Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Отчёт по лабораторным работам по курсу
«Численные Методы»

Студент: К. О. Вахрамян
Преподаватель: Д. Л. Ревизников
Группа: М8О-306Б

Дата:
Оценка:
Подпись:

Москва, 2021

Лабораторная работа №1
«Вычислительные методы линейной алгебры»

Вариант:3

1.1

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.

$$\begin{cases} 9x_1 - 5x_2 - 6x_3 + 3x_4 = -8 \\ x_1 - 7x_2 + x_3 = 38 \\ 3x_1 - 4x_2 + 9x_3 = 47 \\ 6x_1 - x_2 + 9x_3 + 8x_4 = -8 \end{cases}$$

Метод Решения

LU – разложение матрицы A представляет собой разложение матрицы A в произведение нижней и верхней треугольных матриц, т.е.

$$A = LU,$$

где L - нижняя треугольная матрица (матрица, у которой все элементы, находящиеся выше главной диагонали равны нулю, $l_{ij} = 0$ при $i < j$), U - верхняя треугольная матрица (матрица, у которой все элементы, находящиеся ниже главной диагонали равны нулю, $u_{ij} = 0$ при $i > j$).

LU – разложение может быть построено с использованием метода Гаусса. Рассмотрим k - ый шаг метода Гаусса, на котором осуществляется обнуление поддиагональных элементов k - го столбца матрицы $A^{(k-1)}$. С этой целью используется следующая операция:

$$a_{ij}^{(k)} = a_{ij}^{(k-1)} - \mu_i^{(k)} a_{kj}^{(k-1)}, \mu_i^{(k)} = \frac{a_{ik}^{(k-1)}}{a_{kk}^{(k-1)}}, i = k+1, ..., n, j = k, ..., n$$

В терминах матричных операций такая операция эквивалентна умножению $A^{(k)} = M_k A^(k-1)$, где элементы матрицы M_k определяются следующим образом

$$m_{ij}^k = \begin{cases} 1, & i = j \\ 0, & i \neq j, j \neq k \\ -\mu_{k+1}^(k), & i \neq j, j = k \end{cases}$$

В результате прямого хода метода Гаусса получим $A(n-1) = U$,
$A = A^{(0)} = M_1^{-1}A^{(1)} = M_1^{-1}M_2^{-1}...M_{n-1}^{-1}A^{(n-1)}$,
где $A^{(n-1)} = U$ - верхняя треугольная матрица, а $L = M_1^{-1}M_2^{-1}...M_{n-1}^{-1}$ - нижняя треугольная матрица.
В дальнейшем LU – разложение может быть эффективно использовано при решении систем линейных алгебраических уравнений вида $Ax = b$. Действительно, подставляя LU – разложение в СЛАУ, получим $LUx = b$, или $Ux = L^{-1}b$. Т.е. процесс решения СЛАУ сводится к двум простым этапам.

На первом этапе решается СЛАУ $Lz = b$. Поскольку матрица системы - нижняя треугольная, решение можно записать в явном виде:

$$z_1 = b_1, z_i = b_i = \sum_{j=i}^{i-1} l_{ij}z_j, u = 2, .., n$$

На втором этапе решается СЛАУ $Ux = z$ с верхней треугольной матрицей. Здесь, как и на предыдущем этапе, решение представляется в явном виде:

$$x_n = \frac{z_n}{u_{nn}}, x_i = \frac{1}{u_{ii}}(z_i - \sum_{j=i+1}^{n} u_{ij}x_j), i = n-1, ..., 1.$$

Определитель матрицы можно вычислить как элементов матрицы U, стоящих на главной диагонали.

$$det = \prod_{i=1}^{n} u_{ii}$$

Обратну мартицу найдем, n раз решив уравнение

$$LUx_i = e_i, i = 1, ..., n$$

где e_i - столбец с единицей в строке i. Объединив $x_i, i = 1, .., n$ в одну матрицу получим обратную.

Результат работы программы:

```
L:
1.00000 0.00000 0.00000 0.00000
0.11111 1.00000 0.00000 0.00000
0.66667 -0.36207 1.00000 0.00000
```

```
0.33333 0.36207 0.76426 1.00000

U:
9.00000 -5.00000 -6.00000 3.00000
0.00000 -6.44444 1.66667 -0.33333
0.00000 0.00000 13.60345 5.87931
0.00000 0.00000 0.00000 -5.37262

L*U:
9.00000 -5.00000 -6.00000 3.00000
1.00000 -7.00000 1.00000 0.00000
3.00000 -4.00000 9.00000 0.00000
6.00000 -1.00000 9.00000 8.00000

x:
0.00000
-5.00000
3.00000
-5.00000

Det(A) = -4239.00000

A^(-1):
0.11135 -0.14933 0.13258 -0.04176
0.01132 -0.16773 0.03043 -0.00425
-0.03208 -0.02477 0.08044 0.01203
-0.04600 0.11890 -0.18613 0.14225

A * A^(-1):
1.00000 0.00000 0.00000 0.00000
0.00000 1.00000 0.00000 -0.00000
0.00000 0.00000 1.00000 -0.00000
0.00000 0.00000 0.00000 1.00000
```

1.2

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.

$$\begin{cases} 13x_1 - 5x_2 = -66 \\ -4x_1 + 9x_2 - 5x_3 = -47 \\ -x_2 - 13x_3 - 6x_4 = -43 \\ 6x_3 + 20x_4 - 5x_5 = -74 \\ 5x_4 + 5x_5 = 14 \end{cases}$$

Метод Решения

Метод прогонки является одним из эффективных методов решения СЛАУ с трех - диагональными матрицами, возникающих при конечно-разностной аппроксимации задач для обыкновенных дифференциальных уравнений (ОДУ) и уравнений в частных производных второго порядка и является частным случаем метода Гаусса. Рассмотрим следующую СЛАУ:

$$\begin{cases} 0 + b_1x_1 + c_1x_2 = d_1 \\ a_2x_1 + b_2x_2 + c_2x_3 = d_2 \\ \dots \\ a_{n-1}x_{n-2} + b_{n-1}x_{n-1} + c_{n-1}x_n = d_{n-1} \\ a_nx_{n-1} + b_nx_n = d_n, c_n = 0 \end{cases}$$

решение которой будем искать в виде $x_i = P_ix_{i+1} + Q_i, i = 1, .., n$

Прогоночные коэффициенты вычисляются по следующим формулам:

$$P_i = \frac{-c_i}{b_i + a_iP_{i-1}}, Q_i = \frac{d_i - a_iQ_{i-1}}{b_i + a_iP_i}, i = 2, .., n-1$$

$$P_1 = \frac{-c_1}{b_1}, Q_1 = \frac{d_1}{b_1}, i = 1$$

$$P_n = 0, Q_n = \frac{d_n - a_nQ_{n-1}}{b_n + a_n + P_{n-1}}, i = n.$$

Обратный ход метода прогонки осуществляется в соответствии с выражением

$$\begin{cases} x_n = P_nx_{n+1} + Q_n \\ x_{n-1} = P_{n-2}x_n + Q_{n-1} \\ x_{n-2} = P_{n-2}x_{n-1} + Q_{n-2} \\ \dots \\ x_1 = P_1x_2 + Q_1. \end{cases}$$

Результат работы программы:

```
x:
-6.89116
-4.71702
6.42229
-4.10557
6.08446
```

1.3

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.

$$\begin{cases} -23x_1 - 7x_2 + 5x_3 + 2x_4 = -26 \\ 7x_1 - 21x_2 + 4x_3 + 9x_4 = -55 \\ 9x_1 + 5x_2 - 31x_3 - 8x_4 = -58 \\ x_2 - 2x_+10x_4 = -24 \end{cases}$$

Метод Решения
Приведем СЛАУ к эквивалентному виду векторно-матричной форме $x = \beta + \alpha x$.

$$x = \begin{bmatrix} x_1 \\ \dots \\ x_n \end{bmatrix}, \beta = \begin{bmatrix} \beta_1 \\ \dots \\ \beta_n \end{bmatrix}, \alpha = \begin{bmatrix} \alpha_{11} & \dots & \alpha_{1n} \\ \vdots & \dots & \vdots \\ \alpha_{n1} & \dots & \alpha_{nn} \end{bmatrix}$$

Такое приведение может быть выполнено различными способами. Одним из наиболее распространенных является следующий. Разрешим исходную систему относительно неизвестных при ненулевых диагональных элементах $a_{ii} \neq 0, i = 1, .., n$ (если какой-либо коэффициент на главной диагонали равен нулю, достаточно соответствующее уравнение поменять местами с любым другим уравнением). Получим следующие выражения для компонентов вектора β и матрицы α эквивалентной системы:

$$\beta_i = \frac{b_i}{a_{ii}}; a_{ij} = -\frac{a_{ij}}{a_{ii}}, i, j = 1, .., n, i \neq j; a_{ij} = 0, i = j$$

При таком способе приведения исходной СЛАУ к эквивалентному виду метод простых итераций носит название метода Якоби.

