Московский Авиационный Институт
(национальный исследовательский университет)

Факультет прикладной математики и физики
Кафедра вычислительной математики и программирования

Лабораторные работы по курсу «Численные методы»

VI семестр

Студентка: Черыгова Е.Е.
Группа 8О-304Б
Руководитель работы:
Абгарян К. К.

Москва, 2017 г.

Численные методы
Лабораторная работа №1
«Методы решения задач линейной алгебры»
Черыгова Лиза, 8О-304Б

Задание

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

Теоретическая часть

Метод Гаусса

В методе Гаусса матрица СЛАУ с помощью равносильных преобразований преобразуется в верхнюю треугольную матрицу, получающуюся в результате прямого хода. В обратном ходе определяются неизвестные.
Пусть дана СЛАУ

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2 \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = b_n \end{cases}$$

Прямой ход метода Гаусса, запишем расширенную матрицу системы на первом шаге

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & \ldots & a_{1n} & b_1 \\ a_{21} & a_{22} & a_{23} & \ldots & a_{2n} & b_2 \\ a_{31} & a_{32} & a_{33} & \ldots & a_{3n} & b_3 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ a_{n1} & a_{n2} & a_{n3} & \ldots & a_{nn} & b_n \end{pmatrix}$$

На первом шаге алгоритма Гаусса выберем диагональный элемент (если он равен 0, то первую строку переставляем с какой-либо нижележащей строкой) и объявляем $a_{11} \neq = 0$ его ведущим, а соответствующую строку и столбец, на пересечении которых он стоит - ведущими. Обнулим элементы ведущего столбца. Для этого сформируем числа $(-a_{21}/a11), (-a_{31}/a11), \ldots (-a_{n1}/a11)$. Умножая ведущую строку на число $(-a_{21}/a11)$ складывая со второй и ставя результат на место второй строки, получим вместо элемента нуль, а вместо элементов $a_{2j}, j = \overline{2, n}, b_2$ - соответственно элементы

$a^1_{2j} = a_{2j} + a_{1j}(-a_{21}/a11)$ и т.д.

После (n-1)-го шага алгоритма Гаусса получаем следующую расширенную матрицу, содержащую верхнюю треугольную матрицу СЛАУ:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} & b_1 \\ 0 & a^1_{22} & a^1_{13} & \dots & a^1_{1n} & b^1_2 \\ 0 & 0 & a^2_{33} & \dots & a^2_{1n} & b^2_3 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & a^{n-1}_{1n} & b^{n-1}_n \end{pmatrix}$$

Прямой ход алгоритма Гаусса завершен. В обратном ходе алгоритма Гаусса из последнего уравнения сразу определяется x_n, из предпоследнего x_{n-1} - и т.д. Из первого уравнения определяется.

$$\begin{cases} a^{n-1}_{nn} x_n = b^{n-1}_n \\ a^{n-2}_{n-1} x_{n-1} + a^{n-2}_{n-1} x_n = b^{n-2}_n \\ \dots\dots\dots\dots\dots\dots\dots\dots\dots\dots \\ a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1 \end{cases}$$

Откуда получаем решение $x_1, \dots, x_n$.

Определитель

В результате прямого хода метода Гаусса можно вычислить определитель матрицы A исходной СЛАУ:
$detA = (-1)^p a_{11} a^1_{22} a^2_{33} \cdot \dots \cdot a^{n-1}_{nn}$

При этом с помощью множителя $(-1)^p$, где p - число перестановок строк в процессе прямого хода, учитываются соответствующие перемены знаков вследствие перестановок строк. Метод Гаусса можно применить для обращения невырожденной $detA \neq 0$ матрицы.

Обратная матрица

Благодоря методу Гаусса возможно найти обратную матрицу, для этого нужно провести прямой и обратной ход метода Гаусса для расширинной матрицы специального

вида

$$\left(\begin{array}{ccccc|ccccc} a_{11} & a_{12} & a_{13} & \dots & a_{1n} & 1 & 0 & 0 & \dots & 0 \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} & 0 & 1 & 0 & \dots & 0 \\ a_{31} & a_{32} & a_{33} & \dots & a_{3n} & 0 & 0 & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nn} & 0 & 0 & 0 & \dots & 1 \end{array}\right)$$

Матрицы которая получится в правом блоке будет давна обратной матрице.

LU - разложение

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 и U - верхняя треугольная матрица, разложение может быть построено с использованным выше метода Гаусса. Рассмотрим k - ый шаг метода Гаусса, на котором осуществляется обнуление поддиагональных элементов k - го столбца матрицы A^{k-1}. Как было описано выше, с этой целью используется следующая операция:

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

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

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

При этом выражение для обратной операции запишется $A^{k-1} = M_k^{-1} A^k$. В результате прямого хода метода Гаусса получим $A^{n-1} = U$,
$A = A^0 = M_1^{-1} A^1 = \ldots = M_1^{-1} M_2^{-1} \ldots M_{n-1}^{-1} A^{n-1}$, где $U = A^{n-1}$ - верхняя треугольная матрицы, $L = M_1^{-1} M_2^{-1} \ldots M_{n-1}^{-1}$ - нижняя треугольная матрица

Данные

$$\begin{cases} 2x_1 + 7x_2 - 8x_3 + 6x_4 = -39 \\ 4x_1 + 4x_2 + 0x_3 - 7x_4 = 41 \\ -1x_1 - 3x_2 + 6x_3 + 3x_4 = 4 \\ 9x_1 - 7x_2 - 2x_3 - 8x_4 = 113 \end{cases}$$

Вывод программы

```
Исходная матрица коэффициентов:
  2.00000     7.00000    -8.00000     6.00000
  4.00000     4.00000     0.00000    -7.00000
 -1.00000    -3.00000     6.00000     3.00000
  9.00000    -7.00000    -2.00000    -8.00000

Матрица U:
  2.00000     7.00000    -8.00000     6.00000
  0.00000   -10.00000    16.00000   -19.00000
  0.00000     0.00000     2.80000     5.05000
  0.00000     0.00000     0.00000    87.92857

Матрица L:
  1.00000     0.00000     0.00000     0.00000
  2.00000     1.00000     0.00000     0.00000
 -0.50000    -0.05000     1.00000     0.00000
  4.50000     3.85000    -9.85714     1.00000

Произведение L*U:
  2.00000     7.00000    -8.00000     6.00000
  4.00000     4.00000     0.00000    -7.00000
 -1.00000    -3.00000     6.00000     3.00000
  9.00000    -7.00000    -2.00000    -8.00000

det A = -4924.00000
```

```
Решение СЛАУ:
  8.00000
 -3.00000
  2.00000
 -3.00000

Обратная матрица A^-1:
  0.10236     0.07189     0.16125     0.07433
  0.03981     0.11129     0.03493    -0.05443
 -0.00366     0.08672     0.15496    -0.02051
  0.08123    -0.03818     0.11210     0.01137
```

Листинг программы

```
#include "libs.h"

void show(vector <vector <double>> A, int n)
{
for (int i = 0; i < n; ++i)
{
for (int j = 0; j < n; ++j)
{
cout << setw(9) << fixed << setprecision(5) << A[i][j] << "   ";
}
cout << endl;
}
cout << endl;
}

void show(vector <double> x, int n)
{
for (int i = 0; i < n; ++i)
cout << setw(9) << fixed << setprecision(5) << x[i] << "   " << endl;
```

```
cout << endl;
}

double det(vector <vector <double>> U, int n)
{
double det = 1;
for (int i = 0; i < n; ++i)
det *= U[i][i];
return det;
}

vector<vector<double>> transpose(vector <vector<double>> M, int n)
{
for (int i = 0; i < n; ++i)
for (int j = i; j < n; ++j)
swap(M[i][j], M[j][i]);

return M;
}

vector<vector<double>> proisv(vector <vector <double>> A, vector <vector <double
{
vector<vector<double>> R;
R.resize(n);

for (int i = 0; i < n; ++i)
{
R[i].resize(n, 0);
for (int j = 0; j < n; ++j)
for (int k = 0; k < n; ++k)
R[i][j] += A[i][k] * B[k][j];
}
return R;
}
```

```
void LU(vector <vector <double>> A, vector <vector <double>> &L,
vector <vector <double>> &U, int n)
{
for (int j = 0; j < n; ++j)
U[0][j] = A[0][j];

for (int i = 0; i < n; ++i)
L[i][0] = A[i][0] / U[0][0];

for (int i = 1; i < n; ++i)
{
for (int j = i; j < n; ++j)
{
double sum = 0;
for (int k = 0; k < i; ++k)
sum += L[i][k] * U[k][j];
U[i][j] = A[i][j] - sum;

sum = 0;
for (int k = 0; k < i; ++k)
sum += L[j][k] * U[k][i];
L[j][i] = (A[j][i] - sum) / U[i][i];
}
}
}

vector<double> SLAU(vector <vector <double>> A, vector <vector <double>> L,
vector <vector <double>> U, vector <double> b, int n)
{
vector <double> x, z;
x.resize(n, 0);
z.resize(n, 0);

z[0] = b[0];
```

```
for (int i = 1; i < n; ++i)
{
double sum = 0;
for (int j = 0; j < i; ++j)
sum += L[i][j] * z[j];
z[i] = b[i] - sum;
}

x[n - 1] = z[n - 1] / U[n - 1][n - 1];

for (int i = n - 2; i >= 0; --i)
{
double sum = 0;
for (int j = i + 1; j < n; ++j)
sum += U[i][j] * x[j];
x[i] = (z[i] - sum) / U[i][i];
}
return x;
}

vector <vector <double>> InverseMatrix(vector <vector <double>> A, vector <vecto
{
vector <vector<double>> e, x, y;
vector <vector <vector <double>>> newL, newU;
e.resize(n);
x.resize(n);
y.resize(n);
newL.resize(2 * n);
newU.resize(2 * n);

for (int i = 0; i < n; ++i)
{
e[i].resize(n, 0);
x[i].resize(n, 0);
y[i].resize(n, 0);
```

```
e[i][i] = 1;
}

for (int i = 0; i < 2 * n; ++i)
{
newL[i].resize(n);
newU[i].resize(n);
for (int j = 0; j < n; ++j)
{
newL[i][j].resize(n, 0);
newU[i][j].resize(n, 0);
}
}

for (int i = 0; i < n; ++i)
{
LU(L, newL[i], newU[i], n);
LU(U, newL[i + n], newU[i + n], n);
x[i] = SLAU(U, newL[i + n], newU[i + n], SLAU(L, newL[i], newU[i], e[i], n), n);
}
return transpose(x, n);
}

void LU_main()
{
int n;
ifstream test ("test1-LU.txt");
test >> n;

vector <vector <double>> A, L, U;
vector <double> b;

A.resize(n);
L.resize(n);
U.resize(n);
```

```
b.resize(n);
for (int i = 0; i < n; ++i)
{
A[i].resize(n, 0);
L[i].resize(n, 0);
U[i].resize(n, 0);
for (int j = 0; j < n; ++j)
{
test >> A[i][j];
}
test >> b[i];
}

LU(A, L, U, n);
cout << "Исходная матрица коэффициентов:" << endl;
show(A, n);
cout << "Матрица U:" << endl;
show(U, n);
cout << "Матрица L:" << endl;
show(L, n);
cout << "Произведение L*U:" << endl;
show(proisv(L, U, n), n);
cout << "det A = " << det(U, n) << endl << endl;
cout << "Решение СЛАУ: " << endl;
show(SLAU(A, L, U, b, n), n);
cout << "Обратная матрица A^-1:" << endl;
show(InverseMatrix(A, L, U, n), n);

cout << endl << endl;
test.close();
}
```

Задание

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.

Теоретическая часть

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, $a_1 = 0, c_n = 0$:

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

Так же как и метод Гаусса, метод прогонки состоит из двух этапов, прямой ход предназначен для определения прогоночных коэффициентов $P_i, Q_i, i = \overline{1,n}$. Прогоночные коэффициенты вычисляются по формулам

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

$$P_1 = \frac{-c_1}{b_1}, Q_1 = \frac{d_1}{b_1}, P_n = 0, Q_n = \frac{d_n - a_nQ_{n-1}}{b_n + a_nP_{n-1}}$$

Обратный ход метода прогонки

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

Общее число операций в методе прогонки равно 8n+1, т.е. пропорционально числу уравнений. Такие методы решения СЛАУ называют экономичными. Для сравнения число операций в методе Гаусса пропорционально n^3

Данные

$$\begin{cases} 7x_1 - 5x_2 = 38 \\ -6x_1 + 19x_2 - 9x_3 = 14 \\ 6x_2 - 18x_3 + 7x_4 = -45 \\ -7x_3 - 11x_4 - 2x_5 = 30 \\ 5x_4 - 7x_5 = 48 \end{cases}$$

Вывод программы

```
Введите номер задания или 0 для выхода: 2
P[1] = 0.71429, Q[1] = 5.42857
P[2] = 0.61165, Q[2] = 3.16505
P[3] = 0.48848, Q[3] = 4.46545
P[4] = -0.13870, Q[4] = -4.24832
P[5] = 0, Q[5] = -9.00000

X[1] = 9.00000
X[2] = 5.00000
X[3] = 3.00000
X[4] = -3.00000
X[5] = -9.00000
```

Листинг программы

```
#include "libs.h"

vector <double> a, b, c, d, X;

double Q(int i, double P_prev, double Q_prev)
{
if (i == 0)
return d[0] / b[0];

return
```

```
(d[i] - a[i] * Q_prev) / (b[i] + a[i] * P_prev);
}

double P(int i, double P_prev)
{
if (i == 0)
return -c[0] / b[0];

return
-c[i] / (b[i] + a[i] * P_prev);
}

double get_X(int i, double P_prev, double Q_prev, int n)
{
if (i == n - 1)
{
double Qfinal = Q(i, P_prev, Q_prev);
cout << "P[" << i + 1 << "] = 0, Q[" << i + 1 << "] = " << Qfinal << endl;
X[i] = Qfinal;
return Qfinal;
}

double Ptec = P(i, P_prev);
double Qtec = Q(i, P_prev, Q_prev);
cout << "P[" << i + 1 << "] = " << Ptec;
cout << ", Q[" << i + 1 << "] = " << Qtec << endl;

X[i] = Ptec * get_X(i + 1, Ptec, Qtec, n) + Qtec;
return X[i];
}

void Tridiagonal_main()
{
int n;
ifstream test("test2-Tridiagonal.txt");
```

```
test >> n;

a.resize(n, 0);
b.resize(n, 0);
c.resize(n, 0);
d.resize(n, 0);
X.resize(n, 0);
for (int i = 0; i < n; ++i)
{
test >> a[i];
test >> b[i];
test >> c[i];
test >> d[i];
}

X[0] = get_X(0, 0, 0, n);
cout << endl;
for (int i = 0; i < n; ++i)
cout << "X[" << i+1 << "] = " << X[i] << endl;

cout << endl << endl;
test.close();
}
```

Задание

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.

Теоретическая часть

Метод простых итераций

При большом числе уравнений прямые методы решения СЛАУ (за исключением метода прогонки) становятся труднореализуемыми на ЭВМ прежде всего из-за сложности хранения и обработки матриц большой размерности. В то же время характерной особенностью ряда часто встречающихся в прикладных задачах СЛАУ является разреженность матриц. Число ненулевых элементов таких матриц мало по сравнению с их размерностью. Для решения СЛАУ с разреженными матрицами предпочтительнее использовать итерационные методы. Методы последовательных приближений, в которых при вычислении последующего приближения решения используются предыдущие, уже известные приближенные решения, называются итерационными.

Рассмотрим СЛАУ

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2 \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = b_n \end{cases}$$

с не вырожденной матрицей. Приведем СЛАУ к эквивалентному виду

$$\begin{cases} x_1 = \beta_1 + a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n \\ x_2 = \beta_2 + a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ x_n = \beta_n + a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n \end{cases}$$

Такое приведение может быть выполнено различными способами. Одним из наиболее распространенных является следующий. Разрешим систему относительно неизвестных при ненулевых диагональных элементах , $a_i i \neq 0, i = \overline{1,n}$ (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:

$$\beta = \frac{b_i}{a_i i}, \alpha_{ij} = \frac{a_{ij}}{a_i i}, i, j = \overline{1,n}, i \neq j, \alpha_{ij} = 0, i = j, i = \overline{1,n}$$

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

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

$$\begin{cases} x_0 = \beta \\ x_1 = \beta + \alpha x^0 \\ ..... \\ x_k = \beta + \alpha x^{k-1} \end{cases}$$

Метод простых итераций сходится к единственному решению СЛАУ при любом начальном приближении , если какая-либо норма матрицы α эквивалентной системы $||\alpha|| < 1$ Если используется метод Якоби для эквивалентной СЛАУ, то достаточным условием сходимости является диагональное преобладание матрицы A, т.е. для каждой строки матрицы A модули элементов, стоящих на главной диагонали, больше суммы модулей недиагональных элементов. Приведем также необходимое и достаточное условие сходимости метода простых итераций. Для сходимости итерационного процесса, необходимо и достаточно, чтобы спектр матрицы α эквивалентной системы лежал внутри круга с радиусом, равным единице. Процесс итераций останавливается при выполнении условия $\varepsilon^k \leq \varepsilon$, где ε задаваемая точность.

