Московский Авиационный Институт
(Научно-исследовательский университет)

Институт №8 "Информационные технологии и прикладная математика"

Лабораторные работы
по дисциплине:

Численные методы

Студент группы М8О-309Б-18: Рябыкин Алексей Сергеевич
Преподаватель: Сластушенский Юрий Викторович

Москва 2021

Лабораторная работа 1.1
LU-разложение матриц. Метод Гаусса.

Формулировка задания:
Реализовать алгоритм LU - разложения матриц (с выбором главного элемента) в виде программы. Используя разработанное программное обеспечение решить систему линейных алгебраических уравнения (СЛАУ).Для матрицы СЛАУ найти определитель и обратную матрицу.

Вариант 7:

$$\begin{cases} x_1 - 5 \cdot x_2 - 7 \cdot x_3 + x_4 = -75 \\ x_1 - 3 \cdot x_2 - 9 \cdot x_3 - 4 \cdot x_4 = -41 \\ -2 \cdot x_1 + 4 \cdot x_2 + 2 \cdot x_3 + x_4 = 18 \\ -9 \cdot x_1 + 9 \cdot x_2 + 5 \cdot x_3 + 3 \cdot x_4 = 29 \end{cases}$$

Метод решения:
LU разложение матрицы представляет собой разложение матрицы на произведение верхней и нижней треугольных матриц

$$A = LU,$$

где L - нижняя треугольная матрица, U - верхняя треугольная матрица. Реализовано разложение с применением логики метода Гаусса. LU разложение использовано для решения СЛАУ вида $Ax = b$ в два этапа. На первом этапе решается СЛАУ вида $Lz = b$. На втором этапе $Ux = z$. Второй этап эквивалентен обратному ходу метода Гаусса.

Среда разработки: Visual Studio ($C\#$)

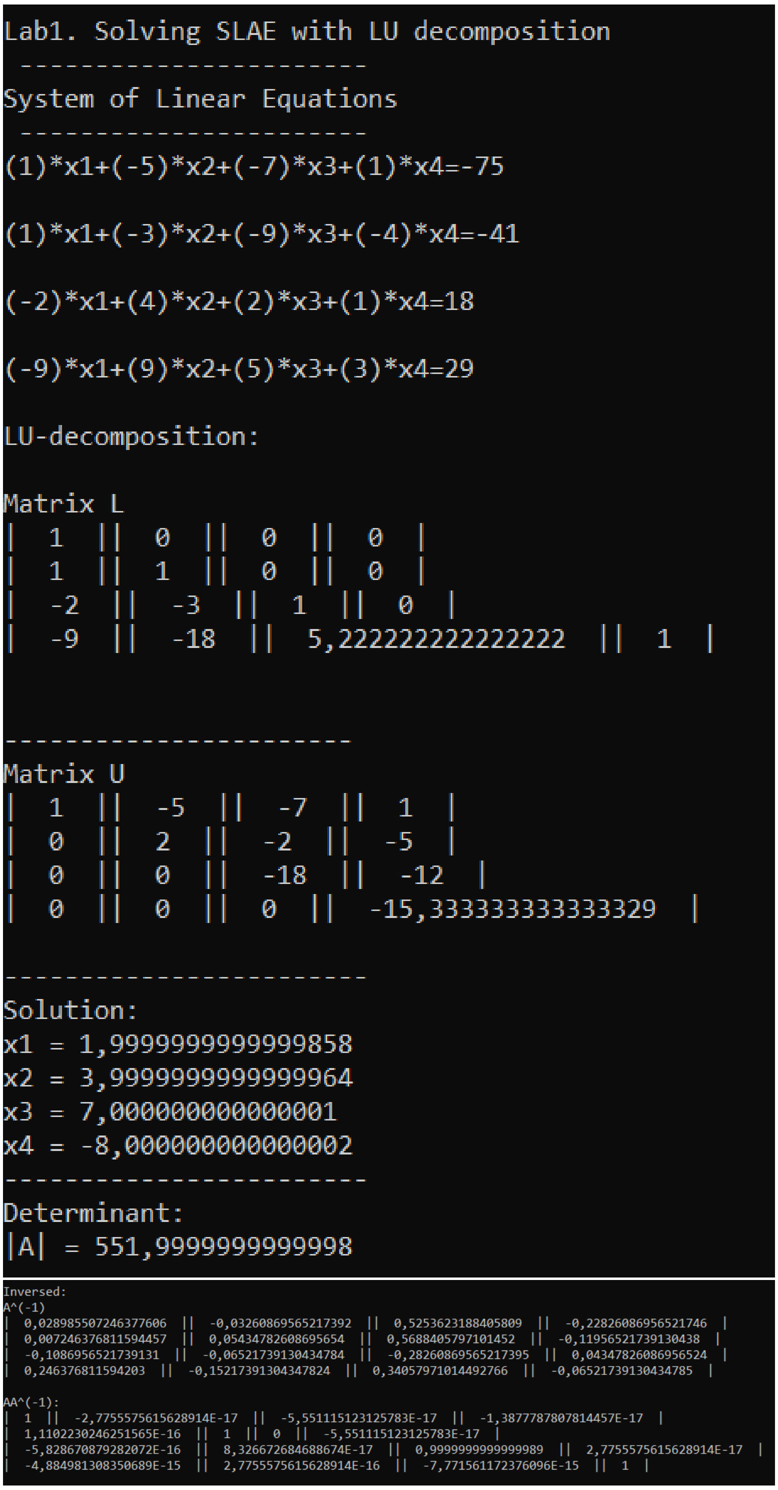
Листинг:

```
public List<object> LUDec(List<double> RP)
{
    Matrix A = new Matrix(_data);
    Matrix L = new Matrix();
    Matrix U = new Matrix();
    Matrix M = new Matrix();
    double sign_det = 1;
    List<Matrix> LU = new List<Matrix>();
    List<double> RP_changed = RP;
    double sum1 = 0, sum2 = 0;
    List<object> answer = new List<object>();
    U = A;
    L = EMatrix();
    for (int i = 0; i < _Dimension; i++)
    {
        M = EMatrix();
        for (int j = i + 1; j < _Dimension; j++)
        {
            M[j, i] = U[j, i] / U[i, i];
            for (int k = 0; k < _Dimension; k++)
                U[j, k] -= M[j, i] * U[i, k];
        }
        L *= M;
    }
```

```
    LU.Add(L);
    LU.Add(U);
    answer.Add(LU);
    answer.Add(RP_changed);
    answer.Add(sign_det);
    return answer;
}



public List<double> SOLVE(Matrix L, Matrix U, List<double> RP)
{
    List<double> x = new List<double>();
    List<double> z = new List<double>();
    double sum = 0;
    //Lz = b
    x.Add(0f);
    z.Add(RP[0]);
    for (int i = 1; i < _Dimension; ++i)
    {
        for (int j = 0; j < i; ++j)
            sum += L[i, j] * z[j];
        z.Add(RP[i]-sum);
        sum = 0;
        x.Add(0f);

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


public Matrix Inversion(Matrix L, Matrix U)
{
    List<double> v = new List<double>();
    for (int i = 0; i < _Dimension; i++)
        v.Add(0f);
    Matrix Inversed = Matrix.EMatrix();
    for (int i = 0; i < _Dimension; ++i)
    {
        v[i] = 1f;
        List<double> x = SOLVE(L, U, v);
```

```
        v[i] = 0;
        for (int j = 0; j < _Dimension; ++j)
        {
            Inversed[j,i] = x[j];
        }
    }
    return Inversed;
```

Пример работы:



```
Lab1. Solving SLAE with LU decomposition
 -----------------------
System of Linear Equations
 -----------------------
(1)*x1+(-5)*x2+(-7)*x3+(1)*x4=-75

(1)*x1+(-3)*x2+(-9)*x3+(-4)*x4=-41

(-2)*x1+(4)*x2+(2)*x3+(1)*x4=18

(-9)*x1+(9)*x2+(5)*x3+(3)*x4=29

LU-decomposition:

Matrix L
|  1  ||  0  ||  0  ||  0  |
|  1  ||  1  ||  0  ||  0  |
|  -2  ||  -3  ||  1  ||  0  |
|  -9  ||  -18  ||  5,22222222222222  ||  1  |


-----------------------
Matrix U
|  1  ||  -5  ||  -7  ||  1  |
|  0  ||  2  ||  -2  ||  -5  |
|  0  ||  0  ||  -18  ||  -12  |
|  0  ||  0  ||  0  ||  -15,3333333333333329  |

 -----------------------
Solution:
x1 = 1,9999999999999858
x2 = 3,9999999999999964
x3 = 7,000000000000001
x4 = -8,000000000000002
 -----------------------
Determinant:
|A| = 551,9999999999998

Inversed:
A^(-1)
|  0,028985507246377606  ||  -0,03260869565217392  ||  0,5253623188405809  ||  -0,22826086956521746  |
|  0,007246376811594457  ||  0,05434782608695654  ||  0,5688405797101452  ||  -0,11956521739130438  |
|  -0,1086956521739131  ||  -0,06521739130434784  ||  -0,28260869565217395  ||  0,04347826086956524  |
|  0,246376811594203  ||  -0,15217391304347824  ||  0,34057971014492766  ||  -0,06521739130434785  |

AA^(-1):
|  1  ||  -2,7755575615628914E-17  ||  -5,551115123125783E-17  ||  -1,3877787807814457E-17  |
|  1,1102230246251565E-16  ||  1  ||  0  ||  -5,551115123125783E-17  |
|  -5,828670879282072E-16  ||  8,326672684688674E-17  ||  0,9999999999999989  ||  2,7755575615628914E-17  |
|  -4,884981308350689E-15  ||  2,7755575615628914E-16  ||  -7,771561172376096E-15  ||  1  |
```

Лабораторная работа 1.2
Метод прогонки

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

Вариант 7:

$$
\begin{cases}
15 \cdot x_1 + 8 \cdot x_2 = 92 \\
2 \cdot x_1 - 15 \cdot x_2 + 4 \cdot x_3 = -84 \\
4 \cdot x_2 + 11 \cdot x_3 + 5 \cdot x_4 = -77 \\
-3 \cdot x_3 + 16 \cdot x_4 - 7 \cdot x_5 = 15 \\
3 \cdot x_4 + 8 \cdot x_5 = -11
\end{cases}
$$

Метод решения:
Метод прогонки - частный случай метода Гаусса для трехдиагональной матрицы. Решение осуществляется в два прохода. Прямой проход позволяет найти прогоночные коэффициенты, на обратном ходе на основе коэффициентов находится решение.

