Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторные работы по курсу «Численные методы»

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

Дата:
Оценка:
Подпись:

Москва, 2022

1 Численные методы линейной алгебры

1 LU-разложение матриц

1.1 Постановка задачи

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

Вариант:

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

1.2 Результаты работы

```
$ cat tests/1/2.txt
4
2 7 -8 6
4 4 0 -7
-1 -3 6 3
9 -7 -2 -8
-39 41 4 113

$ ./lab1_1 <tests/1/2.txt
Решение системы:
8.000   -3.000    2.000   -3.000
Обратная матрица системы:
0.102    0.072    0.161    0.074
0.040    0.111    0.035   -0.054
-0.004    0.087    0.155   -0.021
0.081   -0.038    0.112    0.011

Определитель матрицы системы:
-4924.000
```

1.3 Исходный код

```cpp
#pragma once

#include <vector>
#include <utility>
#include <stdexcept>

#include "matrix.hpp"
#include "vector.hpp"

template <class T>
struct LUP {
  Matrix<T> L;
  Matrix<T> U;
  std::vector<std::pair<size_t, size_t>> P;

  LUP(size_t n): L(n), U(n), P() { }

  LUP(const Matrix<T>& matrix): L(matrix.Size()), U(matrix.Size()), P() {
    matrix.Decompose(L, U, P);
  }

  void Assign(const Matrix<T>& matrix) {
    if (L.Size() != matrix.Size()) {
      L = Matrix<T>(matrix.Size());
      U = Matrix<T>(matrix.Size());
    }
    matrix.Decompose(L, U, P);
  }

  Matrix<T> Compose() {
    return L * U;
  }

  T Det() {
    T res = 1;
    for (size_t i = 0; i < U.Size(); ++i) {
      res *= U[i][i];
    }
    if (P.size() & 1) {
      res = -res;
    }
    return res;
  }

  Vector<T> Solve(Vector<T> b) {
    if (b.Size() != L.Size()) {
      throw std::runtime_error("Dimension mismatch");
```

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

2 Метод прогонки

2.1 Постановка задачи

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

Вариант:

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

2.2 Результаты работы

```
$ cat tests/2/2.txt
5
10 5
3 10 -2
2 -9 -5
5 16 -4
-8 16
-120 -91 5 -74 -56

$ ./lab1_2 <tests/2/2.txt
Решение системы:
-9.000   -6.000    2.000   -7.000   -7.000
```

2.3 Исходный код

```cpp
#pragma once

#include <vector>
#include <iostream>
#include <iomanip>
#include <stdexcept>

#include "vector.hpp"

template <class T>
class TDMatrix {
private:
  size_t _size;

public:
  std::vector<T> a, b, c;

  TDMatrix(size_t n) : _size(n), a(n), b(n), c(n) { }

  TDMatrix(std::vector<T> a, std::vector<T> b, std::vector<T> c) : _size(a.size()), a(a), b(b), c(c) {
    if (a.size() != b.size() || b.size() != c.size()) {
      throw std::runtime_error("Incompatible arrays");
    }
  }

  size_t Size() const {
    return _size;
  }

  Vector<T> Solve(const Vector<T>& vec) {
    if (vec.Size() != _size) {
      throw std::runtime_error("Dimension mismatch");
    }

    Vector<T> p(_size-1), q(_size-1), x(_size);
    p[0] = - c[0] / b[0];
    q[0] = vec[0] / b[0];

    for (size_t i = 1; i < _size-1; ++i) {
      p[i] = - c[i] / (b[i] + a[i] * p[i-1]);
      q[i] = (vec[i] - a[i] * q[i-1]) / (b[i] + a[i] * p[i-1]);
    }

    x[_size-1] = (vec[_size-1] - a[_size-1] * q[_size-2]) / (b[_size-1] + a[_size-1] * p[_size-2]);
    for (int i = _size - 2; i >= 0; --i) {
```

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

3 Итерационные методы решения СЛАУ

3.1 Постановка задачи

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

Вариант:

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

3.2 Результаты работы

```
$ cat tests/3/2.txt
4
24 2 4 -9
-6 -27 -8 -6
-4 8 19 6
4 5 -3 -13
-9 -76 -79 -70
0.0001

$ ./lab1_3 <tests/3/2.txt
Решение методом Якоби:
3.999999 1.999999 -6.999998 8.999998
Количество итераций: 20

Решение методом Зейделя:
4.000000 2.000000 -7.000000 9.000000
Количество итераций: 10
```

3.3 Исходный код

```cpp
#pragma once

#include <cmath>
#include <stdexcept>

#include "matrix.hpp"
#include "vector.hpp"

template <class T>
size_t IterationMethod(const Matrix<T>& A, const Vector<T>& b, const Matrix<T>& M,
    Vector<T>& x, double eps) {
  if ((A.Size() != b.Size()) || (b.Size() != M.Size()) || (M.Size() != x.Size())) {
    throw std::runtime_error("Dimension mismatch");
  }
  Matrix<T> alpha = Matrix<T>::Identity(A.Size()) + M * A;
  Vector<T> beta = -(M * b);
  x = beta;

  T alpha_norm = alpha.Norm();
  size_t count = 0;
  double eps_k;
  Vector<T> next_x(x.Size());

  do {
    next_x = alpha * x + beta;
    if (alpha_norm < 1) {
      eps_k = alpha_norm / (1.0 - alpha_norm) * (next_x - x).Norm();
    } else {
      eps_k = (next_x - x).Norm();
    }
    std::swap(next_x, x);
    ++count;
  } while (eps_k > eps);

  return count;
}

template <class T>
size_t JacobiMethod(const Matrix<T>& A, const Vector<T>& b, Vector<T>& x, double eps)
    {
  Matrix<T> M(A.Size());
  for (size_t i = 0; i < A.Size(); ++i) {
    M[i][i] = -1.0 / A[i][i];
  }
  return IterationMethod(A, b, M, x, eps);
}

