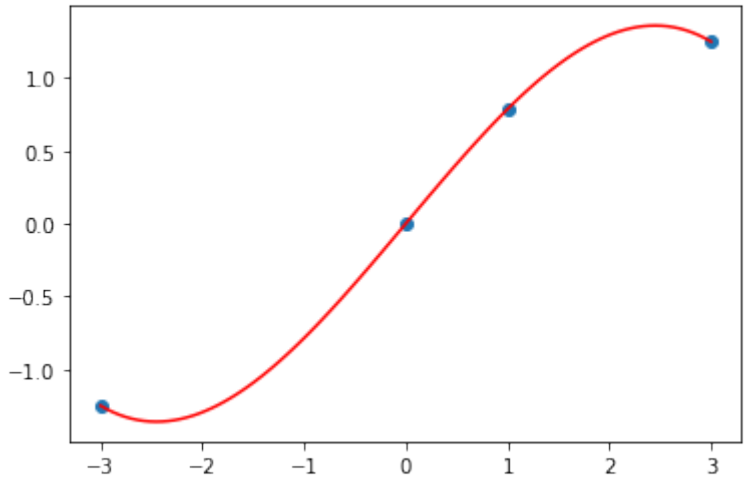
```
$ make
g++ -g -pedantic -std=c++17 -Wall -Wextra -Werror main.cpp -o solution
$ cat tests/1.in
4
-3 -1 1 3
-0.5
$ ./solution <tests/1.in
Интерполяционный многочлен Лагранжа: 0.831529 * x -0.0461312 * x ^ 3
Погрешность в точке X*: 0.0536493
Интерполяционный многочлен Ньютона: 0.831529 * x -0.0461312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

```
#ifndef INTERPOLATOR_HPP
#define INTERPOLATOR_HPP

#include "../polynom.hpp"

using vec = std::vector<double>;

class inter_lagrange {
    vec x;
    vec y;
    size_t n;

public:
    inter_lagrange(const vec & _x, const vec & _y) : x(_x), y(_y), n(x.size()) {};

    polynom operator () () {
        polynom res(vec({0}));
        for (size_t i = 0; i < n; ++i) {
            polynom li(vec({1}));
            for (size_t j = 0; j < n; ++j) {
                if (i == j) {
                    continue;
                }
                polynom xij(vec({-x[j], 1}));
                li = li * xij;
                li = li / (x[i] - x[j]);
            }
            res = res + y[i] * li;
        }
        return res;
    }
};

class inter_newton {
private:
    using vvd = std::vector< std::vector<double> >;
    using vvb = std::vector< std::vector<bool> >;

    vec x;
    vec y;
    size_t n;

    vvd memo;
    vvb calc;

    double f(int l, int r) {
        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 Сплай-интерполяция

## 2.1 Постановка задачи

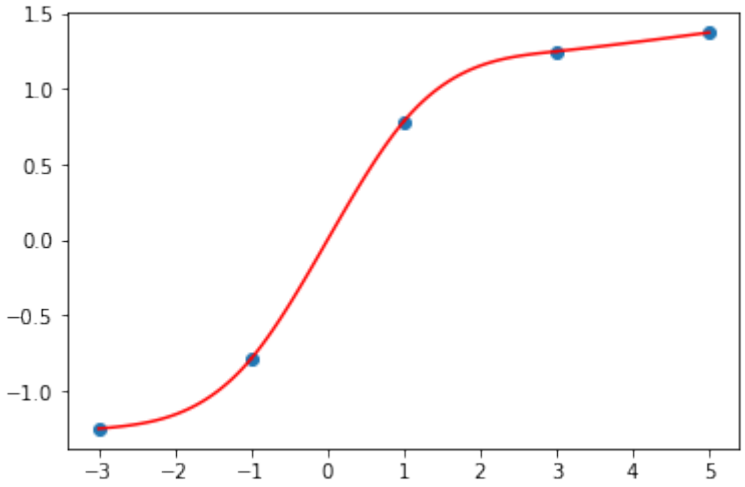
Построить кубический сплайн для функции, заданной в узлах интерполяции, предполагая, что сплайн имеет нулевую кривизну при $x = x_0$ и $x = x_4$. Вычислить значение функции в точке $x = X^*$.