Среда разработки: Visual Studio ($C\#$)

Листинг:

```
static public List<double> TMA(int n, List<double> a, List<double> b,
                                         List<double> c, List<double> d)
{
    List<double> P = new List<double>();
    List<double> Q = new List<double>();

    //direct
    P.Add(-c[0] / b[0]);
    Q.Add(d[0] / b[0]);
    List<double> answer = new List<double>();
    answer.Add(0f);
    for (int i = 1; i < n; i++)
    {
        //if (i < n-1)
        P.Add(-c[i] / (a[i] * P[i-1] + b[i]));
        Q.Add((d[i] - a[i] * Q[i - 1]) / (b[i] + a[i]*P[i-1]));
        answer.Add(0f);
    }

    //back
    answer[n - 1] = Q[n - 1];
    for (int i = n - 2; i > -1; i--)
    {
        answer[i] = P[i] * answer[i + 1] + Q[i];
    }
    return answer;
}
```

Пример работы:

```
Lab 2. Tridiagonal Matrix Algorithm
 -----------------------
a1 = 0 a2 = 2 a3 = 4 a4 = -3 a5 = 3
b1 = 15 b2 = -15 b3 = 11 b4 = 16 b5 = 8
c1 = 8 c2 = 4 c3 = 5 c4 = -7 c5 = 0
d1 = 92 d2 = -84 d3 = -77 d4 = 15 d5 = -11
Answer:
 -----------------------
x1 = 4
x2 = 4
x3 = -8
x4 = -1
x5 = -1
```

Лабораторная работа 1.3
Итерационные методы решения СЛАУ

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

Вариант 7:

$$\begin{cases} 29 \cdot x_1 + 8 \cdot x_2 + 9 \cdot x_3 - 9 \cdot x_4 = 197 \\ -7 \cdot x_1 - 25 \cdot x_2 + 9 \cdot x_4 = -226 \\ x_1 + 6 \cdot x_2 + 16 \cdot x_3 - 2 \cdot x_4 = -95 \\ 7 \cdot x_1 + 4 \cdot x_2 - 2 \cdot x_3 + 17 \cdot x_4 = -58 \end{cases}$$

Метод решения:
Берем начальное приближение, вычисляем реккурентно очередное решение. После вычисления проверяется условия выхода. В методе Зейделя уже подсчитанные компоненты решения на текущей итерации используются для вычисления последующих компонент

Среда разработки: Visual Studio ($C\#$)

Листинг:

```
 public List<object> SIM(List<double> b)
{
    Matrix A = new Matrix(_data);
    List<object> answer = new List<object>();
    List<double> beta = new List<double>();
    Matrix alpha = new Matrix();

    for (int i = 0; i < _Dimension; i++)
    {
        for (int j = 0; j < _Dimension; j++)
        {
            if (i == j)
                alpha[i, j] = 0;
            else
                alpha[i, j] = -A[i, j] / A[i, i];
        }
        beta.Add(b[i] / A[i, i]);
    }
    List<double> x = beta;

    int k = 0;
    double varepsilon_k = 1;
    double varepsilon = 0.0000001;
    double coef = alpha.norm_C() / (1 - alpha.norm_C());
    while (varepsilon_k > varepsilon)
    {
        List<double> x_prev = x;
        x = this.Vecsum(beta, alpha.Multiply_vec(x));
        varepsilon_k = coef * (this.Vecnorm(this.Vecdiff(x, x_prev)));
```

```
            k++;
        }
        answer.Add(k);
        answer.Add(x);
        return answer;
    }


    public List<object> Seidel(List<double> b)
    {
        List<object> answer = new List<object>();
        Matrix A = new Matrix(_data);
        Matrix alpha = new Matrix();
        Matrix B = new Matrix();
        Matrix C = new Matrix();
        List<double> beta = new List<double>();
        for (int i = 0; i < _Dimension; i++)
        {
            for (int j = 0; j < _Dimension; j++)
            {
                if (i == j)
                    alpha[i, j] = 0;
                else
                    alpha[i, j] = -A[i, j] / A[i, i];
            }
            beta.Add(b[i] / A[i, i]);
        }
        double varepsilon_k = 1;
        double varepsilon = 0.0001;
        List<double> x_prev = new List<double>();
        List<double> x = new List<double>();
        for (int i = 0; i < beta.Count; i++)
            x.Add(beta[i]);
        int k = 0;
        double coef = alpha.norm_C() / (1 - alpha.norm_C());
        while (varepsilon_k > varepsilon)
        {
            for (int i = 0; i < x.Count; i++)
                x_prev.Add(x[i]);
          for (int i = 0; i < x.Count; i++)
            {
                x[i] = 0;
                for (int j = 0; j < x.Count; j++)
                    x[i] += alpha[i, j] * x[j];
                x[i] += beta[i];
            }

            varepsilon_k = coef * (this.Vecnorm(this.Vecdiff(x, x_prev)));
            x_prev.Clear();
            k++;
        }

        answer.Add(k);
```

```
        answer.Add(x);

        return answer;
    }
```

Пример работы:

```
Lab3. Simple iteration and Seidel Methods.
-----------------------
System of Linear Equations
 -----------------------
(29)*x1+(8)*x2+(9)*x3+(-9)*x4=197

(-7)*x1+(-25)*x2+(0)*x3+(9)*x4=-226

(1)*x1+(6)*x2+(16)*x3+(-2)*x4=-95

(-7)*x1+(4)*x2+(-2)*x3+(17)*x4=-58

Simple Iteration Method
 -----------------------
x1 = 7,00000000175517
x2 = 6,00000000137473
x3 = -8,9999999998760822
x4 = -3,0000000009134364

Iterations: 40

Seidel Method
 -----------------------
x1 = 7,000000414289881
x2 = 6,000000755686998
x3 = -9,000000006606239
x4 = -3,0000000079959595

Iterations: 10
```

Лабораторная работа 1.4
Метод вращений

Формулировка задания:

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

Вариант 7:

$$\begin{pmatrix} -6 & 6 & -8 \\ 6 & -4 & 9 \\ -8 & 9 & -2 \end{pmatrix}$$

Метод решения:

На каждой итерации выбирается наибольший по модулю внедиагональный элемент, происходит его обнуление при помощи ортогонального преобразования. По достижении состояния, в котором все внедиагональные элементы матрицы малы, на диагонали остаются собственные значения. Собственные вектора - столбцы матрицы, получением путем перемножения матрик ортогональных преобразования.

Среда разработки: Visual Studio ($C\#$)

Листинг:

```
public List<object> Rotation(int n)
        {
            List<object> answer = new List<object>();
            Matrix U_eigen = EMatrix(n);
            Matrix a = new Matrix(_data, n);
            Matrix a_new = a;
            int k = 0;
            int t;
            double sum = 0;
            for (int i = 0; i < a.Dimension; i++)
            {
                for (int j = 0; j < a.Dimension; j++)
                {
                    if (i != j)
                        sum += a[i, j] * a[i, j];
                }
            }
            double varepsilon = 0.001;
            int i_max = 0, j_max = 1;
            Matrix U_k = EMatrix(3);
            while (Math.Sqrt(sum) > varepsilon)
            {

                double max = a[0, 1];
                for (int i = 0; i < a.Dimension - 1; i++)
                    for (int j = i + 1; j < a.Dimension; j++)
                    {
                        if (i != j)
                            if (Math.Abs(a[i, j]) > max)
```

```
                    {
                        i_max = i;
                        j_max = j;
                        max = Math.Abs(a[i, j]);
                    }
                }
            double phi;
            double check = 2 * a[i_max, j_max] / (a[i_max, i_max]
                                                  - a[j_max, j_max]);
            check = Math.Atan(check) / 2;
            if (a[i_max, i_max] == a[j_max, j_max]) phi = Math.PI / 4;
            else phi = check;

            U_k[i_max, j_max] = -Math.Sin(phi);
            U_k[j_max, i_max] = Math.Sin(phi);
            U_k[i_max, i_max] = Math.Cos(phi);
            U_k[j_max, j_max] = Math.Cos(phi);
            max = a[0, 1];
            Matrix tmp = EMatrix(3);
            tmp *= U_k;
            Matrix tmp_1 = U_k.Tranpose();
            a_new = tmp_1 * a;
            a_new = a_new * tmp;
            a = a_new;
            sum = 0;
            for (int i = 0; i < a.Dimension; i++)
            {
                for (int j = 0; j < a.Dimension; j++)
                {
                    if (i > j)
                        sum += a[i, j] * a[i, j];
                }
            }
            k++;
            U_eigen = U_eigen * U_k;
            U_k = EMatrix(3);
            i_max = 0;
            j_max = 1;
        }
        answer.Add(a);
        answer.Add(k);
        answer.Add(U_eigen);
        return answer;

    }
```

Пример работы:

```
Lab4 Rotation Method
----------------------
Matrix:

|  -6  ||  6  ||  -8  |
|  6  ||  -4  ||  9  |
|  -8  ||  9  ||  -2  |


----------------------
Eigen Values

lambda1 = 0,7706967538253657
lambda2 = -19,3441422755157
lambda3 = 6,573445521690332

Iterations: 6
```

```
Eigen vectors:

|  0,16857192233485419  ||  -0,9526252517568733  ||  -0,2531573358911133  |
|  0,7901567125857807  ||  -0,022941230050061948  ||  0,6124753623774777  |
|  -0,5892672369612657  ||  -0,30328011751340395  ||  0,7488559900041479  |

Check: -1,1102230246251565E-16
Check: -3,3306690738754696E-16
Check: -8,326672684688674E-17
--------------------------------------------------
```

Лабораторная работа 1.5
QR-алгоритм

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

Вариант 7:

$$\begin{pmatrix} 9 & 0 & 2 \\ -6 & 4 & 4 \\ -2 & -7 & 5 \end{pmatrix}$$

Метод решения:
QR разложение матрицы представляет собой разложение матрицы на произведение ортогональной и верхней треугольной матриц

$$A = QR$$

Для этого используется матрица Хаусхолдера, позволяющая обращать в нуль поддиагональные элементы столбца матрицы. Она имеет вид:

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

где ν - произвольный ненулевой вектор-столбец, E - единичная матрица, $\nu\nu^\intercal$ - квадратная матрица того же размера. При поиске собственных значений QR разложение происходит на каждой итерации. Каждая итерация двухэтапна: разложение матрицы на Q и R и перемножение их в обратном порядке.