```

```cpp
template <class T>
size_t SeidelMethod(const Matrix<T>& A, const Vector<T>& b, const Matrix<T>& M, Vector
    <T>& x, double eps) {
  if ((A.Size() != b.Size()) || (b.Size() != M.Size()) || (M.Size() != x.Size())) {
    throw std::runtime_error("Dimension mismatch");
  }
  Matrix<T> alpha = Matrix<T>::Identity(A.Size()) + M * A;
  Vector<T> beta = -(M * b);
  x = beta;

  Matrix<T> alpha2 = alpha;
  for (size_t i = 0; i < alpha2.Size(); ++i) {
    for (size_t j = 0; j < i; ++j) {
      alpha2[i][j] = 0;
    }
  }

  T alpha_norm = alpha.Norm();
  T alpha2_norm = alpha2.Norm();
  size_t count = 0;
  double eps_k;
  Vector<T> next_x(x.Size());

  do {
    for (size_t i = 0; i < x.Size(); ++i) {
      next_x[i] = beta[i];
      for (size_t j = 0; j < i; ++j) {
        next_x[i] += alpha[i][j] * next_x[j];
      }
      for (size_t j = i; j < x.Size(); ++j) {
        next_x[i] += alpha[i][j] * x[j];
      }
    }

    if (alpha_norm < 1) {
      eps_k = alpha2_norm / (1.0 - alpha_norm) * (next_x - x).Norm();
    } else {
      eps_k = (next_x - x).Norm();
    }
    std::swap(next_x, x);
    ++count;
  } while (eps_k > eps);

  return count;
}

template <class T>
size_t JacobiSeidelMethod(const Matrix<T>& A, const Vector<T>& b, Vector<T>& x, double
     eps) {
```

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

4 Метод вращений

4.1 Постановка задачи

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

Вариант:

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

4.2 Результаты работы

```
$ cat tests/4/2.txt
3
-9 7 5
7 8 9
5 9 8
0.0001

$ ./lab1_4 <tests/4/2.txt
Собственные значения:
19.532   -0.836  -11.696

Матрица собственных векторов:
0.286   -0.115    0.951
0.691   -0.663   -0.288
0.664    0.740   -0.110

Количество итераций: 6
```

4.3 Исходный код

```cpp
#pragma once

#include <cmath>

#include "matrix.hpp"

template <class T>
size_t RotationMethod(Matrix<T> A, double eps, std::vector<T>& values, Matrix<T>& U) {
  values.assign(A.Size(), 0);
  U = Matrix<T>::Identity(A.Size());

  size_t k, l;
  T max;
  Matrix<T> transform(A.Size());
  double angle, eps_k;
  size_t count = 0;
  do {
    max = 0;
    for (size_t i = 1; i < A.Size(); ++i) {
      for (size_t j = 0; j < i; ++j) {
        if (std::abs(A[i][j]) > max) {
          max = std::abs(A[i][j]);
          l = i;
          k = j;
        }
      }
    }

    angle = 0.5 * std::atan2(2 * A[k][l], A[k][k] - A[l][l]);
    transform = Matrix<T>::Rotation(A.Size(), k, l, angle);

    A = transform.Transpose() * A * transform;
    U = U * transform;

    eps_k = 0;
    for (size_t i = 1; i < A.Size(); ++i) {
      for (size_t j = 0; j < i; ++j) {
        eps_k += A[i][j] * A[i][j];
      }
    }
    eps_k = std::sqrt(eps_k);

    ++count;
  } while (eps_k > eps);

  for (size_t i = 0; i < A.Size(); ++i) {
    values[i] = A[i][i];
```

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

5 QR алгоритм

5.1 Постановка задачи

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

Вариант:

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

5.2 Результаты работы

```
$ cat tests/5/2.txt
3
-6 -4 0
-7 6 -7
-2 -6 -7
0.00001

$ ./lab1_5 <tests/5/2.txt
Собственные значения:
-11.3395 10.0120 -5.6725
Количество итераций: 112
```

5.3 Исходный код

```cpp
#pragma once

#include <cmath>
#include <vector>
#include <complex>

#include "matrix.hpp"
#include "vector.hpp"

template <class T>
void QR_Decomposition(const Matrix<T>& A, Matrix<T>& Q, Matrix<T>& R) {
  R = A;
  Q = Matrix<T>::Identity(A.Size());
  Vector<T> v(A.Size());
  Matrix<T> H(A.Size());
  for (size_t k = 0; k < A.Size() - 1; ++k) {
    for (size_t i = 0; i < k; ++i) {
      v[i] = 0;
    }
    v[k] = R[k][k] * R[k][k];
    for (size_t i = k+1; i < A.Size(); ++i) {
      v[i] = R[i][k];
      v[k] += R[i][k] * R[i][k];
    }
    v[k] = std::sqrt(v[k]) * (R[k][k] >= 0 ? 1 : -1) + R[k][k];

    H = Matrix<T>::Householder(A.Size(), v);
    R = H * R;
    Q = Q * H;
  }
}

template <class T>
size_t QR_Eigenvalues(Matrix<T> A, double eps, std::vector<std::complex<T>>&
    eigenvalues) {
  Matrix<T> Q(A.Size()), R(A.Size());
  size_t iter_count = 0;
  Vector<double> eps_1(A.Size()), eps_2(A.Size()), eps_3(A.Size());
  eigenvalues.assign(A.Size(), std::complex<T>());
  std::vector<std::complex<T>> prev_eigenvalues(A.Size(), 1e18);

  do {
    QR_Decomposition(A, Q, R);
    A = R * Q;

    // вычислениесобственныхзначений
    eigenvalues.swap(prev_eigenvalues);
```

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

2 Решение нелинейных уравнений и систем нелинейных уравнений

1 Решение нелинейных уравнений

1.1 Постановка задачи

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

Вариант:

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

1.2 Результаты работы

```
$ ./lab2_1
0.001
Метод простой итерации:
Корень: -0.58782084
Количество итераций: 49
Корень: 1.05723005
Количество итераций: 11

Метод Ньютона:
Корень: -0.58760883
Количество итераций: 6
Корень: 1.05710366
Количество итераций: 4

$ ./lab2_1
0.00000001
Метод простой итерации:
Корень: -0.58760883
Количество итераций: 127
Корень: 1.05710355
Количество итераций: 29

Метод Ньютона:
```

