Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторные работы по курсу «Численные методы»

Студент: Д. А. Тарпанов
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

```
4
-8 5 8 -6
2 7 -8 -1
-5 -4  1 -6
5 -9 -2 8
-144 25 -21 103
Решение системы:
x1=9.000000
x2=-6.000000
x3=-6.000000
x4=-1.000000
Определитель: 1867.000000
Обратная матрица:
-0.392608 -0.303160 -0.101768 -0.408677
0.049813 0.043921 -0.077129 -0.014997
-0.089448 -0.186395 -0.087306 -0.155865
0.279057 0.192287 -0.044992 0.324585
```

1.3 Исходный код

```
#include "matrix.cpp"

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

template <class T>
class LU {
private:
    const T EPS = 1e-6;

    Matrix<T> L, U;
    T determinant;
    std::vector<std::pair<int,int>> swaps;

    void decompose() {
        int n = U.Size();
        for (int i = 0; i < n; ++i) {
            int maxEl = i;
            for (int j = i + 1; j < n; ++j) {
                if (abs(U[j][i]) > abs(U[maxEl][i])) {
                    maxEl = j;
                }
            }
            if (maxEl != i) {
                std::pair<int,int> swap = std::make_pair(i, maxEl);
                swaps.push_back(swap);
                U.swapRows(i, maxEl);
                L.swapRows(i, maxEl);
                L.swapColumns(i, maxEl);
            }
            for (int j = i + 1; j < n; ++j) {
                if (abs(U[i][i]) < EPS) {
                    continue;
                }
                T m = U[j][i]/U[i][i];
                L[j][i] = m;
                for (int k = 0; k < n; ++k) {
                    U[j][k] -= m * U[i][k];
                }
            }
        }
        if (swaps.size() % 2 == 1) {
            determinant = -1;
        } else determinant = 1;
        for (int i = 0; i < n; ++i) {
            determinant *= U[i][i];
```

```
48          }
49      }
50      void Swap(std::vector<T> &a) {
51          for (auto &i : swaps) {
52              std::swap(a[i.first], a[i.second]);
53          }
54      }
55  public:
56      LU(const Matrix<T> &m) : L(m.Size(), true), U(m){
57          decompose();
58      }
59      friend std::ostream & operator << (std::ostream &out, const LU<T> lu) {
60          out << "Lower matrix:\n" << lu.L << "Upper matrix:\n" << lu.U;
61          return out;
62      }
63      T Determinant() {
64          return determinant;
65      }
66      std::vector<T> solve(std::vector<T> b){
67          int n = b.size();
68          Swap(b);
69          std::vector<T> z(n);
70          for (int i = 0; i < n; ++i) { //Lz = b
71              T summ = b[i];
72              for (int j = 0; j < i; ++j){
73                  summ -= z[j] * L[i][j];
74              }
75              z[i] = summ;
76          }
77          std::vector<T> x(n);
78          for (int i = n - 1; i >= 0; --i) { // Ux = z
79              if (abs(U[i][i]) < EPS) {
80                  continue;
81              }
82              T summ = z[i];
83              for (int j = n - 1; j > i; --j) {
84                  summ -= x[j] * U[i][j];
85              }
86              x[i] = summ/U[i][i];
87          }
88          return x;
89      }
90      ~LU() = default;
91  };
```

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

```
5
10 -1
-8 16 1
6 -16 6
-8 16 -5
5 -13
16 -110 24 -3 87
Решение системы:
x1 = 1.000000
x2 = -6.000000
x3 = -6.000000
x4 = -6.000000
x5 = -9.000000
```

2.3 Исходный код

```
#include <iostream>
#include <vector>

template <class T>
class Tridiag {
private:
    const T EPS = 1e-6;

    int n;
    std::vector<T> a, b, c;
public:
    Tridiag(const int &size) : n(size), a(n), b(n), c(n) {}

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

    std::vector<T> Solve(const std::vector<T> &d) {
        std::vector<T> p(n);
        p[0] = -c[0] / b[0];
        std::vector<T> q(n);
        q[0] = d[0] / b[0];
        for (int i = 1; i < n; ++i) {
            p[i] = -c[i] / (b[i] + a[i]*p[i-1]);
            q[i] = (d[i] - a[i]*q[i-1])/(b[i] + a[i]*p[i-1]);
        }
        std::vector<T> x(n);
        x.back() = q.back();
        for (int i = n - 2; i >= 0; --i) {
            x[i] = p[i] * x[i+1] +q[i];
        }
        return x;
    }
    ~Tridiag() = default;
};
```

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

```
4 0.000001
15 0 7 5
-3 -14 -6 1
-2 9 13 2
4 -1 3 9
176 -111 74 76
Метод простых итераций,решение системы:
x1 = 9.000000
x2 = 5.000000
x3 = 3.000000
x4 = 4.000000
Количество итераций: 88
Метод Зейделя,решение системы:
x1 = 9.000000
x2 = 5.000000
x3 = 3.000000
x4 = 4.000000
Количество итераций: 31
```

3.3 Исходный код

```cpp
#ifndef ITERATION_HPP
#define ITERATION_HPP

#include <cmath>
#include "matrix.cpp"

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

```
3 0.000001
-8 9 6
9 9 1
6 1 8
Собственные значения:
l1 = -13.141391
l2 = 14.757377
l3 = 7.384014
Собственные векторы:
0.903164 0.429261 0.005498
-0.356301 0.756677 -0.548168
-0.239468 0.493127 0.836350
Количество итераций: 7
```

4.3 Исходный код

```
//    A^T = A
#include <cmath>
#include "matrix.cpp"

class Rotation {
private:
    int n;
    Matrix<double> a;
    double eps;
    Matrix<double> v;

    static double EndCondition(const Matrix<double> &m) {
        int n = m.Size();
        double result = 0;
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                if (i == j) {
                    continue;
                }
                result += m[i][j] * m[i][j]; //
            }
        }
        return std::sqrt(result);
    }

    double CalculatePhi(int i, int j) {
        if (std::abs(a[i][i] - a[j][j]) < 1e-9) {
            return std::atan(1.0); //pi/4
        } else {
            return 0.5 * std::atan(2*a[i][j]/(a[i][i] - a[j][j]));
        }
    }

    Matrix<double> CreateRotationMatrix(int i, int j, double phi) {
        Matrix<double> u(n, true);
        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;
    }

    void Build() {
        count = 0;
        while(EndCondition(a) > eps) {
            ++count;
            int max_i = 0, max_j = 1;
```

```
48            for (int i = 0; i < n; ++i) {
49                for (int j = 0; j < n; ++j) {
50                    if (i == j) {
51                        continue;
52                    }
53                    if (std::abs(a[i][j]) > std::abs(a[max_i][max_j])) {
54                        max_i = i;
55                        max_j = j;
56                    }
57                }
58            }
59            double phi = CalculatePhi(max_i, max_j);
60            Matrix<double> u = CreateRotationMatrix(max_i, max_j, phi);
61            v = v * u;
62            a = u.Transponse() * a * u;
63        }
64    }
65
66 public:
67    int count;
68
69    Rotation(const Matrix<double> &_a, double _eps): n(_a.Size()), a(_a), eps(_eps), v(
          n, true){
70        Build();
71    }
72    Matrix<double> getEigenVectors() {
73        return v;
74    }
75    std::vector<double> getEigenValues() {
76        std::vector<double> result(n);
77        for (int i = 0; i < n; ++i) {
78            result[i] = a[i][i];
79        }
80        return result;
81    }
82
83    ~Rotation() = default;
84 };
```

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

```
3 0.00001
0 -1 3
-1 6 -3
-8 4 2
Количество итераций: 27
Собственные значения:
l_1 = 2.31146 + i * 5.10345
l_2 = 2.31146 + i * -5.10345
l_3 = 3.37709
```

5.3 Исходный код

```cpp
#include "matrix.cpp"
#include <complex>
#include <cmath>

class QR {
private:
    static constexpr double INF = 1e18;
    static constexpr std::complex<double> COMP_INF = std::complex<double>(INF, INF);

    int n;
    Matrix<double> a;
    double eps;
    std::vector<std::complex<double>> eigen;

    // V^T * V
    double vtv(const std::vector<double> &vec) {
        double res = 0;
        for (auto elem: vec) {
            res += elem*elem;
        }
        return res;
    }

    double norm2(const std::vector<double> &vec) {
        return std::sqrt(vtv(vec));
    }

    Matrix<double> vvt(const std::vector<double> &vec) {
        int size = vec.size();
        Matrix<double> res(size);
        for (int i = 0; i < size; ++i) {
            for (int j = 0; j < size; ++j) {
                res[i][j] = vec[i] * vec[j];
            }
        }
        return res;
    }

    double sign(double x) {
        if (x < eps) {
            return -1.0;
        } else if (x > eps) {
            return 1.0;
        }
    }


```

```
48      Matrix<double> HH(const std::vector<double> &vec, int el) {
49          std::vector<double> v(vec);
50          v[el] += sign(vec[el]) * norm2(vec);
51          return Matrix<double>(n, true) - (2.0 / vtv(v)) * vvt(v); //E - (2/vtv)*vvt
52      }
53
54      // (ajj - l)(aj1j1 - l) = ajj1 * aj1j
55      // ajj * aj1j1 - ajj * l - aj1j1 * l - ajj1 * aj1j + l*l = 0
56      // l^2 - l * (-ajj - aj1j1) + ajj1 * aj1j - ajj1 * aj1j = 0
57      std::pair<std::complex<double>, std::complex<double>> eq_solver(double ajj, double
            aj1j1, double ajj1, double aj1j) {
58          double a = 1.0;
59          double b = - (ajj + aj1j1);
60          double c = ajj * aj1j1 - aj1j * ajj1;
61          double dd = b * b - 4 * a * c;
62          if (dd > eps) {
63              std::complex<double> bad(NAN, NAN);
64              return std::make_pair(bad, bad);
65          }
66          std::complex<double> d(0.0, std::sqrt(-dd));
67          std::complex<double> x1 = (-b + d) / (2.0 * a);
68          std::complex<double> x2 = (-b - d) / (2.0 * a);
69          return std::make_pair(x1,x2);
70      }
71
72      void calcEigen() {
73          for (int i = 0; i < n; ++i) {
74              if (i < n - 1 && !(std::abs(a[i+1][i]) < eps)) { //if subdiagonal are not
                    small
75                  auto [l1, l2] = eq_solver(a[i][i], a[i+1][i+1], a[i][i+1], a[i+1][i]);
76                  if (std::isnan(l1.real())) {
77                      ++i;
78                      continue;
79                  }
80                  eigen[i] = l1;
81                  eigen[++i] = l2;
82              } else {
83                  eigen[i] = a[i][i];
84              }
85          }
86      }
87
88      bool checkDiag(){
89          for (int i = 0; i < n; ++i) {
90              double sum = 0;
91              for (int j = i + 2; j < n; ++j) {
92                  sum += a[j][i] * a[j][i];
93              }
94              double norm = std::sqrt(sum);
```

```
95              if (!(norm <eps)) {
96                  return false;
97              }
98          }
99          return true;
100     }
101
102     bool checkEps() {
103         if (!checkDiag()) {
104             return false;
105         }
106         std::vector<std::complex<double>> prev(eigen);
107         calcEigen();
108         for (int i = 0; i < n; ++i) {
109             double delta = std::norm(eigen[i] - prev[i]);
110             if (delta > eps) {
111                 return false;
112             }
113         }
114         return true;
115     }
116
117     void build() {
118         iter_count = 0;
119         while (!checkEps()) {
120             ++iter_count;
121             Matrix<double> q(n, true);
122             Matrix<double> r(a);
123             for (int i = 0; i < n - 1; ++i){
124                 std::vector<double> b(n);
125                 for (int j = i; j < n; ++j) {
126                     b[j] = r[j][i];
127                 }
128                 Matrix<double> h = HH(b,i);
129                 q = q * h; // Q_k = H1*H2*...*H_k
130                 r = h * r;
131             }
132             a = r * q;
133         }
134     }
135
136 public:
137     int iter_count;
138
139     QR(const Matrix<double> &_a, double _eps) : n(_a.Size()), a(_a), eps(_eps), eigen(n
        , COMP_INF) {
140         build();
141     }
142     std::vector<std::complex<double>> getEigen() {
```

```
143         calcEigen();
144         return eigen;
145     }
146 
147     ~QR() = default;
148 };
```

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

```
-0.5 0 0.0000001

Метод простой итерации. Количество итераций: 10
x_0 = -0.474626594
Метод Ньютона. Количество итераций: 3
x_0 = -0.474626618
```

1.3 Исходный код

```cpp
#pragma once

#include <cmath>
#include <algorithm>

int iterCount = 0;

double f(double x) {
    return pow(x, 4) - 2 * x - 1;
}

double der(double x) {
    return 4 * pow(x, 3) - 2;
}

double der2(double x) {
    return 12 * pow(x, 2);
}

double phi(double x) {
    return 0.5 * pow(x, 4) - 0.5;
}

double phi_der(double x) {
    return 2 * pow(x,3);
}

double iterations(double l, double r, double eps) {
    iterCount = 0;
    double x_k = (l + r)/2;
    double dx = 1;
    double q = std::max(std::abs(phi_der(l)), std::abs(phi_der(r)));
    double eps_coeff = q / (1 - q);
    while(dx > eps){
        double x_k1 = phi(x_k);
        dx = eps_coeff * std::abs(x_k1 - x_k);
        ++iterCount;
        x_k = x_k1;
    }
    return x_k;
}

double newton(double l, double r, double eps) {
    iterCount = 0;
    double x_k = l;
    if (f(l)*f(r) >=0) {
        return 0;
```

```
48      }
49      if (f(x_k) * der2(x_k) <= 0) {
50          x_k = r;
51      }
52      double dx = 1;
53      while (dx > eps) {
54          double x_k1 = x_k - f(x_k) / der(x_k);
55          dx = std::abs(x_k1 - x_k);
56          ++iterCount;
57          x_k = x_k1;
58      }
59      return x_k;
60  }
```

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

```
1 2
1.3 2.1
0.00001
Решение методом простых итераций: 1.275780 2.059608
Количество итераций: 8
Решение методом Ньютона: 1.275762 2.059607
Количество итераций: 5
```

2.3 Исходный код

```cpp
//
// Created by kng on 23.05.22.
//

#ifndef NM2_2_SOLVER_H
#define NM2_2_SOLVER_H

#include "lu.cpp"

int iter_count = 0;

const double a = 1;

double phi1(double x1, double x2) {
    return std::sqrt(2 * log10(x2) + 1);
}

double phi2(double x1, double x2) {
    return (x1*x1 + a)/(x1 * a);
}

double phi1_der(double x1, double x2) {
    return 1/(x2 * std::sqrt(log(10)) * std::sqrt(2*log(x2) + log(10)));
}

double phi2_der(double x1, double x2) {
    return (a - 1/(x1*x1*a));
}

double phi(double x1, double x2) {
    return phi1_der(x1, x2) * phi2_der(x1, x2);
}

std::pair<double, double> iter_solve(double l1, double r1, double l2, double r2,
    double eps) {
    iter_count = 0;
    double x1_k = r1;
    double x2_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_coeff = q/(1-q);
    double dx = 1;
    while (dx > eps) {
        double x1_k1 = phi1(x1_k, x2_k);
```

```
47          double x2_k1 = phi2(x1_k, x2_k);
48          dx = eps_coeff * (std::abs(x1_k1 - x1_k) + std::abs(x2_k1 - x2_k));
49          ++iter_count;
50          x1_k = x1_k1;
51          x2_k = x2_k1;
52      }
53      return std::make_pair(x1_k, x2_k);
54  }
55
56  double f1(double x1, double x2) {
57      return x1*x1 - 2 * log10(x2) - 1;
58  }
59
60  double f2(double x1, double x2) {
61      return x1*x1 - a*x1*x2 + a;
62  }
63
64  Matrix<double> Jacobi(double x1, double x2) {
65      Matrix<double> result(2);
66      result[0][0] = 2 * x1;
67      result[0][1] = -2/(x2*log(10));
68      result[1][0] = 2 * x1 - a * x2;
69      result[1][1] = -a * x1;
70      return result;
71  }
72
73  double norm(const std::vector<double> &vec) {
74      double res = 0;
75      for (auto elem: vec) {
76          res = std::max(res, std::abs(elem));
77      }
78      return res;
79  }
80
81  std::pair<double, double> newton_solve(double x1_0, double x2_0, double eps) {
82      iter_count = 0;
83      std::vector<double> x_k = {x1_0, x2_0};
84      double dx = 1;
85      while (dx > eps) {
86          double x1 = x_k[0];
87          double x2 = x_k[1];
88          LU<double> J(Jacobi(x1, x2));
89          std::vector<double> f_k = {f1(x1,x2), f2(x1,x2)};
90          std::vector<double> d_x = J.solve(f_k);
91          std::vector<double> x_k1 = x_k - d_x;
92          dx = norm(x_k1 - x_k);
93          ++iter_count;
94          x_k = x_k1;
95      }
```

```
96      return std::make_pair(x_k[0], x_k[1]);
97  }
98
99
100 #endif //NM2_2_SOLVER_H
```

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

```
4
-0.4 -0.1 0.2 0.5
0.1
Интерполяционный многочлен Лагранжа: 7.55373e-05 + 1.99923 * x + 0.00197657
* x^2 + 0.188516 * x^3
Погрешность в точке X*: 0.000111543
Интерполяционный многочлен Ньютона: 7.55373e-05 + 1.99923 * x + 0.00197657
* x^2 + 0.188516 * x^3
Погрешность в точке X*: 0.000111543
```

1.3 Исходный код

```
1  //
2  // Created by kng on 23.05.22.
3  //
4
5  #ifndef NM3_1_INTERPOLATION_H
6  #define NM3_1_INTERPOLATION_H
7
8  #include "polynom.h"
9
10 class Lagrange {
11     std::vector<double> x;
12     std::vector<double> y;
13     int n;
14
15 public:
16     Lagrange(const std::vector<double> & _x, const std::vector<double> & _y) : x(_x), y
           (_y), n(x.size()) {};
17
18     Polynom operator() () {
19         Polynom result(std::vector<double> ({0}));
```

```
20          for (int i = 0; i < n; ++i) {
21              Polynom li(std::vector<double>({1})); //
22              for (int j = 0; j < n; ++j) {
23                  if (i == j) {
24                      continue;
25                  }
26                  Polynom xij(std::vector<double>({-x[j], 1})); // x - x_i
27                  li = li * xij;
28                  li = li / (x[i] - x[j]);
29              }
30              result = result + y[i] * li; // Summ(f_i * (x - x_i)/(x_i - x_j))
31          }
32          return result;
33      }
34  };
35
36  class Newton {
37  private:
38      std::vector<double> x;
39      std::vector<double> y;
40      int n;
41
42      std::vector<std::vector<double>> memo;
43      std::vector<std::vector<bool>> done;
44
45      double f(int l, int r) {
46          if (done[l][r]) {
47              return memo[l][r];
48          }
49          done[l][r] = true;
50          double res;
51          if (l + 1 == r) {
52              res = (y[l] - y[r]) / (x[l] - x[r]); //    1
53          } else {
54              res = (f(l, r - 1) - f(l + 1, r)) / (x[l] - x[r]);
55          }
56          return memo[l][r] = res;
57      }
58  public:
59      Newton(const std::vector<double> & _x, const std::vector<double> & _y) : x(_x), y(
            _y), n(x.size()) {
60          memo.resize(n, std::vector<double>(n));
61          done.resize(n, std::vector<bool> (n));
62      };
63
64      Polynom operator() () {
65          Polynom res(std::vector<double>({y[0]}));
66          Polynom prod(std::vector<double>({-x[0], 1})); // x - x_0
67          int r = 0;
```

```
68          for (int i = 1; i < n; ++i) {
69              res = res + f(0, ++r) * prod;
70              prod = prod * Polynom(std::vector<double>({-x[i], 1})); // (x - x_i)*(x -
                    x_(i+1))....
71          }
72          return res;
73      }
74  };
75
76  #endif //NM3_1_INTERPOLATION_H
```

2 Сплай-интерполяция

2.1 Постановка задачи

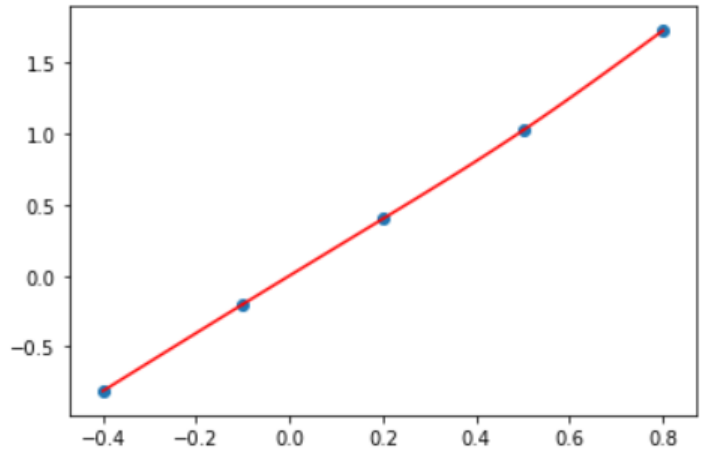
Построить кубический сплайн для функции, заданной в узлах интерполяции, предполагая, что сплайн имеет нулевую кривизну при $x = x_0$ и $x = x_4$. Вычислить значение функции в точке $x = X^*$.

2.2 Консоль

```
5
-0.4 -0.1 0.2 0.5 0.8
-0.81152 -0.20017 0.40136 1.0236 1.7273
0.1
Сплайн: i = 1,a = -0.81152,b = 2.04668,c = 0.00000,d = -0.09833
i = 2,a = -0.20017,b = 2.02013,c = -0.08850,d = 0.12796
i = 3,a = 0.40136,b = 2.00158,c = 0.02667,d = 0.71722
i = 4,a = 1.02360,b = 2.21123,c = 0.67217,d = -0.74685

Значение в точке X*: 0.20134
```

2.3 Результат



2.4 Исходный код

```cpp
//
// Created by kng on 23.05.22.
//

#ifndef NM3_2_SPLINE_H
#define NM3_2_SPLINE_H

#include <cmath>
#include "tridiag.cpp"

class Spline {
    int n;
    std::vector<double> x;
    std::vector<double> y;
    std::vector<double> a, b, c, d;

    void build() {
        std::vector<double> h(n + 1);
        h[0] = NAN;
        for (int i = 1; i <= n; ++i) {
            h[i] = x[i] - x[i - 1];
        }
        std::vector<double> eq_a(n - 1), eq_b(n - 1), eq_c(n - 1), eq_d(n - 1);
        for (int i = 2; i <= n; ++i) { // h_i-1*c_i-1 + 2(h_i-1 + h_i)*c_i + h_i*c_i+1
            = 3[(f_i - f_i-1)/h_i - (f_i-1 - f_i-2)/h_i-1]
            eq_a[i - 2] = h[i - 1];
            eq_b[i - 2] = 2 * (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;
        eq_c.back() = 0;
        Tridiag<double> system(eq_a, eq_b, eq_c);
        std::vector<double> solution = system.Solve(eq_d);
        for (int i = 2; i <= n; ++i) {
            c[i] = solution[i - 2];
        }
        for (int i = 1; i <= n; ++i) {
            a[i] = y[i - 1];
        }
        for (int 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]);
        }
        c[1] = 0.0;
        b[n] = (y[n] - y[n - 1]) / h[n] - (2.0 / 3.0) * h[n] * c[n];
```

```
46          d[n] = -c[n] / (3.0 * h[n]);
47      }
48  public:
49      Spline(const std::vector<double> &_x, const std::vector<double> &_y) {
50          x = _x;
51          y = _y;
52          n = x.size() - 1;
53          a.resize(n + 1);
54          b.resize(n + 1);
55          c.resize(n + 1);
56          d.resize(n + 1);
57          build();
58      }
59
60      friend std::ostream & operator << (std::ostream &out, const Spline &spline) {
61          for (int i = 1; i <= spline.n; ++i) {
62              out << "i = " << i << ", a = " << spline.a[i] << ", b = " << spline.b[i] <<
                        ", c = " << spline.c[i] << ",d = " << spline.d[i] << '\n';
63          }
64          return out;
65      }
66
67      double operator () (double x0) {
68          for (int i = 1; i <= n; ++i) {
69              if (x[i - 1] <= x0 && x0 <= x[i]) {
70                  double x1 = x0 - x[i - 1];
71                  double x2 = x1 * x1;
72                  double x3 = x1 * x1 * x1;
73                  return a[i] + b[i] * x1 + c[i] * x2 + d[i] * x3;
74              }
75          }
76          return NAN;
77      }
78  };
79
80  #endif //NM3_2_SPLINE_H
```

3 Метод наименьших квадратов

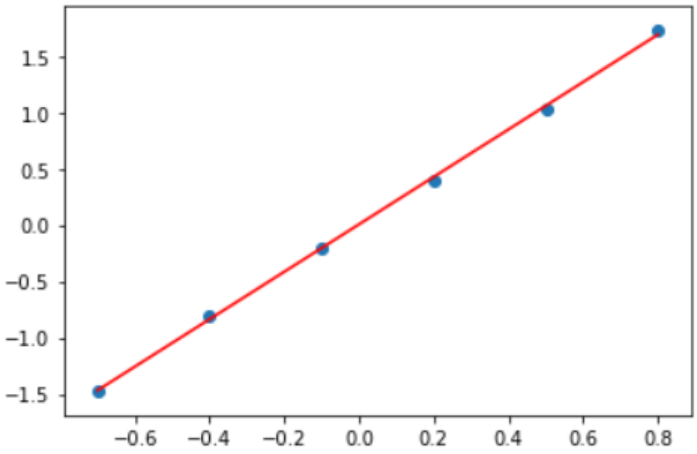
3.1 Постановка задачи

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

```
6
-0.7 -0.4 -0.1 0.2 0.5 0.8
-1.4754 -0.81152 -0.20017 0.40136 1.0236 1.7273
Приближающий многочлен первой степени: 0.00553 2.10670
Сумма квадратов ошибок: 0.00394
Приближающий многочлен второй степени: 0.00553 2.10670 0.00000
Сумма квадратов ошибок: 0.00394
```

3.3 Результат



3.4 Исходный код

```
//
// Created by kng on 23.05.22.
//

#ifndef NM3_3_MNK_H
#define NM3_3_MNK_H

#include <functional>
#include "polynom.h"
#include "lu.cpp"

class MNK {
    int n;
    std::vector<double> x, y;
    int m;
    std::vector<double> a;
    std::vector<std::function<double(double)>> phi;

    void build() {
        Matrix<double> lhs(n, m);
        for (int i = 0; i < n; ++ i) {
            for (int j = 0; j < m; ++ j) {
                lhs[i][j] = phi[j](x[i]);
            }
        }
        Matrix<double> t = lhs.t();
        LU<double> lhs_lu(t * lhs);
        std::vector<double> rhs = t * y;
        a = lhs_lu.solve(rhs);
    }

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

public:

    MNK(const std::vector<double> &_x, const std::vector<double> &_y, const std::vector
        <std::function<double(double)>> _phi) {
        n = _x.size();
        x = _x;
        y = _y;
        m = _phi.size();
```

```
47          a.resize(m);
48          phi = _phi;
49          build();
50      }
51
52      friend std::ostream & operator << (std::ostream &out, const MNK &item) {
53          for (int i = 0; i < item.m; ++i) {
54              if (i) {
55                  out << " ";
56              }
57              out << item.a[i];
58          }
59          return out;
60      }
61
62      double error() {
63          double res = 0;
64          for (int i = 0; i < n; ++i) {
65              res += std::pow(get(x[i]) - y[i], 2.0);
66          }
67          return res;
68      }
69
70      double operator () (double x0) {
71          return get(x0);
72      }
73  };
74
75  #endif //NM3_3_MNK_H
```

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

```
5
-1.0 0.0 1.0 2.0 3.0
-1.7854 0.0 1.7854 3.1071 4.249
1
Первая производная функции в точке x0 = 1.0000,f'(x0) = 1.7854
Вторая производная функции в точке x0 = 1.0000,f''(x0) = 0.0000
```

4.3 Исходный код

```
#ifndef NM3_4_TABLE_FUNCTION_H
#define NM3_4_TABLE_FUNCTION_H

#include <vector>
#include <cmath>

const double EPS = 1e-9;

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

class table_function {
    int n;
    std::vector<double> x, y;
public:
    table_function(const std::vector<double> &_x, const std::vector<double> &_y) {
        x = _x;
        y = _y;
        n = x.size();
    }

    double derivative1(double x0) {
        for (int i = 0; i < n - 1; ++i) {
            if (x[i] < x0 && leq(x0, x[i+1])) { //
                double res = (y[i + 1] - y[i - 1])/(x[i + 1] - x[i - 1]);
                return res;
            }
        }
        return NAN;
    }

    double derivative2(double x0) {
        for (int i = 0; i < n - 1; ++i) {
            if (x[i] < x0 && leq(x0, x[i+1])) {
                double res = (y[i - 1] - 2*y[i] + y[i + 1])/pow((x[i + 1] - x[i]), 2);
                return res;
            }
        }
        return NAN;
    }
};

#endif //NM3_4_TABLE_FUNCTION_H
```

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