Среда разработки: Visual Studio (*C#*)

Листинг:

```
public Matrix HaushodlerMatrix(int index)
        {
            Matrix E = EMatrix(3);
            Matrix H1 = new Matrix(_data,3);
            List<double> x1 = new List<double>();
            List<double> b = new List<double>();
            for (int i = 0; i < H1.Dimension; i++)
            {
                b.Add(H1[i,index]);
            }
            double norm = Vecnorm(b);
            for (int i = 0; i < H1.Dimension; i++)
            {
                if (i == index)
                    x1.Add(H1[i, index] + Math.Sign(H1[i, index]) * norm);
                else if (i < index)
                    x1.Add(0);
                else
```

```
            x1.Add(H1[i,index]);
        }
        Matrix v1 = new Matrix(3);
        for (int i = 0; i < v1.Dimension; i++)
            v1[i, 0] = x1[i];
        Matrix tmp = EMatrix(3);
        tmp = tmp * v1;
        Matrix v1_T = tmp.Tranpose();
        Matrix tmp1 = v1 * v1_T;
        Matrix tmp2 = v1_T * v1;
        Matrix x_tmp = new Matrix(3);
        x_tmp = tmp1 * (1/(tmp2[0, 0]));
        Matrix x = x_tmp * 2;
        H1 = E - x;

        return H1;
    }
    public List<object> QR()
    {
        List<object> answer = new List<object>();
        Matrix A = new Matrix(_data,3);
        List<Matrix> H = new List<Matrix>();
        for (int i = 0; i < _Dimension; ++i)
        {
            Matrix H0 = A.HaushodlerMatrix(i);
            A = H0 * A;
            H.Add(H0);
        }
        for (int i = 0; i < H.Count - 1; i++)
        {
            H[i + 1] = H[i] * H[i + 1]; //Q
        }
        Matrix A0 = EMatrix(3);
        A0 = A * EMatrix(3);
        double varepsilon = 0.000001;

        answer.Add(A); //r
        answer.Add(H[H.Count - 1]); //q
        return answer;
    }
    List<object> Eigen(Matrix A, int index)
    {
        List<object> answer = new List<object>();
        double varepsilon = 0.01;
        double sum = 0, sum1 = 0;
        List<object> res = new List<object>();
        Matrix A_i = EMatrix(3);
        A_i = A_i * A;
        bool flag = true;
        while (flag)
        {
            Matrix Q = (Matrix)A_i.QR()[1];
            Matrix R = (Matrix)A_i.QR()[0];
```

```
        A_i = R * Q;
        Matrix a = EMatrix(3);
        a = A_i * a;
        for (int i = 1; index + i < Dimension; i++)
        {
            sum += a[index + i, index] * a[index + i, index];
        }

        for (int i = 2; index + i < Dimension; i++)
        {
            sum1 += a[index + i, index] * a[index + i, index];
        }
        if (Math.Sqrt(sum1) <= varepsilon &
                        finish_iter_complex(A_i, index, varepsilon))
        {

            res.Add(get_roots(A_i, index));
            res.Add(true);
            res.Add(A_i);
            flag = false;
        }
        else if (Math.Sqrt(sum1) <= varepsilon )
        {

            res.Add(a[index, index]);
            res.Add(false);
            res.Add(A_i);
            flag = false;
        }
        sum = 0;
        sum1 = 0;

    }
    return res;
}
```

Пример работы:

```
Lab5 QR decomposition
---------------------
Matrix:

Matrix
|  9  ||  0  ||  2  |
|  -6  ||  4  ||  4  |
|  -2  ||  -7  ||  5  |

Matrix R
|  -11  ||  0,909090909090909  ||  1,4545454545454546  |
|  7,360160314521759E-16  ||  -8,01077993012343  ||  2,5134989916880377  |
|  9,134316322035658E-16  ||  0,030961752856441205  ||  -6,047034011765138  |

---------------------

Matrix Q
|  -0,8181818181818181  ||  -0,09504174601186763  ||  -0,5670498910252164  |
|  0,5454545454545454  ||  -0,4401837971798905  ||  -0,7132443925775152  |
|  0,18181818181818182  ||  0,8928635344862669  ||  -0,41199133188092885  |

---------------------

Q * R
|  9  ||  1,2212453270876722E-15  ||  1,9999999999999996  |
|  -6,000000000000001  ||  4,0000000000000036  ||  4  |
|  -1,9999999999999998  ||  -7,000000000000002  ||  5  |

---------------------
Eigen Values

lambda1 = 10,027274695913134
lambda2 = 2,545511283827238 + i * (-5,9029537458846155)
lambda3 = 2,545511283827238 - i * (-5,9029537458846155)
```

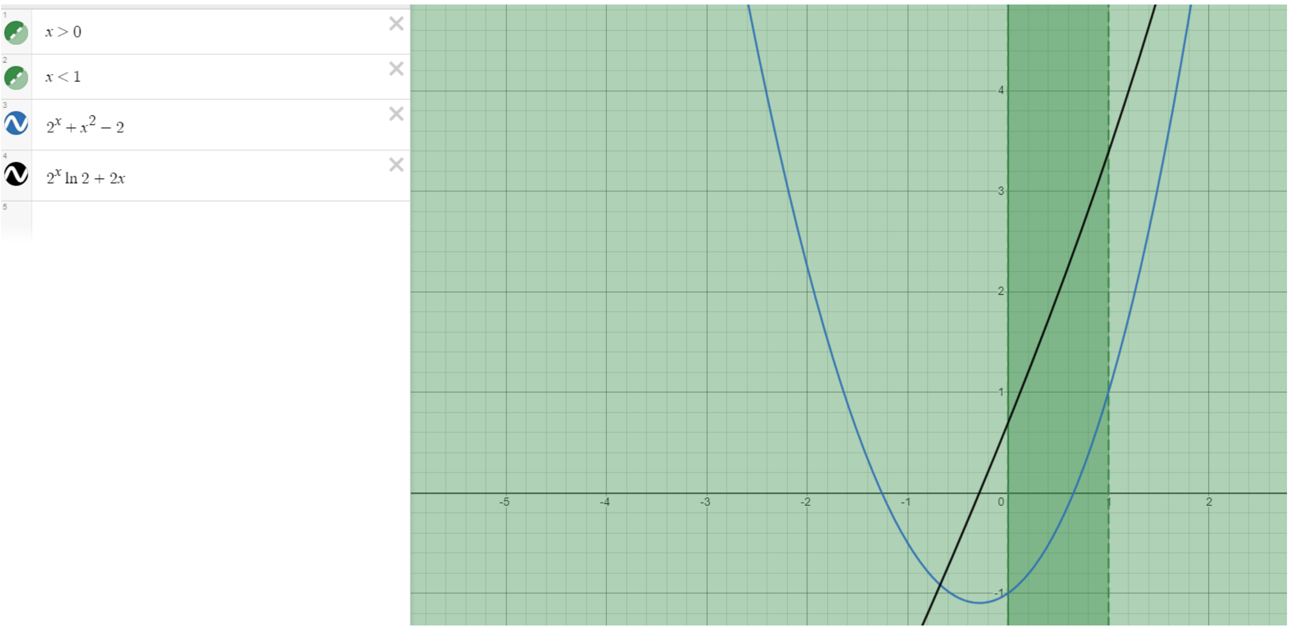
Лабораторная работа 2.1
Решение нелинейных уравнений

Формулировка задания:

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

Вариант 7:

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



Метод решения:

Метод простых итераций задается реккурентной формулой в цикле с критерием окончания.

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

. Метод НЬютона задается

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

Среда разработки: Visual Studio ($C\#$)

Листинг:

```
static public List<double> FixedPointIteration()
{
    double a = 0, b = 1, x_prev = (b - a) / 2 + a;
    double varepsilon = 0.00000000000002;
    int count = 0;
    double max = Math.Abs(Derivation(a));
    double tmp;
    double x = a + 0.0001;
    List<double> answer = new List<double>();
```

```
    while (x <= b)
    {
        tmp = Math.Abs(Derivation(x));
        if (tmp > max)
            max = tmp;
        x += 0.0001;
    }
    x = Phi(x_prev, max);
    double q = getq(a,b, max);
    Console.WriteLine(q + "\n");
    while (q / (1 - q) * Math.Abs(x - x_prev) > varepsilon)
    {
        count++;
        x_prev = x;
        x = Phi(x, max);
    }
    answer.Add(x);
    answer.Add(count);
    answer.Add(f(x));
    return answer;
}


static public List<double> Newton()
{
    List<double> answer = new List<double>();
    double a = 0, b = 1, x_prev = 0;
    double varepsilon = 0.00002;
    int count = 0;
    double x = 1;
    while (Math.Abs(x - x_prev) > varepsilon)
    {
        count++;
        x_prev = x;
        x -= f(x) / Derivation(x);
    }
    answer.Add(x);
    answer.Add(count);
    answer.Add(f(x));
    return answer;
}


 static public double Derivation(double x) =>
    Math.Pow(2,x) * Math.Log(2) + 2*x;
static public double Phi(double x, double max) =>
   x - Math.Sign(Derivation(x)) / max * f(x);

static public double dphi(double x, double max) =>
   Math.Sign(Derivation(x)) / max;
```

Пример работы:

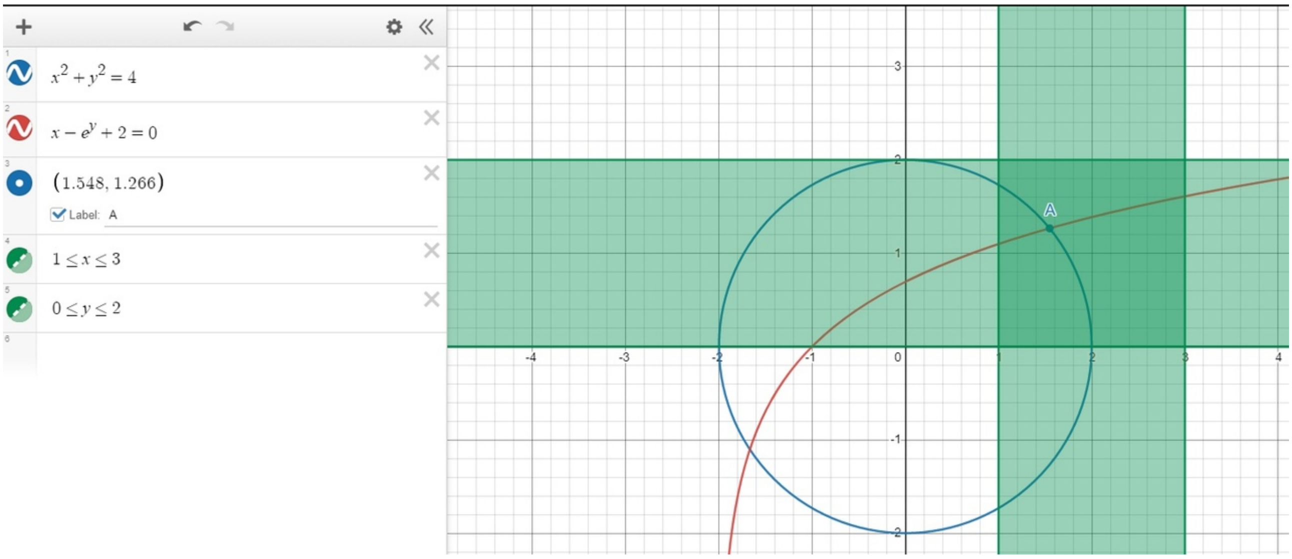
```
Lab1: Fixed Point Iteration Method
============================================
Root x = 0,6534825247841644
Function's value in root = -2,1538326677728037E-14
Iterations:24
============================================

Lab1: Newton Method
============================================
Root x = 0,6534825247849705
Function's value in root = 1,9109158699848194E-12
Iterations:4
============================================
```

Лабораторная работа 2.2
Решение систем нелинейных уравнений

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

Вариант 7:

$$a = 2, \qquad \begin{cases} x_1^2 + x_2^2 - a^2 = 0 \\ x_1 - e^{x_2} + a = 0 \end{cases}$$



Метод решения:
метод простых итераций задается реккурентной формулой в цикле с критерием окончания.