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

1.3 Исходный код

```cpp
#pragma once

#include <functional>
#include <cmath>

template <class T>
using interval_t = std::pair<T, T>;

template <class T>
int sign(T x) {
  if (x > 0) {
    return 1;
  } else if (x < 0) {
    return -1;
  } else {
    return 0;
  }
}

template <class T>
std::pair<T, size_t> IterationMethod(T x, const interval_t<T>& interval, const std::
    function<T(T)>& f, const std::function<T(T)>& df, double eps) {
  T lambda = sign(df(x)) / std::max(std::abs(df(interval.first)), std::abs(df(interval
      .second)));
  double q = std::max(std::abs(1 - lambda * df(interval.first)), std::abs(1 - lambda *
       df(interval.second)));
  q = q / (1 - q);

  double eps_k;
  T next_x;
  size_t iter_count = 0;
  do {
    next_x = x - lambda * f(x);
    eps_k = q * std::abs(next_x - x);

    std::swap(next_x, x);
    ++iter_count;
  } while (eps_k > eps);

  return {x, iter_count};
}
```

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

2 Решение систем нелинейных уравнений

2.1 Постановка задачи

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

Вариант:

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

2.2 Результаты работы

```
$ ./lab2_2
0.001 1
Метод простой итерации:
Решение: -1.32480839 2.51043055
Количество итераций: 7
Решение: 4.44697542 0.93831944
Количество итераций: 9

Метод Ньютона:
Решение: -1.32469087 2.51050556
Количество итераций: 4
Решение: 4.44694707 0.93830348
Количество итераций: 4

$ ./lab2_2
0.00000001 1
Метод простой итерации:
Решение: -1.32469087 2.51050556
Количество итераций: 17
Решение: 4.44694707 0.93830348
Количество итераций: 20

Метод Ньютона:
```

```
Решение: -1.32469087 2.51050556
Количество итераций: 5
Решение: 4.44694707 0.93830348
Количество итераций: 5
```

2.3 Исходный код

```cpp
#pragma once

#include <functional>
#include <vector>
#include <stdexcept>

#include "../linear/matrix.hpp"
#include "../linear/vector.hpp"
#include "../linear/lup.hpp"
#include "../function/optimization.hpp"

template <class T, class... Args>
class Matrix<std::function<T(Args...)>> {
private:
  using function_t = std::function<T(Args...)>;

  std::vector<std::vector<function_t>> _data;
  size_t _size;

  Matrix() : _data(), _size(0) { }

public:
  Matrix(size_t n) : _data(n, std::vector<function_t>(n)), _size(n) { }

  Matrix(std::initializer_list<std::vector<function_t>> list) : _data(list), _size(
      _data.size()) {
    for (const std::vector<function_t>& row: _data) {
      if (row.size() != _size) {
        throw std::runtime_error("Incorrect initializer list");
      }
    }
  }

  std::vector<function_t>& operator[](size_t index) {
    return _data[index];
  }

  const std::vector<function_t>& operator[](size_t index) const {
    return _data[index];
  }

```

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

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

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

3 Приближение функций. Численные дифференцирование и интегрирование

1 Интерполяционные многочлены

1.1 Постановка задачи

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

Вариант:

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

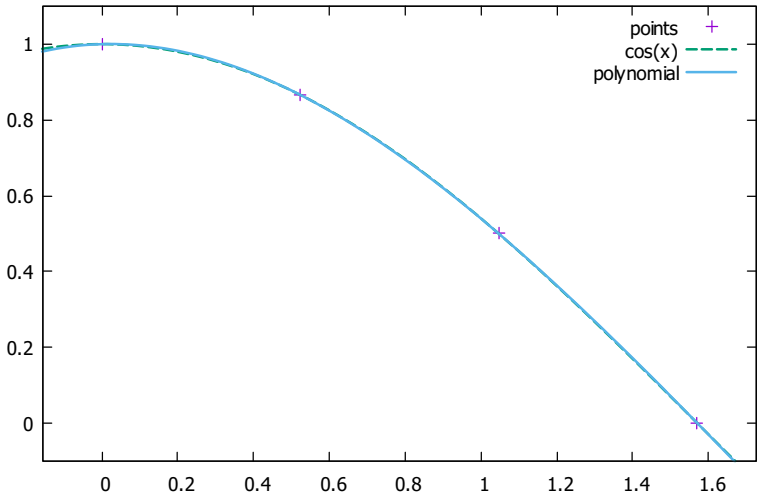
а). $X_i = 0, \frac{\pi}{6}, \frac{2\pi}{6}, \frac{3\pi}{6}$

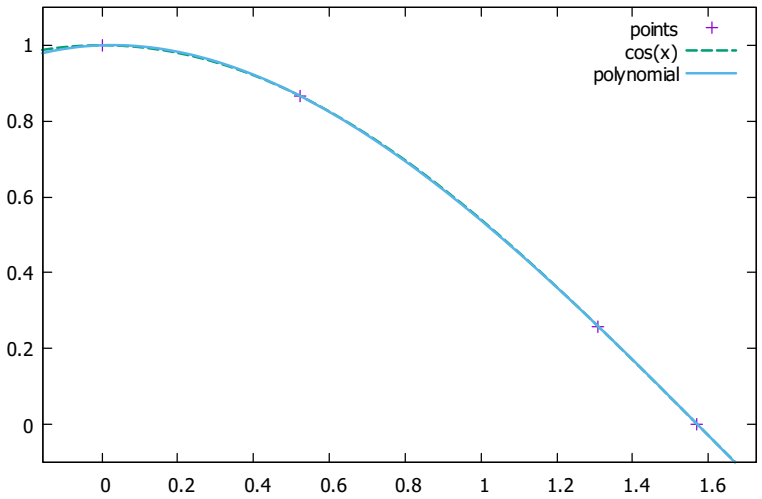
б). $X_i = 0, \frac{\pi}{6}, \frac{5\pi}{12}, \frac{\pi}{2}$