## 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 x0 = -0.5000,f(x0) = -0.4270
```

## 2.3 Результат



## 2.4 Исходный код

```
#ifndef CUBIC_SPLINE_HPP
#define CUBIC_SPLINE_HPP

#include "../lab1_2/tridiag.hpp"

class cubic_spline_t {
    using vec = std::vector<double>;
    using tridiag = tridiag_t<double>;
    size_t n;
    vec x;
    vec y;
    vec a, b, c, d;

    void build_spline() {
        vec h(n + 1);
        h[0] = NAN;
        for (size_t i = 1; i <= n; ++i) {
            h[i] = x[i] - x[i - 1];
        }
        vec eq_a(n - 1);
        vec eq_b(n - 1);
        vec eq_c(n - 1);
        vec eq_d(n - 1);
        for (size_t i = 2; i <= n; ++i) {
            eq_a[i - 2] = h[i - 1];
            eq_b[i - 2] = 2.0 * (h[i - 1] + h[i]);
            eq_c[i - 2] = h[i];
            eq_d[i - 2] = 3.0 * ((y[i] - y[i - 1]) / h[i] - (y[i - 1] - y[i - 2]) / h[i
                - 1]);
        }
        eq_a[0] = 0.0;
        eq_c.back() = 0.0;
        // for (size_t i = 0; i < n - 1; ++i) {
        // printf("%lf %lf %lf %lf\n", eq_a[i], eq_b[i], eq_c[i], eq_d[i]);
        // }
        tridiag system_of_eq(eq_a, eq_b, eq_c);
        vec c_solved = system_of_eq.solve(eq_d);
        for (size_t i = 2; i <= n; ++i) {
            c[i] = c_solved[i - 2];
        }
        for (size_t i = 1; i <= n; ++i) {
            a[i] = y[i - 1];
        }
        for (size_t i = 1; i < n; ++i) {
            b[i] = (y[i] - y[i - 1]) / h[i] - h[i] * (c[i + 1] + 2.0 * c[i]) / 3.0;
            d[i] = (c[i + 1] - c[i]) / (3.0 * h[i]);
        }
```

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

# 3 Метод наименьших квадратов

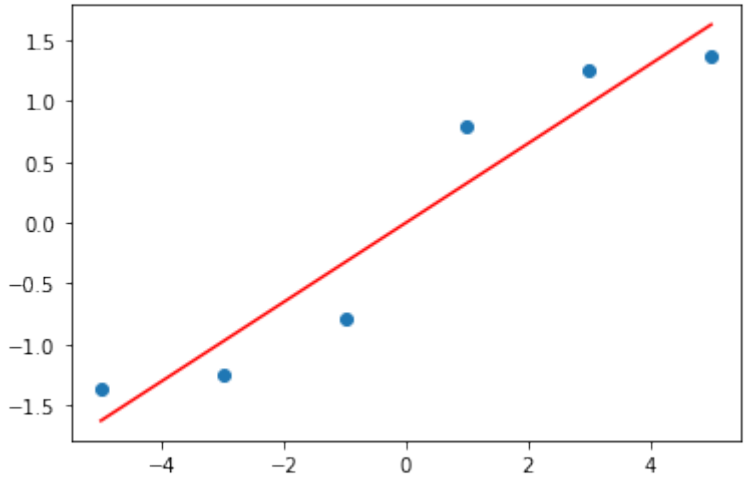
## 3.1 Постановка задачи

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 ошибков: 0.7007
Полученная функция второго порядка: 0.0000 0.3257 -0.0000
Значение суммы квадратов ошибков: 0.7007
```