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

где x - вектор, φ функция вектора. Метод НЬютона задается

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

Среда разработки: Visual Studio ($C\#$)

Листинг:

```
static public Matrix Jacobi(double x1, double x2)
{
    Matrix answer = new Matrix(2);
    answer[0, 0] = der11(x1, x2);
    answer[0, 1] = der12(x1, x2);
    answer[1, 0] = der21(x1, x2);
    answer[1, 1] = der22(x1, x2);
    double det = answer[1, 1] * answer[0, 0] -
        answer[0, 1] * answer[1, 0];
```

```
    return answer;
}
static public Matrix phi(Matrix x)
{
    Matrix answer = new Matrix(2);
    answer[0, 0] = Math.Sqrt(4 - x[1, 0] * x[1,0]);
    answer[1, 0] = Math.Log(x[0, 0] + 2);
    return answer;
}
static public List<double> delta_x(double x1, double x2)
{
    List<double> answer = new List<double>();
    Matrix A = Jacobi(x1, x2);
    List<double> RP = new List<double>() { -f1(x1, x2), -f2(x1, x2) };
    List<object> solution = A.Seidel(RP);
    answer = (List<double>)solution[1];
    return answer;
}


 static public List<object> Newton_forSystem()
{
    List<object> answer = new List<object>();
    List<double> xprev = new List<double>();
    xprev.Add(2);
    xprev.Add(1);
    double varepsilon = 0.001;
    double q = 0.1;
    int count = 0;
    double coef = q / (1 - q);
    Matrix tmp = new Matrix();
    List<double> delta = delta_x(xprev[0], xprev[1]);
    List<double> x = new List<double>();
    for (int i = 0; i < xprev.Count; i++)
        x.Add(xprev[i] + delta[i]);
    List<double> diff = new List<double>();
    List<double> tmp1 = delta_x(xprev[0], xprev[1]);

    for (int i = 0; i < x.Count; i++)
    {
        diff.Add(x[i] - xprev[i]);
    }
    while ( tmp.Vecnorm(tmp1) > varepsilon)
    {
        for (int i = 0; i < x.Count; i++)
            xprev[i] = x[i];
        tmp1 = delta_x(xprev[0], xprev[1]);
        for (int i = 0; i < x.Count; i++)
            x[i] += tmp1[i];
        count++;
        diff.Clear();
        for (int i = 0; i < x.Count; i++)
        {
```

```
                diff.Add(x[i] - xprev[i]);
            }
        }
        answer.Add(x);
        answer.Add(count);
        return answer;
    }
```

Пример работы:

```
Lab2: Fixed Point Iteration Method for system
==========================================
Root x1 = 1,5479929240201846
Root x2 = 1,2663804437385642
Iterations: 16
==========================================

Lab2: Newton Method for system
==========================================
Root x1 = 1,5479921048495415
Root x2 = 1,2663818392051653
Iterations: 3
==========================================
```

Лабораторная работа 3.1
Полиномиальная интерполяция

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

Вариант 7:

$$y = \sqrt{x}, \quad a)X_i = 0, 0.7, 3.4, 5.1; \quad b)X_i = 0, 1.7, 4.0, 5.1; \qquad X^* = 3.0$$

Метод решения:
Многочлен Ньютона рассчитывается исходя из формулы:

$$P_n(x) = f(x_0)+(x-x_0)f(x_1,x_0)+(x-x_0)(x-x_1)f(x_0,x_1,x_2)+\ldots+(x-x_0)(x-x_1)\ldots(x-x_n)f(x_0,x_1,\ldots,x_n)$$

Многочлен Лагранжа:

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

Среда разработки: Visual Studio ($C\#$)

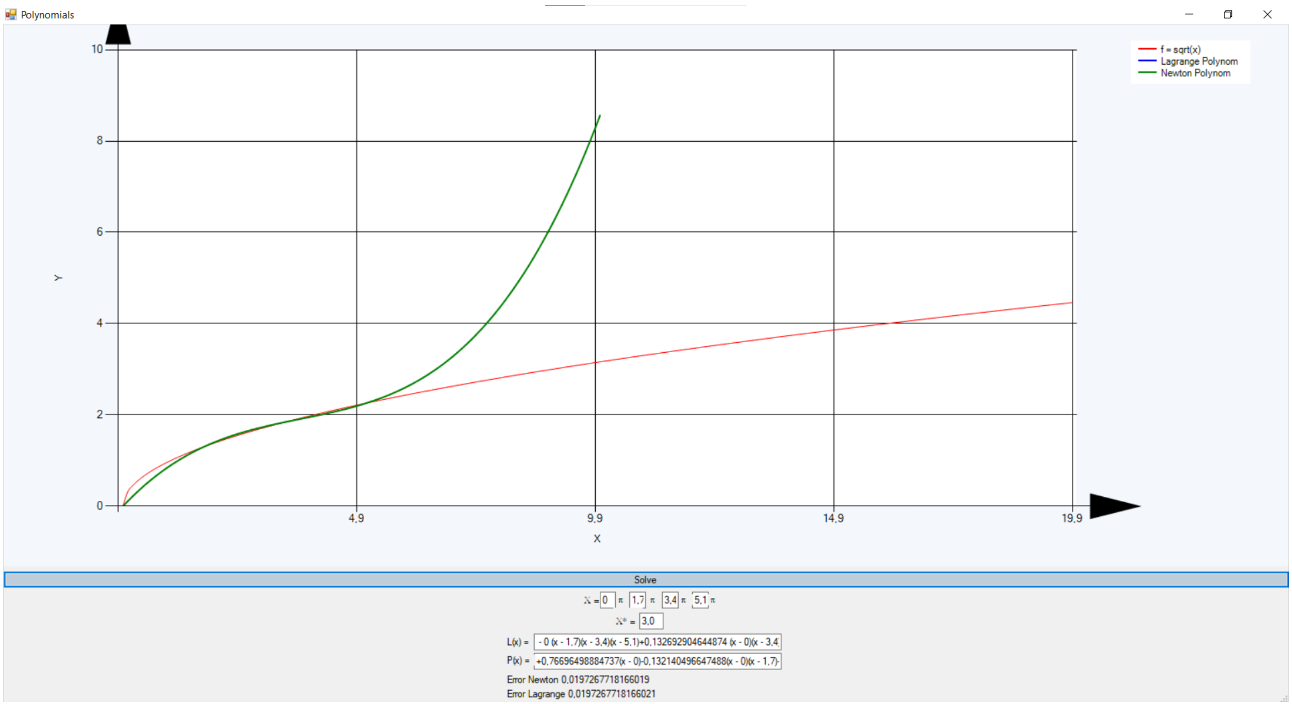
Листинг:

```
public List<object> Lagrange_Polynom(List<double> x, double xfix)
{
    List<object> answer = new List<object>();
    double l = 0d;
    double tmp;
    string str = "";
    string tmp_string;
    double coef;
    for (int i = 0; i < x.Count; i++)
    {
        tmp = 1;
        coef = 1;
        tmp_string = " ";
        for (int j = 0; j < x.Count; j++)
        {
            if (i != j)
            {
                coef /= (x[i] - x[j]);
                tmp *= (xfix - x[j]);
                tmp_string += ("(x - " + Convert.ToString(x[j]) + ")");
            }
        }
        coef *= Math.Sqrt(x[i]);
        if (Math.Sign(coef) == 1)
            str += (Convert.ToString(coef) + tmp_string + "+");
        else
        {
            if (str.Length > 1)
                str.Remove(str.Length - 2, 1);
```

```
            str += (" - " + (Convert.ToString(-coef) + tmp_string + "+"));

        }
        l += tmp * coef;
    }
    str.Remove(str.Length - 1, 1);
    answer.Add(str);
    answer.Add(l);
    string error = Convert.ToString(Math.Abs(Math.Sqrt(xfix) - l));
    answer.Add(error);
    return answer;
}



public List<object> Newton(List<double> x, double xfix)
{
    List<object> answer = new List<object>();
    double p = Math.Sqrt(x[0]);
    double l = 0d;
    double tmp;
    string str = "";
    string tmp_string;
    double coef;
    double tmp2;
    for (int i = 0; i < x.Count-1; i++)
    {
        tmp_string = "";
        tmp = 1;
        List<double> tmp_list = new List<double>();
        for (int j = 0; j <= i; j++)
        {
            tmp_string += ("(x - " + x[j].ToString() + ")");
            tmp *= xfix - x[j];
            tmp_list.Add(x[j]);
        }
        tmp_list.Add(x[i + 1]);
        tmp2 =  (tmp_list.Count > 1) ? f(tmp_list) :
                                        Math.Sqrt(tmp_list[0]);
        p += tmp * tmp2;
        if (Math.Sign(tmp2) == 1)
        {
            str += "+" + Convert.ToString(tmp2) + tmp_string;
        }
        else
            str += "-" + Convert.ToString(-tmp2) + tmp_string;
    }
    answer.Add(str);
    answer.Add(p);
    answer.Add(Math.Abs(p-Math.Sqrt(xfix)));
    return answer;
```

```
}
```

Пример работы:



Лабораторная работа 3.2
Сплайн-интерполяция

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

Вариант 7:

$$X^* = 3.0$$

i	0	1	2	3	4
x_i	0.0	1.7	3.4	5.1	6.8
f_i	0.0	1.3038	1.8439	2.2583	2.6077

Метод решения:
Для построения кубического сплайна необходимо построить n многочленов третьей степени, то есть определить $4n$ неизвестных a_i, b_i, c_i, d_i. Эти коэффициенты ищутся из условий в узлах сетки.