Критерий окончания
$||x^k - x^{k-1}|| \leq \varepsilon$

Метод Зейделя

Метод простых итераций довольно медленно сходится. 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 $x_1^k, \ldots, x_{i-1}^{k+1}$, уже вычисленные на (k+1)-й итерации. Значения остальных компонент берутся из предыдущей итерации. Так же как и в методе простых итераций строится эквивалентная СЛАУ и за начальное приближение принимается вектор правых частей β. Тогда метод Зейделя для известного вектора на

k-ой итерации имеет вид:

$$\begin{cases} x_1^{k+1} = \beta_1 + a_{11}x_1^k + a_{12}x_2^k + \ldots + a_{1n}x_n^k \\ x_2^{k+1} = \beta_2 + a_{21}x_1^{k+1} + a_{22}x_2^{k+1} + \ldots + a_{2n}x_n^{k+1} \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ x_n^{k+1} = \beta_n + a_{n1}x_1^{k+1} + a_{n2}x_2^{k+1} + \ldots + a_{nn}x_n^{k+1} \end{cases}$$

Сходимость и погрешность можно исследовать с помощью формул, выведенных для метода простых итераций.

Критерий окончания
$||x^k - x^{k-1}|| \leq \varepsilon$

Данные

$$\begin{cases} 26x_1 - 9x_2 - 8x_3 + 8x_4 = 20 \\ 9x_1 - 21x_2 - 2x_3 - 7x_4 = -164 \\ -3x_1 + 2x_2 - 18x_3 + 8x_4 = 140 \\ x_1 - 6x_2 - x_3 + 11x_4 = -81 \end{cases}$$

Вывод программы

```
Заданная точность: 0.01000
Исходная матрица коэффициентов:
 10.00000     1.00000     1.00000
  2.00000    10.00000     1.00000
  2.00000     2.00000    10.00000

Решение методом простых итераций.
eps(1) = 0.43211 > 0.01000
eps(2) = 0.11903 > 0.01000
eps(3) = 0.03438 > 0.01000
eps(4) = 0.00973 <= 0.01000
Решение найдено на шаге 4.
X:
  1.00150
  1.00192
```

```
  1.00240


Решение методом Зейделя.
eps(1) = 0.35991
eps(2) = 0.04867
eps(3) = 0.00131 <= 0.01000
Решение найдено на шаге 3.
X:
  1.00018
  0.99994
  0.99998
```

Листинг программы

```
#include "libs.h"

vector<double> Vsum(vector<double> a, vector<double> b)
{
vector<double> sum;
sum.resize(a.size(), 0);
for (unsigned int i = 0; i < a.size(); ++i)
sum[i] = a[i] + b[i];
return sum;
}

vector<double> Vdiff(vector<double> a, vector<double> b)
{
vector<double> diff;
diff.resize(a.size(), 0);
for (unsigned int i = 0; i < a.size(); ++i)
diff[i] = a[i] - b[i];
return diff;
}

vector<double> proisv(vector <vector <double>> A, vector <double> B, int n)
```

```
{
vector<double> R;
R.resize(n,0);

for (int i = 0; i < n; ++i)
for (int j = 0; j < n; ++j)
R[i] += A[i][j] * B[j];
return R;
}

double norm(vector<double> v, int n)
{
double sum = 0;
for (int i = 0; i < n; ++i)
sum += v[i] * v[i];
return sqrt(sum);
}

double norm(vector<vector<double>> M, int n)
{
double sum = 0;
for (int i = 0; i < n; ++i)
for (int j = 0; j < n; ++j)
sum += M[i][j] * M[i][j];
return sqrt(sum);
}

vector<vector<double>> get_alpha(vector<vector<double>> A, int n)
{
vector<vector<double>> alpha;
alpha.resize(n);
for (int i = 0; i < n; ++i)
{
alpha[i].resize(n, 0);
for (int j = 0; j < n; ++j)
```

```
{
if (i != j) alpha[i][j] = -A[i][j] / A[i][i];
else alpha[i][j] = 0;
}
}
return alpha;
}

vector<double> get_beta(vector<vector<double>> A, vector<double>B, int n)
{
vector<double> beta;
beta.resize(n, 0);
for (int i = 0; i < n; ++i)
beta[i] = B[i] / A[i][i];
return beta;
}

bool errorPassed(vector<vector<double>> alpha, vector<double> x_diff, double eps
{
double eps_k = norm(alpha, n) * norm(x_diff, n) / (1 - norm(alpha, n));
cout << fixed << setprecision(5) << eps_k;
if ( eps_k <= eps)
return true;
else
return false;
}

vector<double> SimpleIteration(vector<double> x, vector<vector<double>> alpha, v
{
if (k == 0)
return SimpleIteration(x, alpha, beta, eps, k + 1, n);

vector<double> new_x = Vsum(beta, proisv(alpha, x, n));
cout << "eps(" << k << ") = ";
if (errorPassed(alpha, Vdiff(new_x, x), eps, n))
```

```
{
cout << " <= " << eps << endl << "Решение найдено на шаге " << k << '.' << endl
return new_x;
}
else
{
cout << " > " << eps << endl;
return SimpleIteration(new_x, alpha, beta, eps, k + 1, n);
}
}

double Zeidel_xk(vector<vector<double>> alpha, vector<double> beta,
vector<double> x, vector<double> x_prev, int i, int n)
{
double xk = beta[i];
for (int j = 0; j < i; ++j)
xk += alpha[i][j] * x[j];

for (int j = i; j < n; ++j)
xk += alpha[i][j] * x_prev[j];

return xk;
}

vector<double> Zeidel(vector<vector<double>> alpha, vector<double> beta, double
{
vector<double> x, x_prev = beta;
int count = 1;
x.resize(n, 0);

for (int i = 0; i < n; ++i)
x[i] = Zeidel_xk(alpha, beta, x, x_prev, i, n);

cout << "eps(1) = ";
while (!errorPassed(alpha, Vdiff(x, x_prev), eps, n))
```

```
{
cout << endl;
x_prev = x;
for (int i = 0; i < n; ++i)
x[i] = Zeidel_xk(alpha, beta, x, x_prev, i, n);

cout << "eps(" << ++count << ") = ";
}
cout << " <= " << eps << endl << "Решение найдено на шаге " << count << '.' << en

return x;
}

void SimpleIteration_main()
{
double eps;
int n;
ifstream test("test3-SimpleIteration.txt");
test >> eps >> n;
cout << "Заданная точность: " << eps << endl;

vector <vector <double>> A, alpha;
vector <double> b, beta;
A.resize(n);
b.resize(n);
for (int i = 0; i < n; ++i)
{
A[i].resize(n, 0);
for (int j = 0; j < n; ++j)
test >> A[i][j];

test >> b[i];
}
cout << "Исходная матрица коэффициентов:" << endl;
show(A, n);
```

```
alpha = get_alpha(A, n);
beta = get_beta(A, b, n);

if (norm(alpha, n) < 1)
{
cout << "Решение методом простых итераций." << endl;
show(SimpleIteration(beta, alpha, beta, eps, 0, n), n);
cout << endl;

cout << "Решение методом Зейделя." << endl;
show(Zeidel(alpha, beta, eps, n), n);
cout << endl;
}
else
cout << "Норма матрицы эквивалентной системы >= 1, решений нет." << endl << endl

test.close();
}
```

Задание

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.

Теоретическая часть

Рассмотрим матрицу в $A(n \times n)$ - мерном вещественном пространстве R^n векторов $x = (x_1 x_2 \dots x_n)^T$

1. Собственным вектором x матрицы A называется ненулевой вектор удовлетворяющий равенству $Ax = \lambda x$ где λ - собственное значение матрицы A, соответствующее рассматриваемому собственному вектору.

2. Собственные значения матрицы A с действительными элементами могут быть вещественными различными, вещественными кратными, комплексными попарно сопряженными, комплексными кратными.

3. Классический способ нахождения собственных значений и собственных векторов известен и заключается в следующем (для однородной СЛАУ): $(A-\lambda E)x = \vartheta, \vartheta = (0 \dots 0)^T$ ненулевые решения имеют место при $det(A - \lambda E) = 0$
называют характеристическим уравнением, а выражение в левой части -характеристическим многочленом;

Метод вращений Якоби

Метод вращений Якоби применим только для симметрических матриц $A(n \times n)(A^T = A)$ 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, то $\Lambda = U^T AU$ где - Λ диагональная матрица с собственными значениями на главной диагонали

$$\Lambda = \begin{pmatrix} \lambda_1 & \dots & 0 \\ \dots & \ddots & \dots \\ 0 & \dots & \lambda_n \end{pmatrix}$$

Пусть дана симметрическая матрица A. Требуется для нее вычислить с точностью ε все собственные значения и соответствующие им собственные векторы. Алгоритм метода

вращения следующий:
Пусть известна матрица A^k на k–й итерации, при этом для k=0 $A^0 = A$.
1. Выбирается максимальный по модулю недиагональный элемент a_{ij} матрицы A^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 $a^{k+1})ij$ матрицы A^{k+1}.
3. В матрице вращения на пересечении i-й строки и j-го столбца находится элемент $u_{ij} = -sin\varphi^k$ где - φ^k угол вращения, подлежащий определению. Симметрично относительно главной диагонали расположен элемент $u_{ji} = sin\varphi^k$; Диагональные элементы u^k_{ii} и u^k_{jj} равны соответственно $u_{ii} = cos\varphi^k$, ; другие диагональные элементы $u^k_{mm} = 1, m \neq i, m \neq j, m = \overline{1,n}$, остальные элементы в матрице вращения равны нулю.

Угол вращения φ^k определяется из условий $a^{k+1}_{ij} = 0$

$$\varphi^k = \frac{1}{2} arctg \frac{2a^k_{ij}}{a^k_{ii} - a^k_{jj}}$$

причем если $a^{=}_{ii} a^k_{jj}$, то $\varphi^k = \frac{\pi}{4}$.

4. Строится матрица $A^{k+1} = {U^k}^T A^k U^k$

Критерий окончания

$$(\sum_{i,m,i<m} (a^{k+1}_{im})^2)^{1/2} < \varepsilon$$

и в качестве искомых собственных значений принимаются $\lambda_1 \approx a11^{k+1}, \ldots, \lambda_n \approx ann^{k+1}$

Координатными столбцами собственных векторов матрицы A в единичном базисе будут столбцы матрицы $U = U^0 \cdot \ldots \cdot U^k$, причем эти собственные векторы будут ортогональными между собой.

Данные

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

Вывод программы

```
Заданная точность: 0.01000
Max = 9.00000
l = 1 ; m = 2
```

```
Phi = 0.78540
U(0)  =
  1.00000     0.00000     0.00000
  0.00000     0.70711    -0.70711
  0.00000     0.70711     0.70711

eps(0) = 8.60233 > 0.01000
A(1)  =
 -9.00000     8.48528    -1.41421
  8.48528    17.00000     0.00000
 -1.41421     0.00000    -1.00000

Max = 8.48528
l = 0 ; m = 1
Phi = -0.28914
U(1)  =
  0.95849     0.28513     0.00000
 -0.28513     0.95849     0.00000
  0.00000     0.00000     1.00000

eps(1) = 1.41421 > 0.01000
A(2)  =
-11.52417    -0.00000    -1.35551
  0.00000    19.52417    -0.40323
 -1.35551    -0.40323    -1.00000

Max = 1.35551
l = 0 ; m = 2
Phi = 0.12606
U(2)  =
  0.99207     0.00000    -0.12573
  0.00000     1.00000     0.00000
  0.12573     0.00000     0.99207

eps(2) = 0.40323 > 0.01000
```

```
A(3)  =
-11.69596    -0.05070    -0.00000
 -0.05070    19.52417    -0.40003
 -0.00000    -0.40003    -0.82822

Max = 0.40003
l = 1 ; m = 2
Phi = -0.01965
U(3)  =
  1.00000     0.00000     0.00000
  0.00000     0.99981     0.01964
  0.00000    -0.01964     0.99981

eps(3) = 0.05070 > 0.01000
A(4)  =
-11.69596    -0.05069    -0.00100
 -0.05069    19.53203    -0.00000
 -0.00100    -0.00000    -0.83607

Max = 0.05069
l = 0 ; m = 1
Phi = 0.00162
U(4)  =
  1.00000    -0.00162     0.00000
  0.00162     1.00000     0.00000
  0.00000     0.00000     1.00000

eps(4) = 0.00100 <= 0.01000
Решение найдено на шаге 4.
A:
-11.69604     0.00000    -0.00100
  0.00000    19.53212     0.00000
 -0.00100     0.00000    -0.83607

Собственные значения:
```

```
lambda(1) = -11.69604; lambda(1) = 19.53212; lambda(1) = -0.83607;

Матрица собственных векторов:
  0.95135     0.28590    -0.11488
 -0.28780     0.69137    -0.66270
 -0.11004     0.66352     0.74002

Собственные вектора:
h(1) =
  0.95135
 -0.28780
 -0.11004

h(2) =
  0.28590
  0.69137
  0.66352

h(3) =
 -0.11488
 -0.66270
  0.74002

Проверка:
h(1)*h(2) = -0.00000; h(1)*h(3) = -0.00000; h(2)*h(3) = -0.00000;
```

Листинг программы

```
#include "libs.h"
#define M_PI 3.14159265358979323846



double errorPassed(vector<vector<double>> A, double eps, int n)
{
double sum = 0;
for (int i = 0; i < n; ++i)
```

```
{
for (int j = i + 1; j < n; ++j)
sum += pow(A[i][j], 2);
}

double eps_k = sqrt(sum);
cout << fixed << setprecision(5) << eps_k;
if (eps_k <= eps)
return true;
else
return false;
}

double proisv(vector<double> a, vector <double> b, int n)
{
double c = 0;
for (int i = 0; i < n; ++i)
c += a[i] * b[i];
return c;
}


double get_phi(vector<vector<double>> Aprev, int l, int m)
{
double phi = 0;
if (Aprev[l][l] != Aprev[m][m])
phi = atan(2 * Aprev[l][m] / (Aprev[l][l] - Aprev[m][m])) / 2;
else
phi = M_PI / 4;

return phi;
}

vector<vector<double>> Matrix(vector<vector<double>> Aprev, int n)
{
```

```
vector <vector <double>> U;
U.resize(n);
for (int i = 0; i < n; ++i)
{
U[i].resize(n, 0);
}

for (int i = 0; i < n; ++i)
{
U[i][i] = 1;

}

int l = 0;
int m = 1;
double max = abs(Aprev[0][1]);
for (int i = 0; i < n; ++i)
{
for (int j = i; j < n; ++j)
{
if (i != j)
{
if (abs(Aprev[i][j]) > max)
{
max = abs(Aprev[i][j]);
l = i;
m = j;
}
}
}
}

cout << "Max = " << max << endl;
cout << "l = " << l << " ; m = " << m << endl;
```

```
double phi = get_phi(Aprev, l, m);
cout << "Phi = " << phi << endl;

U[l][m] = -sin(phi);
U[m][l] = sin(phi);
U[l][l] = cos(phi);
U[m][m] = cos(phi);

return U;

}

vector<vector<double>> Yakobi(vector <vector <double>> A, vector <vector <double>
{

Uprev = Matrix(Aprev, n);
cout << "U(" << k << ")  = " << endl;
show(Uprev, n);
SV = proisv(SV, Uprev, n);

A = proisv(proisv(transpose(Uprev, n), Aprev, n), Uprev, n);
cout << "eps(" << k << ") = ";
if (errorPassed(A, eps, n))
{
cout << " <= " << eps << endl << "Решение найдено на шаге " << k << '.' << endl;
cout << "A:" << endl;
show(A, n);
cout << "Собственные значения:" << endl;
cout << "lambda(1) = " << A[0][0] << "; lambda(1) = " << A[1][1] << "; lambda(1)
//SV = proisv(SV, Uprev, n);
cout << endl;
cout << "Матрица собственных векторов:" << endl;
show(SV, n);
vector <double> h1, h2, h3;
h1.resize(n, 0);
```

```
h2.resize(n, 0);
h3.resize(n, 0);

for (int i = 0; i < n; ++i)
h1[i] = SV[i][0];
for (int i = 0; i < n; ++i)
h2[i] = SV[i][1];
for (int i = 0; i < n; ++i)
h3[i] = SV[i][2];
cout << "Собственные вектора:" << endl;
cout << "h(1) = " << endl;
show(h1, n);
cout << "h(2) = " << endl;
show(h2, n);
cout << "h(3) = " << endl;
show(h3, n);

cout << "Проверка:" << endl;
cout << "h(1)*h(2) = " << proisv(h1, h2, n) << "; h(1)*h(3) = " << proisv(h1, h3

return A;
}
else
{
cout << " > " << eps << endl;
Aprev = A;
cout << "A(" << k+1 << ")  = " << endl;
show(Aprev, n);
//SV = proisv(SV, Uprev, n);
return Yakobi(A, Aprev, Uprev, SV, n, k + 1, eps);
}
}

void Yakobi_main()
{
```

```
double eps;
int n, k = 0;
ifstream test("test4-Yakobi.txt");
test >> eps >> n;
cout << "Заданная точность: " << eps << endl;


vector <vector <double>> A, Uprev, SV;

A.resize(n);
Uprev.resize(n);
SV.resize(n);
for (int i = 0; i < n; ++i)
{
A[i].resize(n, 0);
Uprev[i].resize(n, 0);
SV[i].resize(n, 0);
for (int j = 0; j < n; ++j)
test >> A[i][j];
}
for (int i = 0; i < n; ++i)
{
SV[i][i] = 1;

}

Yakobi(A, A, Uprev, SV, n, k, eps);
test.close();
}
```

Численные методы

Лабораторная работа №2

«Методы решения нелинейных уравнений и систем нелинейных уравнений»

Черыгова Лиза, 8О-304Б

Задание

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.

Теоретическая часть

Численное решение нелинейных (алгебраических или трансцендентных) уравнений вида

$$f(x) = 0$$

заключается в нахождении значений x, удовлетворяющих (с заданной точностью) данному уравнению и состоит из следующих основных этапов:
1.Отделение (изоляция, локализация) корней уравнения.
2.Уточнение с помощью некоторого вычислительного алгоритма конкретного выделенного корня с заданной точностью.

Целью первого этапа является нахождение отрезков из области определения функции $f(x)$, внутри которых содержится только один корень решаемого уравнения. Иногда ограничиваются рассмотрением лишь какой-нибудь части области определения, вызывающей по тем или иным соображениям интерес. Для реализации данного этапа используются графические или аналитические способы.

При аналитическом способе отделения корней полезна следующая теорема :
Теорема 2.1. Непрерывная строго монотонная функция имеет и притом единственный нуль на отрезке тогда и только тогда, когда на его концах она принимает значения разных знаков.

Достаточным признаком монотонности функции на отрезке является сохранение знака производной функции.

Графический способ отделения корней целесообразно использовать в том случае, когда имеется возможность построения графика функции $y = f(x)$. Наличие графика исходной функции дает непосредственное представление о количестве и расположении нулей

функции, что позволяет определить промежутки, внутри которых содержится только один корень. Если построение графика функции $y = f(x)$ вызывает затруднение, часто оказывается удобным преобразовать уравнение к эквивалентному виду и построить графики функций $f_1(x) = f_2(x)$. Абсциссы точек пересечения этих графиков будут соответствовать значениям корней решаемого уравнения.

Так или иначе, при завершении первого этапа, должны быть определены промежутки, на каждом из которых содержится только один корень уравнения. Для уточнения корня с требуемой точностью обычно применяется какой-либо итерационный метод, заключающийся в построении числовой последовательности $x^k, k = 0, 1 \ldots$, сходящейся к искомому корню уравнения.

Метод Ньютона (метод касательных)

При нахождении корня уравнения методом Ньютона, итерационный процесс определяется формулой

$$x^{k+1} = x^k - \frac{f(x^k)}{f'(x^k)}, k = 1, 2 \ldots$$

Для начала вычислений требуется задание начального приближения x^0

Условия сходимости метода определяются следующей теоремой :

Теорема 2.2. Пусть на отрезке [a, b] функция 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 $x^k, k = 0, 1, \ldots$, определяемая методом Ньютона, монотонно сходится к корню уравнения $x^* \in (a, b)$ В качестве условия окончания итераций в практических вычислениях часто используется правило $|x^{k-1} - x^k| < \varepsilon$, следовательно $x^* = x^{k+1}$

Метод простых итераций

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

$$x = \varphi(x)$$

Решение ищется путем построения последовательности

$$x^{k+1} = \varphi(x^k), k = 0, 1, \ldots$$

начиная с некоторого заданного значения x^0 . Если $\varphi(x)$ - непрерывная функция, а $x^k, k = 0, 1, \ldots$- сходящаяся последовательность, то значение $x^* = \lim_{k\to\infty} x^k$ является решением уравнения.

Условия сходимости метода и оценка его погрешности определяются теоремой :

Теорема 2.3. Пусть функция $\varphi(x)$ определена и дифференцируема на отрезке [a, b]. Тогда если выполняются условия:

1) $\varphi(x) \in [a,b] \forall x \in [a,b]$ 2) $\exists : |\varphi'(x)| \leqslant q < 1 \forall x \in (a,b)$ то уравнение имеет и притом единственный на [a,b] корень x^*; к этому корню сходится определяемая методом простой итерации последовательность, начинающаяся с любого $x^0 \in [a,b]$.

Окончание счета по методу простых итераций, учитывая его быструю сходимость, можно контролировать путем проверки на малость модуля $|x^{k+1} - x^k|$

Данные

Уравнение:

$x^3 - 2x^2 - 10x + 15 = 0$

$\varepsilon = 0.0001$

Вывод программы

```
Заданная точность: 0.0001
================================================================================
Метод Ньютона:
  k|       xk|       f(xk)|      df/dxk|-f(xk)/(df(xk)/dxk)
  1|  1.36364|    0.180316|    -9.87603|   0.0182579
  2|  1.38189| 0.000703092|    -9.79868|7.17538e-005
  3|  1.38197|
eps(3) = 0.00007175 <= 0.00010000
Решение найдено на шаге 3.
X:1.38196601
================================================================================
Метод простых итераций:
phi(x) [0;1.38750000] => phi(x) [1.3 ; 1.5] для любого x [1.3 ; 1.5]
|phi_diff(x)| [0.01300000;0.07500000] => |phi_diff(x)| < 0.08 = q для любого x [
Условия теоремы выполнены
  k|       xk|     phi(xk)
  1|1.40000000|  1.38240000
  2|1.38240000|  1.38197480
  3|1.38197480|  1.38196619
  4|1.38196619|
eps(4) = 0.00000215 <= 0.00010000
Решение найдено на шаге 4.
X:1.38196619
```

Листинг программы

```
#include "libs.h";


double get_f(double &xk, int k)
{
```

```
double f = 0;
f = pow(xk, 3) - 2 * pow(xk, 2) - 10 * xk + 15;
return f;
}

double get_f_diff(double &xk, int k)
{
double f_diff = 0;
f_diff = 3 * pow(xk, 2) - 4 * xk - 10;
return f_diff;
}

double errorPassed(double xk, double x_prev, int k, double eps)
{
double eps_k = 0;
eps_k = abs(xk - x_prev);
return eps_k;
}

double Newton( double x_prev, int k, double eps)
{
double xk = 0;

if (k == 0)
{
x_prev = 1;
return Newton(x_prev, k + 1, eps);
}



xk = x_prev - (get_f(x_prev, k - 1) / get_f_diff(x_prev, k - 1));
if (errorPassed(xk, x_prev, k, eps) <= eps)
{
cout << setw(3) << k << "|" << setw(9) << xk << "|"  << endl;
cout << "eps(" << k << ") = " << fixed << setprecision(8) << errorPassed(xk, x_p
```

```
cout << " <= " << eps << endl << "Решение найдено на шаге " << k << '.' << endl
return xk;
x_prev = 0;
}
else
{
cout << setw(3) << k << "|" << setw(9) << xk << "|" << setw(12) << get_f(xk, k)
x_prev = xk;
return Newton(x_prev, k + 1, eps);
}
}

double errorPassedSI(double xk, double x_prev, int k, double eps)
{
double q = 0.2;
double eps_k = (q / (1 - q))*abs(xk - x_prev);
}

double get_phi(double x_prev, int k)
{
double phi = 0;
phi = (pow(x_prev, 3) - 2 * pow(x_prev, 2) + 15) / 10;
return phi;
}

double get_phi_diff(double x_prev, int k)
{
double phi_diff = 0;
phi_diff = (3 * pow(x_prev, 2) - 4 * x_prev) / 10;
return phi_diff;
}

void Theorem()
{
double a = 1.3;
```

```
double b = 1.5;

double phi_a = 0;
phi_a = (pow(a, 3) - 2 * pow(a, 2) + 15) / 10;

double phi_b = 0;
phi_b = (pow(b, 3) - 2 * pow(b, 2) + 15) / 10;

cout << "phi(x) [0;" << phi_b << "] => phi(x) [1.3 ; 1.5] для любого x [1.3 ; 1.5

double phi_a_diff = 0;
phi_a_diff = (3 * pow(a, 2) - 4 * a) / 10;

double phi_b_diff = 0;
phi_b_diff = (3 * pow(b, 2) - 4 * b) / 10;

cout << "|phi_diff(x)| [" << 0 << ";" << abs(phi_b_diff) << "] => |phi_diff(x)| <
cout << "Условия теоремы выполнены"<<endl;

}

double SimpleIteration(double x_prev, int k, double eps)
{
double xk = 0;

if (k == 0)
{
x_prev = 1;
return SimpleIteration(x_prev, k + 1, eps);
}

xk = get_phi(x_prev, k - 1);
if (errorPassedSI(xk, x_prev, k, eps)<=eps)
{
```

```
cout << setw(3) << k << "|" << setw(9) << xk << "|" << endl;
cout << "eps(" << k << ") = " << errorPassedSI(xk, x_prev, k, eps);
cout << " <= " << eps << endl << "Решение найдено на шаге " << k << '.' << endl
return xk;
}
else
{
cout << setw(3) << k << "|" << setw(9) << xk << "|" << setw(12) << get_phi(xk, k)
x_prev = xk;
return SimpleIteration(x_prev, k + 1, eps);
}
}


void u_main()
{

double x_prev = 0;
int k = 0;
double eps =0.0001;
cout << "Заданная точность: " << eps << endl;

cout << setw(48) << "=========================================================
cout << "Метод Ньютона: " << endl;
cout << setw(3) << "k" << "|" << setw(9) << "xk" << "|" << setw(12) << "f(xk)" <<
Newton(x_prev, k, eps);

cout << setw(48) << "=========================================================
cout << "Метод простых итераций: " << endl;
Theorem();
cout << setw(3) << "k" << "|" << setw(9) << "xk" << "|" << setw(12) << "phi(xk)"
SimpleIteration(x_prev, k, eps);

}
```

Задание

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.

Теоретическая часть

Решение систем нелинейных уравнений

Систему нелинейных уравнений с n неизвестными можно записать в виде

$$\begin{cases} f_1(x_1, x_2, \ldots, x_n) = 0 \\ f_2(x_1, x_2, \ldots, x_n) = 0 \\ \ldots\ldots\ldots\ldots\ldots\ldots \\ f_n(x_1, x_2, \ldots, x_n) = 0 \end{cases} \tag{1}$$

или, более коротко, в векторной форме

$$\mathbf{f}(\mathbf{x}) = \mathbf{0}$$

где $\mathbf{x}$ - вектор неизвестных величин, $\mathbf{f}$ - вектор-функция

$$\mathbf{x} = (x_1 \ldots x_n)^T, \mathbf{f} = (f_1 \ldots f_n), \mathbf{0} = (0 \ldots 0)$$

В редких случаях для решения такой системы удается применить метод последовательного исключения неизвестных и свести решение исходной задачи к решению одного нелинейного уравнения с одним неизвестным. Значения других неизвестных величин находятся соответствующей подстановкой в конкретные выражения. Однако в подавляющем большинстве случаев для решения систем нелинейных уравнений используются итерационные методы.

В дальнейшем предполагается, что ищется изолированное решение нелинейной системы. Как и в случае одного нелинейного уравнения, локализация решения может осуществляться на основе специфической информации по конкретной решаемой задаче (например, по физическим соображениям), и - с помощью методов математического анализа. При решении системы двух уравнений, достаточно часто удобным является графический способ, когда месторасположение корней определяется как точки пересечения кривых

$f_1(x_1, x_2) = 0, f_2(x_1, x_2) = 0$ на плоскости (x_1, x_2)

Метод Ньютона

Если определено начальное приближение $x^0 = (x_1^0, \ldots, x_n^0)$, итерационный процесс нахождения решения системы методом Ньютона можно представить в виде

В векторно-матричной форме расчетные формулы имеют вид

$$\mathbf{x^{k+1}} = \mathbf{x^k} + \triangle \mathbf{x^k}$$

где вектор приращений $\triangle \mathbf{x^k} = (\triangle x_1^k \ldots \triangle x_n^k) находится из решения уравнения$

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

Здесь $\mathbf{J}(\mathbf{x^k})$ - матрица Якобы первых производных функции $\mathbf{f}(\mathbf{x})$

Из матричных уравнений можем получить формулы итерационного процесса

$$\mathbf{x^{k+1}} = \mathbf{x^k} - \mathbf{J^{-1}}(\mathbf{x^k}) \mathbf{f}(\mathbf{x^k}), k = 0, 1, \ldots$$

При реализации алгоритма метода Ньютона в большинстве случаев предпочтительным является не вычисление обратной матрицы , а нахождение из системы значений приращений и вычисление нового приближения.

Использование метода Ньютона предполагает дифференцируемость функций $f_1(x), f_2(x), \ldots$ невырожденность матрицы Якоби. В случае, если начальное приближение выбрано в достаточно малой окрестности искомого корня, итерации сходятся к точному решению, причем сходимость квадратичная.

В практических вычислениях в качестве условия окончания итераций обычно используется критерий

$$\|x^{k-1} - x^k\| \leqslant \varepsilon$$

где ε - заданная точность

Метод простых итераций

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

$$\mathbf{x} = \varphi(\mathbf{x}), \varphi(\mathbf{x}) = (\varphi_1, \ldots, \varphi_n)$$

где функции $\varphi_1, \ldots, \varphi_n$ - определены и непрерывны в некоторой окрестности искомого изолированного решения $\mathbf{x^0} = (x_1^0, \ldots, x_n^0)^T$ последующие приближения в методе простой итерации находится по формуле в векторном виде

$$\mathbf{x^{k+1}} = \varphi(\mathbf{x^k}), k = 0, 1, \ldots$$

Достаточное условие сходимости итерационного процесса формулируется следующим образом :

Теорема 2.4. Пусть вектор-функция $\varphi(\mathbf{v})$ непрерывна, вместе со своей производной в ограниченной выпуклой замкнутой области G и $\max_{x \in G} \|\varphi'(\mathbf{x})\| \leqslant q \leqslant$, где q - постоянная.