1.2 Результаты работы

```
$ cat tests/1/1.txt
4
0 0.5235987755982988 1.0471975511965976 1.5707963267948966
0.7853981633974483

$ ./lab3_1 <tests/1/1.txt
Многочлен в форме Лагранжа:
0.1139 * x**3 -0.6021 * x**2 + 0.0282 * x**1 + 1.0000
Многочлен в форме Ньютона:
0.1139 * x**3 -0.6021 * x**2 + 0.0282 * x**1 + 1.0000
Погрешность интерполяции:
0.0012

$ cat tests/1/2.txt
4
0 0.5235987755982988 1.3089969389957472 1.5707963267948966
0.7853981633974483

$ ./lab3_1 <tests/1/2.txt
Многочлен в форме Лагранжа:
0.1206 * x**3 -0.6161 * x**2 + 0.0337 * x**1 + 1.0000
```

```
Многочлен в форме Ньютона:
0.1206 * x**3 -0.6161 * x**2 + 0.0337 * x**1 + 1.0000
Погрешность интерполяции:
0.0023
```





1.3 Исходный код

```cpp
#pragma once

#include <vector>
#include <functional>

#include "polynomial.hpp"

template <class T>
Polynomial<T> LagrangePolynomial(const std::vector<T>& x, const std::vector<T>& y) {
  if (x.size() != y.size()) {
    throw std::runtime_error("Incompatible arrays");
  }
  Polynomial<T> p(x.size() - 1);

  T div;
  Polynomial<T> l;
  for (size_t i = 0; i < x.size(); ++i) {
    l.Assign(0, 1);
    div = 1;

    for (size_t j = 0; j < x.size(); ++j) {
      if (j == i) {
        continue;
      }
      div *= x[i] - x[j];
      l *= Polynomial<T>{-x[j], 1}; // x - x_j
    }
    l *= y[i] / div;
    p += l;
  }

  return p;
}

template <class T>
T DividedDifference(const std::vector<T>& x, const std::vector<T>& y) {
  if (x.size() != y.size()) {
    throw std::runtime_error("Incompatible arrays");
  }

  T res = 0, res_i;
  for (size_t i = 0; i < x.size(); ++i) {
    res_i = 1;
    for (size_t j = 0; j < x.size(); ++j) {
      if (i == j) {
        continue;
      }
```

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

2 Интерполяция кубическими сплайнами

2.1 Постановка задачи

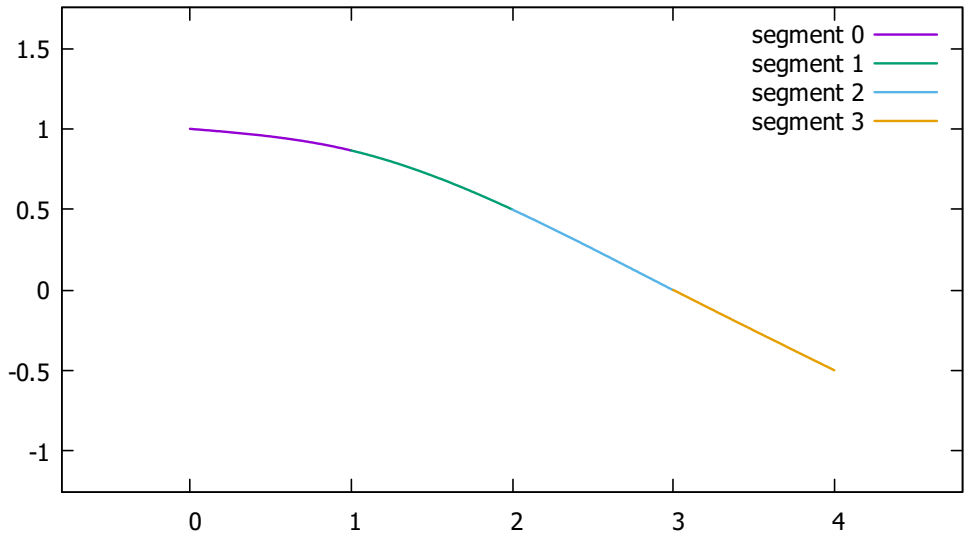
Построить кубический сплайн для функции, заданной в узлах интерполяции, предполагая, что сплайн имеет нулевую кривизну при $x = x_0$ и $x = x_4$. Вычислить значение функции в точке $x = X^*$

Вариант: $X^* = 1.5$

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

2.2 Результаты работы

```
$ ./lab3_2 <tests/2/2.txt
Значение в точке 1.5000: 0.7109
Сегменты кубического сплайна:
[0.0000:1.0000] -0.0526 * x**3 + 0.0000 * x**2 -0.0814 * x**1 + 1.0000
[1.0000:2.0000] 0.0309 * x**3 -0.2504 * x**2 + 0.1691 * x**1 + 0.9165
[2.0000:3.0000] 0.0271 * x**3 -0.2279 * x**2 + 0.1239 * x**1 + 0.9466
[3.0000:4.0000] -0.0054 * x**3 + 0.0651 * x**2 -0.7550 * x**1 + 1.8255
```



2.3 Исходный код

```
#pragma once

#include <vector>
#include <iostream>

#include "polynomial.hpp"
#include "../linear/tridiagonal_matrix.hpp"
#include "../linear/vector.hpp"

template <class T>
class CubicSpline {
public:
  std::vector<T> x;
  std::vector<T> y;
  std::vector<Polynomial<T>> segment;

  CubicSpline(const std::vector<T>& x, const std::vector<T>& y) : x(x), y(y), segment(
      x.size() - 1) {
    if (x.size() != y.size()) {
      throw std::runtime_error("Incompatible arrays");
    }

    std::vector<T> h(x.size() - 1);
    for (size_t i = 1; i < x.size(); ++i) {
      h[i-1] = x[i] - x[i-1];
    }

    size_t n = h.size();
    TDMatrix<T> matrix(n - 1);
    Vector<T> vec(n - 1);

    matrix.b[0] = 2 * (h[0] + h[1]);
    matrix.c[0] = h[1];
    vec[0] = 3 * ((y[2] - y[1]) / h[1] - (y[1] - y[0]) / h[0]);

    for (size_t i = 2; i < n - 1; ++i) {
      matrix.a[i-1] = h[i-1];
      matrix.b[i-1] = 2 * (h[i-1] + h[i]);
      matrix.c[i-1] = h[i];
      vec[i-1] = 3 * ((y[i+1] - y[i]) / h[i] - (y[i] - y[i-1]) / h[i-1]);
    }

    matrix.a[n-2] = h[n-2];
    matrix.b[n-2] = 2 * (h[n-2] + h[n-1]);
    vec[n-2] = 3 * ((y[n] - y[n-1]) / h[n-1] - (y[n-1] - y[n-2]) / h[n-2]);

    std::vector<T> a(n), b(n), c = matrix.Solve(vec), d(n);
```