$S(x_{i-1}) = a_i = a_{i-1} + b_{i-1}(x_{i-1} - x_{i-2}) + c_{i-1}(x_{i-1} - x_{i-2})^2 + d_{i-1}(x_{i-1} - x_{i-2})^3 = f_{i-1}$
$S'(x_{i-1}) = b_i = b_{i-1} + 2c_{i-1}(x_{i-1} - x_{i-2}) + 3d_{i-1}(x_{i-1} - x_{i-2})^2$
$S''(x_{i-1}) = 2c_i = 2c_{i-1} + 6d_{i-1}(x_{i-1} - x_{i-2})^2, \qquad i = 2, 3, \ldots, n$
$S(x_0) = a_1 = f_0$
$S''(x_0) = c_1 = 0$
$S(x_n) = a_n + b_n(x_n - x_{n-1}) + c_n(x_n - x_{n-1})^2 + d_n(x_n - x_{n-1})^3 = f_n$
$S''(x_n) = c_n + 3d_n(x_n - x_{n-1}) = 0$

Среда разработки: Visual Studio ($C\#$)

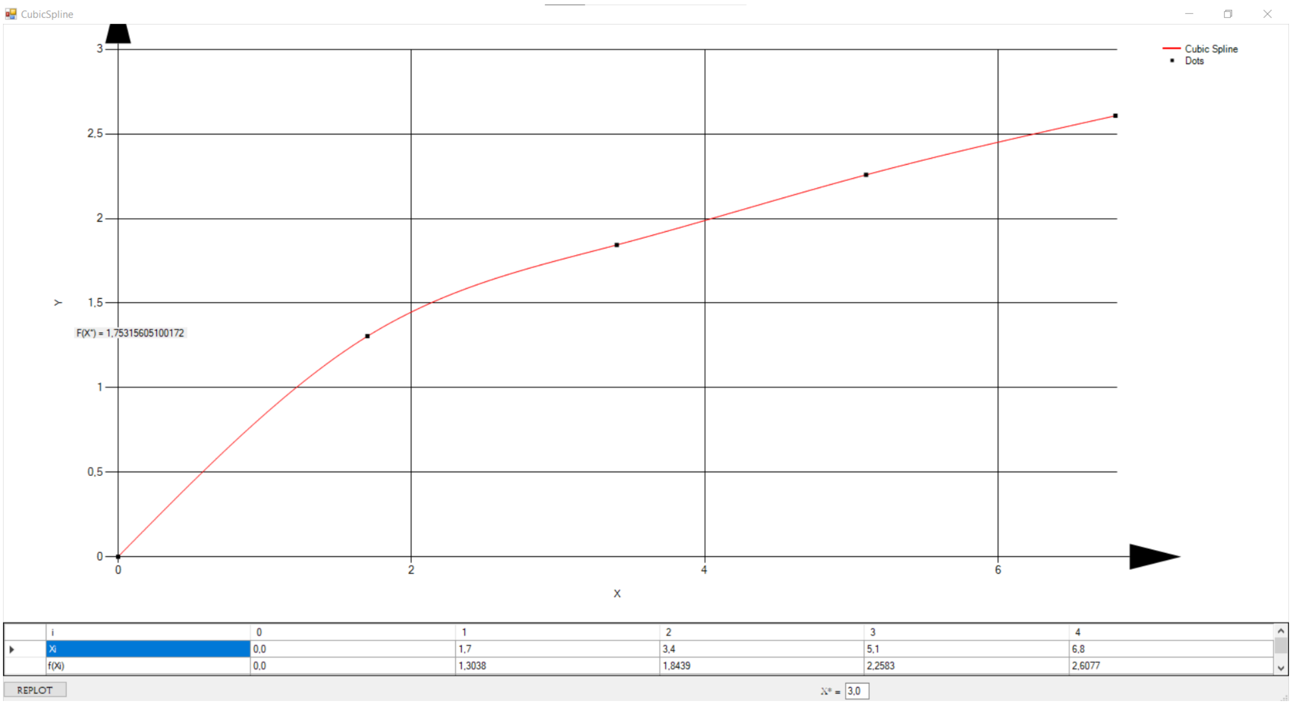
Листинг:

```
public List<object> Spline()
{
    List<object> answer = new List<object>();
    double xfix = Convert.ToDouble(textBox5.Text);
    List<double> x = new List<double>();
    List<double> y = new List<double>();

    for (int i = 0; i < dataGridView1.Rows.Count - 1; i++)
    {
        for (int j = 1; j < dataGridView1.Columns.Count; j++)
        {
            if (i == 0)
                x.Add(Convert.ToDouble(dataGridView1[j, i].Value));
            else
                y.Add(Convert.ToDouble(dataGridView1[j, i].Value));
        }
    }

    for (int i = 0; i < x.Count; i++)
    {
```

```
    chart1.Series["Series2"].Points.AddXY(x[i], y[i]);
}
List<double> a_tma = new List<double>();
a_tma.Add(0); a_tma.Add(h(x, 2)); a_tma.Add(h(x, 3));
List<double> b_tma = new List<double>();
b_tma.Add(2 * (h(x, 1) + h(x, 2))); b_tma.Add(2 * (h(x, 2) +
                                                   h(x, 3)));
b_tma.Add(2 * (h(x, 3) + h(x, 4)));
List<double> d_tma = new List<double>();
d_tma.Add(3 * ((y[2] - y[1]) / h(x, 2) - (y[1] - y[0]) / h(x, 1)));
d_tma.Add(3 * ((y[3] - y[2]) / h(x, 3) - (y[2] - y[1]) / h(x, 2)));
d_tma.Add(3 * ((y[4] - y[3]) / h(x, 4) - (y[3] - y[2]) / h(x, 3)));
List<double> c_tma = new List<double>();
c_tma.Add(h(x,2)); c_tma.Add(h(x, 3)); c_tma.Add(0);
List<double> c = new List<double>();
c.Add(0);
List<double> tmp = TMA(3, a_tma, b_tma, c_tma, d_tma);
for (int i = 0; i < tmp.Count; i++)
    c.Add(tmp[i]);
List<double> a = new List<double>();
for (int i = 0; i < y.Count-1; i++)
{
    a.Add(y[i]);
}
List<double> b = new List<double>();
for (int i = 0; i < y.Count - 2; i++)
    b.Add((y[i + 1] - y[i]) / h(x, i+1) - h(x, i+1) * (c[i + 1] +
                                                  2 * c[i]) / 3);
b.Add((y[4] - y[3]) / h(x, 4) - h(x, 4) * c[3] * 2 / 3);
List<double> d = new List<double>();
for (int i = 0; i < y.Count - 2; i++)
    d.Add((c[i+1] - c[i]) / (3 * h(x, i+1)));
d.Add(-c[3] / (3 * h(x, 4)));
double xprev, yprev, xtmp, ytmp;
List<double> xres = new List<double>(),
    yres = new List<double>();
for (int i = 0; i < a.Count; i++)
{
    xres.Add(x[i]);
    yres.Add(y[i]);
    xtmp = x[i] + 0.01;
    ytmp = s(xtmp, x[i], a[i], b[i], c[i], d[i]);
    xres.Add(xtmp);
    yres.Add(ytmp);
    while (xtmp < x[i+1])
    {
        xtmp = xtmp + 0.01;
        ytmp = s(xtmp, x[i], a[i], b[i], c[i], d[i]);
        xres.Add(xtmp);
        yres.Add(ytmp);

    }
}
```

```
            xres.Add(x[a.Count]);
            yres.Add(y[a.Count]);
            answer.Add(xres);
            answer.Add(yres);
            answer.Add(s(xfix, x[1], a[1], b[1], c[1], d[1]));
            return answer;
        }
```

Пример работы:



Лабораторная работа 3.3
Метод наименьших квадратов

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

Вариант 7:

i	0	1	2	3	4	5
x_i	0.0	0.2	0.4	0.6	0.8	1.0
y_i	1.0	1.0032	1.0512	1.2592	1.8192	3.0

Метод решения:
Аппроксимация многочлена производится с помощью поиска коэффициентов из условия минимума квадратичного отклонения многочлена от таблично-заданной функции

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

Необходимые условия экстремума, записанные в виде:

$$\sum_{i=0}^{n} a_i \sum_{j=0}^{N} x_j^{k+1} = \sum_{j=0}^{N} y_j x_j^k, \qquad k = 0, 1, \ldots, n$$

называют нормальной системой метода наименьших квадратов, является СЛАУ, решая которую, можно построить многочлен, аппроксимирующий таблично заданную функцию и минимизирующий квадратичное отклонение.

Среда разработки: Visual Studio ($C\#$)