Данные

Система:

$x - cos(y) = 1$

$y - sin(x) = 1$

Вывод программы

```
Заданная точность: 0.01
Метод Ньютона:
================================================================================
k|     x1k,    x2k|df1/dx1, df1/dx2|df2/dx2,df2/dx2|     detA1|     detA2|detJ
1|0.821754,1.71736|     1,-0.696707|0.983986,     1|-0.0217539| 0.0326439| 1
2|0.853964,1.73234|     1,-0.680938|0.989279,     1|-0.0322104|-0.0149852| 1
3|0.839157,1.75389|     1,   -0.657| 0.98698,     1| 0.0148076|-0.0215496| 1
4|0.817927,1.74408|
eps(4) = 0.00981087 <= 0.01000000
Решение найдено на шаге 4.
x1:0.81792671   x2:1.74408003

Метод простых итераций:
================================================================================
  k|       x1k|       x2k|phi1( x1k, x2k )|phi2( x1k, x2k )|
  1|0.82175394|1.71735609|      0.82175394|      1.71735609|
  2|0.85396435|1.73234128|      0.85396435|      1.73234128|
  3|0.83915676|1.75389090|      0.83915676|      1.75389090|
  4|0.81792671|1.74408003|      0.81792671|      1.74408003|
  5|0.82758221|1.72972982|      0.82758221|      1.72972982|
  6|0.84173477|1.73629751|      0.84173477|      1.73629751|
  7|0.83525332|1.74579989|      0.83525332|      1.74579989|
  8|0.82588835|1.74146651|      0.82588835|      1.74146651|
  9|0.83015717|1.73515029|      0.83015717|      1.73515029|
```

```
 10|0.83638497|1.73803743|      0.83638497|      1.73803743|
 11|0.83353742|1.74222536|      0.83353742|      1.74222536|
 12|0.82940939|1.74031407|      0.82940939|      1.74031407|
 13|0.83129298|1.73753265|      0.83129298|      1.73753265|
 14|0.83403517|1.73880335|      0.83403517|      1.73880335|
 15|0.83278223|1.74064860|      0.83278223|      1.74064860|
 16|0.83096325|1.73980617|
eps(16) = 0.00758181 <= 0.01000000
Решение найдено на шаге 16.
x1:0.83096325   x2:1.73980617
```

Листинг программы

```
#include "libs.h";

double errorPassed(double x1k, double x2k, double x1_prev, double x2_prev, int k
{
double eps1 = abs(x1k - x1_prev);
double eps2 = abs(x2k - x2_prev);

if (eps1 >= eps2)
{
double eps_k = eps1;
}

double eps_k = eps2;
return eps_k;
}

double det(vector <vector <double>> &A1)
{
double det = 1;
for (int i = 0; i < 2; ++i)
det *= A1[i][i];
return det;
}
```

```
double Newton(double x1_prev, double x2_prev, int k, double eps)
{
double x1k = 0;
double x2k = 0;

if (k == 0)
{
x1_prev = 0.8;
x2_prev = 1.75;
return Newton(x1_prev, x2_prev, k + 1, eps);
}

int n = 2;
vector <vector<double>> A1, A2, Jac;
A1.resize(n);
A2.resize(n);
Jac.resize(n);
for (int i = 0; i < n; ++i)
{
A1[i].resize(n, 0);
A2[i].resize(n, 0);
Jac[i].resize(n, 0);
}

double f1 = x1_prev - cos(x2_prev) - 1;
double f2 = x2_prev - sin(x1_prev) - 1;
double f1_diff_x1 = 1;
double f1_diff_x2 = sin(x2_prev);
double f2_diff_x1 = -cos(x1_prev);
double f2_diff_x2 = 1;

A1[0][0] = f1;
A1[1][0] = f2;
A1[0][1] = f1_diff_x2;
```

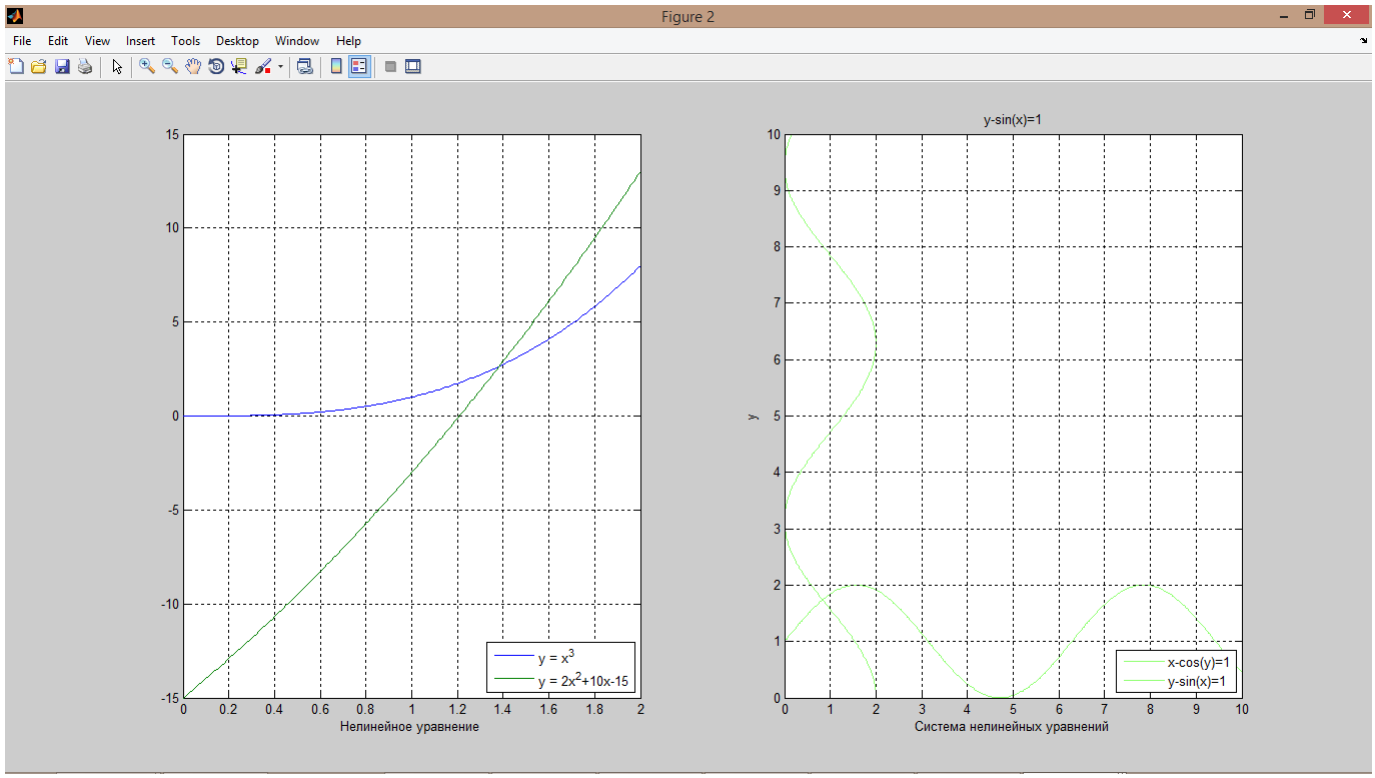
```
A1[1][1] = f2_diff_x2;

A2[0][0] =  f1_diff_x1;
A2[1][0] = f2_diff_x1;
A2[0][1] = f1;
A2[1][1] = f2;

Jac[0][0] = f1_diff_x1;
Jac[0][1] = f1_diff_x2;
Jac[1][0] = f2_diff_x1;
Jac[1][1] = f2_diff_x2;

x1k = x1_prev - (det(A1) / det(Jac));
x2k = x2_prev - (det(A2) / det(Jac));

//cout << "eps(" << k << ") = ";
if (errorPassed(x1k, x2k, x1_prev, x2_prev, k, eps) <= eps)
{
cout << setw(3) << k << "|" << setw(8) << x1k << "," << setw(8) << x2k << "|" <<
cout << "eps(" << k << ") = " << fixed << setprecision(8) << errorPassed(x1k, x2
cout << " <= " << eps << endl << "Решение найдено на шаге " << k << '.' << endl
x2:" << x2k << endl;
return x1k, x2k;
x1_prev = 0;
x2_prev = 0;
}
else
{
cout << setw(3) << k << "|" << setw(8) << x1k << "," << setw(8) << x2k << "|" <<
x1_prev = x1k;
x2_prev = x2k;
return Newton(x1_prev, x2_prev, k + 1, eps);
}
}
```

```
double errorPassedSI(double x1k, double x2k, double x1_prev, double x2_prev, int
{

double eps1 = abs(x1k - x1_prev);
double eps2 = abs(x2k - x2_prev);

if (eps1 >= eps2)
{
double ep = eps1;
}



double ep = eps2;

double eps_k = (q / (1 - q))*ep;

return eps_k;

}

double SimpleIteration(double x1_prev, double x2_prev, int k, double eps, double
{
double x1k = 0;
double x2k = 0;

if (k == 0)
{
x1_prev = 0.8;
x2_prev = 1.75;
return SimpleIteration(x1_prev, x2_prev, k + 1, eps, q);
}

x1k = cos(x2_prev) + 1; //phi1
x2k = sin(x1_prev) + 1; //phi2
```

```
//cout << "eps(" << k << ") = ";
if (errorPassedSI(x1k, x2k, x1_prev, x2_prev, k, eps, q) <= eps)
{
cout << setw(3) << k << "|" << setw(8) << x1k << "|" << setw(8) << x2k << "|" <<
cout << "eps(" << k << ") = " << errorPassedSI(x1k, x2k, x1_prev, x2_prev, k, eps
x2:" << x2k << endl;
return x1k, x2k;
}
else
{
cout << setw(3) << k << "|" << setw(9) << x1k << "|" << setw(9) << x2k << "|" <<
//cout << " > " << eps << endl;
x1_prev = x1k;
x2_prev = x2k;
return SimpleIteration(x1_prev, x2_prev, k + 1, eps, q);
}


}

void su_main()
{
int k = 0;
double x1_prev = 0;
double x2_prev = 0;
double q = 0.9;
double eps = 0.01;
cout << "Заданная точность: " << eps << endl;

cout << "Метод Ньютона: " << endl;
cout << "=========================================================================
cout << setw(3) << "k" << "|" << setw(9) << "x1k," << setw(8) << "x2k" << "|" <<

Newton(x1_prev, x2_prev, k, eps);
cout << endl;
```

```
cout << "Метод простых итераций: " << endl;
cout << "=========================================================================
cout << setw(3) << "k" << "|" << setw(9) << "x1k" << "|" << setw(9) << "x2k" <<
SimpleIteration(x1_prev, x2_prev, k, eps, q);

}
```

Скриншоты



Численные методы

Лабораторная работа №3

«Методы приближения функций. Численное дифференцирование и интегрирование»

Черыгова Лиза, 8О-304Б

Задание

3.1. Используя таблицу значений функции , вычисленных в точках построить интерполяционные многочлены Лагранжа и Ньютона, проходящие через точки . Вычислить значение погрешности интерполяции в точке X^*. **Теоретическая часть Интерполяция**

Пусть на отрезке задано множество несовпадающих точекx_i (интерполяционных узлов), в которых известны значения функции $f_i = f(x_i), i = \overline{0,n}$. Приближающая функция $\varphi(x,a)$ такая, что выполняются равенства

$$\varphi(x_i, a_0, \ldots, a_n) = f(x_i), i = \overline{0,n}$$

называется интерполяционной. Наиболее часто в качестве приближающей функции используют многочлены степени n:

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

Подставляя в $P_n(x)$ значения узлов интерполяции получаем систему линейных алгебраических уравнений относительно коэффициентов a_i:

$$\sum_{i=0}^{n} a_i x^i = f_k, k = \overline{0,n}$$

которая, в случае несовпадения узлов интерполяции имеет единственное решение.

Интерполяционный многочлен Лагранжа Для нахождения интерполяционного многочлена не обязательно решать систему. 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 , так называемые лагранжевы многочлены влияния, которые удовлетворяют условию $l_i(x_j) = 1, i = j, l_i(x_j) = 0, i \neq j$, 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

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

Это интерполяционный многочлен Лагранжа

Недостатком интерполяционного многочлена Лагранжа является необходимость полного пересчета всех коэффициентов в случае добавления дополнительных интерполяционных узлов. Чтобы избежать указанного недостатка используют интерполяционный многочлен в форме Ньютона.

Интерполяционный многочлен Ньютона

Введем понятие разделенной разности. Разделенные разности нулевого порядка совпадают со значениями функции в узлах. Разделенные разности первого порядка обозначаются $f(x_i,$ и определяются через разделенные разности нулевого порядка:

$$f(x_i, x_j) = \frac{f_i - f)j}{x_i - x_j}$$

Разделенная разность порядка n-k+2 определяется соотношениями

$$f(x_i, x_j, x_k, \ldots, x_{n-1}, x_n) = \frac{f(x_i, x_j, x_k, \ldots, x_{n-1}) - f(x_j, x_k, \ldots, x_{n-1}, x_n)}{x_i - x_n}$$

Таким образом, для (n+1)-й точки могут быть построены разделенные разности до n-го порядка; разделенные разности более высоких порядков равны нулю.

Пусть известны значения аппроксимируемой функции в точках. Интерполяционный многочлен, значения которого в узлах интерполяции совпадают со значениями функции может быть записан в виде:

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

Запись многочлена есть так называемый интерполяционный многочлен Ньютона. Отметим, что при добавлении новых узлов первые члены многочлена Ньютона остаются неизменными. Для повышения точности интерполяции в сумму могут быть добавлены новые члены, что требует подключения дополнительных интерполяционных узлов. При этом безразлично, в каком порядке подключаются новые узлы. Этим формула Ньютона выгодно отличается от формулы Лагранжа.

Данные

Функция: $y = tan(x) + x$
X_i: $0\ \frac{\pi}{8}\ \frac{2\pi}{8}\ \frac{3\pi}{8}$
$X^* = \frac{3\pi}{16}$

Вывод программы

```
Введите номер задания или 0 для выхода: 1
Интерполяционный многочлен Лагранжа

  i|    x(i)|    f(i)|  w_diff(x(i))| f(i)/w_diff(x(i))| X*-x(i)
  0|       0|       0|     -0.363355|                -0|0.589049
  1|0.392699|0.806913|      0.121118|           6.66219| 0.19635
  2|0.785398|  1.7854|     -0.121118|          -14.7409|-0.19635
  3|  1.1781| 3.59231|      0.363355|           9.88651|-0.589049

L(X*) = 1.23366
Абсолютная погрешность: 0.0235719
-------------------------------------------------------------------------------
Интерполяционный многочлен Ньютона

i|    x(i)|    f(i)| f(x(i),x(i+1))|f(x(i),x(i+1),x(i+2))|f(x(i),x(i+1),
                                                        x(i+2),x(i+3))
0|       0|       0|               |                     |
                  |        2.05479|                     |
1|0.392699|0.806913|               |            0.556287|
                  |        2.49169|                     |        1.80775
2|0.785398|  1.7854|               |             2.68599|
                  |        4.60127|                     |
3|  1.1781| 3.59231|

P(X*) = 1.23366
Абсолютная погрешность: 0.0235719
```

Листинг программы

```
#include "libs.h"

double w(vector<double> x, double x_et, int n)
{
double w = 1;
for (int i = 0; i < n;++i)
w *= (x_et - x[i]);
return w;
}

double w_diff(vector<double> x, double x_et, int n, int i)
{
double w_diff = 1;
for (int k = 0; k < n; ++k)
{
if (k != i)
w_diff *= x[i] - x[k];
}
return w_diff;
}

double Lagrange(vector<double> x, double x_et, vector <double> y,  int n)
{
for (int i = 0; i < n; ++i)
y[i] = tan(x[i]) + x[i];

vector <double> fu;
fu.resize(n, 0);
for (int i = 0; i < n; ++i)
fu[i] = y[i] / w_diff(x, x_et, n, i);

vector <double> fufufu;
fufufu.resize(n, 0);
for (int i = 0; i < n; ++i)
```

```
fufufu[i] = x_et-x[i];

cout << endl;
cout << setw(3) << "i" << '|' << setw(8) << "x(i)" << '|' << setw(8) << "f(i)" <<
for (int i = 0; i < n; ++i)
cout << setw(3) << i << '|' << setw(8) << x[i] << '|' << setw(8) << y[i] << '|' <

cout << endl;
double res = 0;
for (int i = 0; i < n; ++i)
res += y[i] * w(x, x_et, n) / (x_et - x[i]) / w_diff(x, x_et, n, i);
cout << "L(X*) = "<<res << endl;
double eps = 0;
eps = abs(res - tan(x_et) - x_et);
cout << "Абсолютная погрешность: " << eps << endl;
return res;


}

double f_1(vector<double> x, vector <double> y, int n, int i, int j)
{
double f_1 = 0;
f_1 = (y[i] - y[j]) / (x[i] - x[j]);
return f_1;
}

double f_2(vector<double> x, vector <double> y, int n, int i, int j, int k)
{
double f_2 = 0;
f_2 = (f_1(x, y, n, i, j) - f_1(x, y, n, j, k)) / (x[i] - x[k]);
return f_2;
}

double f_3(vector<double> x, vector <double> y, int n, int i, int j, int k, int
```

```
{
double f_3 = 0;
f_3 = (f_2(x, y, n, i, j, k) - f_2(x, y, n, j, k, l)) / (x[i] - x[l]);
return f_3;
}



double Newton(vector<double> x, double x_et, vector <double> y, int n)
{
cout << endl;
cout << setw(3) << "i" << '|' << setw(8) << "x(i)" << '|' << setw(8) << "f(i)" <<

cout << setw(3) << 0 << '|' << setw(8) << x[0] << '|' << setw(8) << y[0] << '|' <
cout << setw(22) << '|' << setw(21) << f_1(x, y, n, 0, 0 + 1) << '|' << setw(22)

cout << setw(3) << 1 << '|' << setw(8) << x[1] << '|' << setw(8) << y[1] << '|' <
cout << setw(22) << '|' << setw(21) << f_1(x, y, n, 1, 1 + 1) << '|' << setw(22)

cout << setw(3) << 2 << '|' << setw(8) << x[2] << '|' << setw(8) << y[2] << '|' <
cout << setw(22) << '|' << setw(21) << f_1(x, y, n, 2, 2 + 1) << '|' << setw(22)

cout << setw(3) << 3 << '|' << setw(8) << x[3] << '|' << setw(8) << y[3] << '|' <
cout << endl;



double P = 0;
P = y[0] + (x_et - x[0])*f_1(x, y, n, 1, 0) + (x_et - x[0])*(x_et - x[1])*f_2(x,
cout<<"P(X*) = " << P << endl;
double eps = 0;
eps = abs(P - tan(x_et) - x_et);
cout << "Абсолютная погрешность: " << eps << endl;

return P;
```

```
}

void Interpol_main()
{

int n = 4;
double x_et = 3*M_PI/16;
vector <double> x, y;
x.resize(n, 0);
y.resize(n, 0);

x[1] = M_PI / 8;
x[2] = 2 * M_PI / 8;
x[3] = 3 * M_PI / 8;

for (int i = 0; i < n; ++i)
y[i] = tan(x[i]) + x[i];

cout << "Интерполяционный многочлен Лагранжа" << endl;
Lagrange(x, x_et, y, n);
cout << "_____________________________________________________________________
cout << "Интерполяционный многочлен Ньютона" << endl;
Newton(x, x_et, y, n);
}
```

Скриншоты


Cherygova-cg16 Lab7
Lagrange polynom graph


Cherygova-cg16 Lab7
Newton spline

Численные методы
Лабораторная работа №3
«Методы решения нелинейных уравнений и систем нелинейных уравнений»
Черыгова Лиза, 8О-304Б

Задание

3.2. Построить кубический сплайн для функции, заданной в узлах интерполяции, предполагая, что сплайн имеет нулевую кривизну при x^0 и x^4. Вычислить значение функции в точке $X = X^*$.

Теоретическая часть

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:

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

который в узлах интерполяции принимает значения аппроксимируемой функции и непрерывен вместе со своими n-1 производными. Такой кусочно-непрерывный интерполяционный многочлен называется сплайном. Его коэффициенты находятся из условий равенства в узлах сетки значений сплайна и приближаемой функции, а также равенства n-1 производных соответствующих многочленов. На практике наиболее часто используется интерполяционный многочлен третьей степени, который удобно представить как

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

Для построения кубического сплайна необходимо построить n многочленов третьей степени, т.е. определить 4n неизвестных a_i, b_i, c_i, d_i. Эти коэффициенты ищутся из условий в узлах сетки.

Если ввести обозначение $h_i = x_i - x_{i-1}$, и исключить из системы a_i, b_i, d_i то можно

получить систему из n-1 линейных алгебраических уравнений относительно c_i трехдиагональной матрицей

Данные

X^* =1.5

X_i :0.0 0.9 1.8 2.7 3.6

Y_i :0.0 0.72235 1.5609 2.8459 7.7275

Вывод программы

```
c[1] = 0.23504
c[2] = -0.509788
c[3] = 3.45763
```

```
  i|[x[i - 1], x[i]]|       a_i|       b_i|       c_i|       d_i
  1|      [0,   0.9]|         0|  0.732099|         0| 0.0870517
  2|    [0.9,   1.8]|   0.72235|  0.943635|   0.23504| -0.275862
  3|    [1.8,   2.7]|    1.5609|  0.696361| -0.509788|   1.46942
  4|    [2.7,   3.6]|    2.8459|   2.49081|   3.45763|   -1.2806
```

```
X=1.5 принадлежит отрезку [0.9 ; 1.8], на этом отрезке функция представляется ку
f(1.5) = 1.31356
```

Листинг программы

```
#include "libs.h"

vector <double> x, y, a, b, e, d, c, c_i;

double h_i(vector<double>x, int i)
{
double h_i = 0;
h_i = x[i] - x[i - 1];
return h_i;
}

//Решение СЛАУ
double Q(int i, double P_prev, double Q_prev, vector <double> d, vector <double>
{
```

```
if (i == 0)
return d[0] / b[0];

return
(d[i] - a[i] * Q_prev) / (b[i] + a[i] * P_prev);
}

double P(int i, double P_prev, vector <double> e, vector <double> b, vector <dou
{
if (i == 0)
return -e[0] / b[0];

return
-e[i] / (b[i] + a[i] * P_prev);
}

double get_c(int i, double P_prev, double Q_prev, int k, vector <double> a, vect
{
if (i == k - 1)
{
double Qfinal = Q(i, P_prev, Q_prev, d,b,a);
c[i] = Qfinal;
return Qfinal;
}

double Ptec = P(i, P_prev, e,b,a);
double Qtec = Q(i, P_prev, Q_prev, d, b, a);

c[i] = Ptec * get_c(i + 1, Ptec, Qtec, k, a,b,d,e) + Qtec;
return c[i];
}
//--------

double Splaine(vector<double> x, vector<double> y, vector<double> c_i, double x_
{
```

```
vector <double> a_i, b_i, d_i;
a_i.resize(n, 0);
b_i.resize(n, 0);
d_i.resize(n, 0);


for (int i = 1; i < n; ++i)
{
if (i < n - 1)
{
a_i[i] = y[i - 1];
b_i[i] = (y[i] - y[i - 1]) / h_i(x, i) - h_i(x, i)*(c_i[i + 1] + 2 * c_i[i]) / 3
d_i[i] = (c_i[i + 1] - c_i[i]) / (3 * h_i(x, i));
}
else
{
a_i[i] = y[i - 1];
b_i[i] = y[i] - y[i - 1] / h_i(x, i) - 2 * h_i(x, i)*c_i[i] / 3;
d_i[i] = (-1)*c_i[i] / (3 * h_i(x, i));
}

}

cout << endl;

cout << setw(3) << "i" << "|" << setw(16) << "[x[i - 1], x[i]]" << "|" << setw(1
cout << endl;
for (int i = 1; i < n; ++i)
{
cout << setw(3) << i << "|" << setw(5) << "[" << x[i - 1] << "," << setw(6) << x
}

cout << endl;
```

```
cout << "X=1.5 принадледит отрезку [0.9 ; 1.8], на этом отреке функция представля

double f = 0;
f = a_i[2] + b_i[2] * (x_et - x[1]) + c_i[2] * pow((x_et - x[1]), 2) + d_i[2]*pow
cout << "f(1.5) = " << f << endl;
return f;

}

void Splaine_main()
{

int n = 5;
double x_et = 1.5;
int k = 3;
x.resize(n, 0);
y.resize(n, 0);
a.resize(k, 0);
b.resize(k, 0);
e.resize(k, 0);
c.resize(k, 0);
d.resize(k, 0);
c_i.resize(n, 0);

x[0] = 0.0;
x[1] = 0.9;
x[2] = 1.8;
x[3] = 2.7;
x[4] = 3.6;
y[0] = 0.0;

y[1] = 0.72235;
y[2] = 1.5609;
y[3] = 2.8459;
y[4] = 7.7275;
```

```
b[0] = 2 * (h_i(x, 1) + h_i(x, 2));
e[0] = h_i(x, 2);
d[0] = 3 * (((y[2] - y[1]) / h_i(x, 2)) - ((y[1] - y[0]) / h_i(x, 1)));

a[1] = h_i(x, 2);
b[1] = 2 * (h_i(x, 2) + h_i(x, 3));
e[1] = h_i(x, 3);
d[1] = 3 * (((y[3] - y[2]) / h_i(x, 3)) - ((y[2] - y[1]) / h_i(x, 2)));

a[2] = h_i(x, 3);
b[2] = 2 * (h_i(x, 3) + h_i(x,4));
d[2] = 3 * (((y[4] - y[3]) / h_i(x, 4)) - ((y[3] - y[2]) / h_i(x, 3)));

c[0] = get_c(0, 0, 0,k, a,b,d,e);
cout << endl;
for (int i = 0; i < k; ++i)
cout << "c[" << i + 1 << "] = " << c[i] << endl;

c_i[2] = c[0];
c_i[3] = c[1];
c_i[4] = c[2];

Splaine(x, y, c_i, x_et, n);
}
```

Задание

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.

Теоретическая часть Пусть задана таблично в узлах x_j функция $y_i = f(x_j), j = \overline{0, n}$. При этом значения функции y_i определены с некоторой погрешностью, также из физических соображений известен вид функции, которой должны приближенно удовлетворять табличные точки, например: многочлен степени, 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.

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

Минимума можно добиться только за счет изменения коэффициентов многочлена. Необходимые условия экстремума имеют вид

$$\frac{\partial \Phi}{\partial a_k} = 2 \sum_{j=0}^{N} [\sum_{i=0}^{n} a_i x^j - y_j] x_j^k, k = \overline{0, n}$$

Приведем выражение к эквивалентному виду

$$\sum_{i=0}^{n} a_i \sum_{j=0}^{N} x_j^{k+i} = \sum_{j=0}^{N} y_j x_j^k$$

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.

Данные

X_i : -0.9 0.0 0.9 1.8 2.7 3.6

Y_i : -1.2689 0.0 1.2689 2.6541 4.4856 9.9138

Вывод программы

```
Полином 1-й степени
a[1] = -0.190129
```

```
a[2] = 2.24621
       i       0       1       2       3       4       5
     x_i    -0.9       0     0.9     1.8     2.7     3.6
  F1(xi)-2.21171-0.190129 1.83146 3.85304 5.87463 7.89621
Сумма квадратов ошибок Ф = 8.67903

Полином 2-й степени
a[1] = -0.464496
a[2] = 0.874367
a[3] = 0.508089

       i|       0|       1|       2|       3|       4|       5
     x_i|    -0.9|       0|     0.9|     1.8|     2.7|     3.6
  F2(xi)|-0.839875|-0.464496|0.733986| 2.75557| 5.60026| 9.26805

Сумма квадратов ошибок Ф = 2.35571
```

Листинг программы

```
#include "libs.h"

void LU(vector <vector <double>> A, vector <vector <double>> &L,
vector <vector <double>> &U, int n)
{
for (int j = 0; j < n; ++j)
U[0][j] = A[0][j];

for (int i = 0; i < n; ++i)
L[i][0] = A[i][0] / U[0][0];

for (int i = 1; i < n; ++i)
{
for (int j = i; j < n; ++j)
{
double sum = 0;
for (int k = 0; k < i; ++k)
```

```
sum += L[i][k] * U[k][j];
U[i][j] = A[i][j] - sum;

sum = 0;
for (int k = 0; k < i; ++k)
sum += L[j][k] * U[k][i];
L[j][i] = (A[j][i] - sum) / U[i][i];
}
}
}

vector<double> SLAU(vector <vector <double>> A, vector <vector <double>> L,
vector <vector <double>> U, vector <double> b, int n)
{
vector <double> x, z;
x.resize(n, 0);
z.resize(n, 0);

z[0] = b[0];

for (int i = 1; i < n; ++i)
{
double sum = 0;
for (int j = 0; j < i; ++j)
sum += L[i][j] * z[j];
z[i] = b[i] - sum;
}

x[n - 1] = z[n - 1] / U[n - 1][n - 1];

for (int i = n - 2; i >= 0; --i)
{
double sum = 0;
for (int j = i + 1; j < n; ++j)
sum += U[i][j] * x[j];
```

```
x[i] = (z[i] - sum) / U[i][i];
}
return x;
}



double polinom1(vector<double> x, vector<double> y, double sum_x, double sum_x2,
{
vector <vector <double>> A, L, U;
vector <double> b, a , f;

A.resize(k);
L.resize(k);
U.resize(k);
b.resize(k,0);
a.resize(k,0);
f.resize(n, 0);
for (int i = 0; i < k; ++i)
{
A[i].resize(k, 0);
L[i].resize(k, 0);
U[i].resize(k, 0);
}

A[0][0] = (n);
A[1][0] = sum_x;
A[0][1] = sum_x;
A[1][1] = sum_x2;
b[0] = sum_y;
b[1] = sum_yx;

cout << "Полином 1-й степени" << endl;

LU(A, L, U, k);
```

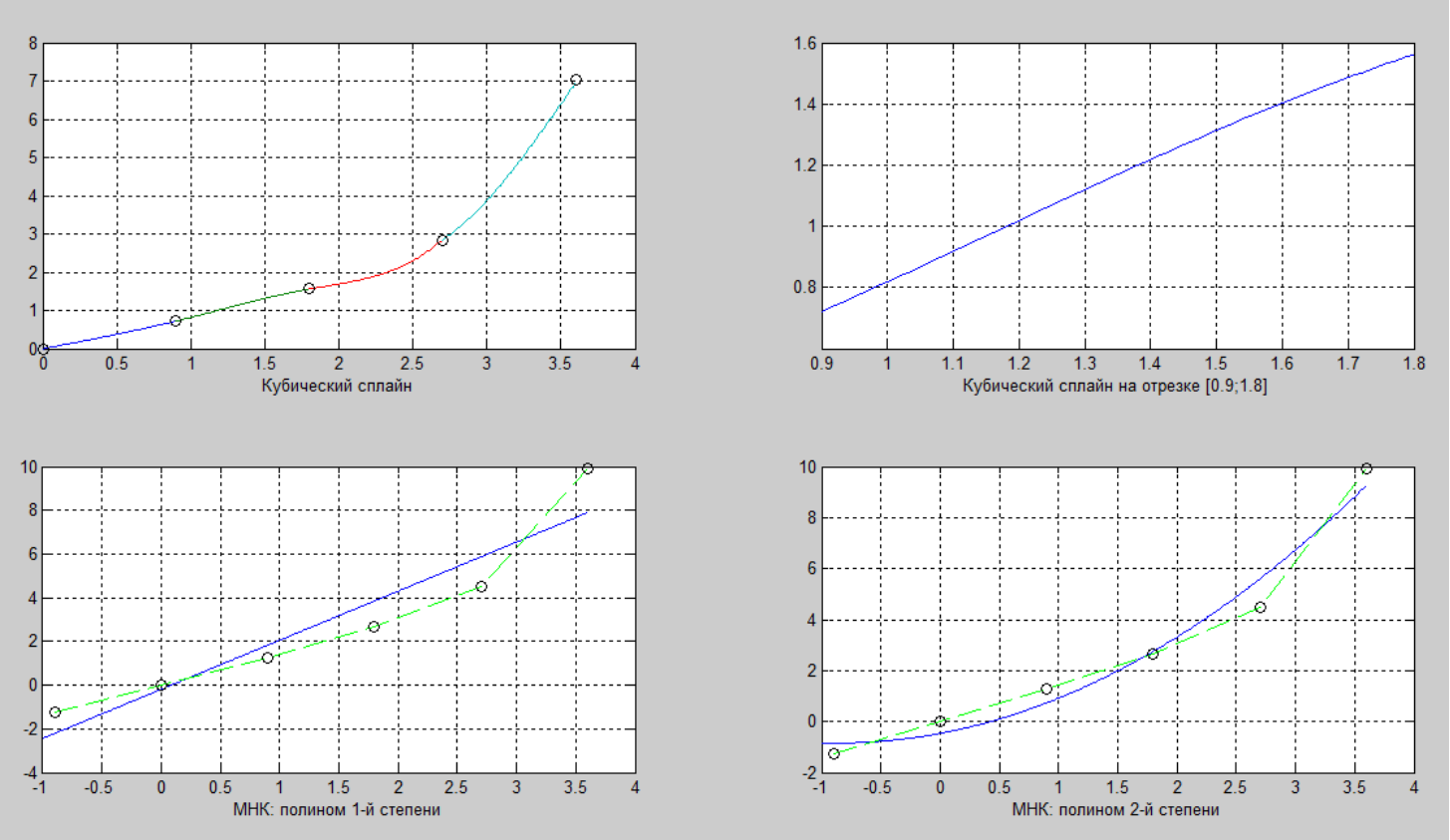
```
a = SLAU(A, L, U, b, k);
for (int i = 0; i < k; ++i)
cout << "a[" << i + 1 << "] = " << a[i] << endl;

for (int i = 0; i < n; ++i)
f[i] = a[0] + a[1] * x[i];
cout << setw(8) << "i" << setw(8) << "0" << setw(8) << "1" << setw(8) << "2" << s
cout << setw(8) << "x_i" << setw(8) << x[0] << setw(8) << x[1] << setw(8) << x[2]
cout << setw(8) << "F1(xi)" << setw(8) << f[0] << setw(8) << f[1] << setw(8) << f

double F = 0;
for (int i = 0; i < n; ++i)
F += pow(f[i] - y[i], 2);

cout << "Сумма квадратов ошибок Ф = " << F << endl;
return F;

}

double polinom2(vector<double> x, vector<double> y, double sum_x, double sum_x2,
{
vector <vector <double>> A, L, U;
vector <double> b, a, f;

A.resize(k+1);
L.resize(k+1);
U.resize(k+1);
b.resize(k + 1, 0);
a.resize(k + 1, 0);
f.resize(n, 0);
for (int i = 0; i < k + 1; ++i)
{
A[i].resize(k + 1, 0);
L[i].resize(k + 1, 0);
U[i].resize(k + 1, 0);
```

```
}

A[0][0] = (n);
A[1][0] = sum_x;
A[2][0] = sum_x2;
A[0][1] = sum_x;
A[1][1] = sum_x2;
A[2][1] = sum_x3;
A[0][2] = sum_x2;
A[1][2] = sum_x3;
A[2][2] = sum_x4;
b[0] = sum_y;
b[1] = sum_yx;
b[2] = sum_yx2;

cout << "Полином 2-й степени" << endl;

LU(A, L, U, k+1);
a = SLAU(A, L, U, b, k+1);
for (int i = 0; i < k+1; ++i)
cout << "a[" << i + 1 << "] = " << a[i] << endl;
cout << endl;

for (int i = 0; i < n; ++i)
f[i] = a[0] + a[1] * x[i]+a[2]*pow(x[i],2);
cout << setw(8) << "i" << "|" << setw(8) << "0" << "|" << setw(8) << "1" << "|" <
cout << setw(8) << "x_i" << "|" << setw(8) << x[0] << "|" << setw(8) << x[1] <<
cout << setw(8) << "F2(xi)" << "|" << setw(8) << f[0] << "|" << setw(8) << f[1] <
cout << endl;

double F = 0;
for (int i = 0; i < n; ++i)
F += pow(f[i] - y[i], 2);

cout << "Сумма квадратов ошибок Ф = " << F << endl;
```

```
return F;

}

void MNK_main()
{

int n;
int k=2;
double sum_x = 0;
double sum_y = 0;
double sum_yx = 0;
double sum_yx2 = 0;
double sum_x2 = 0;
double sum_x3 = 0;
double sum_x4 = 0;
ifstream test("test3-MNK.txt");
test >> n;

vector <double> x, y;

x.resize(n);
y.resize(n);
for (int i = 0; i < n; ++i)
{
test >> x[i];
test >> y[i];
}

for (int i = 0; i < n; ++i)
{
sum_x += x[i];
sum_y += y[i];
sum_yx += y[i]*x[i];
sum_yx2 += y[i]*pow(x[i],2);
```

```
sum_x2 += pow(x[i], 2);
sum_x3 += pow(x[i], 3);
sum_x4 += pow(x[i], 4);
}

polinom1(x,y,sum_x, sum_x2, sum_y, sum_yx, n, k);
cout << endl;
polinom2(x, y, sum_x, sum_x2, sum_x3, sum_x4, sum_y, sum_yx, sum_yx2, n, k);
}
```

Скриншоты



Задание

3.4. Вычислить первую и вторую производную от таблично заданной функции в точке $X = X^*$

Теоретическая часть

Формулы численного дифференцирования в основном используются при нахождении производных от функции $y = f(x)$, заданной таблично. Исходная функция на отрезках заменяется некоторой приближающей, легко вычисляемой функцией.

При решении практических задач, как правило, используются аппроксимации первых и вторых производных.

В первом приближении, таблично заданная функция может быть аппроксимирована отрезками прямой

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

$$y'(x) \approx \frac{y_{i+1} - y_i}{x_{i+1} - x_i} = const$$

производная является кусочно-постоянной функцией и рассчитывается, по формуле с первым порядком точности в крайних точках интервала, и со вторым порядком точности в средней точке интервала .

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

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

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

При равностоящих точках разбиения, данная формула обеспечивает второй порядок точности.

Для вычисления второй производной, необходимо использовать интерполяционный многочлен, как минимум второй степени. После дифференцирования многочлена получаем

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

Данные

X_i: 1.0 2.0 3.0 4.0 5.0

Y_i: 1.0 2.6931 4.0986 5.3863 6.6094

$X* = 3.0$

Вывод программы

```
X* = 3
Производные первого порядка точности

Левосторонняя производная: y‘(3) = 1.4055
Правосторонняя производная: y‘(3) = 1.2877

Поизводная второго порядка точности: y‘(3) = 1.3466

Вторая производная: y‘‘(3) = -0.1178
```

Листинг программы

```
#include "libs.h"

double left_proisv(vector<double> x, vector<double> y)
{
double f_diff = 0;
f_diff = (y[2] - y[1]) / (x[2] - x[1]);
return f_diff;
}

double right_proisv(vector<double> x, vector<double> y)
{
double f_diff = 0;
f_diff = (y[3] - y[2]) / (x[3] - x[2]);
return f_diff;
}

double double_prec_proisv(vector<double> x, vector<double> y, double x_et)
{
double f_diff = 0;

f_diff =(  ( ( (y[3] - y[2]) / (x[3] - x[2]) ) - ( (y[2] - y[1]) / (x[2] - x[1])
```

```
return f_diff;
}

double double_proisv(vector<double> x, vector<double> y)
{
double f_diff = 0;
f_diff = 2 * ((((y[3] - y[2]) / (x[3] - x[2])) - ((y[2] - y[1]) / (x[2] - x[1]))
return f_diff;
}

void Proisv_main()
{

int n;
double x_et;
ifstream test("test3-Proisv.txt");
test >> n;
test >> x_et;

vector <double> x, y;

x.resize(n);
y.resize(n);
for (int i = 0; i < n; ++i)
{
test >> x[i];
test >> y[i];
}

cout << "X* = "<<x_et << endl;

cout << "Производные первого порядка точности"  << endl;
cout << endl;
cout << "Левосторонняя производная: y‘(3) = " << left_proisv(x, y) << endl;
cout << "Правосторонняя производная: y‘(3) = " << right_proisv(x, y) << endl;
```

```
cout << endl;
cout << "Поизводная второго порядка точности: y‘(3) = " << double_prec_proisv(x,
cout << endl;
cout << "Вторая производная: y‘‘(3) = " << double_proisv(x, y) << endl;
}
```

Задание

3.5. Вычислить определенный интеграл методами прямоугольников, трапеций, Симпсона . Оценить погрешность вычислений, используя Метод Рунге-Ромберга. **Теоретическая часть**

Формулы численного интегрирования используются в тех случаях, когда вычислить аналитически определенный интеграл $F = \int_a^b f(x)dx$ не удается. Отрезок [a, b] разбивают точками $x_0, \ldots, x_n$, так что $a = x_0 \leqslant x_0 \ldots \leqslant x_N = b$c достаточно мелким шагом $h_i = x_i - x_{i-1}$ и на одном или нескольких отрезках подынтегральную функцию заменяют такой приближающей, так что она, во-первых, близка , а, во-вторых, интеграл от нее легко вычисляется.

При использовании интерполяционных многочленов различной степени, получают формулы численного интегрирования различного порядка точности.

Заменим подынтегральную функцию, интерполяционным многочленом Лагранжа нулевой степени, проходящим через середину отрезка , 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.

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

Это формула называется формулой трапеции

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

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

Метод Рунге-Ромберга-Ричардсона позволяет получать более высокий порядок точности вычисления. Если имеются результаты вычисления определенного интеграла на сетке с шагом - и на сетке с шагом h $F = F_h + O(h^p)$ и на сетке с шагом kh $F = F_{kh} + O((kh)^p)$,

то

$$\int_a^b f(x)dx \approx F_h + \frac{F_h - F_{kh}}{k^p - 1} + O(h^{p+1})$$

Данные

Функция $y = \frac{1}{(x^4+16)}$
h = 0.5 h = 0.25

Вывод программы

```
Вычисление определённого интеграла с шагом 0.5
Метод прямоугольников с шагом 0.5
F(0) = 0
F(1) = 0.0312424
F(2) = 0.0618864
F(3) = 0.0889993
F(4) = 0.108701
Оценка остаточного члена R = 0.000325521
Метод прямоугольников с шагом 0.25
F(0) = 0
F(1) = 0.0156248
F(2) = 0.0312305
F(3) = 0.0467079
F(4) = 0.0617807
F(5) = 0.0759837
F(6) = 0.0887555
F(7) = 0.0996379
F(8) = 0.108453
Оценка остаточного члена R = 8.13802e-005
Уточнённое значение (метод Рунге-Ромберга-Ричардсона) 0.108371
с абсолютной погрешностью 0.00000126. Точное значение интеграла 0.10837200
--------------------------------------------------------------------------------

Метод трапеций с шагом 0.5
F(0) = 0.00000000
F(1) = 0.04675340
F(2) = 0.07616517
```

```
F(3) = 0.09990404
F(4) = 0.10771654
Оценка остаточного члена R = 0.00065104
Метод трапеций с шагом 0.25
F(0) = 0.00000000
F(1) = 0.02343369
F(2) = 0.03899789
F(3) = 0.05431989
F(4) = 0.06902577
F(5) = 0.08258222
F(6) = 0.09445166
F(7) = 0.10430236
F(8) = 0.10820861
Оценка остаточного члена R = 0.00016276
Уточнённое значение (метод Рунге-Ромберга-Ричардсона) 0.10837263
с абсолютной погрешностью 0.00000063. Точное значение интеграла 0.10837200
--------------------------------------------------------------------------------

Метод симпсона с шагом 0.5
F(0) = 0.00000000
F(1) = 0.05192121
F(2) = 0.07152905
F(3) = 0.10318088
F(4) = 0.10838921
Оценка остаточного члена R = 0.00020552
Метод Симпсона с шагом 0.25
F(0) = 0.00000000
F(1) = 0.02603658
F(2) = 0.03641272
F(3) = 0.05684205
F(4) = 0.06664597
F(5) = 0.08472124
F(6) = 0.09263420
F(7) = 0.10576846
F(8) = 0.10837263
```

```
Оценка остаточного члена R = 0.00001284
F(0) = 0.00000000
F(1) = 0.02603658
F(2) = 0.03641272
F(3) = 0.05684205
F(4) = 0.06664597
F(5) = 0.08472124
F(6) = 0.09263420
F(7) = 0.10576846
F(8) = 0.10837263
Оценка остаточного члена R = 0.00001284
Уточнённое значение (метод Рунге-Ромберга-Ричардсона) 0.10837026
с абсолютной погрешностью 0.00000174. Точное значение интеграла 0.10837200
```

Листинг программы

```
#include "libs.h"

double function(double b)
{
double f = 0;
f = 1 / (pow(b, 4) + 16);
return f;
}

double func_doub_d(double b)
{
double f = 0;
f = 32 * pow(b, 6) / pow((pow(b, 4) + 16), 3) - 12 * pow(b, 2) / pow((pow(b, 4) +
return f;
}

double func_fourth_d(double b)
{
double f = 0;
f = 24 * 16 * 16 * pow(b, 12)*pow(b*b*b*b + 16, -5) - 24 * 16 * 9 * pow(b, 8)*pow
```

```
return f;
}

double Runge_Romb_Rich(double F_h, double F_kh, int p)
{
double RRR = 0;
RRR = F_h + ((F_h - F_kh) / (pow(2, p) - 1));
return RRR;
}

vector<double> rectangle(vector<double> x, vector<double> y, int n, double h)
{
vector<double> F;
F.resize(n + 1, 0);

for (int i = 1; i < n + 1; ++i)
F[i] = h*function(((x[i - 1] + x[i]) / 2)) + F[i - 1];

double R = 0;
double M = func_doub_d(x[0]);
for (int i =0; i < n + 1; ++i)
{
if (func_doub_d(x[i])> M)
M = func_doub_d(x[i]);
}

R = 2*pow(h, 2)*M / 24;

for (int i = 0; i < n + 1; ++i)
cout << "F(" << i << ") = " << F[i] << endl;

cout << "Оценка остаточного члена R = " << R << endl;

//cout << "Уточнённое значение (метод Рунге-Ромберга-Ричардсона) " << Runge_Romb
//cout << "с абсолютной погрешностью " << abs(F_toch - Runge_Romb_Rich(F_toch, F
```

```
return F;

}



vector<double> trapezium( vector<double> x, vector<double> y, int n, double h)
{
vector<double> F;
F.resize(n + 1, 0);
for (int i = 1; i < n + 1; ++i)
{
if (i == 1 )
F[i] = h*(function(x[i - 1]) / 2 + function(x[i])) + F[i - 1];
else
F[i] = h*(function(x[i])) + F[i - 1];
if (i == n)
F[i] = h*(function(x[i]) / 2) + F[i - 1];
}

double R = 0;
double M = func_doub_d(x[0]);
for (int i = 0; i < n + 1; ++i)
{
if (func_doub_d(x[i])> M)
M = func_doub_d(x[i]);
}

R = 2 * pow(h, 2)*M / 12;

for (int i = 0; i < n + 1; ++i)
cout << "F(" << i << ") = " << F[i] << endl;

cout << "Оценка остаточного члена R = " << R << endl;
//cout << "Уточнённое значение (метод Рунге-Ромберга-Ричардсона) " << Runge_Romb
```

```
//cout << "с абсолютной погрешностью " << abs(F_toch - Runge_Romb_Rich(F_toch, F
return F;
}

vector<double> simpson(vector<double> x, vector<double> y, int n, double h)
{
vector<double> F;
F.resize(n + 1, 0);
F[1] = h*(function(x[0]) + 4 * function(x[1])) / 3;
for (int i = 2; i < n + 1; ++i)
{
if (i % 2 != 0 )
F[i] = h*(4*function(x[i]))/3 + F[i - 1];
else
F[i] = h*(2*function(x[i]))/3 + F[i - 1];
if (i == n)
F[i] = h*(function(x[i])) / 3 + F[i - 1];
}

double R = 0;
double M = func_fourth_d(x[0]);
for (int i = 0; i < n + 1; ++i)
{
if (func_fourth_d(x[i])> M)
M = func_fourth_d(x[i]);
}

R = 2 * pow(h, 4)*M / 180;

for (int i = 0; i < n + 1; ++i)
cout << "F(" << i << ") = " << F[i] << endl;
cout << "Оценка остаточного члена R = " << R << endl;
//cout << "Уточнённое значение (метод Рунге-Ромберга-Ричардсона) " << Runge_Romb
//cout << "с абсолютной погрешностью " << abs(F_toch - Runge_Romb_Rich(F_toch, F
return F;
```

```
}


void Integr_main()
{
double x0 = 0;
double x1=2;
double h1 = 0.5;
double h2 = 0.25;
double F_toch = 0.108372;
int n = 4;
int N1 = 8;
//n = (abs(x0) + abs(x1)) / h1;
//N = (abs(x0) + abs(x1)) / h2;

vector <double> x, y, F_rkh, F_tkh, F_skh ;
x.resize(n+1, 0);
y.resize(n+1, 0);
F_rkh.resize(n + 1, 0);
F_tkh.resize(n + 1, 0);
F_skh.resize(n + 1, 0);

for (int i = 1; i < n+1; ++i)
x[i] = x0 + h1 + x[i-1];

vector <double> x2, y2, F_rh, F_th, F_sh;
x2.resize(N1 + 1, 0);
y2.resize(N1 + 1, 0);
F_rh.resize(N1 + 1, 0);
F_th.resize(N1 + 1, 0);
F_sh.resize(N1 + 1, 0);

for (int i = 1; i < N1+1; ++i)
x2[i] = x0 + h2 + x2[i-1];
```

```
cout << "Вычисление определённого интеграла с шагом 0.5" << endl;
cout << "Метод прямоугольников с шагом 0.5" << endl;
//rectangle(x, y, n, h1);
F_rkh = rectangle(x, y, n, h1);
double f_rkh = F_rkh[n];
cout << "Метод прямоугольников с шагом 0.25" << endl;
//rectangle(x2, y2, N1, h2);
F_rh = rectangle(x2, y2, N1, h2);
double f_rh = F_rh[N1];

cout << "Уточнённое значение (метод Рунге-Ромберга-Ричардсона) " << Runge_Romb_R
cout << "с абсолютной погрешностью " << fixed << setprecision(8) << abs(F_toch -
cout << "_____________________________________________________________________
cout << endl;
cout << "Метод трапеций с шагом 0.5" << endl;
//trapezium(x, y, n, h1);
F_tkh = trapezium(x, y, n, h1);
double f_tkh = F_tkh[n];
cout << "Метод трапеций с шагом 0.25" << endl;
//trapezium(x2, y2, N1, h2);
F_th = trapezium(x2, y2, N1, h2);
double f_th = F_th[N1];
cout << "Уточнённое значение (метод Рунге-Ромберга-Ричардсона) " << Runge_Romb_R
cout << "с абсолютной погрешностью " << fixed << setprecision(8) << abs(F_toch -
cout << "_____________________________________________________________________
cout << endl;
cout << "Метод симпсона с шагом 0.5" << endl;
//simpson(x, y, n, h1);
F_skh = simpson(x, y, n, h1);
double f_skh = F_skh[n];
cout << "Метод Симпсона с шагом 0.25" << endl;
simpson(x2, y2, N1, h2);
F_sh = simpson(x2, y2, N1, h2);
double f_sh = F_sh[N1];
cout << "Уточнённое значение (метод Рунге-Ромберга-Ричардсона) " << Runge_Romb_R
```

```
cout << "с абсолютной погрешностью " << fixed << setprecision(8) << abs(F_toch -

cout << endl;
}
```

Численные методы

Лабораторная работа №4

«Методы решения начальных и краевых задач для обыкновенных дифференциальных уравнений (ОДУ) и систем ОДУ»

Черыгова Лиза, 8О-304Б

Задание

4.1. Реализовать методы Эйлера, Рунге-Кутты и Адамса 4-го порядка в виде программ, задавая в качестве входных данных шаг сетки . 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.

Теоретическая часть

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 $x_0 = a$. Введем разностную сетку на отрезке [a, b] $\Omega^k = \{x_k = x_0 + hk\}, k = \overline{0, N}, h = |b - a|/N$. Точки x_k - называются узлами разностной сетки, расстояния между узлами – шагом разностной сетки (h) , а совокупность значений какой либо величины заданных в узлах сетки называется сеточной функцией. Погрешность вычисляется, как разность между истинным и приближенным значением.

Метод Эйлера

Метод Эйлера играет важную роль в теории численных методов решения ОДУ, хотя и не часто используется в практических расчетах из-за невысокой точности. График функции, которая является решением задачи Коши , представляет собой гладкую кривую, проходящую через точку (x_0, y_0) и имеет в этой точке касательную. Тангенс угла наклона касательной к оси Ox равен значению производной от решения в точке и равен значению правой части дифференциального уравнения в точке (x_0, y_0). В случае небольшого шага разностной сетки h график функции и график касательной не успевают сильно разойтись друг от друга и можно в качестве значения решения в узле принять значение касательной , вместо значения неизвестного точного решения . Считая теперь точку начальной и повторяя все предыдущие рассуждения, получим значение в узле.

Формула метода Эйлера

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

Метод Рунге-Кутта 4-го порядка

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

$$y_{k+1} = y_k + \triangle y_k, \triangle y_k = \sum_{i=1}^{p} c_i K_i^k$$

$$K_i^k = hf(x^k + a_i h, y_k + h\sum_{j=1}^{i-1} b_{ij} K_j^k, i = \overline{2,p}$$

Параметры подбираются так, чтобы значение совпадало со значением разложения в точке точного решения в ряд Тейлора с погрешностью $O(h^{p+1})$
для p = 4

$$y_{k+1} = y_k + \triangle y_k, \triangle y_k = \frac{1}{6}(K_1^k + 2K_2^k + 2_3^k + K_4^k)$$

$K_1^k = hf(x_k, y_k), K_2^k = hf(x_k + \frac{1}{2}h, y_k + \frac{1}{2}K_1^k), K_3^k = hf(x_k + \frac{1}{2}h, y_k + \frac{1}{2}K_2^k), K_4^k = hf(x_k + h, y_k + K$

Основным способом контроля точности получаемого численного решения при решении задачи Коши является методы основанные на принципе Рунге-Ромберга-Ричардсона.

Пусть y_k решение задачи Коши полученое методом Рунге-Кутты p – го порядка точности с шагом h в точке x+2h. Пусть y^{2h} решение той же задачи в точке x+2h, полученное тем же методом, но с шагом 2h . Тогда выражение

$$\tilde{y} = y^h + \frac{y^h - y^{2h}}{2^p - 1}$$

апапроксимирует точное решение в точке x+2h y(x+2h) с p+1-ым порядком

Решение задачи Коши для ОДУ второго и более высокого порядка

$$y^{(n)} = f(x, y, y', \ldots, y^{(n-1)})$$

$$y(x^0) = y_0, \ldots y^{(n-1)}(x_0) = y_{0(n-1)}$$

Основной прием используемый при решении заключается в введении новых переменных и сведении задачи для ОДУ высокого порядка к решению системы ОДУ первого порядка.

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.

При использовании интерполяционного многочлена 3-ей степени построенного по значениям подынтегральной функции в последних четырех узлах получим метод Адамса четвертого порядка точности:

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

где f_k значение подынтегральное функции в узлах x_k

В узле x^0 решение известно из начальных условий, а в других трех узлах x^1, x^2, x^3 решения можно получить с помощью подходящего одношагового метода, например: метода Рунге-Кутты четвертого порядка

Данные

Вывод программы

```
Метод Эйлера
  i|   x|          y_k|        y_ist|          Eps|
  0|   1|            3|            3|            0|
  1| 1.1|          3.2|      3.21909|    0.0190909|
  2| 1.2|         3.44|      3.47333|    0.0333333|
  3| 1.3|      3.71463|      3.75923|    0.0446027|
  4| 1.4|      4.02026|      4.07429|    0.0540263|
  5| 1.5|      4.35436|      4.41667|     0.062306|
  6| 1.6|      4.71511|        4.785|    0.0698909|
  7| 1.7|      5.10116|      5.17824|     0.077075|
  8| 1.8|       5.5115|      5.59556|    0.0840538|
  9| 1.9|      5.94536|      6.03632|    0.0909593|
 10|   2|      6.40212|          6.5|     0.097881|

Оценка P-P-P
3       0
3.18667 0.0324242
3.40533 0.068
3.65223 0.106999
```

3.92454 0.149746
4.22014 0.196528

Метод Рунге-Кутта

k	x	y_k	z_k	y_ист	eps_k
0	1	3	2	3	0.000000000000
1	1.100000000000	3.219092790555	2.400000000000	3.219090909091	0.000001881464
2	1.200000000000	3.473336351586	2.762809917355	3.473333333333	0.000003018253
3	1.300000000000	3.759234524857	3.102020202020	3.759230769231	0.000003755626
4	1.400000000000	4.074289982878	3.425685037573	4.074285714286	0.000004268592
5	1.500000000000	4.416671317377	3.738797638148	4.416666666667	0.000004650710
6	1.600000000000	4.785004954280	0.000000000000	4.785000000000	0.000004954280
7	1.700000000000	5.178240503774	0.000000000000	5.178235294118	0.000005209657
8	1.800000000000	5.595560990582	0.000000000000	5.595555555556	0.000005435027
9	1.900000000000	6.036321431054	0.000000000000	6.036315789474	0.000005641580
10	2.000000000000	6.500005836358	0.000000000000	6.500000000000	0.000005836358

Оценка P-P-P

3.000000000000 0.000000000000
3.202140081860 0.016950827231
3.433268361764 0.040064971569
3.690844728641 0.068386040590
3.972866340192 0.101419374094
4.277776380260 0.138890286407

Метод Адамса

k	x	y_k	f(x_k,y_k)	y_ист	eps_k
0	1.000000000000	3.000000000000	2.000000000000	3.000000000000	0.0000000000
1	1.1000000000	3.2190927906	2.4000000000	3.2190909091	0.0000018815
2	1.2000000000	3.4733363516	2.7628099174	3.4733333333	0.0000030183
3	1.3000000000	3.7592345249	3.1020202020	3.7592307692	0.0000037556
4	1.4000000000	4.0744401295	3.4256850376	4.0742857143	0.0001544152
5	1.5000000000	4.4167375016	3.7387976381	4.4166666667	0.0000708349
6	1.6000000000	4.7850759818	0.0000000000	4.7850000000	0.0000759818
7	1.7000000000	5.1781940255	0.0000000000	5.1782352941	0.0000412686

```
  8|  1.8000000000|  5.5954167145|  0.0000000000|  5.5955555556|0.0001388410|
  9|  1.9000000000|  6.0360483679|  0.0000000000|  6.0363157895|0.0002674216|
 10|  2.0000000000|  6.4996124526|  0.0000000000|  6.5000000000|0.0003875474|

Оценка P-P-P
3.0000000000    0.0000000000
3.2021400819    0.0169508272
3.4332683618    0.0400649716
3.6908447286    0.0683860406
3.9728842767    0.1014014376
4.2751629114    0.1415037553
```

Листинг программы

```
#include "libs.h"
//вариант 23

double ff(double x, double y, double z)
{
return z;
}
double gg(double x, double y, double z)
{
return (-x * z +  y + 3*x*x ) / (x*x);
}

vector <double> RRR(vector <double> y, vector <double> y2, vector <double> y_ist
{
for (int i = 0; i < 6; ++i)
{
R[i] = y[i] + (y[i] - y2[i]) / (pow(2, 4) - 1);
}

cout << "Оценка P-P-P" << endl;
for (int i = 0; i < 6; i++){
cout << R[i] << "\t" << fabs(R[i] - y_ist[i]) << endl;
```

```
}
cout << endl;
return R;
}

vector<double> Eiler(vector<double> x, vector<double> y, vector<double> z, vecto
{
vector<double> y2, R;
y2.resize(n, 0);
R.resize(6, 0);
y2[0] = 3;

for (int i = 1; i < n; ++i)
{
y[i] = y[i - 1] + h*ff(x[i - 1], y[i - 1], z[i - 1]);
z[i] = z[i - 1] + h*gg(x[i - 1], y[i - 1], z[i - 1]);
}
return y;
}

vector <double> RK(vector<double> y, vector<double> y_ist, vector<double> x, vec
{
vector<double> K, L;
K.resize(4, 0);
L.resize(4, 0);

for (int i = 1; i < n; ++i)
{
K[0] = h*ff(x0, y0, z0);
L[0] = h*gg(x0, y0, z0);
K[1] = h*ff(x0 + h / 2, y0 + K[0] / 2, z0 + L[0] / 2);
L[1] = h*gg(x0 + h / 2, y0 + K[0] / 2, z0 + L[0] / 2);
K[2] = h*ff(x0 + h / 2, y0 + K[1] / 2, z0 + L[1] / 2);
L[2] = h*gg(x0 + h / 2, y0 + K[1] / 2, z0 + L[1] / 2);
K[3] = h*ff(x0 + h, y0 + K[2], z0 + L[2]);
```

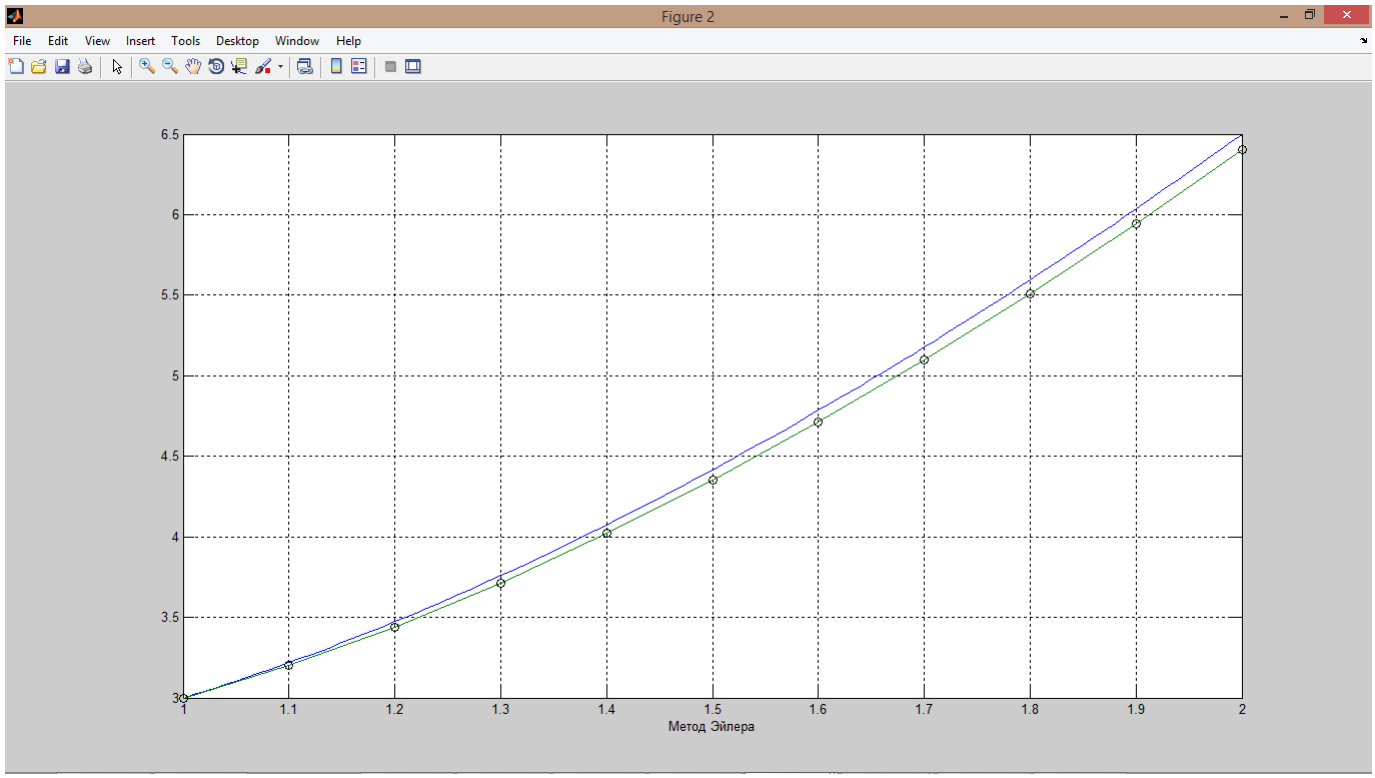
```
L[3] = h*gg(x0 + h, y0 + K[2], z0 + L[2]);

y0 = y0 + (K[0] + 2 * K[1] + 2 * K[2] + K[3]) / 6;
z0 = z0 + (L[0] + 2 * L[1] + 2 * L[2] + L[3]) / 6;
x0 = x0 + h;
y[i] = y0;
z[i] = z0;
}
return y;
}

vector <double> Adams(vector<double> y, vector<double> y_ist, vector<double> x, v
{
vector<double> K, L;
K.resize(4, 0);
L.resize(4, 0);

for (int i = 1; i < 4; ++i)
{
K[0] = h*ff(x0, y0, z0);
L[0] = h*gg(x0, y0, z0);
K[1] = h*ff(x0 + h / 2, y0 + K[0] / 2, z0 + L[0] / 2);
L[1] = h*gg(x0 + h / 2, y0 + K[0] / 2, z0 + L[0] / 2);
K[2] = h*ff(x0 + h / 2, y0 + K[1] / 2, z0 + L[1] / 2);
L[2] = h*gg(x0 + h / 2, y0 + K[1] / 2, z0 + L[1] / 2);
K[3] = h*ff(x0 + h, y0 + K[2], z0 + L[2]);
L[3] = h*gg(x0 + h, y0 + K[2], z0 + L[2]);

y0 = y0 + (K[0] + 2 * K[1] + 2 * K[2] + K[3]) / 6;
z0 = z0 + (L[0] + 2 * L[1] + 2 * L[2] + L[3]) / 6;
x0 = x0 + h;

y[i] = y0;
z[i] = z0;
}
```

```
for (int i = 4; i < n; ++i)
{

z[i] = z[i - 1] + (h / 24)*(55 * gg(x[i - 1], y[i - 1], z[i - 1]) - 59 * gg(x[i -
y[i] = y[i - 1] + (h / 24)*(55 * ff(x[i - 1], y[i - 1], z[i - 1]) - 59 * ff(x[i -

}

return y;
}

void PartI_main()
{
int n = 11;
vector<double> y, y_ist, x, z;
y.resize(n, 0);
y_ist.resize(n, 0);
x.resize(n, 0);
z.resize(n, 0);
vector<double> y2, R;
y2.resize(n, 0);
R.resize(6, 0);
y2[0] = 3;

double h = 0.1;
double z0 = 2;
double y0 = 3;
double x0 = 1;

x[0] = x0;
y[0] = y0;
y_ist[0] = y0;
z[0] = z0;
for (int i = 1; i < n; ++i)
```

```
{
x[i] = x[i - 1] + h;
y_ist[i] = x[i] * x[i] + x[i] + (1/x[i]);
}
// М. Эйлера
y = Eiler(x, y, z, y_ist, n, h);

cout << "Метод Эйлера" << endl;
cout << setw(3) << "i" << "|" << setw(4) << "x" << "|" << setw(14) << "y_k" << "
for (int i = 0; i < n; ++i)
{
cout << setw(3) << i << "|" << setw(4) << x[i] << "|" << setw(14) << y[i] << "|"
}
cout << endl;

for (int i = 1; i < 6; ++i)
{
y2[i] = y2[i - 1] + 2 * h*ff(x[i - 1], y2[i - 1], z[i - 1]);
z[i] = z[i - 1] + h*gg(x[i - 1], y2[i - 1], z[i - 1]);
}

y2 = Eiler(x, y2, z, y_ist, 6, 2*h);
R = RRR(y, y2, y_ist, R);

// М. Р-К
y = RK(y, y_ist, x, z, h, z0, y0, x0, n);

cout << "Метод Рунге-Кутта" << endl;
cout << setw(3) << "k" << "|" << setw(14) << "x" << "|" << setw(14) << "y_k" <<
for (int i = 0; i < n; ++i)
{
cout << setw(3) << i << "|" << setw(14) << x[i] << "|" << setw(14) << y[i] << "|

}
cout << endl;
```

```
y2 = RK(y2, y_ist, x, z, 2*h, z0, y0, x0, 6);
R = RRR(y, y2, y_ist, R);

y = Adams(y, y_ist, x, z, h, z0, y0, x0, n);

cout << "Метод Адамса" << endl;
cout << setw(3) << "k" << "|" << setw(14) << "x" << "|" << setw(14) << "y_k" <<
for (int i = 0; i < n; ++i)
{
cout << setw(3) << i << "|" << setw(14) << x[i] << "|" << setw(14) << y[i] << "|
}cout << endl;

y2 = Adams(y2, y_ist, x, z, 2 * h, z0, y0, x0, 6);
R = RRR(y, y2, y_ist, R);
}
```