В качестве нулевого приближения $x^{(0)}$ вектора неизвестных примем вектор правых частей $x^{(0)} = \beta$ Тогда метод простых итераций примет вид:

$$\begin{cases} x^{(0)} = \beta \\ x^{(1)} = \beta + \alpha x^{(0)} \\ \dots \\ x^{(k)} = \beta + \alpha x^{(k-1)}. \end{cases}$$

Имеет место следующее достаточное условие сходимости метода простых итераций. Метод простых итераций сходится к единственному решению СЛАУ при любом начальном приближении $x^{(0)}$, если какая-либо норма матрицы α эквивалентной системы меньше единицы $\alpha < 1$.
При выполнении достаточного условия сходимости оценка погрешности решения на k - ой итерации дается выражением

$$\varepsilon^{(k)} \leq \varepsilon, \varepsilon^{(k)} = \frac{||\alpha||}{1 - ||\alpha||} ||x^{(k-1)} - x^{(k)}||$$

Поскольку $\alpha < 1$ является только достаточным (не необходимым) условием сходимости метода простых итераций, то итерационный процесс может сходиться и в случае, если оно не выполнено. Тогда критерием окончания итераций может служить неравенство $||x^{(k-1)} - x^{(k)}|| \leq \varepsilon$

Метод простых итераций довольно медленно сходится. Для его ускорения существует метод Зейделя, Для его ускорения существует метод Зейделя, заключающийся в том, что при вычислении компонента x_i^{k+1} вектора неизвестных на $(k+1)$-й итерации используются $x_1^{k+1}, .., x_{i-1}^{k+1}$, уже вычисленные на $(k+1)$-й итерации. Метод Зейделя является методом простых итераций с матрицей правых частей $\alpha = (E - B)^{-1}C$ С и вектором правых частей $(E - B)^{-1}\beta$ и, следовательно, сходимость и погрешность метода Зейделя можно исследовать с помощью формул, выведенных для метода простых итераций

Результат работы программы:

```
JACOBI METHOD

Norm of alpha:
0.952381
A priori estimation:
176
Number of Iteration:
```

```
8
x:
1.00009
2.00004
2.99995
-1.99994


SEIDEL METHOD

Norm of alpha:
0.61905
A priori estimation:
13
Number of Iteration:
4
x:
1.00025
2.00013
3.00005
-2.00000
```

1.4

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.

$$\begin{bmatrix} 5 & 5 & 3 \\ 5 & -4 & 1 \\ 3 & 1 & 2 \end{bmatrix}$$

Метод Решения
Метод вращений Якоби применим только для симметрических матриц $A_{n\times n}$ ($A = A^T$) и решает полную проблему собственных значений и собственных векторов таких матриц. Он основан на отыскании с помощью итерационных процедур матрицы U в преобразовании подобия $\Lambda = U^{-1}AU$, а поскольку для симметрических матриц A матрица преобразования подобия U является ортогональной ($U^{-1} = U^T$), то $\Lambda = U^T AU$, где Λ - диагональная матрица с собственными значениями на главной диагонали.

Алгоритм метода вращения следующий:

1. Выбирается максимальный по модулю недиагональный элемент $a_{ij}^(k)$ матрицы $A^{(k)}(|a_{ij}^{(k)} = \max_{l<m} |a_{lm}^{(k)}$.

2. Ставится задача найти такую ортогональную матрицу U^k , чтобы в результате преобразования подобия $A^{k+1} = U^{(k)T} A^{(k)} U^{(K)}$ произошло обнуление элемента a_{ij}^{k+1} матрицы A^{k+1} . В качестве ортогональной матрицы выбирается матрица вращения.
   Угол вращения $\phi^{(k)}$ определяется из условия $a_{ij}^{(k+1)} = 0$:

$$\phi^{(k)} = \frac{1}{2} \arctan \frac{2a_{ij}^{(k)}}{a_{ii}^{(k)} - a_{jj}^{(k)}}$$

3. Строится матрица $A^{(k+1)} U^{(k)T} A^{(k)} U^{(k)}$

Результат работы программы:

```
Eigenvalue_1 = 8.70547
x_1:
0.83032
0.36023
0.42521
A * x -labda * x:
-0.00000
0.00000
0.00000


Eigenvalue_2 = -6.23937
x_2:
-0.41521
0.90881
0.04089
A * x -labda * x:
-0.00002
-0.00001
0.00004
```

```
Eigenvalue_3 = 0.53391
x_3:
-0.37170
-0.21050
0.90417
A * x -labda * x:
-0.00002
0.00004
0.00000


Number of iterations: 5
```

1.5

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.

$$\begin{bmatrix} 5 & -5 & -6 \\ -1 & -8 & -5 \\ 2 & 7 & -3 \end{bmatrix}$$

Метод Решения

В основе QR-алгоритма лежит представление матрицы в виде $A = QR$, где Q - ортогональная матрица $Q^{-1} = Q^T$, а R - верхняя треугольная. Такое разложение существует для любой квадратной матрицы. Одним из возможных подходов к построению QR разложения является использование преобразования Хаусхолдера, позволяющего обратить в нуль группу поддиагональных элементов столбца матрицы.
Преобразование Хаусхолдера осуществляется с использованием матрицы Хаусхолдера, имеющей следующий вид:

$$H = E - \frac{2}{\nu^T \nu} \nu \nu^T$$

Применяя описанную процедуру с целью обнуления поддиагональных элементов каждого из столбцов исходной матрицы, можно за фиксированное число шагов по-

лучить ее QR – разложение.
Процедура QR - разложения многократно используется в QR -алгоритме вычисления собственных значений.
При отсутствии у матрицы кратных собственных значений последовательность $A^{(k)}$ сходится к верхней треугольной матрице (в случае, когда все собственные значения вещественны) или к верхней квазитреугольной матрице (если имеются комплексно-сопряженные пары собственных значений).
Каждой размерностью комплексно-сопряженной 2×2, т.е. матрица паре $A^{(k)}$ имеет соответствует диагональный блочно-диагональную блок структуру. В качестве критерия окончания итераций для таких блоков может быть использовано следующее условие $\lambda^{(k)} - \lambda^{(k-1)} \leq \varepsilon$.