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

3 Метод наименьших квадратов

3.1 Постановка задачи

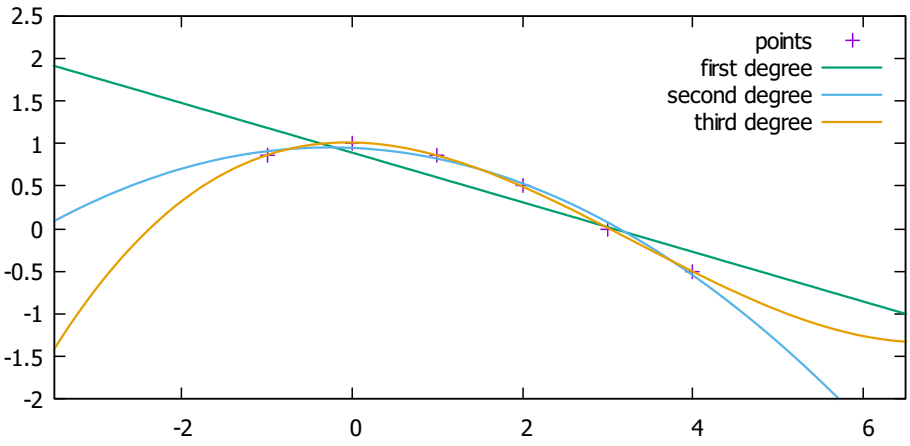
Для таблично заданной функции путем решения нормальной системы МНК найти приближающие многочлены а) 1-ой и б) 2-ой степени. Для каждого из приближающих многочленов вычислить сумму квадратов ошибок. Построить графики приближаемой функции и приближающих многочленов.

Вариант:

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

3.2 Результаты работы

```
$ ./lab3_3 <tests/3/2.txt
Приближающий многочлен 1-ой степени:
-0.2913 * x**1 + 0.8923
Ошибка: 0.2708
Приближающий многочлен 2-ой степени:
-0.0827 * x**2 -0.0431 * x**1 + 0.9475
Ошибка: 0.0152
Приближающий многочлен 3-ой степени:
0.0151 * x**3 -0.1508 * x**2 -0.0174 * x**1 + 1.0110
Ошибка: 0.0003
```



3.3 Исходный код

```
#pragma once

#include <functional>
#include <vector>

#include "../linear/matrix.hpp"
#include "../linear/vector.hpp"
#include "../linear/lup.hpp"

template <class T>
std::vector<T> LSM(const std::vector<T>& x, const std::vector<T>& y, const std::vector
    <std::function<T(T)>>& basis) {
  if (x.size() != y.size()) {
    throw std::runtime_error("Incompatible arrays");
  }

  Matrix<T> Phi(x.size(), basis.size());
  for (size_t i = 0; i < x.size(); ++i) {
    for (size_t j = 0; j < basis.size(); ++j) {
      Phi[i][j] = basis[j](x[i]);
    }
  }
  Matrix<T> PhiT = Phi.Transpose();
```

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

4 Численное дифференцирование

4.1 Постановка задачи

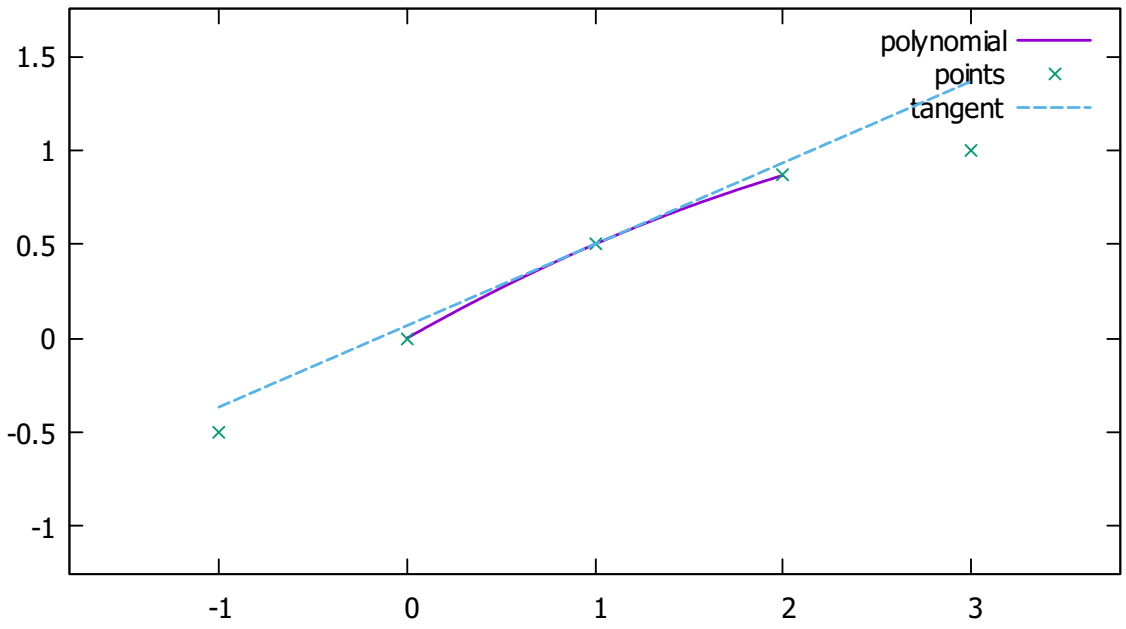
Вычислить первую и вторую производную от таблично заданной функции $y_i = f(x_i)$, $i = 0, \ldots, 4$ в точке $x = X^*$.