```
0 4 1.0 0.5
Метод прямоугольников с шагом 1: 0.040489
Метод трапеций с шагом 1: 0.046395
Метод Симпсона с шагом 1: 0.0141526

Метод прямоугольников с шагом 0.5: 0.0418683
Метод трапеций с шагом 0.5: 0.043442
Метод Симпсона с шагом 0.5: 0.014131

Погрешность вычислений методом прямоугольников: 0.00183909
Погрешность вычислений методом трапеций: 0.00393738
Погрешность вычислений методом Симпсона: 2.304e-05
```

5.3 Исходный код

```cpp
#ifndef NM3_5_INTEGRATE_H
#define NM3_5_INTEGRATE_H

#include <cmath>
#include "interpolation.h"

using func = double(double);

double int_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 int_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 int_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])};
        Lagrange lagr(x, y);
        res += lagr().integrate(x1, x2);
        x1 = x2;
        x2 += h;
    }
    return res / 3;
```

```
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
55 #endif //NM3_5_INTEGRATE_H
```

4 Методы решения обыкновенных дифференциальных уравнений

1 Решение задачи Коши для ОДУ

1.1 Постановка задачи

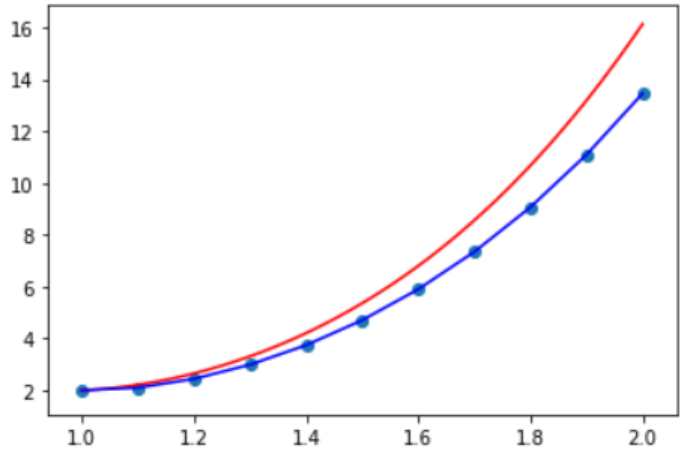
Реализовать методы Эйлера, Рунге-Кутты и Адамса 4-го порядка в виде программ, задавая в качестве входных данных шаг сетки h. С использованием разработанного программного обеспечения решить задачу Коши для ОДУ 2-го порядка на указанном отрезке. Оценить погрешность численного решения с использованием метода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

1.2 Консоль

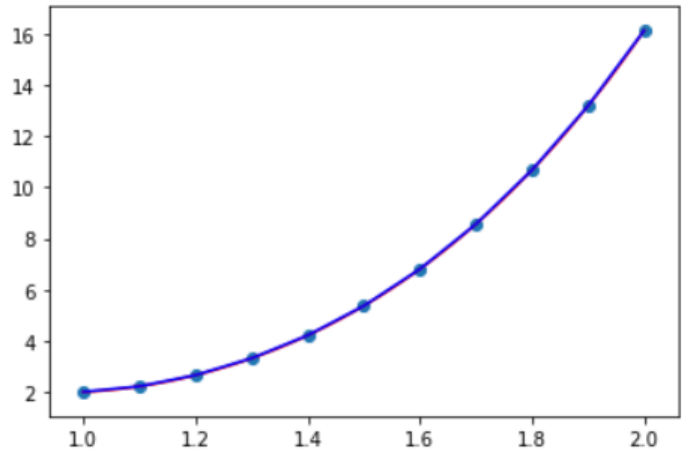
```
1 2
2 1 0.1
Метод Эйлера:
x = [1.000000,1.100000,1.200000,1.300000,1.400000,1.500000,1.600000,1.700000,1.800000
y = [2.000000,2.100000,2.440000,2.988264,3.739862,4.703645,5.896398,7.340012,9.060020
Погрешность вычислений:
1.188858
Метод Рунге-Кутты:
x = [1.000000,1.100000,1.200000,1.300000,1.400000,1.500000,1.600000,1.700000,1.800000
y = [2.000000,2.215465,2.652360,3.311306,4.206041,5.358762,6.797651,8.555485,10.66883
Погрешность вычислений:
0.000027
Метод Адамса:
x = [1.000000,1.100000,1.200000,1.300000,1.400000,1.500000,1.600000,1.700000,1.800000
y = [2.000000,2.215465,2.652360,3.311306,4.205356,5.357862,6.796735,8.554608,10.66802
Погрешность вычислений:
0.000074
```

1.3 Результат

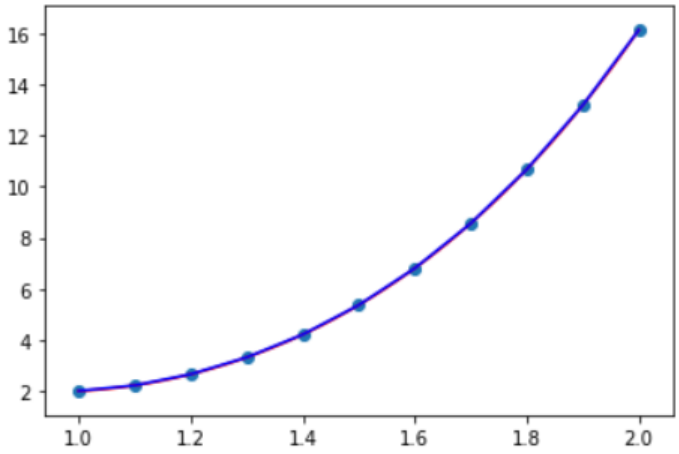
Метод Эйлера



Метод Рунге-Кутты



Метод Адамса



1.4 Исходный код

```
#ifndef NM4_1_DESOLVE_H
#define NM4_1_DESOLVE_H

#include "utils.h"
#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) { // 4
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) { //4
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             /*  */
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             /*  */
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 //NM4_1_DESOLVE_H
```

2 Решение краевых задач

2.1 Постановка задачи

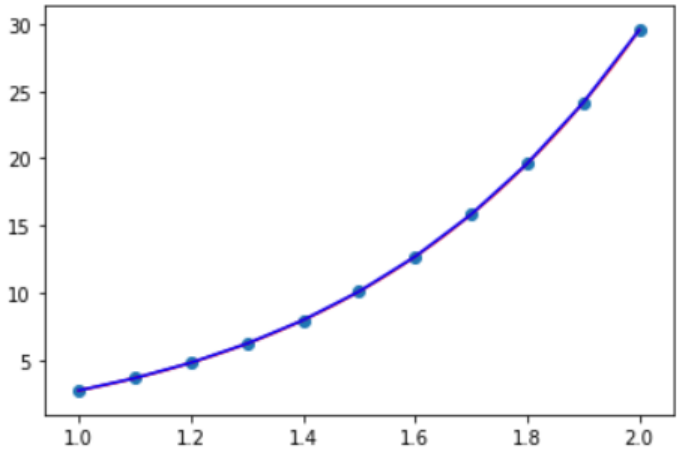
Реализовать метод стрельбы и конечно-разностный метод решения краевой задачи для ОДУ в виде программ. С использованием разработанного программного обеспечения решить краевую задачу для обыкновенного дифференциального уравнения 2-го порядка на указанном отрезке. Оценить погрешность численного решения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

2.2 Консоль

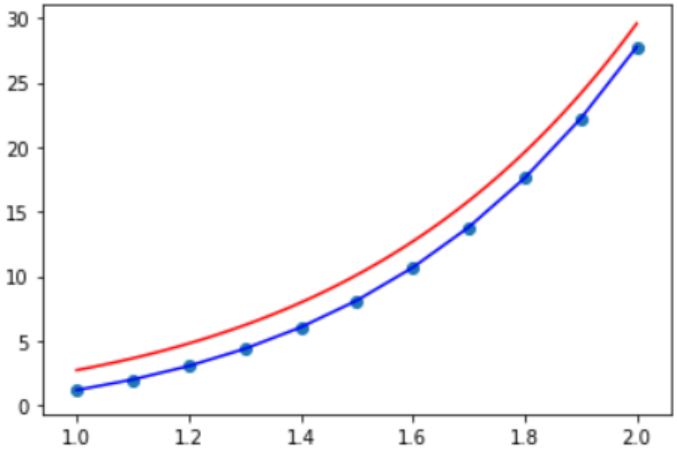
```
0.1 0.00001
Метод стрельбы:
x = [1.000000,1.100000,1.200000,1.300000,1.400000,1.500000,1.600000,1.700000,1.800000
y = [2.718310,3.635057,4.780968,6.201090,7.948142,10.083714,12.679630,15.819517,19.600
Погрешность вычислений:
0.000029
Конечно-разностный метод:
x = [1.000000,1.100000,1.200000,1.300000,1.400000,1.500000,1.600000,1.700000,1.800000
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 NM4_2_SOLVER_H
#define NM4_2_SOLVER_H

#include <cmath>
#include "tridiag.cpp"
#include "desolve.h"

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; //phi = D * y + g * z
             - 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; //      phi(eta) = 0;
    }
    //
    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);
            runge de_solver_prev(a, b, f, g, eta_prev, runge_z0_prev);
            vect sol_prev = de_solver_prev.solve(h);

```

```
46            double runge_z0 = get_start_cond(eta);
47            runge 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
61class fin_dif {
62private:
63    using fx = std::function<double(double)>;
64    using tridiag = Tridiag<double>;
65
66    double a, b;
67    fx p, q, f;
68    double alpha, beta, y0;
69    double delta, gamma, y1;
70
71public:
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] = (alpha - beta/h);
91        c[0] = beta/h;
92        d[0] = y0;
93        a.back() = -gamma/h;
94        b.back() = delta + gamma/h;
```

```
95          d.back() = 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
113 #endif //NM4_2_SOLVER_H
```