Результат работы программы:

```
lambda_1 = -3.6032 + -6.59494i
lambda_2 = -3.6032 + 6.59494i
lambda_3 = 5.2064

QR Algorithm took 26 iterations
```

matrix.hpp

```
1
2  #include <iostream>
3  #include <cstddef>
4  #include <cassert>
5  #include <algorithm>
6  #include <fstream>
7  #include <iomanip>
8  #include <cmath>
9  #include <complex>
10
11
12 long double const delta = 1e-9;
13
14 #pragma once
15
16 class TMatrix {
17 private:
18     size_t rows_;
19     size_t cols_;
20     long double** data_;
21
22
```

```
23
24 public:
25     explicit TMatrix(size_t = 0, long double = 0.);
26     TMatrix(size_t , size_t , long double = 0.);
27     TMatrix(TMatrix const &);
28     ~TMatrix();
29     long double* operator [] (size_t i);
30     long double const* operator [] (size_t i) const;
31     TMatrix operator + (TMatrix const &) const;
32     TMatrix operator - (TMatrix const &) const;
33     TMatrix operator * (TMatrix const &) const;
34     TMatrix operator *= (TMatrix const &);
35     TMatrix operator * (int) const;
36     TMatrix operator * (long double) const;
37     TMatrix& operator = (TMatrix const &);
38     bool operator == (TMatrix const&) const;
39     friend std::ostream& operator << (std::ostream&, TMatrix const& );
40     friend std::istream& operator >> (std::istream&, TMatrix&);
41     std::tuple<TMatrix, TMatrix, std::vector<std::pair<size_t, size_t>>>
           LUdecomposition() const;
42     std::tuple<TMatrix, TMatrix> QRdecomposition() const;
43     long double Determinant() const;
44     TMatrix Inverse() const;
45     TMatrix Transpose() const;
46     size_t Size() const;
47     size_t Get_Rows() const;
48     size_t Get_Cols() const;
49     long double Norm() const;
50     long double Norm_2() const;
51     long double GetSquaredColumnSum(size_t row, size_t col) const;
52     std::pair<size_t, size_t> Change_Without_Zero(size_t i);
53     std::pair<size_t, size_t> Change_With_Max(size_t i);
54     void Swap_Rows(size_t i, size_t j);
55
56
57 };
```

calculation.hpp

```
1 #include "matrix.hpp"
2 #include <tuple>
3 #include <iomanip>
4
5 int Diagonals_Without_Zeros(TMatrix& A, TMatrix& b);
6
7 //Task1
8 TMatrix LU_Solving_System(TMatrix const& L, TMatrix const& U, TMatrix b, std::vector<
      std::pair<size_t, size_t>> const& p);
9 long double LU_Determinant(TMatrix const& U, std::vector<std::pair<size_t, size_t>>
```

```
    const&);
TMatrix LU_Inverse_Matrix(TMatrix const& L, TMatrix const& U, std::vector<std::pair<
    size_t, size_t>> const& p);


//Task2
TMatrix Tridiagonal_Algorithm(TMatrix const& A, TMatrix const& D);

//Task3

std::tuple<TMatrix, int, int, long double> Iterative_Jacobi_Method(TMatrix const& A,
    TMatrix const& b, long double eps, std::ostream& log);
std::tuple<TMatrix, int, int, long double> Seidel_Method(TMatrix const& A, TMatrix
    const& b, long double const eps, std::ostream& log);

//Task4

std::tuple<std::vector<long double>, std::vector<TMatrix>, size_t> Rotation_Method(
    TMatrix const& M, long double const eps, std::ostream& log);

//Task 5
int Sign(long double d);
std::vector<std::complex<long double>> Solve_Quadratic_Equation(TMatrix const& A,
    size_t col);
std::tuple<std::vector<std::complex<long double>>, int> Eigenvalues_Using_QR(TMatrix
    const& A, long double eps, std::ostream& log);
```

Лабораторная работа №2
«Численные методы решения нелинейных уравнений»

2.1

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.

$$\sqrt{1-x^2} - e^x + 0.1 = 0.$$

Метод решение
Метод простой итерации
При использовании метода простой итерации уравнение заменяется эквивалентным уравнением с выделенным линейным членом

$$x = \varphi(x)$$

Решение ищется путем построения последовательности

$$x^{(k+1)} = \varphi(x^{(k)})$$

Теорема. Пусть функция φ определена и дифференцируема на отрезке $[a, b]$. Тогда если выполняются условия:

1. $\varphi \in [a, b], \forall x \in [a, b]$,
2. $\exists q : |\varphi'(x)| \leq q < 1 \forall x \in (a, b)$,

Условия окончания:

$$\varepsilon^{(k)} \leq \varepsilon, \varepsilon^{(k)} = \frac{q}{1-q}|x^{(k+1)} - x^{(k)}|$$

Метод Ньютона (метод касательных)
При нахождении корня уравнения методом Ньютона, итерационный процесс определяется формулой

$$x^{(k+1)} = x^{(k)} - \frac{f(x^{(k)})}{f'(x^{(k)})}$$

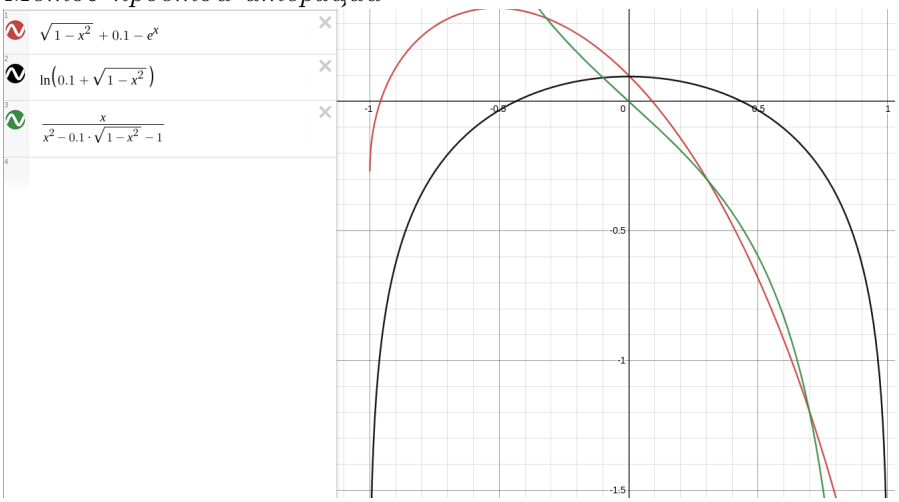
Для начала вычислений требуется задание начального приближения $x^{(0)}$.
Условия сходимости метода определяются следующей теоремой: **Теорема** Пусть на

отрезке $[a, b]$ функция (x) имеет первую и вторую производные постоянного знака и пусть $f(a)f(b) < 0$.Тогда если точка $x^{(0)}$ выбрана на $[a, b]$так, что

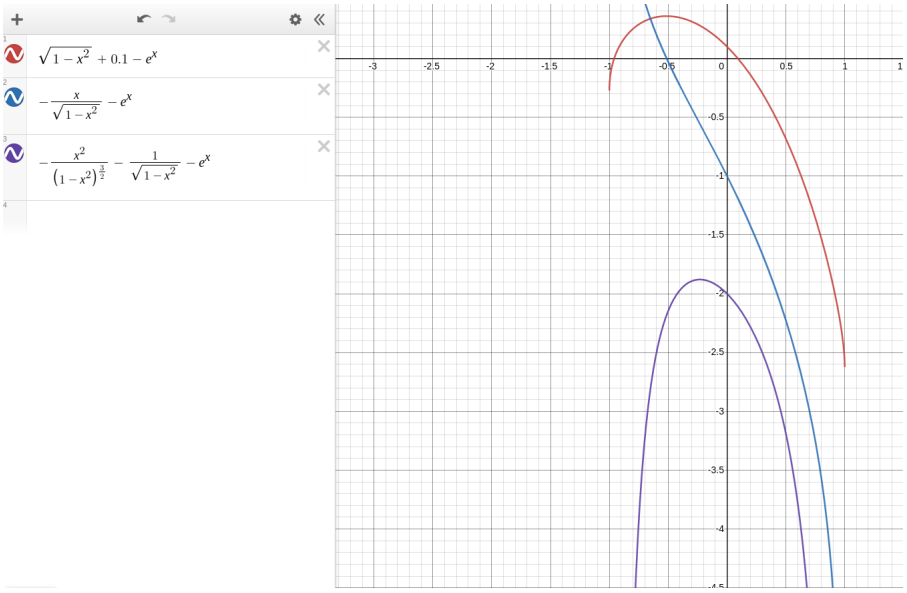
$$f(x^{(0)})f''(x^{(0)}) > 0,$$

то начатая с нее последовательность $x^{(k)}(k = 0, 1, 2, ...)$, определяемая методом Ньютона, монотонно сходится к корню $x^* \in (a, b)$ уравнения.
В качестве условия окончания итераций в практических вычислениях часто используется правило $|x^{(k+1)} - x^{(k)}| < \varepsilon$