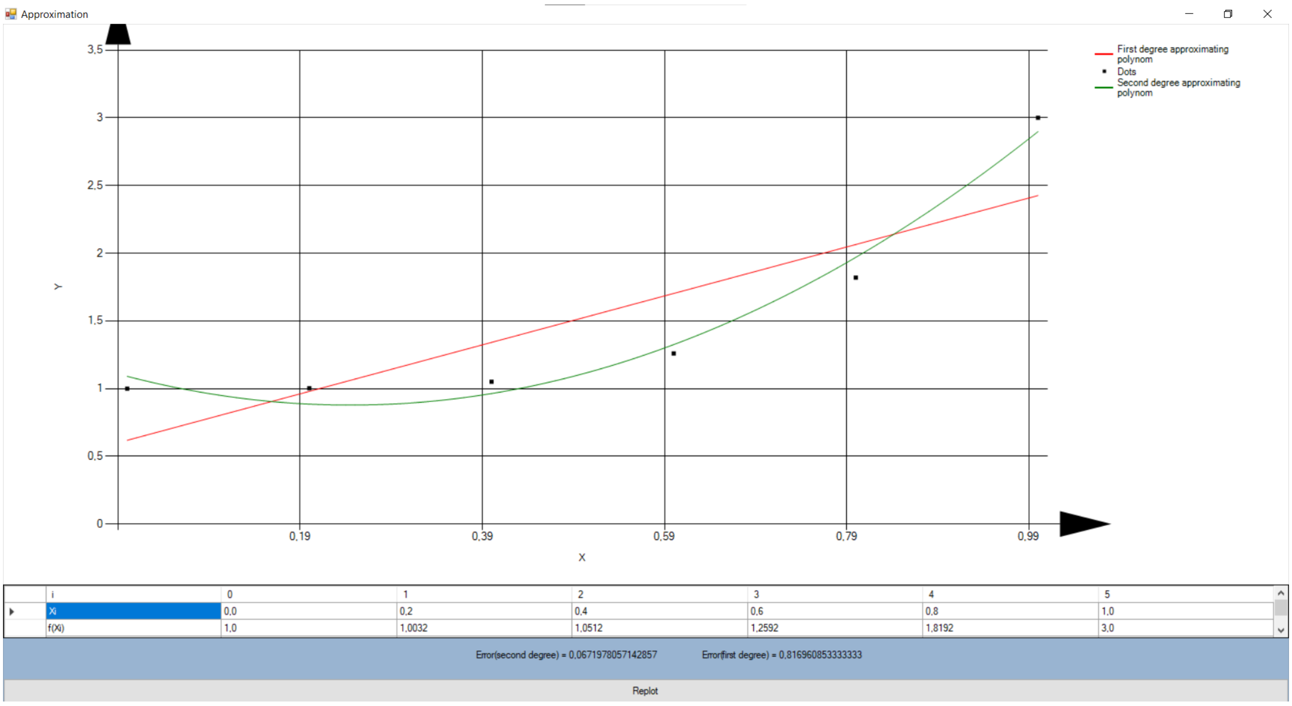
Листинг:

```
public List<object> ApproximationPolynom()
{
    List<object> answer = new List<object>();

    List<double> x = new List<double>();
    List<double> y = new List<double>();

    for (int i = 0; i < dataGridView1.Rows.Count - 1; i++)
    {
        for (int j = 1; j < dataGridView1.Columns.Count; j++)
        {
            if (i == 0)
                x.Add(Convert.ToDouble(dataGridView1[j, i].Value));
            else
                y.Add(Convert.ToDouble(dataGridView1[j, i].Value));
        }
    }
    for (int i = 0; i < x.Count; i++)
```

```
{
    chart1.Series["Series2"].Points.AddXY(x[i], y[i]);
}
Matrix asys = new Matrix(2);
asys[0, 0] = x.Count;
asys[0, 1] = x.Sum();
asys[1, 0] = x.Sum();
double tmp = 0;
foreach (double i in x)
{
    tmp += i * i;
}
asys[1, 1] = tmp;
tmp = 0;
for (int i = 0; i < x.Count; i++)
{
    tmp += x[i] * y[i];

}
List<double> b = new List<double>();
b.Add(y.Sum()); b.Add(tmp);
List<Matrix> LU = (List<Matrix>)asys.LUDec(b)[0];
List<double> a = asys.SOLVE(LU[0], LU[1], b);
List<double> xres = new List<double>(), yres = new List<double>();
xres.Add(x[0]);
yres.Add(f(xres[xres.Count - 1], a));
xres.Add(x[0] + 0.01);
yres.Add(f(xres[xres.Count - 1], a));
while (xres[xres.Count - 1] < x[x.Count - 1])
{
    xres.Add(xres[xres.Count - 1] + 0.01);
    yres.Add(f(xres[xres.Count - 1], a));
}
double err = 0;
for (int i = 0; i < x.Count; i++)
{
    err += (f(x[i], a) - y[i]) * (f(x[i], a) - y[i]);
}
asys = new Matrix(3);
asys[0, 0] = x.Count; asys[1, 0] = x.Sum(); asys[0, 1] = x.Sum();
tmp = 0;
foreach (double i in x)
    tmp += i * i;
asys[0, 2] = tmp; asys[1, 1] = tmp; asys[2, 0] = tmp;
tmp = 0;
foreach (double i in x)
{
    tmp += Math.Pow(i, 3);

}
asys[1, 2] = tmp;
asys[2, 1] = tmp;
tmp = 0;
```

```
    foreach (double i in x)
    {
        tmp += Math.Pow(i, 4);

    }
    asys[2, 2] = tmp;
    b = new List<double>();
    b.Add(y.Sum());
    tmp = 0;
    for (int i = 0; i < x.Count; i++)
    {
        tmp += x[i] * y[i];
    }
    b.Add(tmp);
    tmp = 0;
    for (int i = 0; i < x.Count; i++)
    {
        tmp += x[i] * x[i] * y[i];
    }
    b.Add(tmp);
    LU = (List<Matrix>)asys.LUDec(b)[0];
    a = asys.SOLVE(LU[0], LU[1], b);
    List<double> xres1 = new List<double>(), yres1 = new List<double>();
    xres1.Add(x[0]);
    yres1.Add(f(xres1[xres1.Count - 1], a));
    xres1.Add(x[0] + 0.01);
    yres1.Add(f(xres1[xres1.Count - 1], a));
    while (xres1[xres1.Count - 1] < x[x.Count - 1])
    {
        xres1.Add(xres1[xres1.Count - 1] + 0.01);
        yres1.Add(f(xres1[xres1.Count - 1], a));
    }
    double err1 = 0;
    for (int i = 0; i < x.Count; i++)
    {
        err1 += (f(x[i], a) - y[i]) * (f(x[i], a) - y[i]);
    }
    answer.Add(xres);
    answer.Add(yres);
    answer.Add(err);
    answer.Add(xres1);
    answer.Add(yres1);
    answer.Add(err1);
    return answer;
}
```

Пример работы:



Лабораторная работа 3.4
Численное дифференцирование

Формулировка задания:
Вычислить первую и вторую производную от таблично заданной функции $y_i = f(x_i)$, $i = 0, 1, 2, 3, 4$ в точке $x = X^*$

Вариант 7:

$$X^* = 0.2$$

i	0	1	2	3	4
x_i	-0.2	0.0	0.2	0.4	0.6
y_i	1.7722	1.5708	1.3694	1.1593	0.9273

Метод решения:
Производится процесс построения полинома Ньютона с последующим дифференцированием. Сравнивается со значениями, полученными через три точки - для первой производной с первым порядком точности:

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

для второй производной со вторым порядком точности:

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

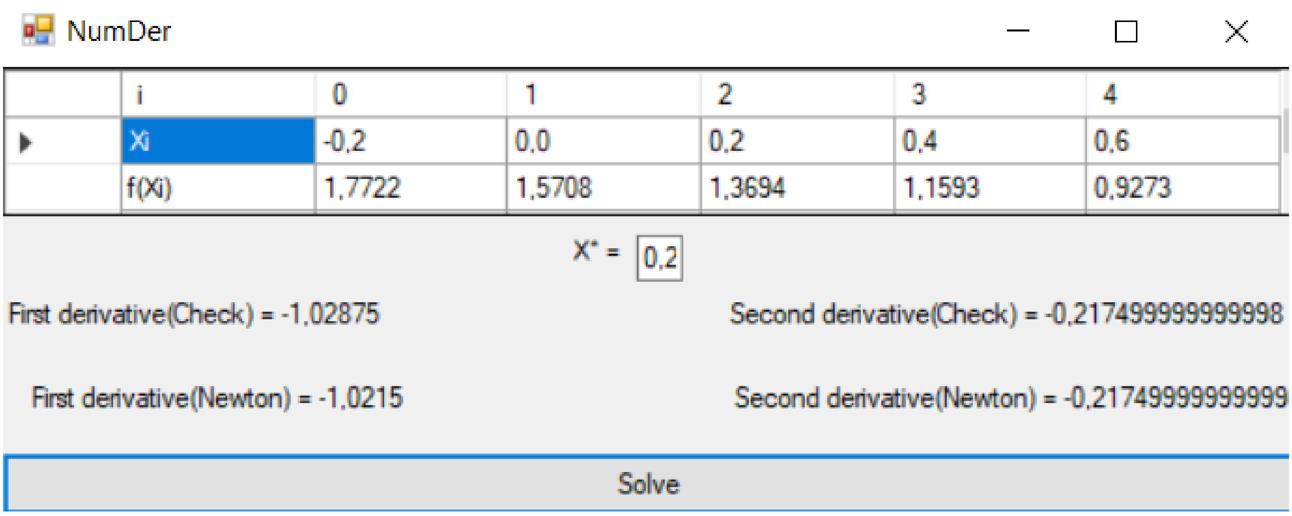
Среда разработки: Visual Studio ($C\#$)

Листинг:

```
public List<List<double>> newtonCoeffs(List<double> x, List<double> y)
{
    List<List<double>> ans = new List<List<double>>();
    ans.Add(new List<double>());
    ans[ans.Count - 1].Add(y[0]);
    var p = y[0];
    List<double> tmpx = new List<double>();
    double tmp, tmpf;
    for (int i = 0; i < x.Count -1; ++i)
    {
        ans.Add(new List<double>());
        tmp = 1;
        tmpx = new List<double>();
        for (int j = 0; j <= i; j++)
        {
            ans[ans.Count - 1].Add(x[j]);
            tmpx.Add(x[j]);
        }
        tmpx.Add(x[i + 1]);
        tmpf = f(tmpx, y);
        p += tmp * tmpf;
```

```
            ans[ans.Count - 1].Insert(0,tmpf);
        }

        return ans;
    }


    public double f(List<double> x, List<double> y)
    {
        if (x.Count == 2)
            return (y[0] - y[1]) / (x[0] - x[1]);
        List<double> tmp1 = new List<double>();
        List<double> tmp2 = new List<double>();
        foreach (var f in x)
        {
            tmp1.Add(f);
            tmp2.Add(f);
        }

        tmp1.RemoveAt(tmp1.Count - 1);
        tmp2.RemoveAt(0);
        List<double> tmp1f = new List<double>();
        List<double> tmp2f = new List<double>();
        foreach (var f in y)
        {
            tmp1f.Add(f);
            tmp2f.Add(f);
        }
        tmp1f.RemoveAt(tmp1f.Count - 1);
        tmp2f.RemoveAt(0);
        return (f(tmp1, tmp1f) - f(tmp2, tmp2f)) / (x[0] - x[x.Count - 1]);
    }
```

Пример работы:



Лабораторная работа 3.5
Численное интегрирование

Формулировка задания:

Вычислить определенный интеграл $F = \int\limits_{x_0}^{x_1} ydx$ методами прямоугольника, трапеций, Симпсона с шагами h_1, h_2. Оценить погрешность вычислений, используя метод Рунге-Ромберга.