Скриншоты



Задание

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. **Теоретическая часть**

Краевая задача для ОДУ второго порядка

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

с граничными условиями, заданными на концах отрезка [a, b]

$$\alpha' y(a)_\beta y'(a) = y_0, \delta y(b) + \gamma y'(b) = y_1$$

где $\alpha, \beta, \delta, gamma$ - такие числа, что $|\alpha + \beta| \neq 0, |\delta + gamma| \neq 0$

4.2.1 - 4.2.2 Метод стрельбы, конечно-разностный метод

Метод стрельбы

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

$$\Phi(\eta) = 0$$

где $\Phi(\eta) = y(b, y_0, \eta) - y_1$, η - некоторое значение тангенса угла наклона касательной к решению в точке x = a, $y(b, y_0, \eta)$ - решение пристрелочной задачи на правом конце

Данное уравнение явдяется алгебраическим и для его решения можно применить метод секущих и получить следующее соотношения для для определения η

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

Конечно-разностный метод решения краевой задачи

Рассмотрим двухточечную краевую задачу для линейного дифференциального уравнения второго порядка на отрезке [a, b]

$$y'' + p(x)y' + q(x)y = f(x), y(a) = y_0, y(b) = y_1$$

Введем разностную сетку на отрезке [a, b] $\Omega^k = \{x_k = x_0 + hk\}, k = \overline{0, N}, h = |b - a|/N$. Точки x_k - называются узлами разностной сетки, расстояния между узлами – шагом разностной сетки (h) , а совокупность значений какой либо величины заданных в узлах сетки называется сеточной функцией. Введем разностную аппроксимацию производных следующим образом:

$$y'_k = \frac{y_{k+1} - y_{k-1}}{2h} + O(h^2), y''_k = \frac{y_{k+1} - 2y_k + y_{k-1}}{h^2} + O(h^2)$$

Подставляя аппроксимации производных получим систему уравнений для нахождения y_k. Приводя подобные и учитывая, что при задании граничных условий первого рода два неизвестных уже фактически определены, получим систему линейных алгебраических уравнений с трехдиагональной матрицей коэффициентов.

Данные

Уравнение $x(2x+1)y'' + 2(x+1)y' - 2y = 0$
$y(1) = 3, y(3) - y'(3) = \frac{31}{9}$

Вывод программы

```
Метод стрельбы:
  j|            nu|           y_k|         y_ist|            Fi|
  0|           0.3|       4.87007|       4.33333|0.536734749107|
  1|0.400000000000|4.989229075570|4.333333333333|0.655895742237|
  2|-0.150428227399|4.333333333333|4.333333333333|0.000000000000|

  i|             x|           y_k|         y_ist|           Eps|
  0|1.000000000000|3.006641213379|3.000000000000|0.006641213379|
  1|1.200000000000|3.065642942899|3.033333333333|0.032309609566|
  2|1.400000000000|3.157381308973|3.114285714286|0.043095594687|
  3|1.600000000000|3.270947504018|3.225000000000|0.045947504018|
  4|1.800000000000|3.399794382948|3.355555555556|0.044238827392|
  5|2.000000000000|3.539754967333|3.500000000000|0.039754967333|
  6|2.200000000000|3.688051042250|3.654545454545|0.033505587705|
  7|2.400000000000|3.842759133450|3.816666666667|0.026092466783|
  8|2.600000000000|4.002505288359|3.984615384615|0.017889903744|
  9|2.800000000000|4.166281921461|4.157142857143|0.009139064318|
 10|3.000000000000|4.333333333333|4.333333333333|-0.000000000000|
Оценка P-P-P
3.025247881667  0.025247881667
3.089921885376  0.056588552042
3.180743327615  0.066457613330
3.289601908098  0.064601908098
3.411308111690  0.055752556135
3.542405298488  0.042405298488


Конечно-разностный метод:
  i|             x|           y_k|         y_ist|           Eps|
  0|1.000000000000|3.006641213379|3.000000000000|-0.006641213379|
  1|1.200000000000|3.031918072362|3.033333333333|0.001415260971|
  2|1.400000000000|3.111284273510|3.114285714286|0.003001440776|
  3|1.600000000000|3.220428265826|3.225000000000|0.004571734174|
  4|1.800000000000|3.349478958316|3.355555555556|0.006076597240|
  5|2.000000000000|3.492491363597|3.500000000000|0.007508636403|
  6|2.200000000000|3.645672110648|3.654545454545|0.008873343898|
  7|2.400000000000|3.806487235703|3.816666666667|0.010179430963|
  8|2.600000000000|3.973179786018|3.984615384615|0.011435598597|
  9|2.800000000000|4.144493304885|4.157142857143|0.012649552257|
 10|3.000000000000|4.319505519996|4.333333333333|0.013827813338|
Оценка P-P-P
3.025247881667  0.025247881667
2.937964866890  0.095368466444
2.941532351559  0.172753362726
```

```
2.982435700396  0.242564299604
3.046146971183  0.309408584373
3.124443298135  0.375556701865
```

Листинг программы

```
#include "libs.h"

double f(double x, double y, double z)
{
return z;
}
double g(double x, double y, double z)
{
return (-2 * (x + 1)*z + 2 * y) / (x*(2 * x + 1));
}

double p(double x)
{
return 2 * (x + 1) / (x*(2 * x + 1));
}

double q(double x)
{
return - 2 / (x*(2 * x + 1));
}

vector <double> RR(vector <double> y, vector <double> y2, vector <double> y_ist, vector <double
{
for (int i = 0; i < 6; ++i)
{
R[i] = y[i] + (y[i] - y2[i]) / (pow(2, 4) - 1);
}

cout << "Оценка P-P-P" << endl;
for (int i = 0; i < 6; i++){
cout << R[i] << "\t" << fabs(R[i] - y_ist[i]) << endl;
}
cout << endl;
return R;
}


vector<double> gauss(vector<vector<double>> a, vector<double> y, int n)
{
vector <double> x;
double max;
int k, index;
const double eps = 0.00001;  // точность
x.resize(n, 0);
k = 0;
```

```
while (k < n)
{
// Поиск строки с максимальным a[i][k]
max = abs(a[k][k]);
index = k;
for (int i = k + 1; i < n; i++)
{
if (abs(a[i][k]) > max)
{
max = abs(a[i][k]);
index = i;
}
}
// Перестановка строк
if (max < eps)
{
// нет ненулевых диагональных элементов
cout << "Решение получить невозможно из-за нулевого столбца ";
cout << index << " матрицы A" << endl;
//return 0;
}
for (int j = 0; j < n; j++)
{
double temp = a[k][j];
a[k][j] = a[index][j];
a[index][j] = temp;
}
double temp = y[k];
y[k] = y[index];
y[index] = temp;
// Нормализация уравнений
for (int i = k; i < n; i++)
{
double temp = a[i][k];
if (abs(temp) < eps) continue; // для нулевого коэффициента пропустить
for (int j = 0; j < n; j++)
a[i][j] = a[i][j] / temp;
y[i] = y[i] / temp;
if (i == k)  continue; // уравнение не вычитать само из себя
for (int j = 0; j < n; j++)
a[i][j] = a[i][j] - a[k][j];
y[i] = y[i] - y[k];
}
k++;
}
// обратная подстановка
for (k = n - 1; k >= 0; k--)
{
x[k] = y[k];
```

```
for (int i = 0; i < k; i++)
y[i] = y[i] - a[i][k] * x[k];
}
return x;
}


vector <double> RK_n(vector<double> y, vector<double> x, double nu, double h, double y0, int n)
{

double y1 = 0;
vector<double> K, L, Fi;
K.resize(4, 0);
L.resize(4, 0);
Fi.resize(n, 0);

for (int i = 0; i < n; ++i)
{
K[0] = h*f(x[i], y0, nu);
L[0] = h*g(x[i], y0, nu);
K[1] = h*f(x[i] + h / 2, y0 + K[0] / 2, nu + L[0] / 2);
L[1] = h*g(x[i] + h / 2, y0 + K[0] / 2, nu + L[0] / 2);
K[2] = h*f(x[i] + h / 2, y0 + K[1] / 2, nu + L[1] / 2);
L[2] = h*g(x[i] + h / 2, y0 + K[1] / 2, nu + L[1] / 2);
K[3] = h*f(x[i] + h, y0 + K[2], nu + L[2]);
L[3] = h*g(x[i] + h, y0 + K[2], nu + L[2]);

y0 = y0 + (K[0] + 2 * K[1] + 2 * K[2] + K[3]) / 6;
nu = nu + (L[0] + 2 * L[1] + 2 * L[2] + L[3]) / 6;
y[i] = y0;

}

return y;
}

vector<double> Shooting(vector<double> y, vector<double> y_ist, vector<double> x, vector<double
{
vector<double> Fi, y_RR;
Fi.resize(n, 0);
y_RR.resize(n, 0);

y_RR = RK_n(y, x, nu[0], h, y0, n);
y[0] = y_RR[n-1];
Fi[0] = fabs(y[0] - y_ist[n - 1]);

for (int i = 1; i < n; ++i)
{
y_RR = RK_n(y, x, nu[i], h, y0, n);
```

```
y[i] = y_RR[n-1];
Fi[i] = fabs(y[i] - y_ist[n - 1]);
nu[i + 1] = nu[i] - (((nu[i] - nu[i - 1]) / (y[i] - y[i - 1])) * (y[i] - y_ist[n - 1]));
if (Fi[i] < 0.0001 && m==1)
{
cout << "Метод стрельбы:" << endl;

cout << setw(3) << "j" << "|" << setw(14) << "nu" << "|" << setw(14) << "y_k" << "|" << setw(14
for (int j = 0; j <= i; ++j)
{
cout << setw(3) << j << "|" << setw(14) << nu[j] << "|" << setw(14) << y[j] << "|" << setw(14)
}
cout << endl;
cout << setw(3) << "i" << "|" << setw(14) << "x" << "|" << setw(14) << "y_k" << "|" << setw(14)
for (int i = 0; i <n; ++i)
{
cout << setw(3) << i << "|" << setw(14) << x[i] << "|" << setw(14) << y_RR[i] << "|" << setw(14
}
for (int i = 0; i < n; ++i)
y[i] = 0;
break;
}
}


return y_RR;
}


void PartII_main()
{

int n = 11;
int k = 10;
vector<double> y, y_ist, x, z, z2, nu, b, b2, y2, R;
vector <vector <double>> A, A2;

A.resize(k);
b.resize(k,0);
A2.resize(5);
b2.resize(5, 0);
for (int i = 0; i < k; ++i)
{
A[i].resize(k, 0);
}
for (int i = 0; i < 5; ++i)
{
A2[i].resize(k, 0);
}
```

```
y.resize(n, 0);
y_ist.resize(n, 0);
x.resize(n, 0);
z.resize(n, 0);
z2.resize(n, 0);
nu.resize(n, 0);
y2.resize(6, 0);
R.resize(6, 0);

double h = 0.2;
double z0 = 0;
double y0 = 3;
double x0 = 1;

x[0] = x0;
y[0] = y0;
y2[0] = y0;
y_ist[0] = y0;
nu[0] = 0.3;
nu[1] = 0.4;
for (int i = 1; i < n; ++i)
{
x[i] = x[i-1] + h;
y_ist[i] = 1/x[i] + 1+x[i];
}

y = Shooting(y, y_ist, x, nu, h, y0, n,1);
y2 = Shooting(y2, y_ist, x, nu, 2*h, y0, 6,0);
R = RR(y, y2, y_ist, R);
cout << endl;

A[0][0] = -2 + q(x[1])*h*h;
A[0][1] = p(x[1])*h/2 + 1;

A[k-1][k-2] = 1;
A[k-1][k-1] = - 0.8;

for (int i = 1; i < k-1 ; ++i)
{
A[i][i-1] = 1 - (p(x[i + 1])*h/2);
A[i][i] = -2 + q(x[i+1])*h*h;
A[i][i+1] = 1 + (p(x[i + 1])*h/2);
}

b[0] = y0*(p(x[1])*h/2 - 1);
b[k-1] = 31 * 0.2 / 9;

z = gauss(A, b, k);
```

```
for (int i = 0; i < k; ++i)
{
y[i + 1] = z[i];
}

cout << "Конечно-разностный метод:" << endl;
cout << setw(3) << "i" << "|" << setw(14) << "x" << "|" << setw(14) << "y_k" << "|" << setw(14)
for (int i = 0; i <n; ++i)
{
cout << setw(3) << i << "|" << setw(14) << x[i] << "|" << setw(14) << y[i] << "|" << setw(14) <
}

A2[0][0] = -2 + q(x[1]) * 4 * h * h;
A2[0][1] = p(x[1])*2*h / 2 + 1;

A2[4][3] = 1;
A2[4][4] = -0.8;

for (int i = 1; i < 4; ++i)
{
A2[i][i - 1] = 1 - (p(x[i + 1])*2*h / 2);
A2[i][i] = -2 + q(x[i + 1]) * 4*h * h;
A2[i][i + 1] = 1 + (p(x[i + 1])*2*h / 2);
}

b2[0] = y0*(p(x[1])*2*h / 2 - 1);
b2[4] = 31 * 0.2 / 9;

z2 = gauss(A2, b2, 5);
for (int i = 0; i < 5; ++i)
{
y2[i + 1] = z2[i];
}
R = RR(y, y2, y_ist, R);
}
```