### 3.3 Результат



### 3.4 Исходный код

```
#ifndef MINIMAL_SQUARE_HPP
#define MINIMAL_SQUARE_HPP

#include "../lab1_1/lu.hpp"
#include "../polynom.hpp"

class minimal_square_t {
    using vec = std::vector<double>;
    using matrix = matrix_t<double>;
    using lu = lu_t<double>;

    using func = std::function<double(double)>;
    using vf = std::vector<func>;

    size_t n;
    vec x;
    vec y;
    size_t m;
    vec a;
    vf phi;

    void build() {
        matrix lhs(n, m);
        for (size_t i = 0; i < n; ++i) {
            for (size_t j = 0; j < m; ++j) {
                lhs[i][j] = phi[j](x[i]);
            }
        }
        matrix lhs_t = lhs.t();
        lu lhs_lu(lhs_t * lhs);
        vec rhs = lhs_t * y;
        a = lhs_lu.solve(rhs);
    }

    double get(double x0) {
        double res = 0.0;
        for (size_t i = 0; i < m; ++i) {
            res += a[i] * phi[i](x0);
        }
        return res;
    }

public:
    minimal_square_t(const vec & _x, const vec & _y, const vf & _phi) {
        if (_x.size() != _y.size()) {
            throw std::invalid_argument("Sizes does not match");
        }
```

```
48          n = _x.size();
49          x = _x;
50          y = _y;
51          m = _phi.size();
52          a.resize(m);
53          phi = _phi;
54          build();
55      }
56
57      friend std::ostream & operator << (std::ostream & out, const minimal_square_t &
            item) {
58          for (size_t i = 0; i < item.m; ++i) {
59              if (i) {
60                  out << ' ';
61              }
62              out << item.a[i];
63          }
64          return out;
65      }
66
67      double mse() {
68          double res = 0;
69          for (size_t i = 0; i < n; ++i) {
70              res += std::pow(get(x[i]) - y[i], 2.0);
71          }
72          return res;
73      }
74
75      double operator () (double x0) {
76          return get(x0);
77      }
78  };
79
80  #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

```
#ifndef TABLE_FUNCTION_HPP
#define TABLE_FUNCTION_HPP

#include <exception>
#include <vector>

const double EPS = 1e-9;

bool leq(double a, double b) {
    return (a < b) or (std::abs(b - a) < EPS);
}

class table_function_t {
    using vec = std::vector<double>;
    size_t n;
    vec x;
    vec y;

public:
    table_function_t(const vec & _x, const vec & _y) {
        if (_x.size() != _y.size()) {
            throw std::invalid_argument("Sizes does not match");
        }
        x = _x;
        y = _y;
        n = x.size();
    }

    double derivative1(double x0) {
        for (size_t i = 0; i < n - 2; ++i) {
            /* x in (x_i, x_i+1] */
            if (x[i] < x0 and leq(x0, x[i + 1])) {
                double dydx1 = (y[i + 1] - y[i + 0]) / (x[i + 1] - x[i + 0]);
                double dydx2 = (y[i + 2] - y[i + 1]) / (x[i + 2] - x[i + 1]);
                double res = dydx1 + (dydx2 - dydx1) * (2.0 * x0 - x[i] - x[i + 1]) / (x
                    [i + 2] - x[i]);
                return res;
            }
        }
        return NAN;
    }

    double derivative2(double x0) {
        for (size_t i = 0; i < n - 2; ++i) {
            /* x in (x_i, x_i+1] */
            if (x[i] < x0 and leq(x0, x[i + 1])) {
                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 $F = \int_{X_0}^{X_1} ydx$, 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

```cpp
#ifndef INTEGRATE_HPP
#define INTEGRATE_HPP

#include <cmath>
#include "../lab3_1/interpolator.hpp"

using func = double(double);

double integrate_rect(double l, double r, double h, func f) {
    double x1 = l;
    double x2 = l + h;
    double res = 0;
    while (x1 < r) {
        res += h * f((x1 + x2) * 0.5);
        x1 = x2;
        x2 += h;
    }
    return res;
}

double integrate_trap(double l, double r, double h, func f) {
    double x1 = l;
    double x2 = l + h;
    double res = 0;
    while (x1 < r) {
        res += h * (f(x1) + f(x2));
        x1 = x2;
        x2 += h;
    }
    return res * 0.5;
}

using vec = std::vector<double>;

double integrate_simp(double l, double r, double h, func f) {
    double x1 = l;
    double x2 = l + h;
    double res = 0;
    while (x1 < r) {
        vec x = {x1, (x1 + x2) * 0.5, x2};
        vec y = {f(x[0]), f(x[1]), f(x[2])};
        inter_lagrange lagr(x, y);
        res += lagr().integrate(x1, x2);
        x1 = x2;
        x2 += h;
    }
    return res;
```

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

# 4 Методы решения обыкновенных дифференциальных уравнений

## 1 Решение задачи Коши для ОДУ

### 1.1 Постановка задачи

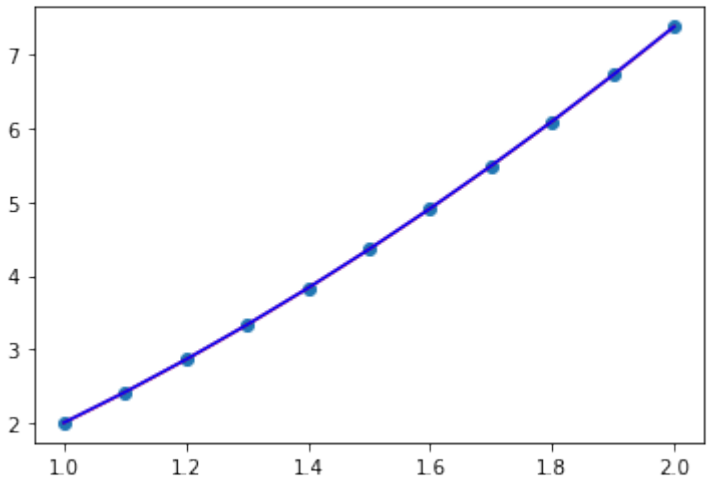
Реализовать методы Эйлера, Рунге-Кутты и Адамса 4-го порядка в виде программ, задавая в качестве входных данных шаг сетки $h$. С использованием разработанного программного обеспечения решить задачу Коши для ОДУ 2-го порядка на указанном отрезке. Оценить погрешность численного решения с использованием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

### 1.2 Консоль

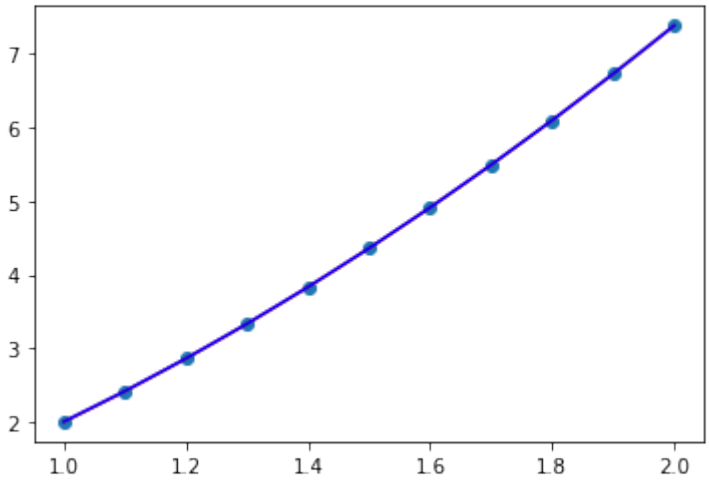
```
$ make
g++ -g -pedantic -std=c++17 -Wall -Wextra -Werror main.cpp -o solution
$ cat tests/1.in
1 2
2 4 0.1
$ ./solution <tests/1.in
Метод Эйлера:
x = [1.000000,1.100000,1.200000,1.300000,1.400000,1.500000,1.600000,1.700000,1.800000
y = [2.000000,2.414842,2.858788,3.331076,3.831064,4.358201,4.912010,5.492073,6.098021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

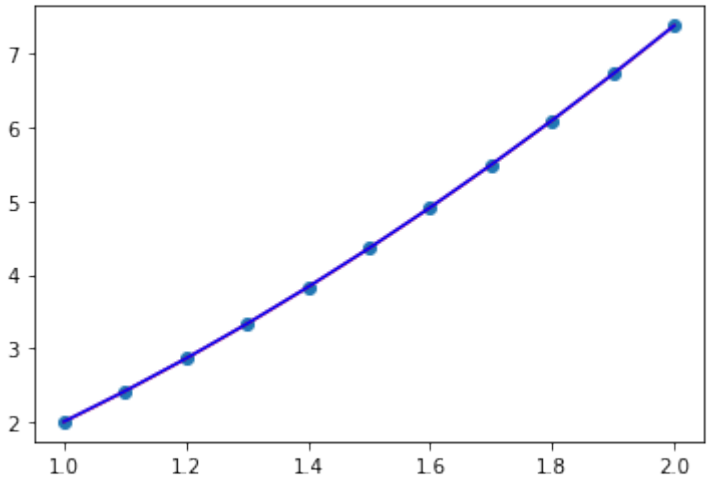
Метод Эйлера




Метод Рунге-Кутты



Метод Адамса



### 1.4 Исходный код

```cpp
#ifndef SIMPLE_DESOLVE_HPP
#define SIMPLE_DESOLVE_HPP

#include "../de_utils.hpp"
#include <functional>

/* f(x, y, z) */
using func = std::function<double(double, double, double)>;
using vect = std::vector<tddd>;
using vec = std::vector<double>;

const double EPS = 1e-9;

bool leq(double a, double b) {
    return (a < b) or (std::abs(b - a) < EPS);
}

class euler {
private:
    double l, r;
    func f, g;
    double y0, z0;

public:
    euler(const double _l, const double _r,
        const func _f, const func _g,
        const double _y0, const double _z0) : l(_l), r(_r), f(_f), g(_g), y0(_y0), z0(
            _z0) {}

    vect solve(double h) {
        vect res;
        double xk = l;
        double yk = y0;
        double zk = z0;
        res.push_back(std::make_tuple(xk, yk, zk));
        while (leq(xk + h, r)) {
            double dy = h * f(xk, yk, zk);
            double dz = h * g(xk, yk, zk);
            xk += h;
            yk += dy;
            zk += dz;
            res.push_back(std::make_tuple(xk, yk, zk));
        }
        return res;
    }
};

```

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

```
94
95      double calc_tuple(func f, tddd xyz) {
96          return f(std::get<0>(xyz), std::get<1>(xyz), std::get<2>(xyz));
97      }
98
99      vect solve(double h) {
100         if (l + 3.0 * h > r) {
101             throw std::invalid_argument("h is too big");
102         }
103         runge first_points(l, l + 3.0 * h, f, g, y0, z0);
104         vect res = first_points.solve(h);
105         size_t cnt = res.size();
106         double xk = std::get<0>(res.back());
107         double yk = std::get<1>(res.back());
108         double zk = std::get<2>(res.back());
109         while (leq(xk + h, r)) {
110             /* Predictor */
111             double dy = (h / 24.0) * (55.0 * calc_tuple(f, res[cnt - 1])
112                                     - 59.0 * calc_tuple(f, res[cnt - 2])
113                                     + 37.0 * calc_tuple(f, res[cnt - 3])
114                                     - 9.0 * calc_tuple(f, res[cnt - 4]));
115             double dz = (h / 24.0) * (55.0 * calc_tuple(g, res[cnt - 1])
116                                     - 59.0 * calc_tuple(g, res[cnt - 2])
117                                     + 37.0 * calc_tuple(g, res[cnt - 3])
118                                     - 9.0 * calc_tuple(g, res[cnt - 4]));
119             double xk1 = xk + h;
120             double yk1 = yk + dy;
121             double zk1 = zk + dz;
122             res.push_back(std::make_tuple(xk1, yk1, zk1));
123             ++cnt;
124             /* Corrector */
125             dy = (h / 24.0) * (9.0 * calc_tuple(f, res[cnt - 1])
126                              + 19.0 * calc_tuple(f, res[cnt - 2])
127                              - 5.0 * calc_tuple(f, res[cnt - 3])
128                              + 1.0 * calc_tuple(f, res[cnt - 4]));
129             dz = (h / 24.0) * (9.0 * calc_tuple(g, res[cnt - 1])
130                              + 19.0 * calc_tuple(g, res[cnt - 2])
131                              - 5.0 * calc_tuple(g, res[cnt - 3])
132                              + 1.0 * calc_tuple(g, res[cnt - 4]));
133             xk += h;
134             yk += dy;
135             zk += dz;
136             res.pop_back();
137             res.push_back(std::make_tuple(xk, yk, zk));
138         }
139         return res;
140     }
141 };
142
```

```
143 double runge_romberg(const vect & y_2h, const vect & y_h, double p) {
144     double coef = 1.0 / (std::pow(2, p) - 1.0);
145     double res = 0.0;
146     for (size_t i = 0; i < y_2h.size(); ++i) {
147         res = std::max(res, coef * std::abs(std::get<1>(y_2h[i]) - 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

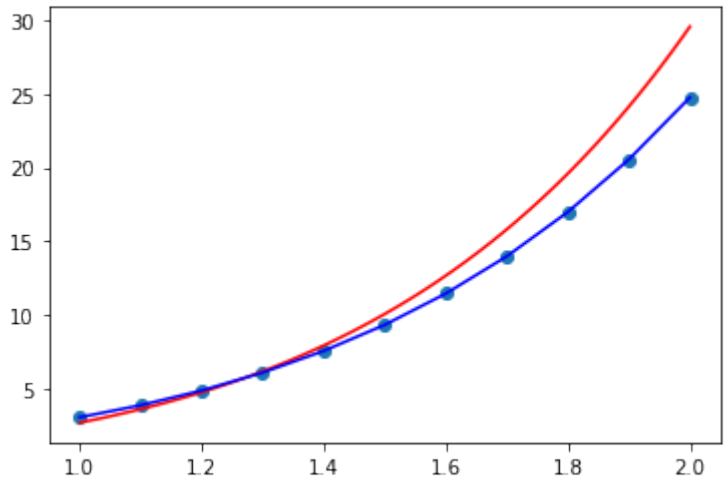
Реализовать метод стрельбы и конечно-разностный метод решения краевой задачи для ОДУ в виде программ. С использованием разработанного программного обеспечения решить краевую задачу для обыкновенного дифференциального уравнения 2-го порядка на указанном отрезке. Оценить погрешность численного решения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

## 2.2 Консоль

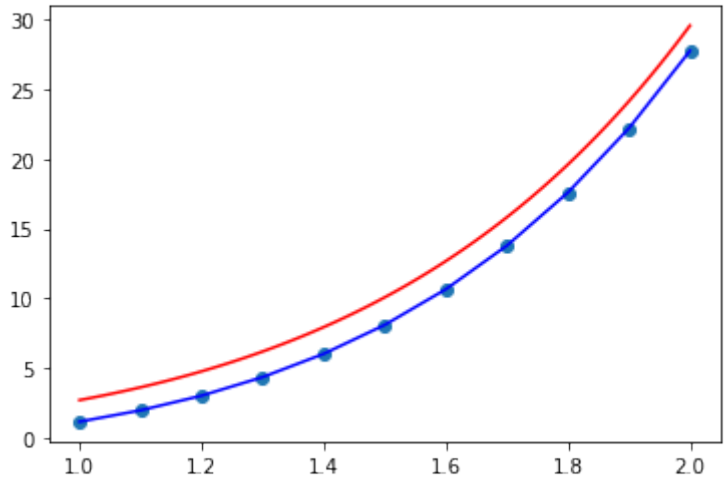
```
$ make
g++ -g -pedantic -std=c++17 -Wall -Wextra -Werror main.cpp -o solution
$ cat tests/1.in
0.1 0.0001
$ ./solution <tests/1.in
Метод стрельбы:
x = [1.000000,1.100000,1.200000,1.300000,1.400000,1.500000,1.600000,1.700000,1.800000
y = [3.082723,3.898207,4.896683,6.111204,7.580066,9.347569,11.464879,13.991012,16.993
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

```
#ifndef BOUNDARY_SOLVER_HPP
#define BOUNDARY_SOLVER_HPP

#include <cmath>
#include "../lab1_2/tridiag.hpp"
#include "../lab4_1/simple_desolve.hpp"

class shooting {
private:
    double a, b;
    func f, g;
    double alpha, beta, y0;
    double delta, gamma, y1;

public:
    shooting(const double _a, const double _b,
        const func _f, const func _g,
        const double _alpha, const double _beta, const double _y0,
        const double _delta, const double _gamma, const double _y1)
        : a(_a), b(_b), f(_f), g(_g),
        alpha(_alpha), beta(_beta), y0(_y0),
        delta(_delta), gamma(_gamma), y1(_y1) {}

    double get_start_cond(double eta) {
        return (y0 - alpha * eta) / beta;
    }

    double get_eta_next(double eta_prev, double eta, const vect sol_prev, const vect
        sol) {
        double yb_prev = std::get<1>(sol_prev.back());
        double zb_prev = std::get<2>(sol_prev.back());
        double phi_prev = delta * yb_prev + gamma * zb_prev - y1;
        double yb = std::get<1>(sol.back());
        double zb = std::get<2>(sol.back());
        double phi = delta * yb + gamma * zb - y1;
        return eta - (eta - eta_prev) / (phi - phi_prev) * phi;
    }

    vect solve(double h, double eps) {
        double eta_prev = 1.0;
        double eta = 0.8;
        while (1) {
            double runge_z0_prev = get_start_cond(eta_prev);
            euler de_solver_prev(a, b, f, g, eta_prev, runge_z0_prev);
            vect sol_prev = de_solver_prev.solve(h);

            double runge_z0 = get_start_cond(eta);
```

```
47              euler de_solver(a, b, f, g, eta, runge_z0);
48              vect sol = de_solver.solve(h);
49
50              double eta_next = get_eta_next(eta_prev, eta, sol_prev, sol);
51              if (std::abs(eta_next - eta) < eps) {
52                  return sol;
53              } else {
54                  eta_prev = eta;
55                  eta = eta_next;
56              }
57          }
58      }
59  };
60
61  class fin_dif {
62  private:
63      using fx = std::function<double(double)>;
64      using tridiag = tridiag_t<double>;
65
66      double a, b;
67      fx p, q, f;
68      double alpha, beta, y0;
69      double delta, gamma, y1;
70
71  public:
72      fin_dif(const double _a, const double _b,
73          const fx _p, const fx _q, const fx _f,
74          const double _alpha, const double _beta, const double _y0,
75          const double _delta, const double _gamma, const double _y1)
76          : a(_a), b(_b), p(_p), q(_q), f(_f),
77          alpha(_alpha), beta(_beta), y0(_y0),
78          delta(_delta), gamma(_gamma), y1(_y1) {}
79
80      vect solve(double h) {
81          size_t n = (b - a) / h;
82          vec xk(n + 1);
83          for (size_t i = 0; i <= n; ++i) {
84              xk[i] = a + h * i;
85          }
86          vec a(n + 1);
87          vec b(n + 1);
88          vec c(n + 1);
89          vec d(n + 1);
90          b[0] = h * alpha - beta;
91          c[0] = beta;
92          d[0] = h * y0;
93          a.back() = -gamma;
94          b.back() = h * delta + gamma;
95          d.back() = h * y1;
```

```
96          for (size_t i = 1; i < n; ++i) {
97              a[i] = 1.0 - p(xk[i]) * h * 0.5;
98              b[i] = -2.0 + h * h * q(xk[i]);
99              c[i] = 1.0 + p(xk[i]) * h * 0.5;
100             d[i] = h * h * f(xk[i]);
101         }
102         tridiag sys_eq(a, b, c);
103         vec yk = sys_eq.solve(d);
104         vect res;
105         for (size_t i = 0; i <= n; ++i) {
106             res.push_back(std::make_tuple(xk[i], yk[i], NAN));
107         }
108         return res;
109     }
110 };
111 
112 #endif /* BOUNDARY_SOLVER_HPP */
```