Метод простой итерации



Метод Ньютона



Результат работы программы:

```
Newton Method:
x = 0.0914901
Number of iterations: 5

Simple Iterations Method:
x = 0.0914901
Number of iterations: 8
```

2.2

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

$$\begin{cases} (x_1^2 + 4^2)x_2 - 4^3 = 0 \\ (x_1 - 2)^2 + (x_2 - 2)^2 - 4^2 = 0 \end{cases}$$

Метод решения
Метод Ньютона
Если определено начальное приближение $x(0)$, итерационный процесс нахождения решения системы методом Ньютона можно представить в виде

$$\begin{cases} x_1^{(k+1)} = x_1^{(k)} = \Delta x_1^{(k)} \\ \dots \\ x_n^{(k+1)} = x_n^{(k)} + \Delta x_n^{(k)} \end{cases}$$

где вектор приращений $\Delta x^{(k)}$ находится из решения уравнения

$$f(x^{(k)}) + J(x^{(k)})\Delta x^{(k)} = 0$$

$J(x)$ - матрица Якоби первых производных вектор-функции $f(x)$
Итерационная формула:

$$x^{(k+1)} = x^{(k)} - J^{-1}(x^{(k)})f(x^{(k)})$$

Критерий остановки:

$$||x^{(k+1)} - x^{(k)}|| \leq \varepsilon$$

Метод простой итерации
При использовании метода простой итерации система уравнений приводится к эквивалентной системе специального вида в векторной форме:

$$x = \varphi(x)$$

Если выбрано некоторое начальное приближение $x^{(0)}$ последующие приближения в методе простой итерации находятся по формулам

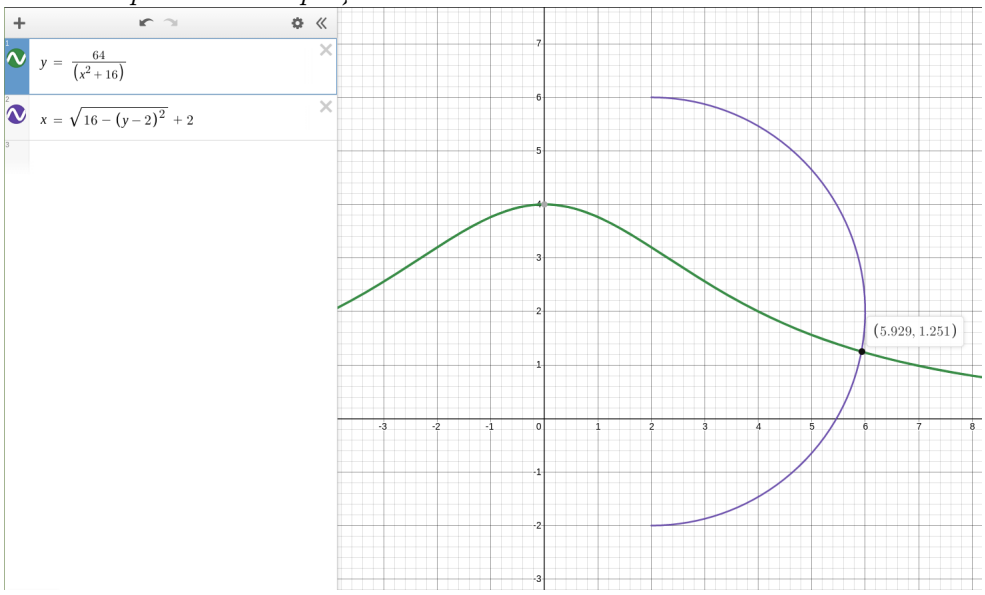
$$x^{(k+1)} = \varphi(x^{(k)}$$

Критерий остановки:

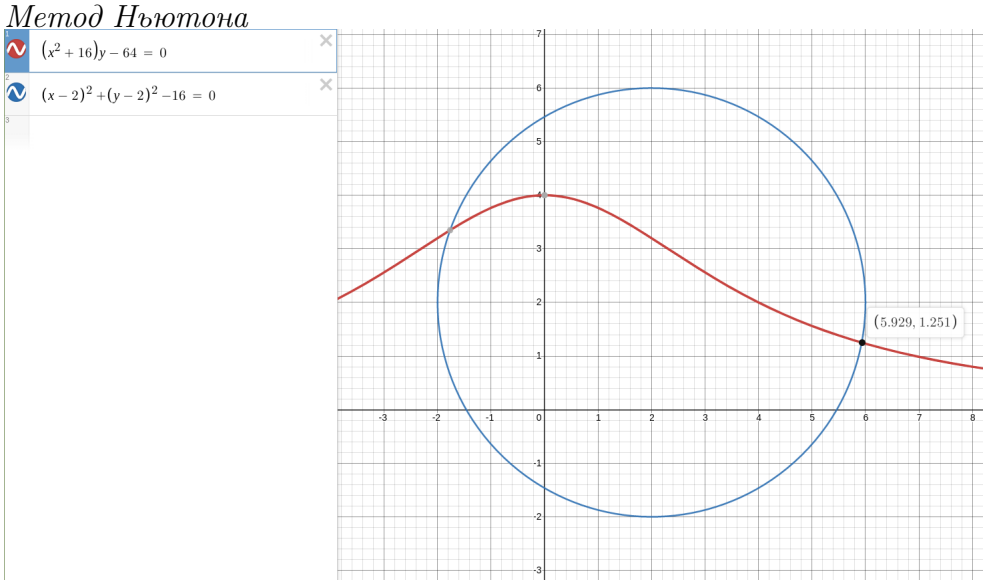
$$\varepsilon^{(k)} \leq \varepsilon, \varepsilon^{(k)} = \frac{q}{1-q}||x^{(k+1)} - x^{(k)}||$$

где q выбирается из условия $\max_{x \in G} ||\varphi'(x)|| \leq q < 1$

Метод простой итерации



Метод Ньютона



Результат работы программы

```
Newton Method:
x:
5.92926
1.25107

Number of iterations: 3

Simple Iterations Method:
x:
5.92799
1.25221

Number of iterations: 4
```

```
1
2  #include <iostream>
3  #include <cstddef>
4  #include <cassert>
5  #include <algorithm>
6  #include <fstream>
7  #include <iomanip>
8  #include <cmath>
```

```
9  #include <complex>
10
11
12 long double const delta = 1e-9;
13
14 #pragma once
15
16 class TMatrix {
17 private:
18     size_t rows_;
19     size_t cols_;
20     long double** data_;
21
22
23
24 public:
25     explicit TMatrix(size_t = 0, long double = 0.);
26     TMatrix(size_t , size_t , long double = 0.);
27     TMatrix(TMatrix const &);
28     ~TMatrix();
29     long double* operator [] (size_t i);
30     long double const* operator [] (size_t i) const;
31     TMatrix operator + (TMatrix const &) const;
32     TMatrix operator - (TMatrix const &) const;
33     TMatrix operator * (TMatrix const &) const;
34     TMatrix operator *= (TMatrix const &);
35     TMatrix operator * (int) const;
36     TMatrix operator * (long double) const;
37     TMatrix& operator = (TMatrix const &);
38     bool operator == (TMatrix const&) const;
39     friend std::ostream& operator << (std::ostream&, TMatrix const& );
40     friend std::istream& operator >> (std::istream&, TMatrix&);
41     std::tuple<TMatrix, TMatrix, std::vector<std::pair<size_t, size_t>>>
           LUdecomposition() const;
42     std::tuple<TMatrix, TMatrix> QRdecomposition() const;
43     long double Determinant() const;
44     TMatrix Inverse() const;
45     TMatrix Transpose() const;
46     size_t Size() const;
47     size_t Get_Rows() const;
48     size_t Get_Cols() const;
49     long double Norm() const;
50     long double Norm_2() const;
51     long double GetSquaredColumnSum(size_t row, size_t col) const;
52     std::pair<size_t, size_t> Change_Without_Zero(size_t i);
53     std::pair<size_t, size_t> Change_With_Max(size_t i);
54     void Swap_Rows(size_t i, size_t j);
55
56
```

```
57
58
59
60 };
```

```
1  #include <cmath>
2  #include <fstream>
3  #include <iostream>
4  #include <tuple>
5  #include <iomanip>
6
7  #include "matrix.hpp"
8
9
10 TMatrix LU_Solving_System(TMatrix const& L, TMatrix const& U, TMatrix b, std::vector<
       std::pair<size_t, size_t>> const& p);
11 long double LU_Determinant(TMatrix const& U, std::vector<std::pair<size_t, size_t>>
       const&);
12 TMatrix LU_Inverse_Matrix(TMatrix const& L, TMatrix const& U, std::vector<std::pair<
       size_t, size_t>> const& p);
13 TMatrix Seidel_Method(TMatrix const& A, TMatrix const& b, long double const eps);
14 int Sign(long double d);
15
16
17
18 //Task1
19
20
21 std::tuple<long double, int> Newton_Method(long double a, long double b, long double
       eps, std::ostream& log);
22 std::tuple<long double, int> Simple_Iterations_Method(long double a, long double b,
       long double eps, std::ostream& log);
23
24
25 //Task2
26
27 std::tuple<TMatrix, int> Dimentional_Newton_Method(long double eps, std::ostream& log)
       ;
28 std::tuple<TMatrix, int> Dimentional_Simple_Iterations_Method(long double eps, std::
       ostream& log);
```

Лабораторная работа №3
«Методы приближения функций»

3.1

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

$$y = \tan x, 1)X_i = 0, \frac{\pi}{8}, \frac{2\pi}{8}, \frac{3\pi}{8} 2)X_i = \frac{\pi}{8}, \frac{\pi}{3}, \frac{3\pi}{8}$$

Метод решения:
Наиболее часто в качестве приближающей функции используют многочлены степени n:

$$P_n(x) = \sum_{i=0}^{n} a_i x^i$$

Произвольный многочлен может быть записан в виде:

$$L_n(x) = \sum_{i=0}^{n} f_i l_i(x).$$

Здесь $l_i(x)$ – многочлены степени n так называемые лагранжевы многочлены влияния, которые удовлетворяют условию $l_i(x_j) = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases}$ и, соответственно, $l_i(x) = \prod_{j=0, j\neq i}^{n} \frac{(x-x_i)}{(x_i-x_j)}$ а интерполяционный многочлен запишется в виде

$$L_n(x) = \sum_{i=0}^{n} f_i \prod_{j=0, j\neq i}^{n} \frac{(x - x_j)}{(x_i - x_j)}$$

Интерполяционный многочлен, записанный в этой форме, называется ин- терполяционным многочленом Лагранжа.
Если ввести функцию $\omega_{n+1}(x) = (x - x_0)...(x - x_n)$, то выражение для интерполяционного многочлена Лагранжа примет вид:

$$L_n(x) = \sum_{i=0}^{n} f_i \frac{\omega_{n+1}(x)}{(x - x_i)\omega'_{n+1}(x_i)}$$

Интерполяционный многочлен, значения которого в узлах интерполяции совпадают со значениями функции $f(x)$ может быть записан в виде:

$$P_n(x) = f(x_0) + (x - x_0)f(x_1, x_0) + ... + (x - x_0)(x - x_1)...(x - x_n)f(x_0, ..., x_n)$$

Запись многочлена в формуле есть так называемый интерполяционный многочлен Ньютона.

Результат работы программы:

```
Lagrange Polynomial:
L(x) =  + 3.419910x(x -0.785398)(x -1.178097)-8.256393x(x -0.392699)(x -1.178097)
+ 6.644232x(x -0.392699)(x -0.785398)
L(X*) = 0.64460678118654746171500 85 f(X*) = 0.66817863791929887895548 75
Absolute error of interpolation:0.02357185673275141724047899

Newton Polynomial:
P(x) =  + 1.054786x + 0.556287x(x -0.392699) + 1.807749x(x -0.392699)(x -0.785398)
P(X*) = 0.64460678118654746171500 85 f(X*) = 0.66817863791929887895548 75
Absolute error of interpolation:0.02357185673275141724047899
```

3.2

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 $x = X^*$.

Метод решения

Использование одной интерполяционной формулы на большом числе узлов нецелесообразно. Интерполяционный многочлен может проявить свои колебательные свойства, его значения между узлами могут сильно отличаться от значений интерполируемой функции. Одна из возможностей преодоления этого недостатка заключается в применении сплайн-интерполяции. Суть сплайн-интерполяции заключается в определении интерполирующей функции по формулам одного типа для различных непересекающихся промежутков и в стыковке значений функции и её производных на их границах.

Наиболее широко применяемым является случай, когда между любыми двумя точками разбиения исходного отрезка строится многочлен n-й степени:

$$S(x) = \sum_{k=0}^{n} a_{ik}x^k, x_{i-1} \leq x \leq x_i, i = 1, .., n$$

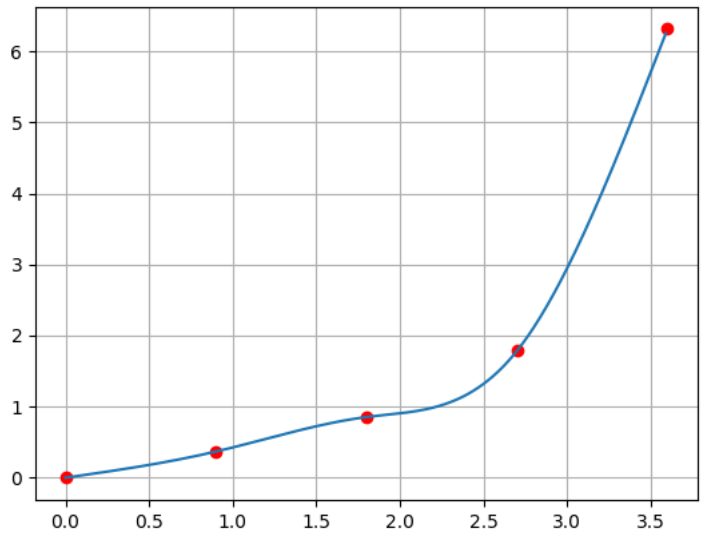
На практике наиболее часто используется интерполяционный многочлен третьей степени, который удобно представить как

$$S(x) = a_i b_i (x - x_{i-1}) + c_i (x - x_{i-1})^2 + d_i (x - x_i)^3$$

Для построения кубического сплайна необходимо построить n многочленов третьей степени, т.е. определить $4n$ неизвестных a_i, b_i, c_i, d_i . Необходимо решить СЛАУ для трехдиагональной матрицы методом прогонки.

Результат работы программы:

```
f(x) = 0.724543
```



3.3

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.

i	0	1	2	3	4	5
x_i	-0.9	0.0	0.9	1.8	2.7	3.6
y_i	-0.36892	0.0	0.36892	0.85408	1.7856	6.3138

Метод решения:
Пусть задана таблично в узлах x_j функция $y_j = f(x_j), j = 0, 1, ..., N$. При этом значения функции y_j определены с некоторой погрешностью, также из физических соображений известен вид функции, которой должны приближенно удовлетворять табличные точки, например: многочлен степени n , у которого неизвестны коэффициенты $a_i, F_n(x) = \sum_{i=0}^{n} a_i x^i$ Неизвестные коэффициенты будем находить из условия минимума квадратичного отклонения многочлена от таблично заданной функции.

$$\Phi = \sum_{j=0}^{N} (F_n(x_j) - j_j)^2$$

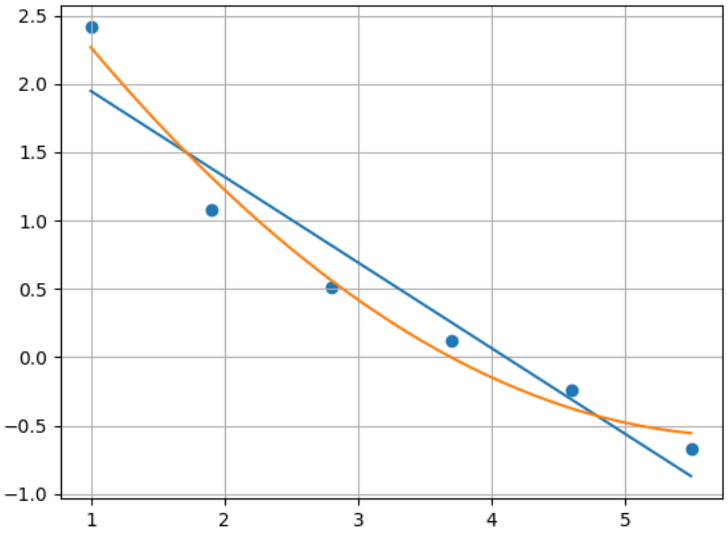
Минимума Ф можно добиться только за счет изменения коэффициентов многочлена $F_n(x)$. Необходимые условия экстремума имеют вид

$$\frac{\partial \Phi}{\partial a_k} = 2 \sum_{j=0}^{N} (\sum_{j=0}^{n} a_i x_j^i - u_j) x_j^k = 0, k = 0, 1, .., n.$$

Система называется нормальной системой метода наименьших квадратов (МНК) представляет собой систему линейных алгебраических уравнений относительно коэффициентов a_i . Решив систему, построим многочлен $F_n(x)$, приближающий функцию $f(x)$ и минимизирующий квадратичное отклонение.

Результат работы программы:

```
F1(x) = 2.575567-0.627578x
PHI = 0.470122
F2(x) = 3.547567-1.398066x + 0.118537x^2
PHI = 0.125953
```



3.4

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

3. $X^* = 2.0$

i	0	1	2	3	4
x_i	1.0	1.5	2.0	2.5	3.0
y_i	0.0	0.40547	0.69315	0.91629	1.0986

Метод решения:
Формулы численного дифференцирования в основном используются при нахождении производных от функции $y = f(x)$, заданной таблично. Исходная функция.
При решении практических задач, как правило, используются аппроксимации первых и вторых производных.
В первом приближении, таблично заданная функция может быть аппроксимирована

При использовании для аппроксимации таблично заданной функции интерполяционного многочлена второй степени имеем:

$$y'(x) \approx \varphi'(x) = y_i + \frac{y_{i+1} - y_i}{x_{i+1} - x_i} + \frac{\frac{y_{i+2} - y_{i+1}}{x_{i+2} - x_{i+1}} - \frac{y_{i+1} - y_i}{x_{i+1} - x_i}}{x_{i+2} - x_i}(2x - x_i - x_{i+1}), x \in [x_i, x_{i+1}]$$

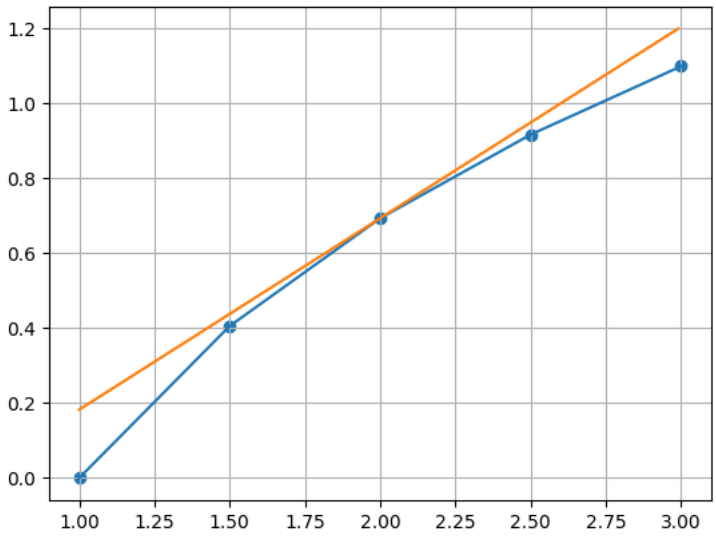
Для вычисления второй производной, необходимо использовать интерполяционный

многочлен, как минимум второй степени. После дифференцирования многочлена получаем:

$$y''(x) \approx \varphi''(x) = 2\frac{\frac{y_{i+2}-y_{i+1}}{x_{i+2}-x_{i+1}} - \frac{y_{i+1}-y_i}{x_{i+1}-x_i}}{x_{i+2}-x_i}, x \in [x_i, x_{i+1}$$

Результат работы программы:

```
0.51082 -0.25816
```



3.5

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:

3. $y = \frac{x}{(3x+4)^3}$, $X_0 = -1$, $X_k = 1$, $h_1 = 0.5$, $h_2 = 0.25$;

Метод решения:
Формулы численного интегрирования используются в тех случаях, когда вычислить аналитически определенный интеграл $F = \int_a^b ydx$ не удается. Отрезок $[a, b]$ разбивают точками $x_0, .., x_N$ достаточно мелким шагом $h_i = x_i = x_{i-1}$ и на одном или

нескольких отрезках h_i подынтегральную функцию $f(x)$ заменяют такой приближающей $\varphi(x)$, так что она, во-первых, близка $f(x)$, а, во-вторых, интеграл от $\varphi(x)$ легко вычисляется.
Заменим подынтегральную функцию, интерполяционным многочленом Лагранжа нулевой степени, проходящим через середину отрезка – точку $x_i = (x_{i-1} + x_i)/2$, получим формулу прямоугольников.

$$\int_a^b f(x)dx \approx \sum_{i=1}^{N} h_i f(\frac{x_{i-1} + x_i}{2})$$

В случае таблично заданных функций удобно в качестве узлов интерполяции выбрать начало и конец отрезка интегрирования, т.е. заменить функцию $f(x)$ многочленом Лагранжа первой степени.

$$F = \int_a^b f(x)dx \approx \frac{1}{2} \sum_{i=1}^{N} (f_i + f_{i-1})h_i$$

Эта формула носит название формулы трапеций.

Для повышения порядка точности формулы численного интегрирования заменим подынтегральную кривую параболой – интерполяционным многочленом второй степени, выбрав в качестве узлов интерполяции концы и середину отрезка интегрирования: $x_{i-1}, x_{i-1/2} = (x_{i-1} + x_i)/2, x_i$

Для случая $h_i = \frac{x_i - x_{i-1}}{2}$, получим формулу Симпсона (парабол)

$$F = \int_a^b f(x)dx \approx \frac{1}{3} \sum_{i=1}^{N} (f_{i-1} + 4f_{i-1/2}f_i)h_i$$

Оценка Рунге-Ромберга:

$$\varphi = \frac{F_h - F_{h/2}}{k^p - 1}$$

Результат работы программы:

```
Exact Solution: -0.12245

h1 = 0.5
Rectangle Method: -0.0724458
```

```
Trapezoid Method: -0.265249
Simpson Method: -0.185511

h2 = 0.25
Rectangle Method: -0.103192
Trapezoid Method: -0.168088
Simpson Method: -0.132747

Runge Romberg Richardson Method:
for Rectangle Method: -0.0307457
for Trapezoid Method: 0.032387
for Simpson Method: 0.00753763
```

```
1
2  #include <iostream>
3  #include <cstddef>
4  #include <cassert>
5  #include <algorithm>
6  #include <fstream>
7  #include <iomanip>
8  #include <cmath>
9  #include <complex>
10
11
12 long double const delta = 1e-9;
13
14 #pragma once
15
16 class TMatrix {
17 private:
18     size_t rows_;
19     size_t cols_;
20     long double** data_;
21
22
23
24 public:
25     explicit TMatrix(size_t = 0, long double = 0.);
26     TMatrix(size_t , size_t , long double = 0.);
27     TMatrix(TMatrix const &);
28     ~TMatrix();
29     long double* operator [] (size_t i);
30     long double const* operator [] (size_t i) const;
31     TMatrix operator + (TMatrix const &) const;
32     TMatrix operator - (TMatrix const &) const;
33     TMatrix operator * (TMatrix const &) const;
```

```
    TMatrix operator *= (TMatrix const &);
    TMatrix operator * (int) const;
    TMatrix operator * (long double) const;
    TMatrix& operator = (TMatrix const &);
    bool operator == (TMatrix const&) const;
    friend std::ostream& operator << (std::ostream&, TMatrix const& );
    friend std::istream& operator >> (std::istream&, TMatrix&);
    std::tuple<TMatrix, TMatrix, std::vector<std::pair<size_t, size_t>>>
        LUdecomposition() const;
    std::tuple<TMatrix, TMatrix> QRdecomposition() const;
    long double Determinant() const;
    TMatrix Inverse() const;
    TMatrix Transpose() const;
    size_t Size() const;
    size_t Get_Rows() const;
    size_t Get_Cols() const;
    long double Norm() const;
    long double Norm_2() const;
    long double GetSquaredColumnSum(size_t row, size_t col) const;
    std::pair<size_t, size_t> Change_Without_Zero(size_t i);
    std::pair<size_t, size_t> Change_With_Max(size_t i);
    void Swap_Rows(size_t i, size_t j);


};
```

```
#pragma once
#include <string>
#include <iomanip>
#define ld long double
#include "matrix.hpp"
#include "Segment.hpp"


//Task1

std::tuple<std::string, ld, ld> Lagrange_Polynomial(std::vector<ld> const &, ld, std::
    ostream& log);
std::tuple<std::string, ld, ld> Newton_Polynomial(std::vector<ld> const &, ld, std::
    ostream&);

//Task2

std::tuple<std::vector<TSegment>, ld> Cubic_Spline(std::vector<ld> const& x, std::
    vector<ld> const& f, ld Point);

//Task3
std::tuple<std::string, ld, std::string, ld> Least_Square_Method(std::vector<ld> const
    & x, std::vector<ld> const& y);

```

```
21 //Task4
22
23 std::tuple<ld, ld> Numerical_Differentiation(std::vector<ld> const& x, std::vector<ld>
       const& y, ld Point);
24
25 //Task5
26
27 ld Rectangle_Method(ld x_0, ld x_k, ld h);
28 ld Trapezoid_Method(ld x_0, ld x_k, ld h);
29 ld Simpson_Method(ld x_0, ld x_k, ld h);
30 ld Runge_Romberg_Richardson_Method (ld F_half,ld F,ld p);
```

```
1  #pragma once
2  #include <iostream>
3  #include <cstddef>
4  #include <cassert>
5  #include <algorithm>
6  #include <fstream>
7  #include <iomanip>
8  #include <cmath>
9  #include <complex>
10
11 #define ld long double
12
13 struct TSegment {
14     ld a, b, c, d;
15     void Print(std::ostream& os) {
16         os << std::setprecision(5) << std::fixed << a << ' ' << b << ' ' << c << ' ' <<
               d << '\n';
17     };
18 };
```

Лабораторная работа №4
«методы решения обыкновенных дифференциальных уравнений»

4.1

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.

$$y'' - 2y - 4x^2e^x = 0,\ y(0) = 3,\ y'(0) = 0,\ x \in [0,1], h = 0.1\, y = e^{x^2} = e^{x*\sqrt{2}} + e^{-x\sqrt{2}}$$

Метод решения:
Рассматривается задача Коши для одного дифференциального уравнения первого порядка разрешенного относительно производной

$$y' = f(x,y), y(x_0) = y_0$$

Требуется найти решение на отрезке $[a,b]$, где $x_0 = a$
Метод Эйлера (явный).
Метод Эйлера играет важную роль в теории численных методов решения ОДУ, хотя и не часто используется в практических расчетах из-за невысокой точности. Вывод расчетных соотношений для этого метода может быть произведен несколькими способами: с помощью геометрической интерпретации, с использованием разложения в ряд Тейлора, конечно разностным методом (с помощью разностной аппроксимации производной), квадратурным способом (использованием эквивалентного интегрального уравнения).
Формула метода Эйлера:

$$y_{k+1} = y_k + hf(x_k, y_k)$$

Метод Рунге-Кутты:
Семейство явных методов Рунге-Кутты р-го порядка записывается в виде совокупности формул:

$$y_{k+1} = y_k = \Delta y_k \qquad \Delta y_k = \sum_{i=1}^{p} c_i K_i^k \qquad K_i^k = hf(x_k + a_i h, y_k + h\sum_{j=1}^{i-1} b_{ij} K_j^k)$$

чтобы решить ДУ вторго порядка, нужно делать замену и решить следующую систему:

$$\begin{cases} y' = f(x,y,z) \\ z' = g(x,y,z) \end{cases} \quad y(x_0) = y_0 \qquad\qquad z(x_0) = z_0$$

Многошаговые методы. Метод Адамса.
Многошаговые методы решения задачи Коши характеризуются тем, что решение в текущем узле зависит от данных не в одном предыдущем узле, как это имеет место в одношаговых методах, а от нескольких предыдущих узлах. Многие многошаговые методы различного порядка точности можно конструировать с помощью квадратурного способа (т.е. с использованием эквивалентного интегрального уравнения).
При использовании интерполяционного многочлена 3-ей степени построенного по значениям подынтегральной функции в последних четырех узлах получим метод Адамса четвертого порядка точности:

$$y_{k+1} = y_k + \frac{h}{24}(55f_k - 59f_{k-1} + 37f_{k-2} - 9f_{k-3})$$

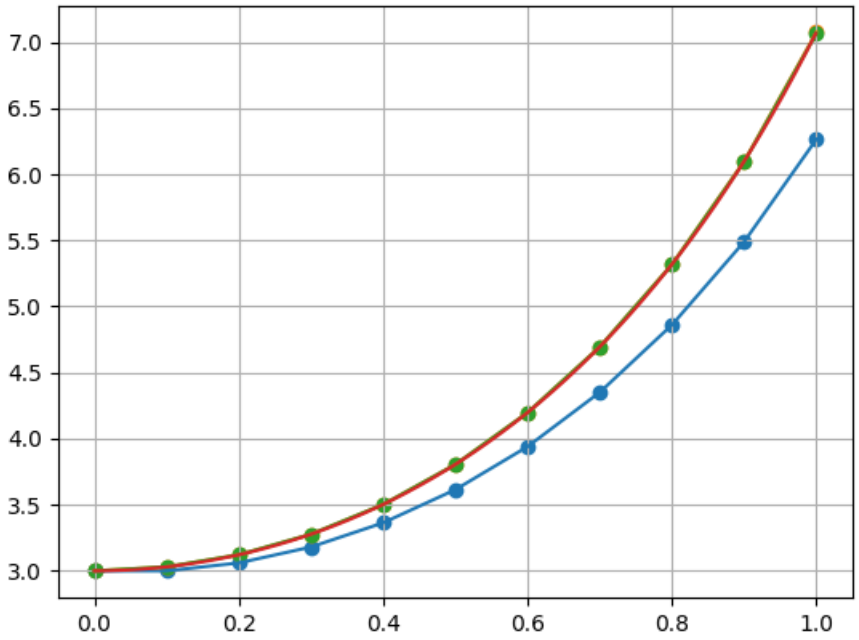
Результат работы программы:

```
Euler's Method
0       1       2       3       4       5       6       7       8       9       10
x_k:  0.00000 0.10000 0.20000 0.30000 0.40000 0.50000 0.60000 0.70000 0.80000
0.90000 1.00000
y_k:  3.00000 3.00000 3.06000 3.18040 3.36367 3.61449 3.94009 4.35082 4.86099
5.49017 6.26513
eps_k:0.00000 0.03008 0.06135 0.09649 0.13846 0.19072 0.25749 0.34419 0.45798
0.60859 0.80952
Runge:0.00000 0.01500 0.03023 0.04700 0.06674 0.09106 0.12190 0.16174 0.21381
0.28246 0.37374


Runge-Kutta's Method
0       1       2       3       4       5       6       7       8       9       10
x_k:  0.00000 0.10000 0.20000 0.30000 0.40000 0.50000 0.60000 0.70000 0.80000
0.90000 1.00000
y_k:  3.00000 3.03008 3.12134 3.27689 3.50213 3.80520 4.19757 4.69499 5.31895
6.09873 7.07459
eps_k:0.00000 0.00000 0.00000 0.00000 0.00000 0.00001 0.00001 0.00002 0.00003
0.00004 0.00006
Runge:0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000 0.00000
0.00000 0.00000


Adams's Method
0       1       2       3       4       5       6       7       8       9       10
```

```
x_k:  0.00000 0.10000 0.20000 0.30000 0.40000 0.50000 0.60000 0.70000 0.80000
0.90000 1.00000
y_k:  3.00000 3.03008 3.12134 3.27689 3.50203 3.80479 4.19672 4.69341 5.31622
6.09423 7.06738
eps_k:0.00000 0.00000 0.00000 0.00000 0.00010 0.00041 0.00086 0.00160 0.00275
0.00453 0.00727
Runge:0.00000 0.00000 -0.00000 -0.00000 0.00001 0.00002 0.00005 0.00010 0.00017
0.00027 0.00044
```



4.2

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.

$$x^2(x+1)y'' - 2y = 0 \qquad y'(1) = -1 \qquad 2y(2) - 4y'(2) = 4 \qquad y(x) = \frac{1}{x} + 1$$

Метод решения:
Метод стрельбы
Суть метода заключена в многократном решении задачи Коши для приближенного нахождения решения краевой задачи.

Пусть надо решить краевую задачу на отрезке $[a, b]$. Вместо исходной задачи формулируется задача Коши с начальными условиями

$$y(a) = y_0 \qquad\qquad y'(b) = \eta$$

Необходимо найти корень уравнения $\Phi(\eta) = 0$
Следующее значение искомого корня определяется по соотношению

$$\eta_{j+2} = \eta_{j+1} - \frac{\eta_{j+1} - \eta_j}{\Phi(\eta_{j+1}) - \Phi(\eta_j)}\Phi(\eta_{j+1})$$

Конечно-разностный метод решения краевой задачи
Введем разностную аппроксимацию производных следующим образом:

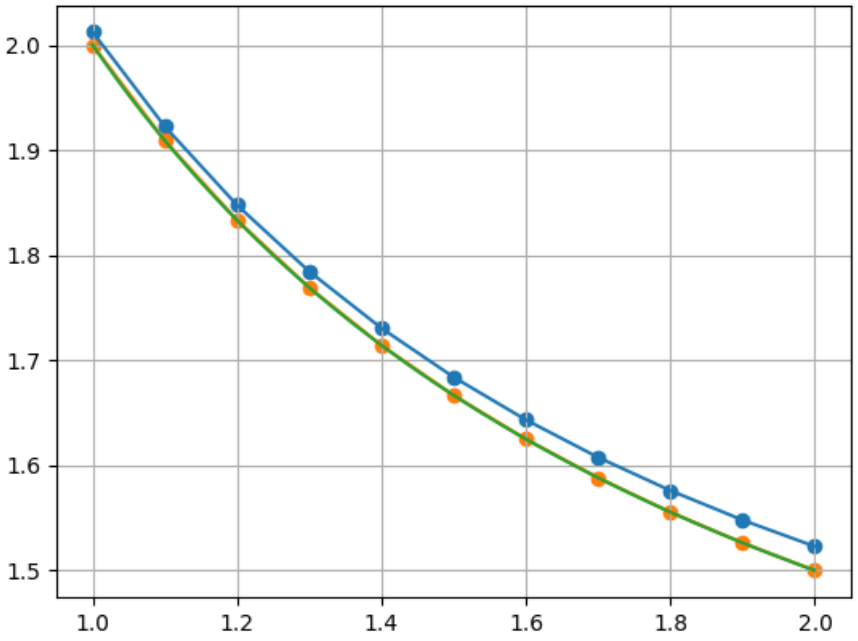
$$y'_k = \frac{y_{k+1} - y_{k-1}}{2h}; \qquad\qquad y''_k = \frac{y_{k+1} - 2y_k + y_{k-1}}{h^2}$$

Подставив эти аппроксиимации в исходное уравнение получим трехдиагональную матрицу, которую решим методом прогонки.
Результат работы программы:

```
Finite Defference Method
0       1       2       3       4       5       6       7       8       9       10
x_k:  1.00000 1.10000 1.20000 1.30000 1.40000 1.50000 1.60000 1.70000 1.80000
1.90000 2.00000
y_k:  2.01240 1.92246 1.84765 1.78451 1.73055 1.68395 1.64333 1.60765 1.57610
1.54802 1.52289
eps_k:0.01240 0.01337 0.01432 0.01528 0.01626 0.01728 0.01833 0.01942 0.02054
0.02170 0.02289
Runge:-0.00930 -0.01002 -0.01074 -0.01146 -0.01220 -0.01296 -0.01375 -0.01456
-0.01540 -0.01627 -0.01717


Shooting Method
0       1       2       3       4       5       6       7       8       9       10
x_k:  1.00000 1.10000 1.20000 1.30000 1.40000 1.50000 1.60000 1.70000 1.80000
1.90000 2.00000
```

```
y_k:  2.00002 1.90911 1.83335 1.76925 1.71431 1.66669 1.62502 1.58826 1.55558
1.52634 1.50003
eps_k:0.00002 0.00002 0.00002 0.00002 0.00002 0.00002 0.00002 0.00002 0.00003
0.00003 0.00003
Runge:-0.00001 -0.00002 -0.00002 -0.00002 -0.00002 -0.00002 -0.00002 -0.00002
-0.00002 -0.00003 -0.00003
```




```cpp
#pragma once
#include <fstream>
#include <iostream>
#include <cmath>
#include <vector>
#include <tuple>
#include <iomanip>
#include <limits>
#define ld long double


ld eps(ld y1, ld y2);
ld Runge(ld y_half, ld y, int p);

//Task1
ld y_exact1(ld x);

//Euler
```

```
19 std::tuple<std::vector<ld>, std::vector<ld>> Euler_Method(ld h, ld x_0, ld x_1, ld y_0
       , ld z_0, std::ostream* log = nullptr);
20 std::vector<ld> Runge_Estimate_for_Euler(std::vector<ld> const & y, ld h, ld x_0, ld
       x_1, ld y_0, ld z_0);
21
22 //Runge-Kutta
23
24 std::tuple<std::vector<ld>, std::vector<ld>, std::vector<ld>> Runge_Kutta_Method(ld h,
        ld x_0, ld x_1, ld y_0, ld z_0, std::ostream* log = nullptr);
25 std::vector<ld> Runge_Estimate_for_Runge_Kutta(std::vector<ld> const & y, ld h, ld x_0
       , ld x_1, ld y_0, ld z_0);
26
27 //Adams
28 std::tuple<std::vector<ld>, std::vector<ld>> Adams_Method(ld h, ld x_0, ld x_1, ld y_0
       , ld z_0, std::ostream* log = nullptr);
29 std::vector<ld> Runge_Estimate_for_Adams(std::vector<ld> const & y, ld h, ld x_0, ld
       x_1, ld y_0, ld z_0);
30
31 //Task2
32
33 ld y_exact2(ld x);
34 std::tuple<std::vector<ld>, std::vector<ld>> Finite_Difference_Method(ld h, ld x_0, ld
        x_1, ld a_0, ld a_1, ld alpha, ld b_0, ld b_1, ld beta, std::ostream* log =
       nullptr);
35 std::vector<ld> Runge_Estimate_for_Finite_Defference(std::vector<ld> const & y,ld h,
       ld x_0, ld x_1, ld a_0, ld a_1, ld alpha, ld b_0, ld b_1, ld beta);
36
37 ld y_exact3(ld x);
38 std::tuple<std::vector<ld>, std::vector<ld>> Shooting_Method(ld h, ld x_0, ld x_1, ld
       a_0, ld a_1, ld alpha, ld b_0, ld b_1, ld beta, std::ostream* log = nullptr);
39 std::vector<ld> Runge_Estimate_for_Shooting(std::vector<ld> const & y,ld h, ld x_0, ld
        x_1, ld a_0, ld a_1, ld alpha, ld b_0, ld b_1, ld beta);
```

1 Выводы

Выполнив лабораторные работы по курсу Численные методы, я закрепил знания математического анализа, линейной алгебры, дифференциальных уравнений. Реализовал множество алгоритмов численного решения всевозможных задач, и написал немало кода на языке C++.
С кодом можно ознакомиться, перейдя по ссылке:
https://github.com/vebcreatex7/NM