Вариант: $X^* = 1.0$

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

4.2 Результаты работы

```
$ ./lab3_4 <tests/4/2.txt
Первая производная: 0.4330
Вторая производная: -0.1340
```



4.3 Исходный код

```
#pragma once

#include <vector>
#include <stdexcept>

template <class T>
T TableFirstDerivative(const std::vector<T>& x, const std::vector<T>& y, T value) {
  if (x.size() != y.size()) {
    throw std::runtime_error("Incompatible arrays");
  }

  if (x.size() < 3) {
    throw std::runtime_error("Too few points");
  }

  if (value < x[0] || value > x.back()) {
    throw std::runtime_error("Out of range");
  }

  size_t i = 0;
  T min = value - x[0];
  for (size_t j = 1; j < x.size(); ++j) {
    if (min > std::abs(value - x[j])) {
      min = std::abs(value - x[j]);
      i = j - 1;
    }
  }

  if (i == x.size() - 2) {
    --i;
  }

  return (y[i+1] - y[i]) / (x[i+1] - x[i]) + ((y[i+2] - y[i+1]) / (x[i+2] - x[i+1]) -
      (y[i+1] - y[i]) / (x[i+1] - x[i])) / (x[i+2] - x[i]) * (2 * value - x[i] - x[i
      +1]);
}

template <class T>
T TableSecondDerivative(const std::vector<T>& x, const std::vector<T>& y, T value) {
  if (x.size() != y.size()) {
    throw std::runtime_error("Incompatible arrays");
  }

  if (x.size() < 3) {
    throw std::runtime_error("Too few points");
  }

```

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

5 Численное интегрирование

5.1 Постановка задачи

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

Вариант:

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

5.2 Результаты работы

```
$ cat tests/5/1.txt
0 4 1

$ ./lab3_5 <tests/5/1.txt
Метод прямоугольников: 0.05816
Погрешность: 0.01816

Метод трапеций: 0.06597
Погрешность: 0.00345

Метод Симпсона: 0.06942
Погрешность: 0.00038

$ cat tests/5/2.txt
0 4 0.5

$ ./lab3_5 <tests/5/2.txt
Метод прямоугольников: 0.06550
Погрешность: 0.00734

Метод трапеций: 0.06941
Погрешность: 0.00114

Метод Симпсона: 0.07055
Погрешность: 0.00008
```

5.3 Исходный код

```cpp
#pragma once

#include <functional>
#include <cmath>

#include "polynomial.hpp"
#include "interpolation_polynomial.hpp"

template <class T>
T RectangleMethod(const std::function<T(T)>& f, T a, T b, T h) {
  T res = 0;
  int count = std::floor((b - a) / h);
  T x = a;
  for (int i = 0; i < count; ++i, x += h) {
    res += f(x) * h;
  }
  return res;
}

template <class T>
T TrapeziumMethod(const std::function<T(T)>& f, T a, T b, T h) {
  T res = 0;
  int count = std::floor((b - a) / h);
  T prev_x = a, x = a + h;
  for (int i = 0; i < count; ++i, prev_x = x, x += h) {
    res += (f(prev_x) + f(x)) / 2 * h;
  }
  return res;
}

template <class T>
T SimpsonMethod(const std::function<T(T)>& f, T a, T b, T h) {
  T res = 0;
  int count = std::floor((b - a) / h) / 2;
  T x = a, next_x;
  for (int i = 0; i < count; ++i, x = next_x) {
    next_x = x + h + h;
    Polynomial<T> p = LagrangePolynomial<T>({x, x+h, next_x}, {f(x), f(x+h), f(next_x)
        });
    res += p.Integrate(x, next_x);
  }
  return res;
}

template <class T>
T RungeRombergError(const std::function<T(std::function<T(T)>, T, T, T)>&
    IntegrationMethod, const std::function<T(T)>& f, T a, T b, T h, T r, int p) {
```

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

4 Численные методы решения обыкновенных дифференциальных уравнений

1 Решение задачи Коши для ОДУ второго порядка

1.1 Постановка задачи

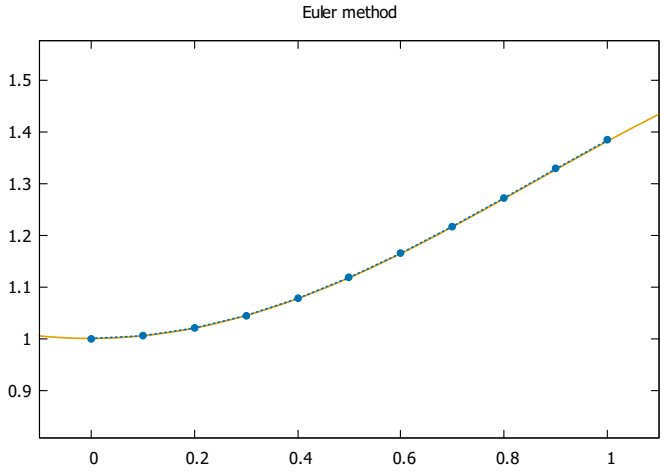
Реализовать методы Эйлера, Рунге-Кутты и Адамса 4-го порядка в виде программ, задавая в качестве входных данных шаг сетки h. С использованием разработанного программного обеспечения решить задачу Коши для ОДУ 2-го порядка на указанном отрезке. Оценить погрешность численного решения с использованием метода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

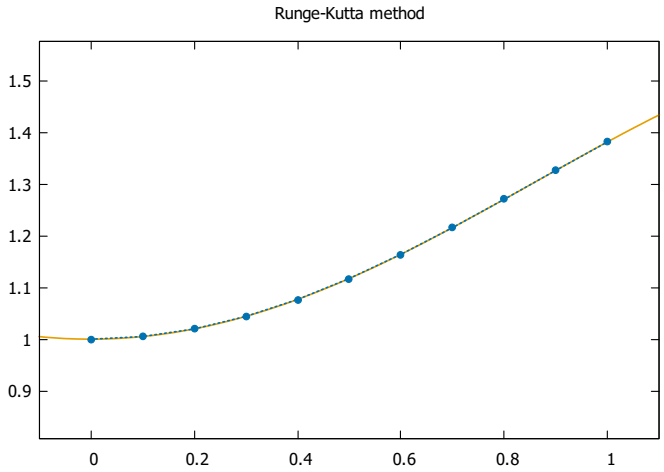
Вариант:

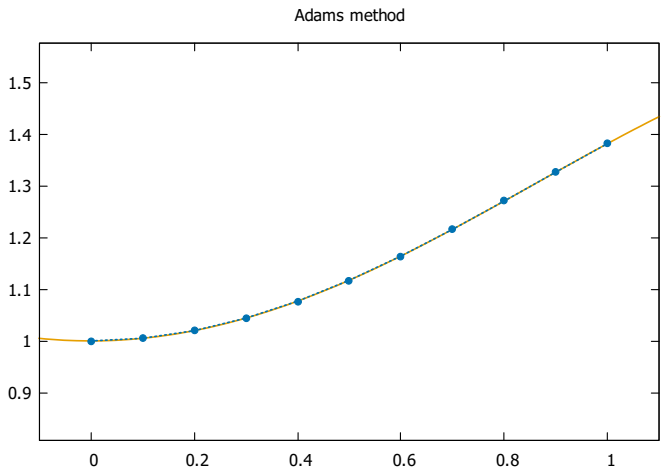
$y'' + y - 2\cos x = 0$
$y(0) = 0$
$y'(0) = 1$
$x \in [0, 1],\ h = 0.1$
Точное решение: $y = x \sin x + \cos x$

1.2 Результаты работы

```
$ cat tests/1/1.txt
0 1
1
0
0.1
$ ./lab4_1 <tests/1/1.txt
Метод Эйлера:
1 1.005 1.01988 1.04418 1.07717 1.11783 1.16488 1.21681 1.27188 1.32817 1.38362
Погрешность: 0.00181351

Метод Рунге-Кутты:
1 1.00499 1.0198 1.04399 1.07683 1.1173 1.16412 1.21579 1.27059 1.3266 1.38177
Погрешность: 1.11895e-06

Метод Адамса:
1 1.00499 1.0198 1.04399 1.07683 1.11729 1.16412 1.2158 1.27059 1.32661 1.38178
Погрешность: 1.60996e-06
```


Euler method
1.5
1.4
1.3
1.2
1.1
1
0.9
0
0.2
0.4
0.6
0.8
1


Runge-Kutta method
1.5
1.4
1.3
1.2
1.1
1
0.9
0
0.2
0.4
0.6
0.8
1


Adams method
1.5
1.4
1.3
1.2
1.1
1
0.9
0
0.2
0.4
0.6
0.8
1

1.3 Исходный код

```cpp
#pragma once

#include <functional>
#include <vector>

template <class T>
using function_t = std::function<T(T, T, T)>;

const double EPS = 1e-6;

template <class T>
std::vector<std::vector<T>> EulerMethod(const function_t<T>& f, const function_t<T>& g
    , T y0, T z0, T start, T end, T h) {
  std::vector<std::vector<T>> res(3, std::vector<T>());
  T x = start, y = y0, z = z0;
  res[0].push_back(x);
  res[1].push_back(y);
  res[2].push_back(z);

  T pred_y, pred_z, next_x, next_y, next_z;
  while (x + h < end || std::abs(end - x - h) < EPS) {
    pred_y = y + h * f(x, y, z);
    pred_z = z + h * g(x, y, z);

    next_x = x + h;
    next_y = y + h * (f(x, y, z) + f(next_x, pred_y, pred_z)) / 2;
    next_z = z + h * (g(x, y, z) + g(next_x, pred_y, pred_z)) / 2;

    x = next_x;
    y = next_y;
    z = next_z;

    res[0].push_back(x);
    res[1].push_back(y);
    res[2].push_back(z);
  }

  return res;
}

template <class T>
std::vector<std::vector<T>> RungeKuttaMethod(const function_t<T>& f, const function_t<
    T>& g, T y0, T z0, T start, T end, T h) {
  std::vector<std::vector<T>> res(3, std::vector<T>());
  T x = start, y = y0, z = z0;
  res[0].push_back(x);
  res[1].push_back(y);
```

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

```
            19 * f(res[0][k], res[1][k], res[2][k]) -
            5 * f(res[0][k-1], res[1][k-1], res[2][k-1]) +
                f(res[0][k-2], res[1][k-2], res[2][k-2])) / 24;

    dz = h * (9 * g(x + h, pred_y, pred_z) +
            19 * g(res[0][k], res[1][k], res[2][k]) -
            5 * g(res[0][k-1], res[1][k-1], res[2][k-1]) +
                g(res[0][k-2], res[1][k-2], res[2][k-2])) / 24;

    x += h;
    y += dy;
    z += dz;
    k += 1;

    res[0].push_back(x);
    res[1].push_back(y);
    res[2].push_back(z);
  }

  return res;
}

template <class T>
T RungeRombergError(const std::vector<T>& uh, const std::vector<T>& urh, int p) {
  double r = 2;
  double coeff = 1. / (std::pow(r, p) - 1);
  T err = 0;
  for (size_t i = 0; i < urh.size(); ++i) {
    err = std::max(err, std::abs(uh[2 * i] - urh[i]) * coeff);
  }
  return err;
}
```

2 Решение краевой задачи для ОДУ второго порядка

2.1 Постановка задачи

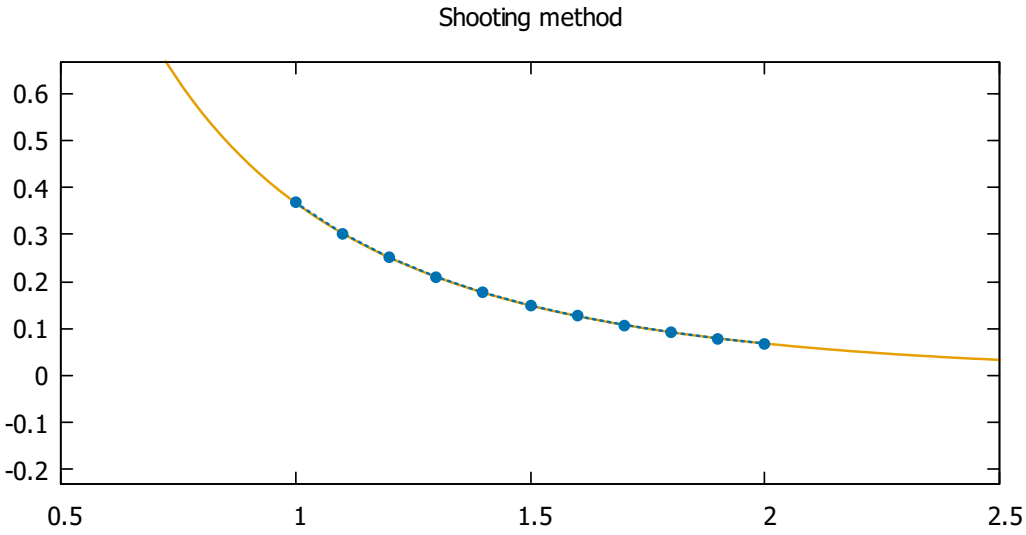
Реализовать метод стрельбы и конечно-разностный метод решения краевой задачи для ОДУ в виде программ. С использованием разработанного программного обеспечения решить краевую задачу для обыкновенного дифференциального уравнения 2-го порядка на указанном отрезке. Оценить погрешность численного решения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

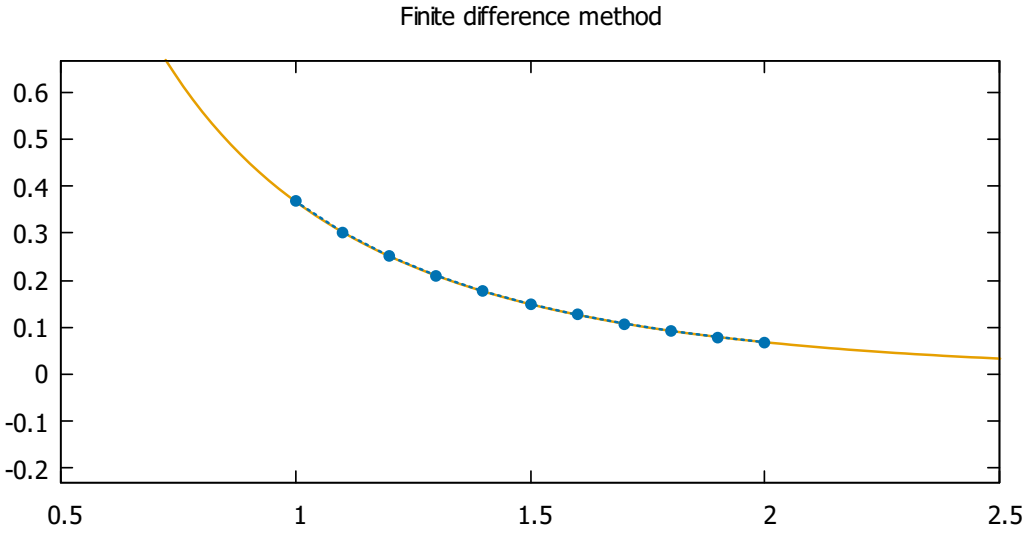
Вариант:

$xy'' + 2y' - xy = 0$
$y(1) = e^{-1}$
$y(2) = 0.5e^{-2}$
Точное решение: $y = \dfrac{e^{-x}}{x}$

2.2 Результаты работы

```
0.1
Метод стрельбы:
0.367879 0.302612 0.250998 0.209643 0.176143 0.148756 0.126187 0.107462 0.0918336
0.0787208 0.0676676
Погрешность: 1.69614e-05

Метод конечных разностей:
0.367879 0.302618 0.251008 0.209655 0.176156 0.148768 0.126198 0.107471 0.0918396
0.0787239 0.0676676
Погрешность: 1.51834e-05
```


Shooting method
0.6
0.5
0.4
0.3
0.2
0.1
0
-0.1
-0.2
0.5
1
1.5
2
2.5


Finite difference method
0.6
0.5
0.4
0.3
0.2
0.1
0
-0.1
-0.2
0.5
1
1.5
2
2.5

2.3 Исходный код

```cpp
#pragma once

#include <functional>
#include <vector>

#include "cauchy_problem.hpp"
#include "../linear/tridiagonal_matrix.hpp"

template <class T>
std::vector<std::vector<T>> ShootingMethod(const function_t<T>& f, const function_t<T>& g, T y1, T y2, T start, T end, T h) {
  std::function<T(T)> Phi = [y2](T y){ return y - y2; };

  T mu0 = 1, mu1 = 2, phi0, phi1, dphi;
  std::vector<std::vector<T>> solution = RungeKuttaMethod(f, g, y1, mu0, start, end, h);
  phi0 = Phi(solution[1].back());
  solution = RungeKuttaMethod(f, g, y1, mu1, start, end, h);
  phi1 = Phi(solution[1].back());

  while (std::abs(phi1) > EPS) {
    dphi = (phi1 - phi0) / (mu1 - mu0);

    mu0 = mu1;
    mu1 -= phi1 / dphi;

    phi0 = phi1;
    solution = RungeKuttaMethod(f, g, y1, mu1, start, end, h);
    phi1 = Phi(solution[1].back());
  }

  return solution;
}

template <class T>
std::vector<std::vector<T>> FiniteDifferenceMethod(const std::function<T(T)>& p, const std::function<T(T)>& q, const std::function<T(T)>& r, T alpha1, T beta1, T gamma1, T alpha2, T beta2, T gamma2, T start, T end, T h) {
  std::vector<std::vector<T>> res(2, std::vector<T>());
  T x = start;
  res[0].push_back(x);

  std::vector<T> a, b, c, d;

  a.push_back(0);
  b.push_back(-alpha1 / h + beta1);
  c.push_back(alpha1 / h);
```

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