Вариант 7:

$$y = \frac{1}{3x^2 + 4x + 2}, \qquad X_0 = -2, \quad X_k = 2, \quad h_1 = 1.0, \quad h_2 = 0.5$$

Метод решения:

Метод прямоугольника реализован в соответствии с формулой:

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

Метод трапеций:

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

Метод Симпсона:

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

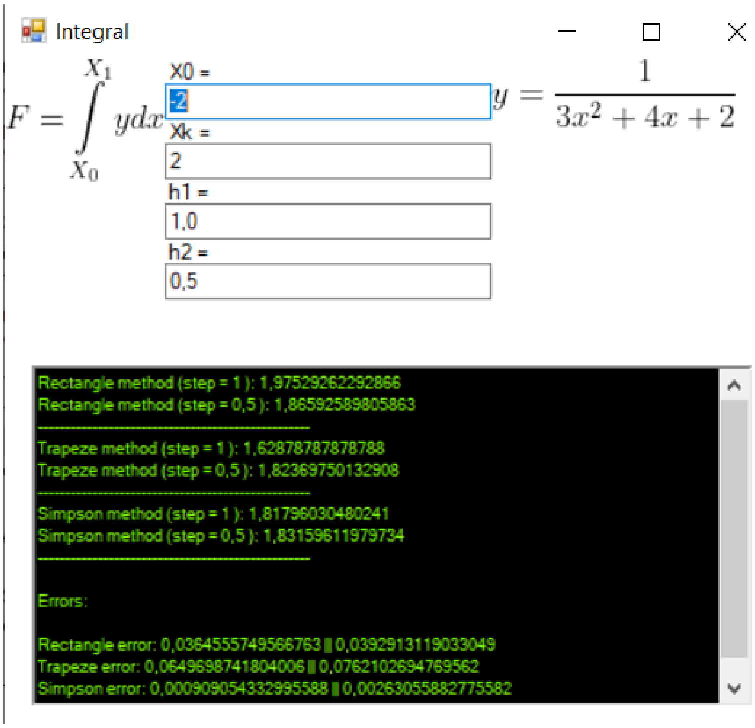
Среда разработки: Visual Studio ($C\#$)

Листинг:

```
public double rectangle_method(List<double> x, double h)
{
    double sum = 0;
    for (int i = 0; i < x.Count - 1; i++)
        sum += f((x[i + 1] + x[i]) / 2);
    return h * sum;
}
public double trapeze_method(List<double> y, double h)
{
    int n = y.Count - 1;
    double sum = 0;
    for (int i = 1; i < n -1; i++)
    {
        sum += y[i];
    }
    return h * (y[0] / 2 + sum + y[n]);
}
public double Simpson_method(List<double> y, double h)
{
    int n = y.Count - 1;
    double sum = 0;
```

```
        for (int i = 1; i < n ; i++)
        {
            sum += f(y[i - 1]) + 4 * f((y[i - 1] + y[i]) / 2f) + f(y[i]);
        }
        return sum * h / 6;
    }

    public string Runge_Romberg_Rithardson(List<double> steps,
    List<double> rec, List<double> trapeze, List<double> sim,
    string error_end)
    {
        List<string> answer = new List<string>();
        error_end += "\nErrors: \n";
        double k = steps[0] / steps[1];
        double analitic_solution = 1.8574186872187473;
        List<double> err_rec = new List<double>();
        err_rec.Add(Math.Abs(rec[0] - rec[1]) / (k * k - 1));
        err_rec.Add(Math.Abs(rec[0] - analitic_solution) / (k * k - 1));
        List<double> err_trapeze = new List<double>();
        err_trapeze.Add(Math.Abs(trapeze[0] - trapeze[1]) / (k * k - 1));
        err_trapeze.Add(Math.Abs(trapeze[0] - analitic_solution)
        / (k * k - 1));
        List<double> err_sim = new List<double>();
        err_sim.Add(Math.Abs(sim[0] - sim[1]) / (Math.Pow(k, 4) - 1));
        err_sim.Add(Math.Abs(sim[0] - analitic_solution)
        / (Math.Pow(k, 4) - 1));
        error_end += "\nRectangle error: " +
            err_rec[0].ToString() + " || " + err_rec[1].ToString() + "\n";
        error_end += "Trapeze error: " +
            err_trapeze[0].ToString() + " || " +
                            err_trapeze[1].ToString() + "\n";
        error_end += "Simpson error: " +
            err_sim[0].ToString() + " || "
                            + err_sim[1].ToString() + "\n";
        return error_end;

    }
```

Пример работы:



Лабораторная работа 4.1
Решение задачи Коши для ОДУ

Формулировка задания:

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

Вариант 7:

Задача Коши

$$y'' - 4xy' + (4x^2 - 2)y = 0$$
$$y(0) = 1$$
$$y'(0) = 1$$
$$x \in [0, 1], \quad h = 0.1$$

Точное решение:

$$y = (1 + x)e^{x^2}$$

Метод решения:

Метод Эйлера - одношаговый метод, формула которого:

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

Глобальная погрешность - линейная функция, поэтому метод Эйлера имеет первый порядок точности.

Метод Рунге-Кутты записывается в виде совокупности формул:

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

$$i = 2, 3, \ldots, p$$

Реализован метод четвертого порядка:

$$y_k k + 1 = y_k + \Delta y_k$$
$$\Delta y_k = \frac{1}{6}(K_1^k + 2K_2^k + 2K_3^k + K_4^k)$$
$$K_1^k = hf(x_k, y_k)$$
$$K_2^k = hf(x_k + \frac{1}{2}h, y_k + \frac{1}{2}K_1^k)$$
$$K_3^k = hf(x_k + \frac{1}{2}h, y_k + \frac{1}{2}K_2^k)$$
$$K_4^k = hf(x_k + h, y_k + K_3^k)$$

Методу Адамса соответствует формула:

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

где f_k значение подынтегральной функции в узле x_k.

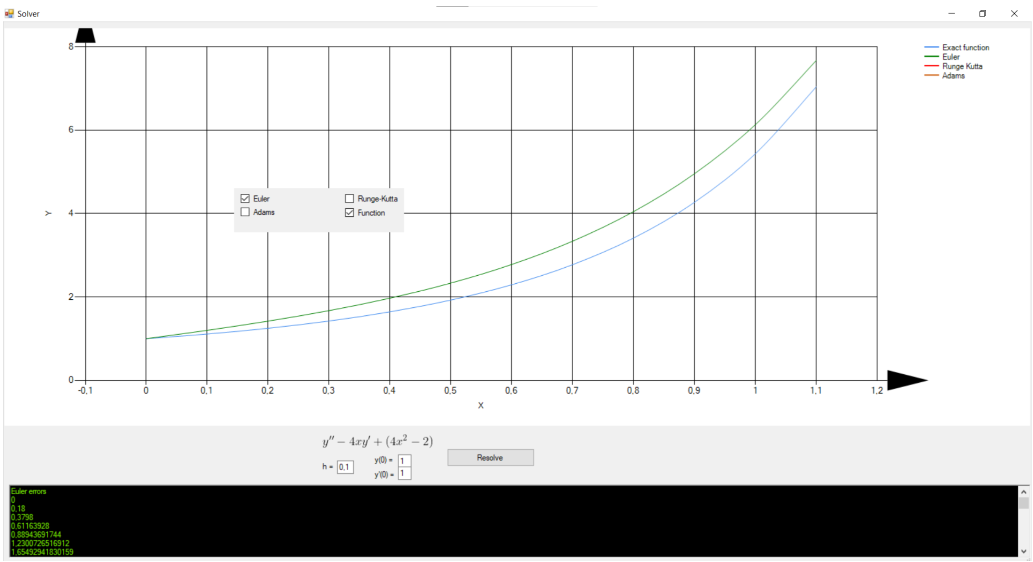
Среда разработки: Visual Studio ($C\#$)

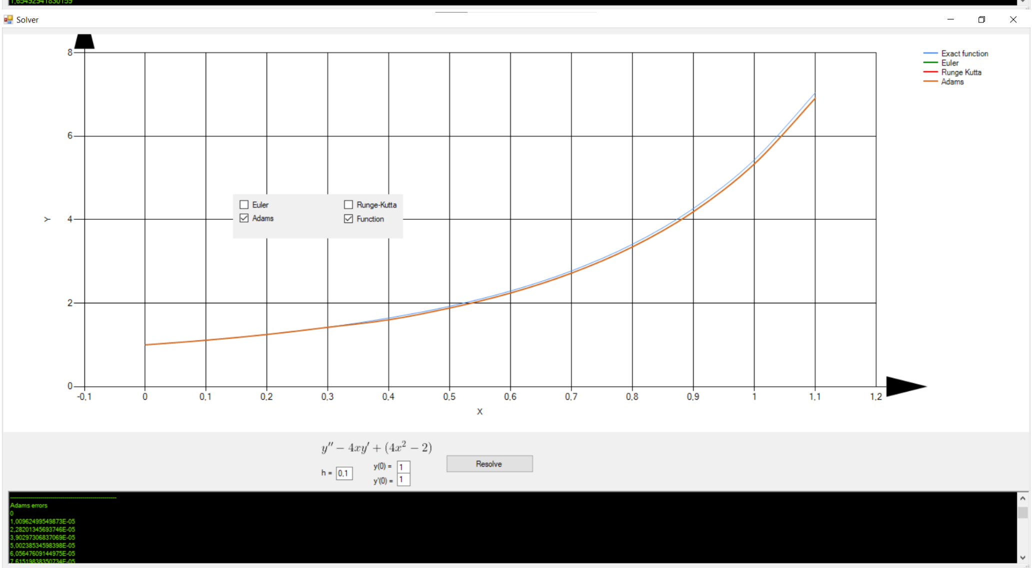
Листинг:

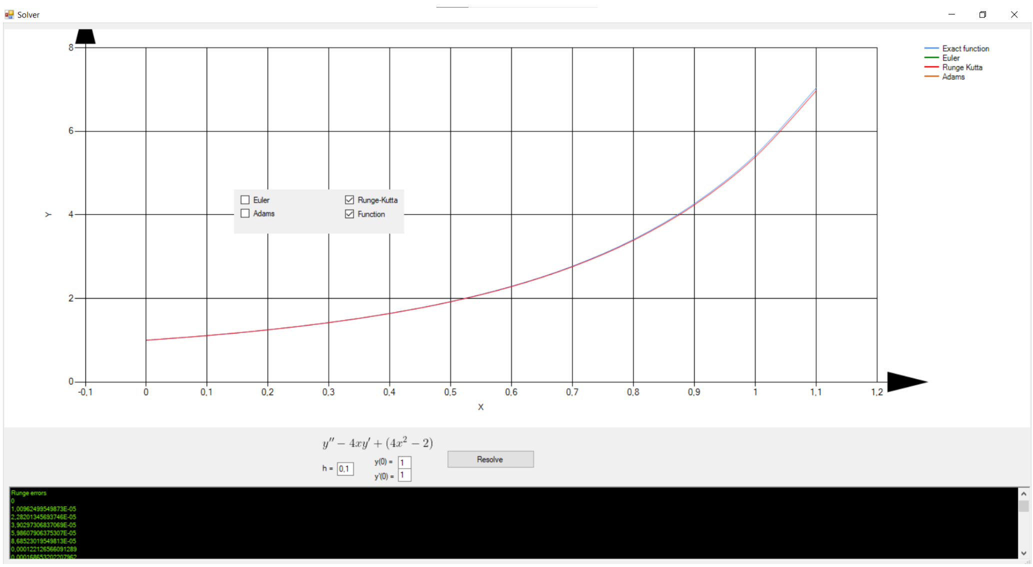
```
public List<double> Euler(List<double> x, double a, double b, double h)
{
    List<double> y = new List<double>();
    List<double> z = new List<double>();
    z.Add(2); y.Add(1);
    for (int i = 0; i < x.Count; i++)
    {
        z.Add(z[i] + h * f(x[i], y[i], z[i]));
        y.Add(y[i] + h * z[i]);
    }

    return y;
}
public List<List<double>> Runge_Kutta(List<double> x, double a, double b,
double h)
{
    List<List<double>> answer = new List<List<double>>();
    List<double> y = new List<double>();
    List<double> k = new List<double>() { 0, 0, 0, 0 };
    List<double> l = new List<double>() { 0, 0, 0, 0 };
    //for (int i = 0)
    List<double> z = new List<double>();
    z.Add(1);
    y.Add(1);
    for (int i = 0; i < x.Count; i++)
    {
        l[0] = h * f(x[i], y[i], z[i]);
        k[0] = h * g(x[i], y[i], z[i]);
        l[1] = h * f(x[i] + h / 2, y[i] + k[0] / 2, z[i] + l[0] / 2);
        k[1] = h * g(x[i] + l[0] / 2, y[i] + l[0] / 2, z[i] + l[0] / 2);
        l[2] = h * f(x[i] + h / 2, y[i] + k[1] / 2, z[i] + l[1] / 2);
        k[2] = h * g(x[i] + l[1] / 2, y[i] + l[1] / 2, z[i] + l[1] / 2);
        l[3] = h * f(x[i] + h / 2, y[i] + k[2], z[i] + l[2]);
        k[3] = h * g(x[i] + l[2], y[i] + l[2], z[i] + l[2]);
        z.Add(z[i] + delta(l));
        y.Add(y[i] + delta(k));
    }
    answer.Add(y);
    answer.Add(z);
    return answer;
}

public double g(double x, double y, double z) => z;
public List<double> Adams(List<double> x, double a, double b, double h,
List<double> y_runge, List<double> z_runge)
{
    List<double> y = new List<double>();
    List<double> z = new List<double>();
    z.Add(2);
    z.Add(z_runge[1]);
```

```
        z.Add(z_runge[2]);
        z.Add(z_runge[3]);
        y.Add(1);
        y.Add(y_runge[1]);
        y.Add(y_runge[2]);
        y.Add(y_runge[3]);
        for (int i = 3; i < x.Count - 1; i++)
        {
            double tmp1 = z[i] + h * (55 * f(x[i], y[i], z[i]) - 59 *
            f(x[i - 1], y[i - 1], z[i - 1]) +
                37 * f(x[i - 2], y[i - 2], z[i - 2]) - 9 * f(x[i - 3],
                y[i - 3], z[i - 3])) / 24;
            double tmp2 = y[i] + h * (55 * (z[i]) - 59 * (z[i - 1])
                + 37 * (z[i - 2]) - 9 * (z[i - 3])) / 24;
            z.Add(tmp1);
            y.Add(tmp2);
        }
        return y;
    }
```

Пример работы:







Solver
Exact function
Euler
Runge Kutta
Adams
Euler
Adams
Runge-Kutta
Function
Y
X
$y'' - 4xy' + (4x^2 - 2)$
h = 0,1
y(0) = 1
y'(0) = 1
Resolve
Runge errors
0
1,00962499549873E-05
2,28201345693746E-05
3,90297306837069E-05
5,98607906375307E-05
8,68523019549813E-05
0,000122126566091289

Лабораторная работа 4.2
Решение краевых задач

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

Вариант 7:
Краевая задача

$$(2x+1)y'' + 4xy' - 4y = 0$$
$$y'(0) = -1$$
$$y'(1) + 2y(1) = 3$$

Точное решение

$$y(x) = x + e^{-2x}$$

Метод решения:
Метод стрельбы: берем некое начальное значение $\eta = \eta_0$, решаем задачу Коши методом Рунге-Кутта. Сравнивая решение этой задачи со значением y_1 в правом конце отрезка корректируем угол наклона касательной к решению в левом конце отрезка. Решение исходной задачи эквивалентно:

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

где

$$\Phi(\eta) = y(b, y_0, \eta) - y_1$$

Конечно-разностный метод заключается в решении СЛАУ с трехдиагональной матрицей коэффициентов:

$$\begin{cases} (-2 + h^2 q(x_1))y_1 + (1 + \frac{p(x_1)h}{2})y_2 = h^2 f(x_1) - (1 - \frac{p(x_1)h}{2})y_a \\ (1 - \frac{p(x_k)h}{2})y_{k-1} + (-2 + h^2 q(x_k))y_k + (1 + \frac{p(x_k)h}{2})y_{k+1} = h^2 f(x_k) \qquad k = 2, \ldots, N-2 \\ (1 - \frac{p(x_{N-1})h}{2})y_{N-1} + (-2 + h^2 q(x_{N-1}))y_{N-1} = h^2 f(x_{N-1}) - (1 + \frac{p(x_{N-1})h}{2})y_b \end{cases}$$

Среда разработки: Visual Studio ($C\#$)

Листинг:

```
public List<List<double>> finite_difference_method(double a = 0,
double b = 1, double alpha = 0,
double beta = 1, double delta = 2,
    double gamma = 1, double y0 = -1,
    double y1 = 3, double h = 0.1)
{
    List<List<double>> points = new List<List<double>>();
    int n = (int)((b - a) / h);
    List<double> x = new List<double>();
    for (double i = a; i < b; i += h)
    {
```

```
            x.Add(i);

        }
        List<double> a_tma = new List<double>();
        List<double> b_tma = new List<double>();
        List<double> c_tma = new List<double>();
        List<double> d_tma = new List<double>();
        a_tma.Add(0);
        for (int i = 0; i < n - 1; i++)
            a_tma.Add(1 - p(x[i]) * h / 2);
        a_tma.Add(-gamma);
        b_tma.Add(alpha * h - beta);
        for (int i = 0; i < n-1; i++)
            b_tma.Add(q(x[i]) * h *h -2 );
        b_tma.Add(delta * h + gamma);
        c_tma.Add(beta);
        for (int i = 0; i < n - 1; i++)
            c_tma.Add(1 + p(x[i]) * h / 2);
        c_tma.Add(0);
        d_tma.Add(y0 * h);
        for (int i = 0; i < n - 1; i++)
            d_tma.Add(f(x[i]) * h * h);
        d_tma.Add(y1 * h);

        List<double> y = TMA(a_tma.Count, a_tma, b_tma, c_tma, d_tma);
        points.Add(x);
        points.Add(y);
        return points;
    }



    public List<List<double>> shooting_method(double a = 0, double b = 1,
    double alpha = 0, double beta = 1,
    double delta = 2, double gamma = 1,
    double y0 = -1, double y1 = 3,
    double h = 0.1)
    {

        double n_prev = 100, n = 10;
        double y_der = (y0 - alpha * n_prev) / beta;
        var x = new List<double>();
        List<double> n0 = new List<double>();
        n0.Add(n_prev);
        n0.Add(n);
        for (double i = a; i < b+h; i += h)
            x.Add(i);
        var ans_prev = Runge_Kutta(x, a, b, h, n_prev, y_der);
        y_der = (y0 - alpha * n) / beta;
        var ans = Runge_Kutta(x, a, b, h, n, y_der);
        int iter = 2;

        while (check_finish(x, ans[0], b, delta, gamma, y1))
```

```
    {
        iter++;
        double tmp = n;
        n = get_n(n_prev, n, ans_prev, ans,x);
        n0.Add(n);
        n_prev = tmp;
        ans_prev.Clear();
        foreach (var f in ans)
            ans_prev.Add(f);
        y_der = (y0 - alpha * n) / beta;
        ans = Runge_Kutta(x, a, b, h, n, y_der);

    }

    ans.Insert(0, x);
    List<double> iters = new List<double>();
    iters.Add(iter);
    ans.Add(iters);
    ans.Add(n0);
    return ans;
}


public double get_n(double n_prev, double n,
List<List<double>> ans_prev, List<List<double>> ans, List<double> x,
double b = 1, double delta = 2, double gamma = 1,
double y1 = 3)
{
    //List<double> x = new List<double>();

    List<double> y = new List<double>();
    foreach (var f in ans_prev[0])
        y.Add(f);
    double y_der = first_der(x, y, b);
    double phi_n_prev = delta * y[y.Count - 1] + gamma * y_der - y1;
    y.Clear();

    foreach (var f in ans[0])
        y.Add(f);
    y_der = first_der(x, y, b);
    double phi_n = delta * y[y.Count - 1] + gamma * y_der - y1;
    return n - (n - n_prev) / (phi_n - phi_n_prev) * phi_n;

}
public bool check_finish(List<double> x, List<double >y, double b = 1,
double delta = 2, double gamma = 1, double y1 = 1, double eps = 1E-12)
{
    int iter;
    double y_der = first_der(x, y, b);

    return (Math.Abs(delta * y[y.Count - 1] + gamma * y_der - y1) > eps);
}
```

```
public List<List<double>> Runge_Romberg_method(
List<List<double>> res_finite, List<List<double>> res_shoot, double k)
{

    List<List<double>> ans = new List<List<double>>();
    List<double> err_finite = new List<double>();
    List<double> err_shooting = new List<double>();
    for (int i = 0; i < res_finite[0].Count; i++)
    {
        err_finite.Add(Math.Abs(res_finite[0][i] - res_finite[1][i])
                                              / (Math.Pow(k, 1) - 1)) ;
    }
    for (int i = 0; i < res_shoot[0].Count; i++)
    {
        err_shooting.Add(Math.Abs(res_shoot[0][i] - res_shoot[1][i])
                                             / (Math.Pow(k, 1) - 1));
    }

    ans.Add(err_finite);
    ans.Add(err_shooting);
   // ans.Add(err_adams);

    return ans;
}

public List<List<double>> exact_error(List<double> res_finite,
List<double> res_shoot, List<double> exact)
{
    //res[0] -euler
    List<List<double>> ans = new List<List<double>>();
    List<double> err_finite = new List<double>();
    List<double> err_shoot = new List<double>();
    for (int i = 0; i < Math.Min(res_finite.Count, exact.Count); i++)
        err_finite.Add(Math.Abs(res_finite[i] - exact[i]));
    for (int i = 0; i < Math.Min(res_shoot.Count, exact.Count); i++)
        err_shoot.Add(Math.Abs(res_shoot[i] - exact[i]));
    ans.Add(err_finite);
    ans.Add(err_shoot);
    return ans;

}
```

Пример работы: