МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ

ФГБОУ ВО «Брянский государственный технический университет»

Кафедра «Компьютерные технологии и системы»

Направление подготовки

10.05.04- «Информационно-аналитические системы безопасности»

Профиль – «Автоматизация информационно-аналитической деятельности»

**Отчет по лабораторным работам**

по дисциплине: «Численные методы»

|  |  |
| --- | --- |
|  | Выполнили:  Студента группы О-19-ИАС-аид-С  Колотвин Артём Павлович  «\_\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_ 2022 г |
|  | Принял:  к.т.н., доц. каф. «КТС»  Леонов Юрий Алексеевич  «\_\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_ 2022 г. |

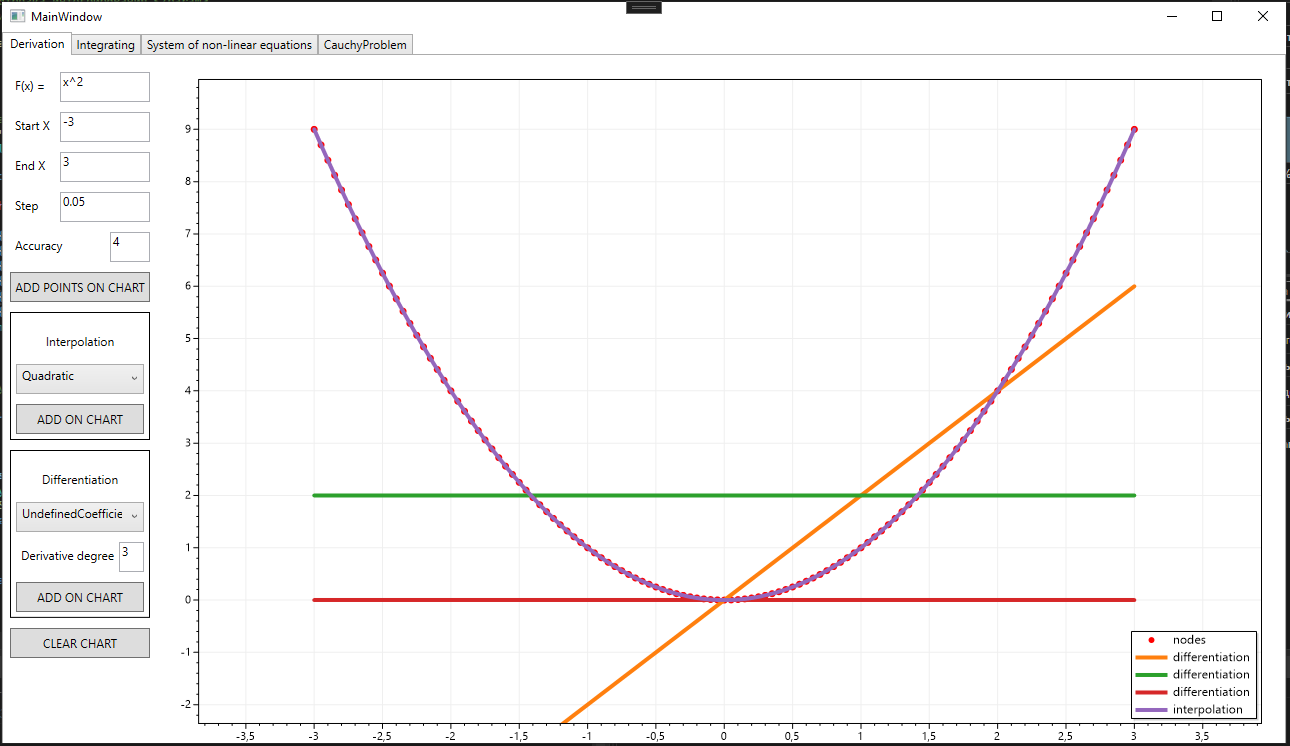
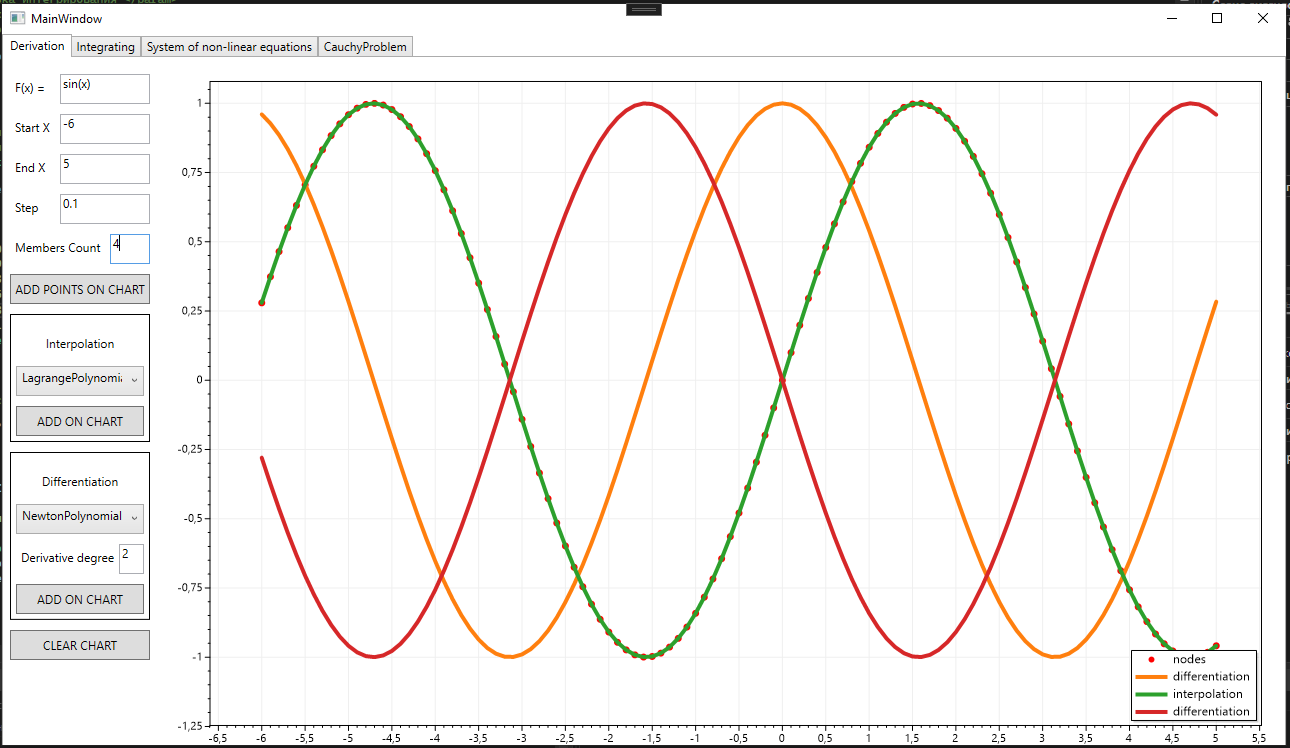
**Брянск 2022**

## Лабораторная работа №1

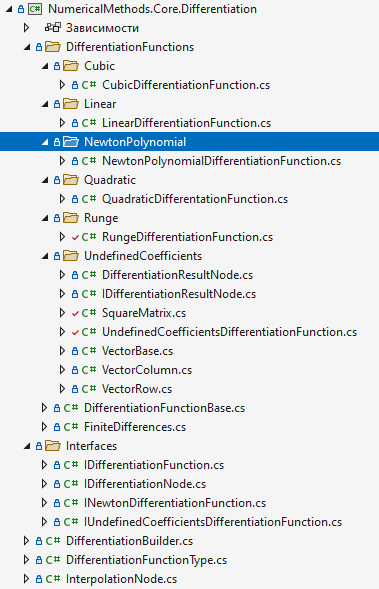
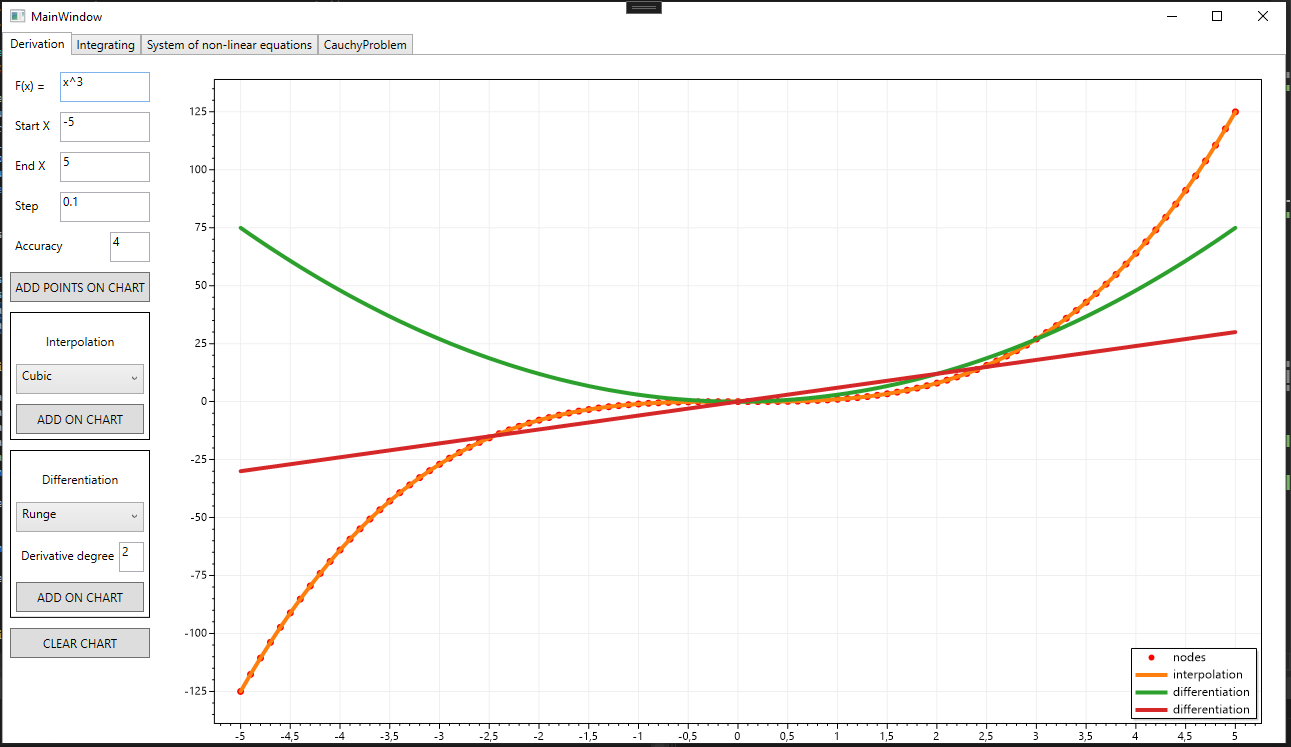
### «Решение задач численного дифференцирования»

**Цель работы**: изучение методов численного дифференцирования, вычисление первой и второй производных заданной функции с использованием интерполяционного многочлена Ньютона.

**Скриншоты работы программы**



**Структура проекта**



**Листинг программы**

Листинг класса DifferentiationBuilder

using NumericalMethods.Core.Differentiation.Interfaces;

using NumericalMethods.Core.Differentiation.DifferentiationFunctions.Linear;

using NumericalMethods.Core.Differentiation.DifferentiationFunctions.Quadratic;

using NumericalMethods.Core.Differentiation.DifferentiationFunctions.NewtonPolynomial;

using NumericalMethods.Core.Differentiation.DifferentiationFunctions.UndefinedCoefficients;

using NumericalMethods.Core.Differentiation.DifferentiationFunctions.Cubic;

using NumericalMethods.Core.Differentiation.DifferentiationFunctions.Runge;

namespace NumericalMethods.Core.Differentiation;

public static class DifferentiationBuilder

{

public static IDifferentiationFunction? Build(IEnumerable<IDifferentiationNode> differentiation\_nodes, DifferentiationFunctionType function\_type, double step, int derrivative\_degree)

{

return function\_type switch

{

DifferentiationFunctionType.Linear => new LinearDifferentiationFunction(differentiation\_nodes, step, derrivative\_degree),

DifferentiationFunctionType.Quadratic => new QuadraticDifferentationFunction(differentiation\_nodes, step, derrivative\_degree),

DifferentiationFunctionType.Cubic => new CubicDifferentiationFunction(differentiation\_nodes, step, derrivative\_degree),

\_ => throw new NotImplementedException()

};

}

public static INewtonDifferentiationFunction CreateNewton(IEnumerable<IDifferentiationNode> differentiationNodes, double step, int derivative\_degree, int numberOfMembers)

{

return new NewtonPolynomialDifferentiationFunction(differentiationNodes, step, derivative\_degree, numberOfMembers);

}

public static IUndefinedCoefficientsDifferentiationFunction CreateUndefinedCoefficients(IEnumerable<IDifferentiationNode> differentiationNodes, double step, int derrivative\_degre, int count\_coefficients\_c)

{

return new UndefinedCoefficientsDifferentiationFunction(differentiationNodes, step, derrivative\_degre, count\_coefficients\_c);

}

public static IDifferentiationFunction CreateRunge(IEnumerable<IDifferentiationNode> differentiationNodes, double step, int derrivative\_degre, int accuracy\_order, int number\_of\_used\_points)

{

return new RungeDifferentiationFunction(differentiationNodes, step, derrivative\_degre, accuracy\_order, number\_of\_used\_points);

}

}

Листинг класса DifferentiationFunctionType

namespace NumericalMethods.Core.Differentiation;

public enum DifferentiationFunctionType

{

Linear,

Quadratic,

Cubic,

NewtonPolynomials,

UndefinedCoefficients,

Runge

}

Листинг класса InterpolationNode

using NumericalMethods.Core.Approximation.Interfaces;

namespace NumericalMethods.Core.Differentiation;

public record class InterpolationNode(double X, double Y) : IInterpolationNode;

Листинг класса IDifferentiationFunction

namespace NumericalMethods.Core.Differentiation.Interfaces;

public interface IDifferentiationFunction

{

public double? Calculate(double argument);

}

Листинг класса IDifferentiationNode

namespace NumericalMethods.Core.Differentiation.Interfaces;

public interface IDifferentiationNode

{

public double X { get; init; }

public double Y { get; init; }

}

Листинг класса INewtonDifferentiationFunction

namespace NumericalMethods.Core.Differentiation.Interfaces

{

public interface INewtonDifferentiationFunction

{

public double? Calculate(double argument);

}

}

Листинг класса IUndefinedCoefficientsDifferentiationFunction

using NumericalMethods.Core.Differentiation.DifferentiationFunctions.UndefinedCoefficients;

namespace NumericalMethods.Core.Differentiation.Interfaces;

public interface IUndefinedCoefficientsDifferentiationFunction

{

public IEnumerable<IDifferentiationResultNode> Calculate();

}

Листинг класса DifferentiationFunctionBase

using NumericalMethods.Core.Approximation;

using NumericalMethods.Core.Approximation.Interfaces;

using NumericalMethods.Core.Differentiation.Interfaces;

using NumericalMethods.Core.Differentiation;

namespace NumericalMethods.Core.Differentiation.DifferentiationFunctions

{

internal abstract class DifferentiationFunctionBase

{

protected readonly double \_step;

protected readonly int \_derivative\_degree;

protected readonly IInterpolationFunction \_interpolation\_function;

protected readonly IDifferentiationNode \_center\_node;

protected readonly IDifferentiationNode \_first\_node;

protected readonly IDifferentiationNode \_last\_node;

protected readonly IEnumerable<IDifferentiationNode> \_nodes;

public DifferentiationFunctionBase(IEnumerable<IDifferentiationNode> differentiationNodes, double step, int derrivative\_degree)

{

\_step = step;

\_nodes = differentiationNodes;

\_derivative\_degree = derrivative\_degree;

\_first\_node = differentiationNodes.First();

\_last\_node = differentiationNodes.Last();

\_center\_node = differentiationNodes.OrderBy((node) => node.X).ElementAt(differentiationNodes.Count() / 2);

IEnumerable<IInterpolationNode> mapped\_nodes = differentiationNodes.Select(node => new InterpolationNode(node.X, node.Y));

\_interpolation\_function = InterpolationBuilder.Build(mapped\_nodes, InterpolationFunctionType.Cubic);

}

private double? GetCenterFiniteDifference(double argument) => \_interpolation\_function.Calculate(argument + \_step) - \_interpolation\_function.Calculate(argument - \_step);

protected double? GetCenterFiniteDifference(double argument, int degree) => GetCenterFiniteDifferenceRecursive(argument, degree, 1);

private double? GetCenterFiniteDifferenceRecursive(double argument, int degree, int depth)

{

return depth == degree ?

GetCenterFiniteDifference(argument)

: GetCenterFiniteDifferenceRecursive(argument + \_step, degree, depth + 1) - GetCenterFiniteDifferenceRecursive(argument - \_step, degree, depth + 1);

}

private double? GetLeftFiniteDifference(double argument) => \_interpolation\_function.Calculate(argument) - \_interpolation\_function.Calculate(argument - \_step);

protected double? GetLeftFiniteDifference(double argument, int degree) => GetLeftFiniteDifferenceRecursive(argument, degree, 1);

private double? GetLeftFiniteDifferenceRecursive(double argument, int degree, int depth)

{

return depth == degree ?

GetLeftFiniteDifference(argument)

: GetLeftFiniteDifferenceRecursive(argument, degree, depth + 1) - GetLeftFiniteDifferenceRecursive(argument - \_step, degree, depth + 1);

}

private double? GetRightFiniteDifference(double argument) => \_interpolation\_function.Calculate(argument + \_step) - \_interpolation\_function.Calculate(argument);

protected double? GetRightFiniteDifference(double argument, int degree) => GetRightFiniteDifferenceRecursive(argument, degree, 1);

private double? GetRightFiniteDifferenceRecursive(double current\_argument, int degree, int depth)

{

return depth == degree ?

GetRightFiniteDifference(current\_argument)

: GetRightFiniteDifferenceRecursive(current\_argument + \_step, degree, depth + 1) - GetRightFiniteDifferenceRecursive(current\_argument, degree, depth + 1);

}

}

Листинг класса FiniteDifferences

namespace NumericalMethods.Core.Differentiation.DifferentiationFunctions;

public enum FiniteDifferences

{

Left,

Center,

Right

}

Листинг класса LinearDifferentiationFunction

using NumericalMethods.Core.Differentiation.DifferentiationFunctions;

using NumericalMethods.Core.Differentiation.Interfaces;

namespace NumericalMethods.Core.Differentiation.DifferentiationFunctions.Linear;

internal class LinearDifferentiationFunction : DifferentiationFunctionBase, IDifferentiationFunction

{

private readonly INewtonDifferentiationFunction \_newton\_funtion;

public LinearDifferentiationFunction(IEnumerable<IDifferentiationNode> interpolation\_nodes, double step, int derrivative\_degree)

: base(interpolation\_nodes, step, derrivative\_degree)

{

\_newton\_funtion = DifferentiationBuilder.CreateNewton(interpolation\_nodes, step, derrivative\_degree, derrivative\_degree + 1);

}

public double? Calculate(double argument) => \_newton\_funtion.Calculate(argument);

}

Листинг класса QuadraticDifferentationFunction

using NumericalMethods.Core.Differentiation;

using NumericalMethods.Core.Differentiation.DifferentiationFunctions;

using NumericalMethods.Core.Differentiation.Interfaces;

namespace NumericalMethods.Core.Differentiation.DifferentiationFunctions.Quadratic;

internal class QuadraticDifferentationFunction : DifferentiationFunctionBase, IDifferentiationFunction

{

private INewtonDifferentiationFunction \_newton\_function;

public QuadraticDifferentationFunction(IEnumerable<IDifferentiationNode> differentiation\_nodes, double step, int derrivative\_degree)

: base(differentiation\_nodes, step, derrivative\_degree)

{

\_newton\_function = DifferentiationBuilder.CreateNewton(differentiation\_nodes, step, derrivative\_degree, derrivative\_degree + 2);

}

public double? Calculate(double argument) => \_newton\_function.Calculate(argument);

}

Листинг класса CubicDifferentiationFunction

using NumericalMethods.Core.Differentiation.Interfaces;

namespace NumericalMethods.Core.Differentiation.DifferentiationFunctions.Cubic;

internal class CubicDifferentiationFunction : DifferentiationFunctionBase, IDifferentiationFunction

{

private readonly INewtonDifferentiationFunction \_newton\_function;

public CubicDifferentiationFunction(IEnumerable<IDifferentiationNode> differentiationNodes, double step, int derrivative\_degree)

: base(differentiationNodes, step, derrivative\_degree)

{

\_newton\_function = DifferentiationBuilder.CreateNewton(differentiationNodes, step, derrivative\_degree, derrivative\_degree + 3);

}

public double? Calculate(double argument) => \_newton\_function.Calculate(argument);

}

Листинг класса NewtonPolynomialDifferentiationFunction

using MathNet.Symbolics;

using NumericalMethods.Core.Differentiation.DifferentiationFunctions;

using NumericalMethods.Core.Differentiation.Interfaces;

using System.Globalization;

using System.Text;

using System.Text.Json;

using System.Text.RegularExpressions;

namespace NumericalMethods.Core.Differentiation.DifferentiationFunctions.NewtonPolynomial;

internal class NewtonPolynomialDifferentiationFunction : DifferentiationFunctionBase, INewtonDifferentiationFunction

{

private readonly Dictionary<int, int> factorialCache = new Dictionary<int, int>() { { 1, 1 } };

private int \_numberOfMembers;

private SymbolicExpression \_functionExpression;

private List<string> \_other\_variables;

private List<int> \_degrees;

private readonly double last\_x\_for\_right\_finite\_difference;

private readonly double first\_x\_for\_left\_finite\_difference;

public NewtonPolynomialDifferentiationFunction(IEnumerable<IDifferentiationNode> differentiationNodes, double step, int derrivative\_degree, int numberOfMembers)

: base(differentiationNodes, step, derrivative\_degree)

{

\_numberOfMembers = numberOfMembers;

string? function = GetNewtonPolynomialDerrivative(\_derivative\_degree, \_numberOfMembers);

\_functionExpression = SymbolicExpression.Parse(function);

\_other\_variables = \_functionExpression

.CollectVariables()

.Select(variable => variable.ToString())

.Where(variable => variable.Contains('y')).ToList();

\_degrees = \_other\_variables

.Select(variable => int.Parse(variable[1..])).ToList();

int max\_finite\_difference\_degree = \_numberOfMembers - 1;

last\_x\_for\_right\_finite\_difference = \_first\_node.X + step \* max\_finite\_difference\_degree;

first\_x\_for\_left\_finite\_difference = \_last\_node.X - step \* max\_finite\_difference\_degree;

}

private int CalculateFactorial(int value)

{

if (factorialCache.TryGetValue(value, out int cachedValue))

{

return cachedValue;

}

else

{

if (value < 2) return 1;

int calculatedValue = value \* CalculateFactorial(value - 1);

factorialCache.Add(value, calculatedValue);

return calculatedValue;

}

}

private string GetNewtonPolynomial(int numberOfMembers)

{

switch (numberOfMembers)

{

case < 2: return "f";

case 2: return "f + t\*y1";

default:

{

StringBuilder result = new StringBuilder("f + t\*y1");

for (int i = 3; i <= numberOfMembers; i++)

{

int n = i - 1;

int factorial = CalculateFactorial(n);

List<string> bracketsList = new List<string>();

for (int j = 0; j < n - 1; j++)

{

bracketsList.Add($"(t - {j + 1})");

}

string bracketsString = string.Join(" \* ", bracketsList);

string member = $"t \* {bracketsString}/{factorial}\*y{n}";

result.Append($" + {member}");

}

return result.ToString();

}

}

}

private string? GetNewtonPolynomialDerrivative(int derrivativeDegree, int numberOfMembers)

{

if (derrivativeDegree < 1 || derrivativeDegree + 1 > numberOfMembers) return null;

string newtonPolynomial = GetNewtonPolynomial(numberOfMembers);

SymbolicExpression result = SymbolicExpression.Parse(newtonPolynomial).Expand();

for (int i = 0; i < derrivativeDegree; i++)

{

result = result.Differentiate("t");

}

return result.ToString();

}

public double? Calculate(double argument)

{

List<double?> variablesValues = new List<double?>();

FiniteDifferences difference\_type = ChoiceDifference(argument);

double t\_value = 1;

double denominator = 1;

switch (difference\_type)

{

case FiniteDifferences.Right:

variablesValues = \_degrees.Select(finiteDifferenceDegree => GetRightFiniteDifference(argument, finiteDifferenceDegree)).ToList();

t\_value = 0;

denominator = Math.Pow(\_step, \_derivative\_degree);

break;

case FiniteDifferences.Left:

variablesValues = \_degrees.Select(finiteDifferenceDegree => GetLeftFiniteDifference(argument, finiteDifferenceDegree)).ToList();

t\_value = 1;

denominator = Math.Pow(\_step, \_derivative\_degree);

break;

case FiniteDifferences.Center:

variablesValues = \_degrees.Select(finiteDifferenceDegree => GetCenterFiniteDifference(argument, finiteDifferenceDegree)).ToList();

t\_value = 1;

denominator = Math.Pow(2 \* \_step, \_derivative\_degree);

break;

};

if (variablesValues.Any(variableValue => variableValue is null)) return null;

Dictionary<string, FloatingPoint> values = new Dictionary<string, FloatingPoint>()

{

{"t", t\_value }

};

for (int i = 0; i < \_other\_variables.Count; i++)

{

values.Add(\_other\_variables[i], variablesValues[i]);

}

return \_functionExpression.Evaluate(values).RealValue / denominator;

}

private FiniteDifferences ChoiceDifference(double argument)

{

if (argument <= last\_x\_for\_right\_finite\_difference) return FiniteDifferences.Right;

else if (argument >= first\_x\_for\_left\_finite\_difference) return FiniteDifferences.Left;

else return FiniteDifferences.Center;

}

}

Листинг класса UndefinedCoefficientsDifferentiationFunction

using MathNet.Symbolics;

using NumericalMethods.Core.Differentiation.Interfaces;

namespace NumericalMethods.Core.Differentiation.DifferentiationFunctions.UndefinedCoefficients

{

internal class UndefinedCoefficientsDifferentiationFunction : DifferentiationFunctionBase, IUndefinedCoefficientsDifferentiationFunction

{

private readonly int \_count\_coefficients\_c;

SymbolicExpression[] \_ys\_before\_last\_derivative\_expressions;

SymbolicExpression[] \_ys\_last\_derivative\_expressions;

private double[] \_current\_xs;

private double[] \_current\_ys;

public UndefinedCoefficientsDifferentiationFunction(IEnumerable<IDifferentiationNode> differentiationNodes, double step, int derrivative\_degree, int count\_coefficients\_c)

: base(differentiationNodes, step, derrivative\_degree)

{

\_count\_coefficients\_c = count\_coefficients\_c;

\_current\_xs = differentiationNodes.Select(node => node.X).ToArray();

\_current\_ys = differentiationNodes.Select(node => node.Y).ToArray();

\_ys\_before\_last\_derivative\_expressions = new SymbolicExpression[\_count\_coefficients\_c];

\_ys\_last\_derivative\_expressions = new SymbolicExpression[\_count\_coefficients\_c];

for (int i = 0; i < \_count\_coefficients\_c; i++)

{

\_ys\_before\_last\_derivative\_expressions[i] = $"(x - x0)^{i}";

}

\_ys\_last\_derivative\_expressions = \_ys\_before\_last\_derivative\_expressions.Select(y => y.Differentiate("x")).ToArray();

}

public IEnumerable<IDifferentiationResultNode> Calculate()

{

for (int i = 0; i < \_derivative\_degree; i++)

{

List<double> cs = SolveLinearSystem().ToList();

int start\_new\_items\_index = 1;

int end\_new\_items\_index = \_current\_xs.Length - (\_count\_coefficients\_c - 2);

double[] new\_ys = new double[end\_new\_items\_index];

double[] new\_xs = \_current\_xs[start\_new\_items\_index..end\_new\_items\_index];

for (int j = start\_new\_items\_index; j < end\_new\_items\_index; j++)

{

new\_ys[j - 1] = \_current\_ys[(j - 1)..(j - 1 + \_count\_coefficients\_c)].Zip(cs, (y, c) => y \* c).Sum();

}

\_current\_xs = new\_xs;

\_current\_ys = new\_ys;

}

return \_current\_xs.Zip(\_current\_ys, (x, y) => new DifferentiationResultNode(x, y));

}

private VectorColumn SolveLinearSystem()

{

SquareMatrix matrix = CreateMatrix();

VectorColumn vectorB = CreateVectorB();

SquareMatrix invertedMatrix = matrix.Invert();

return invertedMatrix \* vectorB;

}

private VectorColumn CreateVectorB()

{

IDictionary<string, FloatingPoint> values = new Dictionary<string, FloatingPoint>()

{

{ "x", \_current\_xs[1] },

{ "x0", \_current\_xs[0] }

};

double[] vectorBody = \_ys\_last\_derivative\_expressions

.Select((yExpression, index) => yExpression.Evaluate(values).RealValue)

.ToArray();

return new VectorColumn(vectorBody);

}

private SquareMatrix CreateMatrix()

{

double[,] matrixBody = new double[\_count\_coefficients\_c, \_count\_coefficients\_c];

for (int i = 0; i < \_count\_coefficients\_c; i++)

{

for (int j = 0; j < \_count\_coefficients\_c; j++)

{

IDictionary<string, FloatingPoint> values = new Dictionary<string, FloatingPoint>()

{

{ "x", \_current\_xs[j] },

{ "x0", \_current\_xs[0] }

};

matrixBody[i, j] = \_ys\_before\_last\_derivative\_expressions[i].Evaluate(values).RealValue;

}

}

return new SquareMatrix(matrixBody);

}

}

}

Листинг класса RungeDifferentiationFunction

using NumericalMethods.Core.Differentiation.Interfaces;

namespace NumericalMethods.Core.Differentiation.DifferentiationFunctions.Runge

{

internal class RungeDifferentiationFunction : DifferentiationFunctionBase, IDifferentiationFunction

{

private readonly int \_accuracy\_order;

private readonly int \_number\_of\_used\_points;

public RungeDifferentiationFunction(IEnumerable<IDifferentiationNode> differentiationNodes, double step, int derrivative\_degree, int accuracy\_order, int number\_of\_used\_points)

: base(differentiationNodes, step, derrivative\_degree)

{

\_accuracy\_order = accuracy\_order;

\_number\_of\_used\_points = number\_of\_used\_points;

}

public double? Calculate(double argument)

{

return CalculateRecursive(argument, 1);

}

private double? CalculateRecursive(double argument,int depth)

{

Func<double, double?> f = \_interpolation\_function.Calculate;

double? xh = (f(argument) - f(argument - \_step)) / \_step;

double? xkh = (f(argument) - f(argument - \_step \* \_number\_of\_used\_points)) / (\_step \* \_number\_of\_used\_points);

if (xh == null || xkh == null)

{

xh = (f(argument + \_step) - f(argument)) / \_step;

xkh = (f(argument + \_step \* \_number\_of\_used\_points) - f(argument)) / (\_step \* \_number\_of\_used\_points);

}

int new\_depth = depth + 1;

if(depth != \_derivative\_degree)

{

xh = (CalculateRecursive(argument, new\_depth) - CalculateRecursive(argument - \_step, new\_depth)) / \_step;

xkh = (CalculateRecursive(argument, new\_depth) - CalculateRecursive(argument - \_step \* \_number\_of\_used\_points, new\_depth)) / (\_step \* \_number\_of\_used\_points);

if (xh == null || xkh == null)

{

xh = (CalculateRecursive(argument+\_step,new\_depth) - CalculateRecursive(argument,new\_depth)) / \_step;

xkh = (CalculateRecursive(argument + \_step \* \_number\_of\_used\_points,new\_depth) - CalculateRecursive(argument, new\_depth)) / (\_step \* \_number\_of\_used\_points);

}

}

return xh + (xh - xkh) / (Math.Pow(\_number\_of\_used\_points, \_accuracy\_order) - 1);

}

}

}

Листинг класса IDifferentiationResultNode

namespace NumericalMethods.Core.Differentiation.DifferentiationFunctions.UndefinedCoefficients;

public interface IDifferentiationResultNode

{

public double X { get; init; }

public double Y { get; init; }

}

Листинг класса DifferentiationResultNode

namespace NumericalMethods.Core.Differentiation.DifferentiationFunctions.UndefinedCoefficients;

public record class DifferentiationResultNode(double X, double Y) : IDifferentiationResultNode;

Листинг класса SquareMatrix

using MathNet.Symbolics;

namespace NumericalMethods.Core.Differentiation.DifferentiationFunctions.UndefinedCoefficients;

public class SquareMatrix

{

private readonly double[,] \_data;

public int Size { get; set; }

public SquareMatrix(int size)

{

\_data = new double[size, size];

Size = size;

}

public SquareMatrix(double[,] data)

{

Size = data.GetLength(0);

\_data = new double[Size, Size];

FillDataFrom(data);

}

public double this[int row, int col]

{

get

{

return row >= 0 && row < \_data.GetLength(0) && col >= 0 && col < \_data.GetLength(1)

? \_data[row, col]

: throw new ArgumentOutOfRangeException("row or col invalid");

}

set

{

\_data[row, col] = row >= 0 && row < \_data.GetLength(0) && col >= 0 && col < \_data.GetLength(1)

? value

: throw new ArgumentOutOfRangeException("row or col invalid");

}

}

public static VectorColumn operator \* (SquareMatrix matrix, VectorColumn vector)

{

VectorColumn result = new VectorColumn(vector.Size);

for (int i = 0; i < matrix.Size; i++)

{

double row\_sum = 0;

for (int j = 0; j < matrix.Size; j++)

{

row\_sum += matrix[i, j] \* vector[j];

}

result[i] = row\_sum;

}

return result;

}

public static SquareMatrix operator - (SquareMatrix matrix)

{

for (int i = 0; i < matrix.Size; i++)

{

for (int j = 0; j < matrix.Size; j++)

{

matrix[i, j] = -matrix[i, j];

}

}

return matrix;

}

private void FillDataFrom(double[,] data)

{

for (int i = 0; i < Size; i++)

{

for (int j = 0; j < Size; j++)

{

this[i, j] = data[i, j];

}

}

}

/// <summary> Вычисляет норму матрицы </summary>

public double GetNorm()

{

double max = 0;

for (int i = 0; i < Size; i++)

{

double row\_sum = 0;

for (int j = 0; j < Size; j++)

{

row\_sum += Math.Abs(this[i, j]);

}

if (row\_sum > max) max = row\_sum;

}

return max;

}

/// <summary> Вычисляет определитель матрицы </summary>

public double GetDeterminant()

{

int size = \_data.GetLength(0);

switch (size)

{

case 1: return \_data[0, 0];

case 2: return (\_data[0, 0] \* \_data[1, 1]) - (\_data[1, 0] \* \_data[0, 1]);

default:

double determinant = 0;

for (int column = 0; column < size; column++)

{

determinant += (column % 2 == 0 ? 1 : -1) \* \_data[0, column] \* CreateMatrixWithoutRowAndColumn(0, column).GetDeterminant();

}

return determinant;

}

}

/// <summary> Создает урезанную матрицу </summary>

/// <param name="index\_row">Удаляемая строка</param>

/// <param name="index\_column">Удаляемый столбец</param>

/// <returns>Новая урезанная матрица</returns>

public SquareMatrix CreateMatrixWithoutRowAndColumn(int index\_row, int index\_column)

{

SquareMatrix new\_matrix = new SquareMatrix(Size - 1);

List<List<double>> elements\_new\_matrix = new List<List<double>>();

for (int i = 0; i < this.Size; i++)

{

if (i == index\_row) continue;

elements\_new\_matrix.Add(new List<double>());

for (int j = 0; j < this.Size; j++)

{

if (j == index\_column) continue;

elements\_new\_matrix.Last().Add(this[i, j]);

}

}

for (int i = 0; i < new\_matrix.Size; i++)

{

for (int j = 0; j < new\_matrix.Size; j++)

{

new\_matrix[i, j] = elements\_new\_matrix[i][j];

}

}

return new\_matrix;

}

/// <summary> Инвертирует текущую матрицу</summary>

public SquareMatrix Invert()

{

double determinant = this.GetDeterminant();

SquareMatrix transposed = this.CreateAlgebraicAddition().Transpose();

for (int i = 0; i < Size; i++)

{

for (int j = 0; j < Size; j++)

{

this[i, j] = transposed[i, j] / determinant;

}

}

return this;

}

/// <summary> Создает алгебраическое дополнение </summary>

/// <returns>Новая матрица - алгебраическое дополнение текущей</returns>

public SquareMatrix CreateAlgebraicAddition()

{

SquareMatrix algebraic\_addition\_matrix = new SquareMatrix(Size);

for (int i = 0; i < Size; i++)

{

for (int j = 0; j < Size; j++)

{

algebraic\_addition\_matrix[i, j] = ((i + j) % 2 is 0 ? 1 : -1) \* CreateMatrixWithoutRowAndColumn(i, j).GetDeterminant();

}

}

return algebraic\_addition\_matrix;

}

/// <summary> Транспонирует текущую матрицу </summary>

public SquareMatrix Transpose()

{

double[,] new\_data = new double[Size, Size];

for (int i = 0; i < Size; i++)

{

for (int j = 0; j < Size; j++)

{

new\_data[i, j] = this[j, i];

}

}

FillDataFrom(new\_data);

return this;

}

/// <summary> Создает матрицу Якоби </summary>

public static SquareMatrix CreateJacobiMatrix(IEnumerable<SymbolicExpression> functions, Dictionary<string, FloatingPoint> values)

{

var variables = functions.First().CollectVariables();

int count\_params = variables.Count();

SquareMatrix resultMatrix = new SquareMatrix(count\_params);

for (int i = 0; i < resultMatrix.Size; i++)

{

for (int j = 0; j < resultMatrix.Size; j++)

{

string currentVariableName = variables.ElementAt(j).ToString();

resultMatrix[i, j] = functions.ElementAt(i)

.Differentiate(variables.ElementAt(j))

.Evaluate(values).RealValue;

}

}

return resultMatrix;

}

}

Листинг класса VectorBase

namespace NumericalMethods.Core.Differentiation.DifferentiationFunctions.UndefinedCoefficients;

public abstract class VectorBase

{

private readonly double[] \_data;

public VectorBase(double[] data)

{

\_data = data;

}

public VectorBase(int size)

{

\_data = new double[size];

}

public double this[int index]

{

get => \_data[index];

set => \_data[index] = value;

}

public double GetNormM()

{

return \_data.Select(vector => Math.Abs(vector)).Max();

}

public List<double> ToList()

{

return \_data.ToList();

}

public int Size { get => \_data.Length; }

}

Листинг класса VectorColumn

namespace NumericalMethods.Core.Differentiation.DifferentiationFunctions.UndefinedCoefficients

{

public class VectorColumn : VectorBase

{

public VectorColumn(int size) : base(size)

{

}

public VectorColumn(double[] data) : base(data)

{

}

public static VectorColumn operator -(VectorColumn vector)

{

for (int i = 0; i < vector.Size; i++)

{

vector[i] = -vector[i];

}

return vector;

}

public static VectorColumn operator +(VectorColumn vectorOne, VectorColumn vectorTwo)

{

if (vectorOne.Size > vectorTwo.Size)

{

VectorColumn result = new VectorColumn(vectorOne.Size);

for (int i = 0; i < vectorTwo.Size; i++)

{

result[i] = vectorOne[i] + vectorTwo[i];

}

for (int i = vectorTwo.Size; i < result.Size; i++)

{

result[i] = vectorOne[i];

}

return result;

} else

{

VectorColumn result = new VectorColumn(vectorTwo.Size);

for (int i = 0; i < vectorOne.Size; i++)

{

result[i] = vectorOne[i] + vectorTwo[i];

}

for (int i = vectorOne.Size; i < result.Size; i++)

{

result[i] = vectorTwo[i];

}

return result;

}

}

public static VectorColumn operator -(VectorColumn vectorOne, VectorColumn vectorTwo)

{

return vectorOne + -vectorTwo;

}

}

}

Листинг класса VectorRow

namespace NumericalMethods.Core.Differentiation.DifferentiationFunctions.UndefinedCoefficients;

public class VectorRow : VectorBase

{

public VectorRow(int size) : base(size)

{

}

public VectorRow(double[] data) : base(data)

{

}

}

Листинг класса MainWindow.xaml

using NumericalMethods.Core.Approximation;

using NumericalMethods.Core.Approximation.Interfaces;

using NumericalMethods.Core.Differentiation;

using NumericalMethods.Core.Differentiation.DifferentiationFunctions.UndefinedCoefficients;

using NumericalMethods.Core.Differentiation.Interfaces;

using NumericalMethods.Infrastructure.Integration;

using NumericalMethods.Infrastructure.Integration.Interfaces;

using NumericalMethods.Infrastructure.NonLinearEquationsSystems;

using NumericalMethods.Core.CauchyProblem;

using NumericalMethods.Infrastructure.NonLinearEquationsSystems.Methods;

using org.mariuszgromada.math.mxparser;

using ScottPlot;

using System;

using System.Collections.Generic;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows;

using System.Windows.Controls;

using System.Windows.Input;

using Expression = org.mariuszgromada.math.mxparser.Expression;

using Point = NumericalMethods.WPFApplication.Differentiation.Point;

namespace NumericalMethods.WPFApplication

{

/// <summary>

/// Interaction logic for MainWindow.xaml

/// </summary>

public partial class MainWindow : Window

{

private readonly List<Point> \_points = new List<Point>();

private Function \_current\_function;

private double \_start\_x;

private double \_end\_x;

private double \_step;

private int \_order;

private ResultTable \_resultTable;

public MainWindow()

{

InitializeComponent();

Width = 1300;

Height = 750;

InitializeDifferentiation();

}

private void InitializeDifferentiation()

{

foreach (var item in Enum.GetValues(typeof(InterpolationFunctionType)))

{

Differentiation\_InterpolationFunctionTypeComboBox.Items.Add(item.ToString());

}

Differentiation\_InterpolationFunctionTypeComboBox.SelectedItem = Differentiation\_InterpolationFunctionTypeComboBox.Items[0];

foreach (var item in Enum.GetValues(typeof(DifferentiationFunctionType)))

{

Differentiation\_FunctionTypeComboBox.Items.Add(item.ToString());

}

Differentiation\_FunctionTypeComboBox.SelectedItem = Differentiation\_InterpolationFunctionTypeComboBox.Items[0];

Differentiation\_FunctionTextBox.Text = "x^2";

Differentiation\_StartXTextBox.Text = "-3";

Differentiation\_EndXTextBox.Text = "3";

Differentiation\_StepTextBox.Text = "1";

Differentiation\_NumberOfMembers.Text = "2";

DerivativeDegreeTextBox.Text = "1";

Differentiation\_MainChart.Plot.Legend(enable: true);

}

private void Differentiation\_AddNodesOnChart\_Click(object sender, RoutedEventArgs e)

{

if (String.IsNullOrEmpty(Differentiation\_FunctionTextBox.Text) ||

String.IsNullOrEmpty(Differentiation\_StartXTextBox.Text) ||

String.IsNullOrEmpty(Differentiation\_EndXTextBox.Text) ||

String.IsNullOrEmpty(Differentiation\_StepTextBox.Text))

{

MessageBox.Show("Для начала рассчёта, заполните все поля");

return;

}

\_points.Clear();

\_current\_function = new Function("f(x) = " + Differentiation\_FunctionTextBox.Text.Trim());

double start\_x = new Expression(Differentiation\_StartXTextBox.Text.Trim()).calculate();

double end\_x = new Expression(Differentiation\_EndXTextBox.Text.Trim()).calculate();

double step = new Expression(Differentiation\_StepTextBox.Text.Trim()).calculate();

int roundNumbers = step.ToString().Contains(',') ? step.ToString().Split(',')[1].Length % 16 : 1;

for (double x = start\_x; Math.Round(x, roundNumbers) <= end\_x; x += step)

{

double y = \_current\_function.calculate(x);

\_points.Add(new Point(x, y));

}

double[] xs = \_points.Select(node => node.X).ToArray();

double[] ys = \_points.Select(node => node.Y).ToArray();

Differentiation\_MainChart.Plot.AddScatter(xs, ys, lineWidth: 0, markerShape: MarkerShape.filledCircle, color: System.Drawing.Color.Red, markerSize: 7, label: "nodes");

Differentiation\_MainChart.Refresh();

}

private void Differentiation\_FunctionTypeComboBox\_SelectionChanged(object sender, SelectionChangedEventArgs e)

{

Differentiation\_AccuracyGrid.Visibility = Visibility.Collapsed;

Differentiation\_NumberOfMembersGrid.Visibility = Visibility.Collapsed;

Differentiation\_Accuracy.Text = "";

Differentiation\_NumberOfMembers.Text = "";

DifferentiationFunctionType differentiationFunctionType = (DifferentiationFunctionType)Enum.Parse(typeof(DifferentiationFunctionType), ((ComboBox)sender).SelectedValue.ToString());

if(differentiationFunctionType == DifferentiationFunctionType.NewtonPolynomials)

{

Differentiation\_NumberOfMembersGrid.Visibility = Visibility.Visible;

}

if(differentiationFunctionType == DifferentiationFunctionType.Runge || differentiationFunctionType == DifferentiationFunctionType.UndefinedCoefficients)

{

Differentiation\_AccuracyGrid.Visibility = Visibility.Visible;

}

}

private void Differentiation\_AddOnChartInterpolationButton\_Click(object sender, RoutedEventArgs e)

{

if(\_points.Count == 0)

{

MessageBox.Show("Добавьте точки");

return;

}

string? function\_type\_string = Differentiation\_InterpolationFunctionTypeComboBox.SelectedValue.ToString();

if (function\_type\_string is null) return;

InterpolationFunctionType interpolation\_type = Enum.Parse<InterpolationFunctionType>(function\_type\_string);

IInterpolationFunction? interpolation\_function = InterpolationBuilder.Build(\_points, interpolation\_type);

if (interpolation\_function is null) return;

var xs = new List<double>();

var ys = new List<double>();

double start\_x = new Expression(Differentiation\_StartXTextBox.Text.Trim()).calculate();

double end\_x = new Expression(Differentiation\_EndXTextBox.Text.Trim()).calculate();

double step = new Expression(Differentiation\_StepTextBox.Text.Trim()).calculate();

int roundNumbers = step.ToString().Contains(',') ? step.ToString().Split(',')[1].Length % 16 : 1;

for (double x = start\_x; Math.Round(x, roundNumbers) <= end\_x; x += step)

{

var y = interpolation\_function.Calculate(x);

if (y is null) continue;

xs.Add(x);

ys.Add((double)y);

}

Differentiation\_MainChart.Plot.AddScatter(xs.ToArray(), ys.ToArray(), lineWidth: 4, markerSize: 0, label: "interpolation");

Differentiation\_MainChart.Refresh();

Differentiation\_MainChart.Refresh();

}

private void Differentiation\_AddOnChartButton\_Click(object sender, RoutedEventArgs e)

{

if (\_points.Count == 0)

{

MessageBox.Show("Добавьте точки");

return;

}

var xs = new List<double>();

var ys = new List<double>();

\_start\_x = new Expression(Differentiation\_StartXTextBox.Text.Trim()).calculate();

\_end\_x = new Expression(Differentiation\_EndXTextBox.Text.Trim()).calculate();

\_step = new Expression(Differentiation\_StepTextBox.Text.Trim()).calculate();

int roundNumbers = \_step.ToString().Contains(',') ? \_step.ToString().Split(',')[1].Length % 16 : 1;

int derivative\_degree = int.Parse(DerivativeDegreeTextBox.Text.Trim());

string? function\_type\_string = Differentiation\_FunctionTypeComboBox.SelectedValue.ToString();

if (function\_type\_string is null) return;

DifferentiationFunctionType interpolation\_type = (DifferentiationFunctionType)Enum.Parse(typeof(DifferentiationFunctionType), function\_type\_string);

switch (interpolation\_type)

{

case DifferentiationFunctionType.NewtonPolynomials:

if (String.IsNullOrEmpty(Differentiation\_NumberOfMembers.Text))

{

MessageBox.Show("Введите number of members");

return;

}

int numberOfMembers = int.Parse(Differentiation\_NumberOfMembers.Text);

INewtonDifferentiationFunction newton\_function = DifferentiationBuilder.CreateNewton(\_points, \_step, derivative\_degree, numberOfMembers);

for (double x = \_start\_x; Math.Round(x, roundNumbers) <= \_end\_x; x += \_step)

{

double? y = newton\_function.Calculate(x);

if (y is null) continue;

xs.Add(x);

ys.Add((double)y);

}

break;

case DifferentiationFunctionType.UndefinedCoefficients:

if (String.IsNullOrEmpty(Differentiation\_Accuracy.Text))

{

MessageBox.Show("Введите accuracy");

return;

}

int count\_coefficients\_c = int.Parse(Differentiation\_Accuracy.Text);

List<Point> additional\_right\_points = new List<Point>();

for (int i = 1; i <= (count\_coefficients\_c - 2) \* derivative\_degree; i++)

{

double new\_x = \_end\_x + \_step \* i;

double new\_y = \_current\_function.calculate(new\_x);

Point additional\_point = new Point(new\_x, new\_y);

additional\_right\_points.Add(additional\_point);

}

List<Point> additional\_left\_point = new List<Point>();

for (int i = derivative\_degree; i >= 1; i--)

{

double new\_x = \_start\_x - \_step \* i;

double new\_y = \_current\_function.calculate(new\_x);

Point additional\_point = new Point(new\_x, new\_y);

additional\_left\_point.Add(additional\_point);

}

IEnumerable<Point> new\_points = additional\_left\_point.Concat(\_points.Concat(additional\_right\_points));

IUndefinedCoefficientsDifferentiationFunction undefined\_coefficients\_function = DifferentiationBuilder.CreateUndefinedCoefficients(new\_points, \_step, derivative\_degree, count\_coefficients\_c);

IEnumerable<IDifferentiationResultNode> result = undefined\_coefficients\_function.Calculate();

xs = result.Select(node => node.X).ToList();

ys = result.Select(node => node.Y).ToList();

break;

case DifferentiationFunctionType.Runge:

{

if (String.IsNullOrEmpty(Differentiation\_Accuracy.Text))

{

MessageBox.Show("Введите accuracy");

return;

}

IDifferentiationFunction differentiation\_function = DifferentiationBuilder.CreateRunge(\_points, \_step, derivative\_degree, 1, 2);

for (double x = \_start\_x; Math.Round(x, roundNumbers) <= \_end\_x; x += \_step)

{

double? y = differentiation\_function.Calculate(x);

if (y is null) continue;

xs.Add(x);

ys.Add((double)y);

}

break;

}

default:

{

IDifferentiationFunction differentiation\_function = DifferentiationBuilder.Build(\_points, interpolation\_type, \_step, derivative\_degree);

for (double x = \_start\_x; Math.Round(x, roundNumbers) <= \_end\_x; x += \_step)

{

double? y = differentiation\_function.Calculate(x);

if (y is null) continue;

xs.Add(x);

ys.Add((double)y);

}

break;

}

}

Differentiation\_MainChart.Plot.AddScatter(xs.ToArray(), ys.ToArray(), lineWidth: 4, markerSize: 0, label: "differentiation");

Differentiation\_MainChart.Refresh();

}

private void Differentiation\_ClearChartButton\_Click(object sender, RoutedEventArgs e)

{

Differentiation\_MainChart.Plot.Clear();

Differentiation\_MainChart.Refresh();

}

}

}

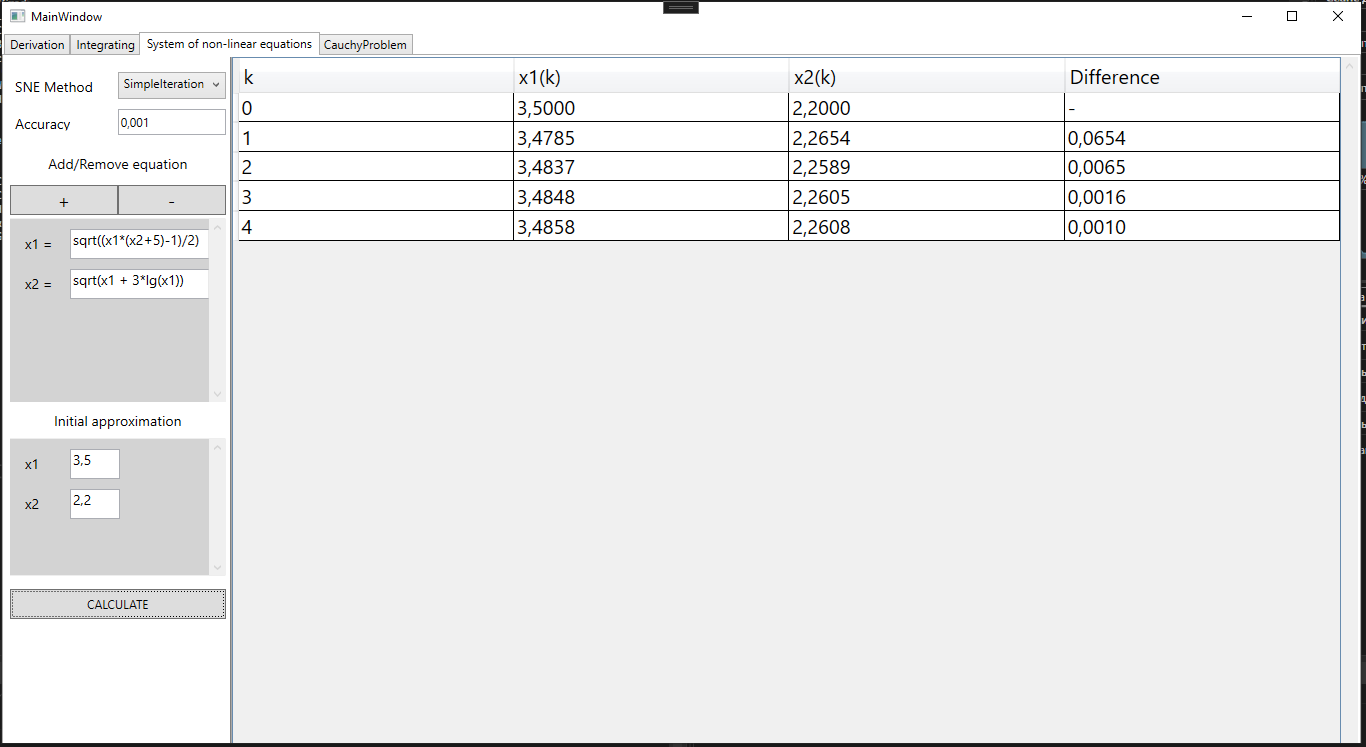
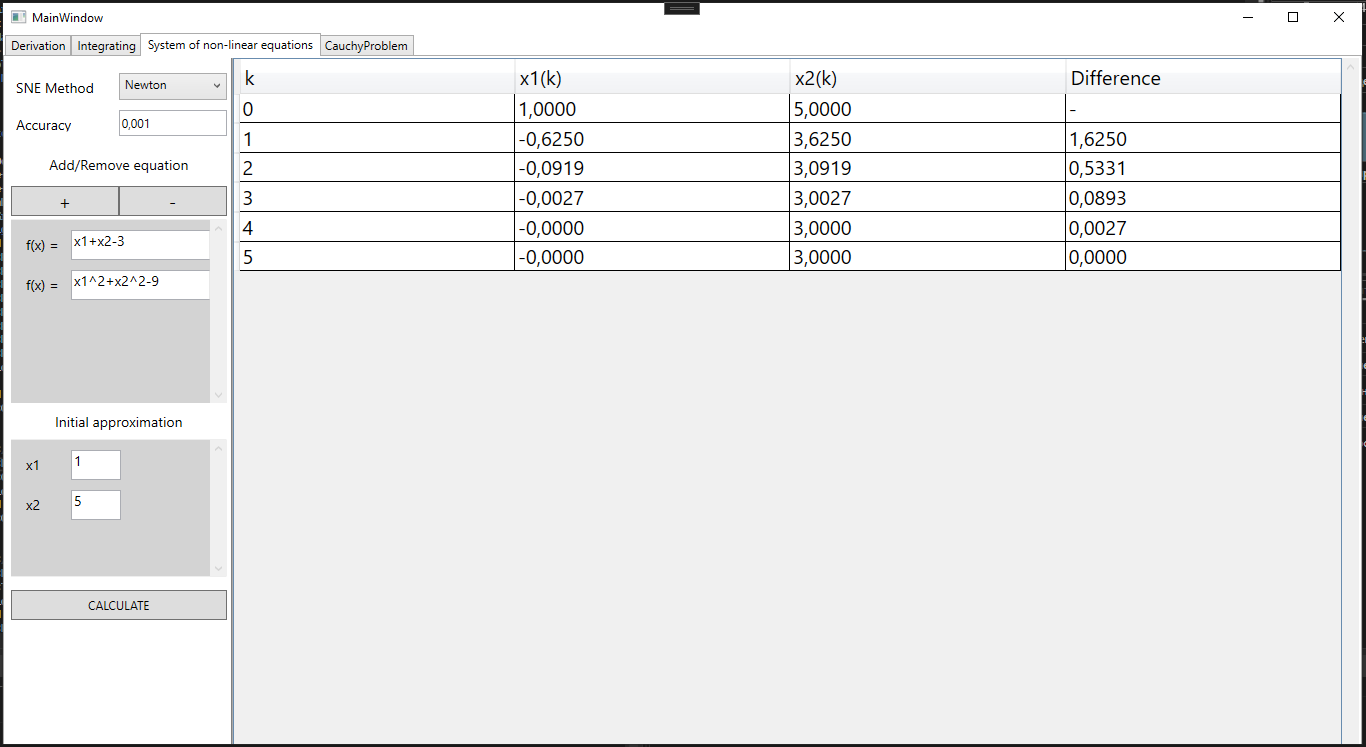
**Вывод:** в ходе выполнения лабораторной работы были изучены методы численного дифференцирования, вычисление первой и второй производных заданной функции с использованием интерполяционного многочлена Ньютона.

## Лабораторная работа №2

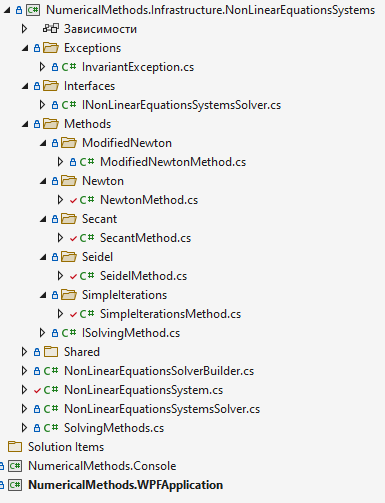
### «Решение систем нелинейных уравнений»

**Цель работы**: программная реализация методов решения систем нелинейных уравнений: метод простых итераций, метод Зейделя, метод Ньютона, модифицированный метод Ньютона, метод секущих.

**Скриншоты работы программы**



**Структура проекта**



**Листинг работы программы**

Листинг класса ISolvingMethod

using MathNet.Symbolics;

namespace NumericalMethods.Infrastructure.NonLinearEquationsSystems.Methods;

public interface ISolvingMethod

{

public IEnumerable<IEnumerable<double>> SolveWithSteps(NonLinearEquationsSystem system, double eps, Dictionary<string,FloatingPoint> initialGuess);

}

Листинг класса SimpleIterationsMethod

using MathNet.Symbolics;

using NumericalMethods.Infrastructure.NonLinearEquationsSystems.Shared;

namespace NumericalMethods.Infrastructure.NonLinearEquationsSystems.Methods.SimpleIterations

{

public class SimpleIterationsMethod : ISolvingMethod

{

private List<SymbolicExpression> functionExpressions;

private double eps;

private Dictionary<string, FloatingPoint> currentValues;

private List<double> differences;

public IEnumerable<IEnumerable<double>> SolveWithSteps(NonLinearEquationsSystem system, double eps, Dictionary<string,FloatingPoint> initialGuess)

{

Dictionary<string, FloatingPoint> sortedInitialGuess = initialGuess.OrderBy(x => x.Key).ToDictionary(el => el.Key, el => el.Value);

if (Math.Abs(SquareMatrix.CreateJacobiMatrix(system.FunctionExpressions,initialGuess).GetNorm()) >= 1)

throw new Exception("Данное приближение не подходит под условие сходимости попробуйте дургое");

this.currentValues = sortedInitialGuess;

this.functionExpressions = system.FunctionExpressions.ToList();

this.eps = eps;

differences = new List<double>() { double.MinValue };

List<List<double>> results = (List<List<double>>)SolveRecursive(new List<List<double>>() { sortedInitialGuess.Select(el=> el.Value.RealValue).ToList() });

for(int i = 0; i < results.Count; i++)

{

results[i].Add(differences[i]);

}

return results;

}

private IEnumerable<IEnumerable<double>> SolveRecursive(List<List<double>> results)

{

for (int i = 0; i < results.Last().Count; i++)

currentValues[currentValues.ElementAt(i).Key] = results.Last()[i];

results.Add(new List<double>());

functionExpressions.ForEach((functionExpression) => results.Last().Add(functionExpression.Evaluate(currentValues).RealValue));

differences.Add(FindMaxDifference(results.ElementAt(results.Count - 1), results.ElementAt(results.Count - 2)));

if (differences.Last() > eps){

results = (List<List<double>>)SolveRecursive(results);

}

return results;

}

private double FindMaxDifference(List<double> nextValues, List<double> currentValues)

{

return nextValues.Zip(currentValues, (nextValue, currentValue) =>

{

return Math.Abs(nextValue - currentValue);

}).Max();

}

}

}

Листинг класса SeidelMethod

using MathNet.Symbolics;

using NumericalMethods.Infrastructure.NonLinearEquationsSystems.Shared;

namespace NumericalMethods.Infrastructure.NonLinearEquationsSystems.Methods.Seidel

{

internal class SeidelMethod : ISolvingMethod

{

private List<SymbolicExpression> functionExpressions;

private double eps;

private Dictionary<string, FloatingPoint> currentValues;

private List<double> differences;

public IEnumerable<IEnumerable<double>> SolveWithSteps(NonLinearEquationsSystem system, double eps, Dictionary<string, FloatingPoint> initialGuess)

{

Dictionary<string, FloatingPoint> sortedInitialGuess = initialGuess.OrderBy(x => x.Key).ToDictionary(el => el.Key, el => el.Value);

if (Math.Abs(SquareMatrix.CreateJacobiMatrix(system.FunctionExpressions, initialGuess).GetNorm()) >= 1)

throw new Exception("Данное приближение не подходит под условие сходимости попробуйте дургое");

this.currentValues = sortedInitialGuess;

this.functionExpressions = system.FunctionExpressions.ToList();

this.eps = eps;

differences = new List<double>() { double.MinValue };

List<List<double>> results = (List<List<double>>)SolveRecursive(new List<List<double>>() { sortedInitialGuess.Select(el => el.Value.RealValue).ToList() });

for (int i = 0; i < results.Count; i++)

{

results[i].Add(differences[i]);

}

return results;

}

private IEnumerable<IEnumerable<double>> SolveRecursive(List<List<double>> results)

{

for (int i = 0; i < results.Last().Count; i++)

currentValues[currentValues.ElementAt(i).Key] = results.Last()[i];

results.Add(new List<double>());

int countFunctionExpressions = functionExpressions.Count();

for (int i = 0; i < countFunctionExpressions; i++)

{ results.Last().Add(functionExpressions.ElementAt(i).Evaluate(currentValues).RealValue);

currentValues[currentValues.ElementAt(i).Key] = results.Last().Last();

}

differences.Add(FindMaxDifference(results.ElementAt(results.Count - 1), results.ElementAt(results.Count - 2)));

if (differences.Last() > eps)

{

results = (List<List<double>>)SolveRecursive(results);

}

return results;

}

private double FindMaxDifference(List<double> nextValues, List<double> currentValues)

{

return nextValues.Zip(currentValues, (nextValue, currentValue) =>

{

return Math.Abs(nextValue - currentValue);

}).Max();

}

}

}

Листинг класса NewtonMethod

using MathNet.Symbolics;

using NumericalMethods.Infrastructure.NonLinearEquationsSystems.Shared;

using System.Xml.Linq;

namespace NumericalMethods.Infrastructure.NonLinearEquationsSystems.Methods.Newton;

public class NewtonMethod : ISolvingMethod

{

IEnumerable<IEnumerable<double>> ISolvingMethod.SolveWithSteps(NonLinearEquationsSystem system, double eps, Dictionary<string,FloatingPoint> initialGuess)

{

Dictionary<string, FloatingPoint> sortedInitialGuess = initialGuess.OrderBy(x => x.Key).ToDictionary(el => el.Key, el => el.Value);

VectorColumn lastVectorX = new VectorColumn(sortedInitialGuess.Select(el => el.Value.RealValue).ToArray());

double delta = double.MaxValue;

List<List<double>> results = new List<List<double>>() { lastVectorX.ToList()};

results.Last().Add(double.MinValue);

while (delta > eps)

{

Dictionary<string, FloatingPoint> values = new();

for (int i = 0; i < lastVectorX.Size; i++)

{

values.Add(sortedInitialGuess.ElementAt(i).Key, lastVectorX[i]);

}

VectorColumn? yVector = new VectorColumn(system.FunctionExpressions

.Select(function => function.Evaluate(values))

.Select(fp => fp.RealValue)

.ToArray());

VectorColumn newVectorX = lastVectorX + -SquareMatrix.CreateJacobiMatrix(system.FunctionExpressions, values).Invert() \* yVector;

delta = (newVectorX - lastVectorX).GetNormM();

List<double> result = newVectorX.ToList();

result.Add(delta);

results.Add(result);

lastVectorX = newVectorX;

}

return results;

}

}

Листинг класса ModifiedNewtonMethod

using MathNet.Symbolics;

using NumericalMethods.Infrastructure.NonLinearEquationsSystems.Shared;

namespace NumericalMethods.Infrastructure.NonLinearEquationsSystems.Methods.ModifiedNewton

{

internal class ModifiedNewtonMethod:ISolvingMethod

{

IEnumerable<IEnumerable<double>> ISolvingMethod.SolveWithSteps(NonLinearEquationsSystem system, double eps, Dictionary<string, FloatingPoint> initialGuess)

{

Dictionary<string, FloatingPoint> sortedInitialGuess = initialGuess.OrderBy(x => x.Key).ToDictionary(el => el.Key, el => el.Value);

VectorColumn lastVectorX = new VectorColumn(sortedInitialGuess.Select(el => el.Value.RealValue).ToArray());

SquareMatrix minusInternalJacobiMatrix = -SquareMatrix.CreateJacobiMatrix(system.FunctionExpressions, initialGuess).Invert();

double delta = double.MaxValue;

List<List<double>> results = new List<List<double>>() { lastVectorX.ToList() };

results.Last().Add(double.MinValue);

while (delta > eps)

{

Dictionary<string, FloatingPoint> values = new();

for (int i = 0; i < lastVectorX.Size; i++)

{

values.Add(sortedInitialGuess.ElementAt(i).Key, lastVectorX[i]);

}

VectorColumn? yVector = new VectorColumn(system.FunctionExpressions

.Select(function => function.Evaluate(values))

.Select(fp => fp.RealValue)

.ToArray());

VectorColumn newVectorX = lastVectorX + minusInternalJacobiMatrix \* yVector;

delta = (newVectorX - lastVectorX).GetNormM();

List<double> result = newVectorX.ToList();

result.Add(delta);

results.Add(result);

lastVectorX = newVectorX;

}

return results;

}

}

}

Листинг класса SecantMethod

using MathNet.Symbolics;

using NumericalMethods.Infrastructure.NonLinearEquationsSystems.Shared;

namespace NumericalMethods.Infrastructure.NonLinearEquationsSystems.Methods.Secant

{

internal class SecantMethod:ISolvingMethod

{

IEnumerable<IEnumerable<double>> ISolvingMethod.SolveWithSteps(NonLinearEquationsSystem system, double eps, Dictionary<string, FloatingPoint> initialGuess)

{

Dictionary<string, FloatingPoint> sortedInitialGuess = initialGuess.OrderBy(x => x.Key).ToDictionary(el=>el.Key,el=>el.Value);

VectorColumn lastVectorX = new VectorColumn(sortedInitialGuess.Select(el => el.Value.RealValue).ToArray());

SquareMatrix lastA = SquareMatrix.CreateJacobiMatrix(system.FunctionExpressions, sortedInitialGuess);

List<List<double>> results = new List<List<double>>() { lastVectorX.ToList() };

results.Last().Add(double.MinValue);

VectorColumn? YVector = new VectorColumn(system.FunctionExpressions

.Select(function => function.Evaluate(sortedInitialGuess))

.Select(fp => fp.RealValue)

.ToArray());

VectorColumn s = -lastA.Invert() \* YVector;

VectorColumn newVectorX = lastVectorX + s;

double delta = s.GetNormM();

List<double> result = newVectorX.ToList();

result.Add(delta);

results.Add(result);

lastVectorX = newVectorX;

VectorColumn? newYVector;

VectorColumn deltaYVector;

SquareMatrix newA;

while (delta > eps)

{

for (int i = 0; i < lastVectorX.Size; i++)

{

sortedInitialGuess[sortedInitialGuess.ElementAt(i).Key] = lastVectorX[i];

}

newYVector = new VectorColumn(system.FunctionExpressions

.Select(function => function.Evaluate(sortedInitialGuess))

.Select(fp => fp.RealValue)

.ToArray());

deltaYVector = newYVector - YVector;

newA = lastA + ((deltaYVector - lastA \* s) \* s.Transposition()/(s.Transposition() \* s).Value);

s = -newA.Invert() \* newYVector;

newVectorX = lastVectorX + s;

delta = s.GetNormM();

result = newVectorX.ToList();

result.Add(delta);

results.Add(result);

lastA = newA;

YVector = newYVector;

lastVectorX = newVectorX;

}

return results;

}

}

}

Листинг класса INonLinearEquationsSystemsSolver

using MathNet.Symbolics;

namespace NumericalMethods.Infrastructure.NonLinearEquationsSystems.Interfaces;

public interface INonLinearEquationsSystemsSolver

{

public IEnumerable<IEnumerable<double>> SolveWithSteps(NonLinearEquationsSystem system, double eps, Dictionary<string, FloatingPoint> initialGuess);

}

Листинг класса NonLinearEquationsSolverBuilder

using NumericalMethods.Infrastructure.NonLinearEquationsSystems.Interfaces;

using NumericalMethods.Infrastructure.NonLinearEquationsSystems.Methods.Newton;

using NumericalMethods.Infrastructure.NonLinearEquationsSystems.Methods.SimpleIterations;

using NumericalMethods.Infrastructure.NonLinearEquationsSystems.Methods.Seidel;

using NumericalMethods.Infrastructure.NonLinearEquationsSystems.Methods.ModifiedNewton;

using NumericalMethods.Infrastructure.NonLinearEquationsSystems.Methods.Secant;

namespace NumericalMethods.Infrastructure.NonLinearEquationsSystems;

public record class NonLinearEquationsSolverBuilder()

{

public INonLinearEquationsSystemsSolver Build(SolvingMethods SolvingMethod)

{

return SolvingMethod switch

{

SolvingMethods.Newton => new NonLinearEquationsSystemsSolver(new NewtonMethod()),

SolvingMethods.SimpleIterations => new NonLinearEquationsSystemsSolver(new SimpleIterationsMethod()),

SolvingMethods.Seidel => new NonLinearEquationsSystemsSolver(new SeidelMethod()),

SolvingMethods.ModifiedNewton => new NonLinearEquationsSystemsSolver(new ModifiedNewtonMethod()),

SolvingMethods.Secant => new NonLinearEquationsSystemsSolver(new SecantMethod()),

\_ => throw new NotImplementedException()

};

}

}

Листинг класса NonLinearEquationsSystem

using MathNet.Symbolics;

using NumericalMethods.Infrastructure.NonLinearEquationsSystems.Exceptions;

using System.Runtime.CompilerServices;

namespace NumericalMethods.Infrastructure.NonLinearEquationsSystems

{

public class NonLinearEquationsSystem

{

public NonLinearEquationsSystem(IEnumerable<string> functions)

{

Functions = functions;

FunctionExpressions = functions

.Select(function => SymbolicExpression.Parse(function))

.ToList();

CheckInvariants();

}

private void CheckInvariants()

{

InvariantException.ThrowIf(

isViolated: CheckFunctionsIsNotNull(),

message: $"{nameof(Functions)} не должно быть null");

InvariantException.ThrowIf(

isViolated: CheckFunctionsHaveAnyEquation(),

message: $"{nameof(Functions)} должно иметь хотябы одно уравнение");

InvariantException.ThrowIf(

isViolated: CheckNumberVariablesEqualsNumberEquations(),

message: $"Количество аргументов в {nameof(Functions)} должно совпадать с количеством уравнений");

}

private bool CheckFunctionsIsNotNull()

{

return Functions is null;

}

private bool CheckFunctionsHaveAnyEquation()

{

return !Functions.Any();

}

private bool CheckNumberVariablesEqualsNumberEquations()

{

return FunctionExpressions.First().CollectVariables().Count() != FunctionExpressions.Count();

}

public IEnumerable<string> Functions { get; }

public IEnumerable<SymbolicExpression> FunctionExpressions { get;}

}

}

Листинг класса NonLinearEquationsSystemsSolver

using NumericalMethods.Infrastructure.NonLinearEquationsSystems.Interfaces;

using NumericalMethods.Infrastructure.NonLinearEquationsSystems.Methods;

using MathNet.Symbolics;

namespace NumericalMethods.Infrastructure.NonLinearEquationsSystems;

public record class NonLinearEquationsSystemsSolver(ISolvingMethod SolvingMethod) : INonLinearEquationsSystemsSolver

{

public IEnumerable<IEnumerable<double>> SolveWithSteps(NonLinearEquationsSystem system, double eps, Dictionary<string, FloatingPoint> initialGuess)

{

return SolvingMethod.SolveWithSteps(system, eps, initialGuess);

}

}

Листинг класса SolvingMethods

namespace NumericalMethods.Infrastructure.NonLinearEquationsSystems;

public enum SolvingMethods

{

SimpleIterations,

Seidel,

Newton,

ModifiedNewton,

Secant

}

Листинг класса MainWindow.xaml.cs

namespace NumericalMethods.WPFApplication

{

/// <summary>

/// Interaction logic for MainWindow.xaml

/// </summary>

public partial class MainWindow : Window

{

private readonly List<Point> \_points = new List<Point>();

private Function \_current\_function;

private double \_start\_x;

private double \_end\_x;

private double \_step;

private int \_order;

private ResultTable \_resultTable;

public MainWindow()

{

InitializeComponent();

Width = 1300;

Height = 750;

InitializeNonLinerEquation();

}

private void InitializeNonLinerEquation()

{

Enum.GetNames(typeof(SolvingMethods)).ToList().ForEach(nameSolvingMethod =>

{

SNE\_methodComboBox.Items.Add(nameSolvingMethod);

});

SNE\_methodComboBox.SelectedItem = SNE\_methodComboBox.Items[0];

if (SNE\_systemOfEquationsGrid.RowDefinitions.Count == 0)

SNE\_RemoveEquationButton.IsEnabled = false;

}

private void InitialDefaulSystemOfEquations()

{

Canvas SNE\_equationsCanvas;

int start\_position = 10;

if (SNE\_systemOfEquationsGrid.RowDefinitions.Count < 1)

start\_position = 10;

else

start\_position += 40 \* SNE\_systemOfEquationsGrid.RowDefinitions.Count;

RowDefinition rowDefinition = new RowDefinition();

rowDefinition.Height = new GridLength(52, GridUnitType.Pixel);

SNE\_equationsCanvas = new Canvas();

SNE\_systemOfEquationsGrid.RowDefinitions.Add(rowDefinition);

SNE\_systemOfEquationsGrid.Children.Add(SNE\_equationsCanvas);

Label equationLabel = new Label();

equationLabel.Content = "f(x) = ";

equationLabel.FontSize = 14;

equationLabel.Margin = new Thickness(10, start\_position, 0, 0);

SNE\_equationsCanvas.Children.Add(equationLabel);

TextBox equationOfSystem = new TextBox();

equationOfSystem.Margin = new Thickness(60, start\_position, 20, 0);

equationOfSystem.FontSize = 14;

equationOfSystem.Width = 200;

equationOfSystem.Height = 30;

SNE\_equationsCanvas.Children.Add(equationOfSystem);

}

private void InitialOtherSystemOfEquations()

{

Canvas SNE\_equationsCanvas;

int start\_position = 10;

if (SNE\_systemOfEquationsGrid.RowDefinitions.Count < 1)

start\_position = 10;

else

start\_position += 40 \* SNE\_systemOfEquationsGrid.RowDefinitions.Count;

RowDefinition rowDefinition = new RowDefinition();

rowDefinition.Height = new GridLength(52, GridUnitType.Pixel);

SNE\_equationsCanvas = new Canvas();

SNE\_systemOfEquationsGrid.RowDefinitions.Add(rowDefinition);

SNE\_systemOfEquationsGrid.Children.Add(SNE\_equationsCanvas);

Label equationLabel = new Label();

equationLabel.Content = $"x{SNE\_systemOfEquationsGrid.RowDefinitions.Count} = ";

equationLabel.FontSize = 14;

equationLabel.Margin = new Thickness(10, start\_position, 0, 0);

SNE\_equationsCanvas.Children.Add(equationLabel);

TextBox equationOfSystem = new TextBox();

equationOfSystem.Margin = new Thickness(60, start\_position, 20, 0);

equationOfSystem.FontSize = 14;

equationOfSystem.Width = 200;

equationOfSystem.Height = 30;

SNE\_equationsCanvas.Children.Add(equationOfSystem);

}

private void GenerateInitialApproximation()

{

Canvas SNE\_initialApproximationCanvas;

int start\_position = 10;

if (SNE\_initialApproximationGrid.RowDefinitions.Count < 1)

start\_position = 10;

else

start\_position += 40 \* SNE\_initialApproximationGrid.RowDefinitions.Count;

RowDefinition rowDefinition = new RowDefinition();

rowDefinition.Height = new GridLength(52, GridUnitType.Pixel);

SNE\_initialApproximationCanvas = new Canvas();

SNE\_initialApproximationGrid.RowDefinitions.Add(rowDefinition);

SNE\_initialApproximationGrid.Children.Add(SNE\_initialApproximationCanvas);

Label equationLabel = new Label();

equationLabel.Content = $"x{SNE\_systemOfEquationsGrid.RowDefinitions.Count}";

equationLabel.FontSize = 14;

equationLabel.Margin = new Thickness(10, start\_position, 0, 0);

SNE\_initialApproximationCanvas.Children.Add(equationLabel);

TextBox equationOfSystem = new TextBox();

equationOfSystem.Margin = new Thickness(60, start\_position, 20, 0);

equationOfSystem.FontSize = 14;

equationOfSystem.Width = 50;

equationOfSystem.Height = 30;

SNE\_initialApproximationCanvas.Children.Add(equationOfSystem);

}

private void GenerateResultTable(IEnumerable<IEnumerable<double>> result, IEnumerable<string> variablesNames, SolvingMethods solvingMethod)

{

variablesNames = variablesNames.OrderBy(el => el);

DataTable dataTable = new DataTable();

SNE\_ResultDataGrid.ColumnWidth = new DataGridLength(1, DataGridLengthUnitType.Star);

dataTable.Columns.Add("k");

foreach (var variableName in variablesNames)

{

dataTable.Columns.Add($"{variableName}(k)");

}

dataTable.Columns.Add(solvingMethod == SolvingMethods.Secant ? "Sk" : "Difference");

int countResult = result.Count();

for (int i = 0; i < countResult; i++)

{

object[] row = new object[variablesNames.Count() + 2];

row[0] = i;

for (int j = 0; j < result.ElementAt(i).Count() - 1; j++)

{

row[j + 1] = string.Format("{0:F4}", result.ElementAt(i).ElementAt(j));

}

row[row.Length - 1] = result.ElementAt(i).Last() == double.MinValue ? "-" : string.Format("{0:F4}", result.ElementAt(i).Last());

dataTable.Rows.Add(row);

}

SNE\_ResultDataGrid.ItemsSource = dataTable.AsDataView();

}

private void SNE\_AddEquationButton\_Click(object sender, RoutedEventArgs e)

{

SNE\_ResultDataGrid.ItemsSource = null;

string method\_type = SNE\_methodComboBox.SelectedItem.ToString();

if (method\_type == SolvingMethods.Seidel.ToString() || method\_type == SolvingMethods.SimpleIterations.ToString()) { InitialOtherSystemOfEquations(); }

else { InitialDefaulSystemOfEquations(); }

GenerateInitialApproximation();

SNE\_RemoveEquationButton.IsEnabled = true;

}

private void SNE\_RemoveEquationButton\_Click(object sender, RoutedEventArgs e)

{

SNE\_systemOfEquationsGrid.RowDefinitions.Remove(SNE\_systemOfEquationsGrid.RowDefinitions.Last());

SNE\_systemOfEquationsGrid.Children.Remove(SNE\_systemOfEquationsGrid.Children[SNE\_systemOfEquationsGrid.Children.Count - 1]);

SNE\_initialApproximationGrid.RowDefinitions.Remove(SNE\_initialApproximationGrid.RowDefinitions.Last());

SNE\_initialApproximationGrid.Children.Remove(SNE\_initialApproximationGrid.Children[SNE\_initialApproximationGrid.Children.Count - 1]);

if (SNE\_systemOfEquationsGrid.RowDefinitions.Count < 1)

SNE\_RemoveEquationButton.IsEnabled = false;

}

private void SNE\_CalculateButton\_Click(object sender, RoutedEventArgs e)

{

if (SNE\_systemOfEquationsGrid.Children.Count == 0 || SNE\_initialApproximationGrid.Children.Count == 0)

{

MessageBox.Show("Добавьте уравнения и начальное приближение");

return;

}

List<string> functions = new List<string>();

foreach (var childrenGrid in SNE\_systemOfEquationsGrid.Children)

{

Canvas canvas = childrenGrid as Canvas;

foreach (var childrenCanvas in canvas.Children)

{

if (childrenCanvas is TextBox)

{

TextBox textBox = childrenCanvas as TextBox;

if (String.IsNullOrEmpty(textBox.Text))

{

MessageBox.Show("Заполните все поля, связанные с функциями");

return;

}

functions.Add(textBox.Text.Trim());

}

}

}

Dictionary<string, MathNet.Symbolics.FloatingPoint> initialGuess = new Dictionary<string, MathNet.Symbolics.FloatingPoint>();

string initialGuessKey = null;

MathNet.Symbolics.FloatingPoint initialGuessValue = null;

foreach (var childrenGrid in SNE\_initialApproximationGrid.Children)

{

Canvas canvas = childrenGrid as Canvas;

foreach (var childrenCanvas in canvas.Children)

{

if (childrenCanvas is Label)

{

Label label = childrenCanvas as Label;

initialGuessKey = label.Content.ToString();

}

if (childrenCanvas is TextBox)

{

TextBox textBox = childrenCanvas as TextBox;

if (String.IsNullOrEmpty(textBox.Text))

{

MessageBox.Show("Заполните все поля, связанные с начальным приближением");

return;

}

initialGuessValue = double.Parse(textBox.Text);

}

}

initialGuess.Add(initialGuessKey, initialGuessValue);

}

if (String.IsNullOrEmpty(SNE\_AccuracyTextBox.Text))

{

MessageBox.Show("Заполните поле точность");

return;

}

IEnumerable<IEnumerable<double>> res;

try

{

res = new NonLinearEquationsSolverBuilder().Build((SolvingMethods)Enum.Parse(typeof(SolvingMethods), SNE\_methodComboBox.SelectedValue.ToString())).SolveWithSteps(new NonLinearEquationsSystem(functions), double.Parse(SNE\_AccuracyTextBox.Text), initialGuess);

}

catch (Exception ex)

{

MessageBox.Show(ex.Message);

return;

}

res = res.Select(row => row.Select(item => Math.Round(item, 4)));

GenerateResultTable(res, initialGuess.Select(el => el.Key), (SolvingMethods)Enum.Parse(typeof(SolvingMethods), SNE\_methodComboBox.SelectedValue.ToString()));

}

}

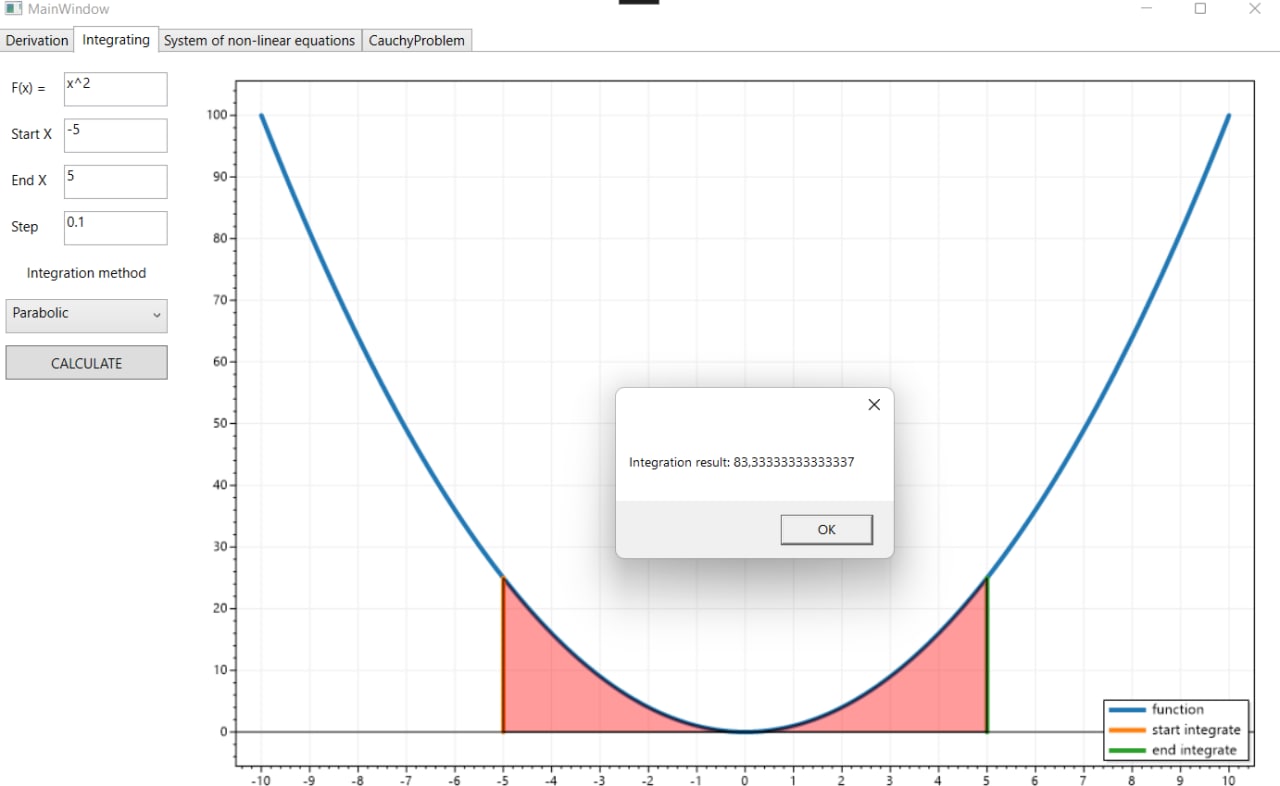
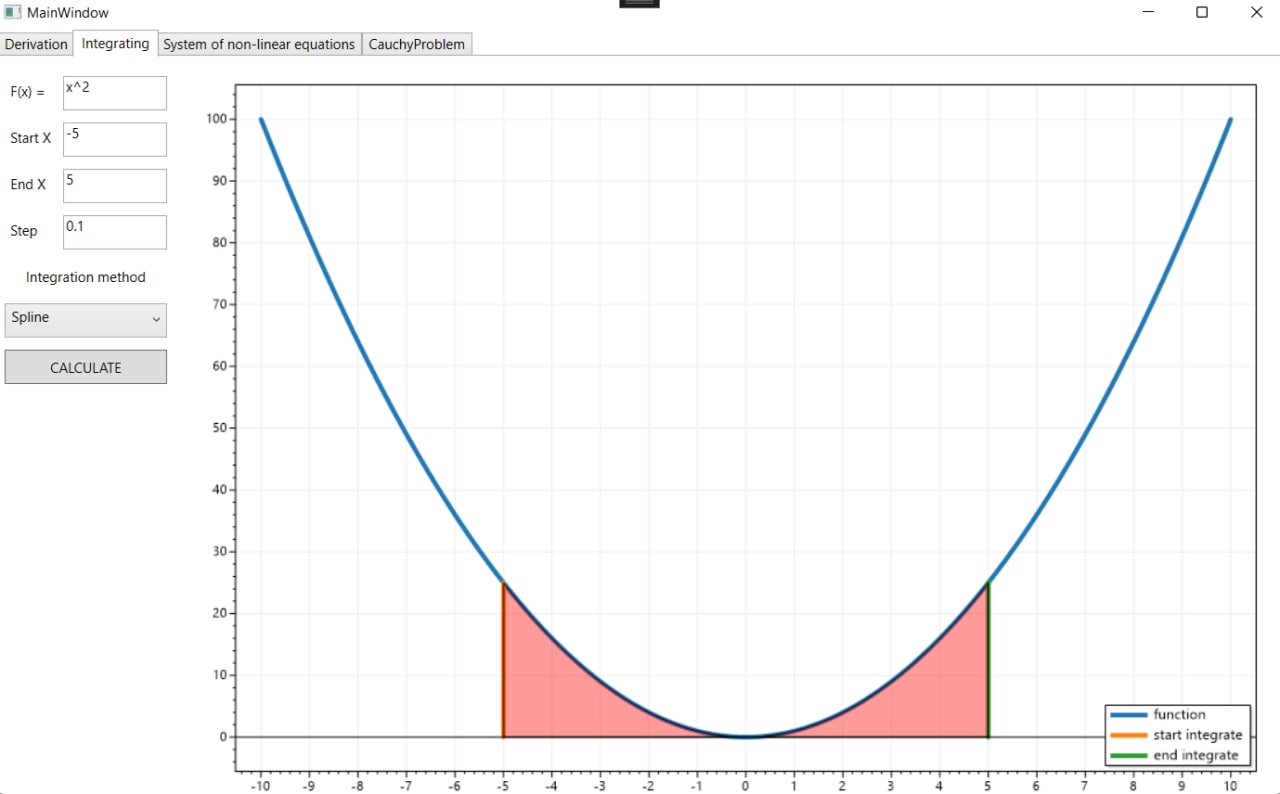
**Вывод**: в ходе выполнения лабораторной работы были программно реализованы методы решения систем нелинейных уравнений.

## Лабораторная работа №3

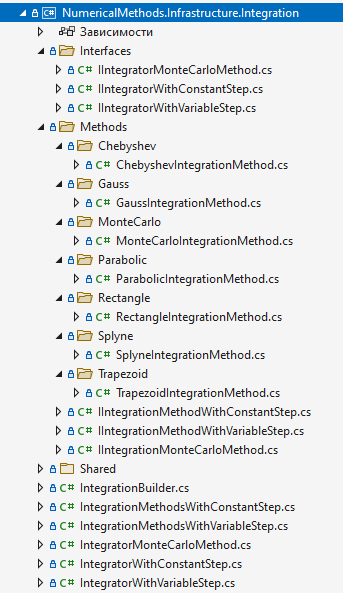
### «Решение задач численного интегрирования»

**Цель работы**: ознакомление с методами численного интегрирования и их реализация на языке программирования C# WPF.

**Скриншоты работы программы**



**Структура проекта**



**Листинг работы программы**

Листинг класса ChebyshevIntegrationMethod

using MathNet.Symbolics;

using NumericalMethods.Infrastructure.Integration.Shared;

using NumericalMethods.Infrastructure.NonLinearEquationsSystems;

using NumericalMethods.Infrastructure.NonLinearEquationsSystems.Interfaces;

namespace NumericalMethods.Infrastructure.Integration.Methods.Chebyshev;

public class ChebyshevIntegrationMethod : IIntegrationMethodWithVariableStep

{

/// <summary> Интегрирование отрезка функции </summary>

/// <param name="function"> Подинтегральная функция </param>

/// <param name="start"> Начало отрезка интегрирования </param>

/// <param name="end"> Конец отрезка интегрирования </param>

/// <param name="count\_nodes"> Количество разбиений отрезка интегрирования </param>

/// <returns></returns>

public double Integrate(string function, double start, double end, int count\_nodes)

{

SymbolicExpression functionExpression = SymbolicExpression.Parse(function);

IEnumerable<double> tRoots = GetRootsForCountNodes(count\_nodes);

IEnumerable<double> xCoefs = tRoots.Select(tRoot => GetRealX(start, end, tRoot));

IEnumerable<double> aCoefs = Enumerable.Repeat(2.0 / count\_nodes, count\_nodes);

IEnumerable<double> aCoefsMultiplyedYs = aCoefs.Zip(aCoefs, (aCoef, xCoef) => aCoef \* functionExpression.EvaluateX(xCoef));

return (end - start) / 2 \* aCoefsMultiplyedYs.Sum();

}

/// <summary> Функция зависимости x от t </summary>

/// <param name="a"> Начало отрезка интегрирования </param>

/// <param name="b"> Конеч отрезка интегрирования </param>

/// <param name="t"> Значение коэффициента t </param>

private static double GetRealX(double a, double b, double t)

{

return (a + b) / 2 + (b - a) / 2 \* t;

}

/// <summary> Возвращает значения корней Чебышева для определенного количества разбиений отрезка интегрирования </summary>

/// <param name="countNodes"> Количество разбиений </param>

/// <exception cref="NotImplementedException"></exception>

private static IEnumerable<double> GetRootsForCountNodes(int countNodes)

{

return countNodes switch

{

3 => new double[3] { 0.707107, 0, -0.707107 },

4 => new double[4] { 0.794654, 0.187592, -0.794654, -0.187592 },

5 => new double[5] { 0.832498, 0.374541, 0, -0.832498, -0.374541 },

6 => new double[6] { 0.866247, 0.422519, 0.266635, -0.866247, -0.422519, -0.266635 },

7 => new double[7] { 0.883862, 0.529657, 0.323912, 0, -0.883862, -0.529657, -0.323912 },

9 => new double[9] { 0.911589, 0.601019, 0.528762, 0.167906, 0, -0.911589, -0.601019, -0.528762, -0.167906 },

\_ => throw new NotImplementedException($"Для размерности {countNodes} метод не реализован")

};

}

/// <summary> Создает задачу поиска начального приближения для вычисления значений корней Чебышева </summary>

private Task<IEnumerable<double>?> CreateRootFinder(List<string> expressions, CancellationToken token)

{

return Task.Run(() =>

{

List<string> expressionsCopy = expressions.ToList();

NonLinearEquationsSystem snu = new NonLinearEquationsSystem(expressionsCopy);

INonLinearEquationsSystemsSolver solver = new NonLinearEquationsSolverBuilder().Build(SolvingMethods.Newton);

bool success = false;

Random random = new Random();

IEnumerable<double> randomInitial = new double[snu.FunctionExpressions.Count()].Select(value => random.Next(-1, 2) + random.NextDouble());

IEnumerable<double> findedRoots = null;

while (success is false)

{

try

{

if (token.IsCancellationRequested) break;

//findedRoots = solver.Solve(snu, 0.001d, randomInitial);

if (findedRoots.Any(root => double.IsNaN(root) || root is < -1 or > 1))

{

randomInitial = randomInitial.Select(value => random.Next(-1, 2) + random.NextDouble());

continue;

}

success = true;

} catch { randomInitial = randomInitial.Select(value => random.Next(-1, 2) + random.NextDouble()); }

}

return findedRoots;

}, token);

}

Листинг класса GaussIntegrationMethod

using MathNet.Symbolics;

using NumericalMethods.Infrastructure.Integration.Shared;

namespace NumericalMethods.Infrastructure.Integration.Methods.Gauss

{

internal class GaussIntegrationMethod : IIntegrationMethodWithVariableStep

{

public double Integrate(string function, double start, double end, int count\_nodes)

{

SymbolicExpression func = SymbolicExpression.Parse(function);

double[] acoef = new double[count\_nodes];

double[] x = new double[count\_nodes];

for (int i = 1; i <= count\_nodes; i++)

{

double ti = T(count\_nodes - 1, i, count\_nodes);

double pd = PD(count\_nodes, ti);

double diva = (1 - ti \* ti) \* pd \* pd;

acoef[i - 1] = 2 / diva;

x[i - 1] = GetRealX(start, end, ti);

}

return (end - start) / 2 \* acoef.Zip(x, (ac, xc) => ac \* func.EvaluateX(xc)).Sum();

}

double GetRealX(double a, double b, double t)

{

return (a + b) / 2 + (b - a) / 2 \* t;

}

double T(int k, int i, int n)

{

if (k == 0)

{

double t1 = Math.PI \* (4 \* i - 1);

double t2 = 4 \* n + 2;

return Math.Cos(t1 / t2);

}

double tprev = T(k - 1, i, n);

return tprev - P(n, tprev) / PD(n, tprev);

}

double P(double n, double t)

{

return n switch

{

0 => 1,

1 => t,

\_ => (2 \* (n - 1) + 1) / ((n - 1) + 1) \* t \* P(n - 1, t)

- (n - 1) / ((n - 1) + 1) \* P(n - 2, t)

};

}

double PD(double n, double t)

{

double left = n / (1 - t \* t);

double right = P(n - 1, t) - t \* P(n, t);

return left \* right;

}

}

}

Листинг класса MonteCarloIntegrationMethod

using MathNet.Symbolics;

using NumericalMethods.Infrastructure.Integration.Shared;

namespace NumericalMethods.Infrastructure.Integration.Methods.MonteCarlo

{

internal class MonteCarloIntegrationMethod : IIntegrationMonteCarloMethod

{

public double Integrate(string function, double start, double end, int count\_points)

{

SymbolicExpression func = SymbolicExpression.Parse(function);

Random rnd = new Random();

double x = start, sumValuesFunctions = 0;

for (int i = 0; i < count\_points; i++)

{

while(x <= start || x > end)

x = rnd.Next((int)start, (int)end) + rnd.NextDouble();

sumValuesFunctions += func.EvaluateX(x);

x = start;

}

return (end - start) / count\_points \* sumValuesFunctions;

}

}

}

Листинг класса ParabolicIntegrationMethod

using MathNet.Symbolics;

using NumericalMethods.Infrastructure.Integration.Shared;

namespace NumericalMethods.Infrastructure.Integration.Methods.Parabolic;

public class ParabolicIntegrationMethod : IIntegrationMethodWithConstantStep

{

public double Integrate(string function, double start, double end, double step)

{

SymbolicExpression func = SymbolicExpression.Parse(function);

double valueEvenFunction = 0, valueOddFunction = 0;

int i = 0;

for (double x = start + step; x < end - step; x += step)

if (++i % 2 == 0)

valueEvenFunction += func.EvaluateX(x);

else

valueOddFunction += func.EvaluateX(x);

return step / 3 \* (func.EvaluateX(start) + (4 \* valueOddFunction) + (2 \* valueEvenFunction) + func.EvaluateX(end));

}

}

Листинг класса RectangleIntegrationMethod

using MathNet.Symbolics;

using NumericalMethods.Infrastructure.Integration.Shared;

namespace NumericalMethods.Infrastructure.Integration.Methods.Rectangle;

internal class RectangleIntegrationMethod : IIntegrationMethodWithConstantStep

{

public double Integrate(string function, double start, double end, double step)

{

SymbolicExpression func = SymbolicExpression.Parse(function);

double resulted\_sum = 0;

for (double xi = start + (step / 2); xi < end; xi += step)

resulted\_sum += step \* func.EvaluateX(xi);

return resulted\_sum;

}

}

Листинг класса SplyneIntegrationMethod

using NumericalMethods.Infrastructure.Integration.Methods.Trapezoid;

using MathNet.Symbolics;

using NumericalMethods.Infrastructure.Integration.Shared;

namespace NumericalMethods.Infrastructure.Integration.Methods.Splyne

{

public record class SplyneIntegrationMethod : IIntegrationMethodWithConstantStep

{

public double Integrate(string function, double start, double end, double step)

{

SymbolicExpression func = SymbolicExpression.Parse(function);

double inaccuracyTrapezoidMethod = 0;//Погрешность метода трапеции

double trapezoidIntegral = new TrapezoidIntegrationMethod().Integrate(function, start, end, step); // Первая часть функции Сплайна

for (double x = start + step; x < end; x += step)

inaccuracyTrapezoidMethod += Math.Pow(step, 3) \* func.Derivative(x, 2);

return trapezoidIntegral - (inaccuracyTrapezoidMethod / 12);

}

}

}

Листинг класса TrapezoidIntegrationMethod

using MathNet.Symbolics;

using NumericalMethods.Infrastructure.Integration.Shared;

namespace NumericalMethods.Infrastructure.Integration.Methods.Trapezoid;

public class TrapezoidIntegrationMethod : IIntegrationMethodWithConstantStep

{

public double Integrate(string function, double start, double end, double step)

{

SymbolicExpression func = SymbolicExpression.Parse(function);

double sumElement = 0;

for (double x = start + step; x < end; x += step)

sumElement += step \* (func.EvaluateX(x - step) + func.EvaluateX(x));

return sumElement / 2;

}

}

Листинг класса IntegrationBuilder

using NumericalMethods.Infrastructure.Integration.Interfaces;

using NumericalMethods.Infrastructure.Integration.Methods.Parabolic;

using NumericalMethods.Infrastructure.Integration.Methods.Rectangle;

using NumericalMethods.Infrastructure.Integration.Methods.Splyne;

using NumericalMethods.Infrastructure.Integration.Methods.Trapezoid;

using NumericalMethods.Infrastructure.Integration.Methods.MonteCarlo;

using NumericalMethods.Infrastructure.Integration.Methods.Gauss;

using NumericalMethods.Infrastructure.Integration.Methods.Chebyshev;

namespace NumericalMethods.Infrastructure.Integration;

public record class IntegrationBuilder

{

public IIntegratorWithConstantStep Build(string function, IntegrationMethodsWithConstantStep method)

{

return method switch

{

IntegrationMethodsWithConstantStep.Rectangle => new IntegratorWithConstantStep(new RectangleIntegrationMethod(), function),

IntegrationMethodsWithConstantStep.Trapeze => new IntegratorWithConstantStep(new TrapezoidIntegrationMethod(), function),

IntegrationMethodsWithConstantStep.Parabolic => new IntegratorWithConstantStep(new ParabolicIntegrationMethod(), function),

IntegrationMethodsWithConstantStep.Spline => new IntegratorWithConstantStep(new SplyneIntegrationMethod(), function),

\_ => new IntegratorWithConstantStep(new SplyneIntegrationMethod(), function)

};

}

public IIntegratorWithVariableStep Build(string function, IntegrationMethodsWithVariableStep method)

{

return method switch

{

IntegrationMethodsWithVariableStep.Gauss => new IntegratorWithVariableStep(new GaussIntegrationMethod(),function),

IntegrationMethodsWithVariableStep.Chebyshev => new IntegratorWithVariableStep(new ChebyshevIntegrationMethod(), function),

\_ => throw new NotImplementedException()

};

}

public IIntegratorMonteCarloMethod BuildMonteCarlo(string function)

{

return new IntegratorMonteCarloMethod(new MonteCarloIntegrationMethod(), function);

}

}

Листинг класса IntegratorMonteCarloMethod

using NumericalMethods.Infrastructure.Integration.Interfaces;

using NumericalMethods.Infrastructure.Integration.Methods;

namespace NumericalMethods.Infrastructure.Integration

{

internal record class IntegratorMonteCarloMethod(IIntegrationMonteCarloMethod IntegrationMethod, string Function) : IIntegratorMonteCarloMethod

{

public double Integrate(double start, double end, int count\_points)

{

return IntegrationMethod.Integrate(Function, start, end, count\_points);

}

}

}

Листинг класса IntegrationMethodsWithConstantStep

namespace NumericalMethods.Infrastructure.Integration;

public enum IntegrationMethodsWithConstantStep

{

Rectangle = 0,

Trapeze = 1,

Parabolic = 2,

Spline = 3

}

Листинг класса IntegrationMethodsWithVariableStep

namespace NumericalMethods.Infrastructure.Integration;

public enum IntegrationMethodsWithVariableStep

{

Gauss = 0,

Chebyshev = 1

}

Листинг класса IntegratorMonteCarloMethod

using NumericalMethods.Infrastructure.Integration.Interfaces;

using NumericalMethods.Infrastructure.Integration.Methods;

namespace NumericalMethods.Infrastructure.Integration

{

internal record class IntegratorMonteCarloMethod(IIntegrationMonteCarloMethod IntegrationMethod, string Function) : IIntegratorMonteCarloMethod

{

public double Integrate(double start, double end, int count\_points)

{

return IntegrationMethod.Integrate(Function, start, end, count\_points);

}

}

}

Листинг класса IntegratorWithConstantStep

using NumericalMethods.Infrastructure.Integration.Interfaces;

using NumericalMethods.Infrastructure.Integration.Methods;

namespace NumericalMethods.Infrastructure.Integration;

internal record class IntegratorWithConstantStep(IIntegrationMethodWithConstantStep IntegrationMethod, string Function) : IIntegratorWithConstantStep

{

public double Integrate(double start, double end, double step)

{

return IntegrationMethod.Integrate(Function, start, end, step);

}

}

Листинг класса IntegratorWithVariableStep

using NumericalMethods.Infrastructure.Integration.Interfaces;

using NumericalMethods.Infrastructure.Integration.Methods;

namespace NumericalMethods.Infrastructure.Integration;

internal record class IntegratorWithVariableStep(IIntegrationMethodWithVariableStep IntegrationMethod, string Function) : IIntegratorWithVariableStep

{

public double Integrate(double start, double end, int count\_nodes)

{

return IntegrationMethod.Integrate(Function, start, end, count\_nodes);

}

}

Листинг класса IIntegratorMonteCarloMethod

namespace NumericalMethods.Infrastructure.Integration.Interfaces

{

public interface IIntegratorMonteCarloMethod

{

public double Integrate(double start, double end, int count\_points);

}

}

Листинг класса IIntegratorWithConstantStep

namespace NumericalMethods.Infrastructure.Integration.Interfaces;

public interface IIntegratorWithConstantStep

{

public double Integrate(double start, double end, double step);

}

Листинг класса IIntegratorWithVariableStep

namespace NumericalMethods.Infrastructure.Integration.Interfaces;

public interface IIntegratorWithVariableStep

{

public double Integrate(double start, double end, int count\_nodes);

}

Листинг класса IIntegrationMethodWithConstantStep

using NumericalMethods.Infrastructure.Integration.Interfaces;

namespace NumericalMethods.Infrastructure.Integration.Methods;

internal interface IIntegrationMethodWithConstantStep

{

public double Integrate(string function, double start, double end, double step);

}

Листинг класса IIntegrationMethodWithVariableStep

using NumericalMethods.Infrastructure.Integration.Interfaces;

namespace NumericalMethods.Infrastructure.Integration.Methods;

internal interface IIntegrationMethodWithVariableStep

{

public double Integrate(string function, double start, double end, int count\_nodes);

}

Листинг класса IIntegrationMonteCarloMethod

namespace NumericalMethods.Infrastructure.Integration.Methods

{

internal interface IIntegrationMonteCarloMethod

{

public double Integrate(string function, double start, double end, int count\_points);

}

}

Листинг класса MainWindow.xaml

namespace NumericalMethods.WPFApplication

{

/// <summary>

/// Interaction logic for MainWindow.xaml

/// </summary>

public partial class MainWindow : Window

{

private readonly List<Point> \_points = new List<Point>();

private Function \_current\_function;

private double \_start\_x;

private double \_end\_x;

private double \_step;

private int \_order;

private ResultTable \_resultTable;

public MainWindow()

{

InitializeComponent();

Width = 1300;

Height = 750;

InitializeIntegration();

}

private void InitializeIntegration()

{

foreach (var item in Enum.GetValues(typeof(IntegrationMethodsWithConstantStep)))

{

Integration\_MethodComboBox.Items.Add(item.ToString());

}

foreach(var item in Enum.GetValues(typeof(IntegrationMethodsWithVariableStep)))

{

Integration\_MethodComboBox.Items.Add(item.ToString());

}

Integration\_MethodComboBox.Items.Add("MonteCarlo");

Integration\_MethodComboBox.SelectedItem = Integration\_MethodComboBox.Items[0];

Integration\_FunctionTextBox.Text = "x^2";

Integration\_StartXTextBox.Text = "-5";

Integration\_EndXTextBox.Text = "5";

Integration\_StepTextBox.Text = "0.1";

Integration\_MainChart.Plot.Legend(enable: true);

}

private void Integration\_MethodComboBox\_SelectionChanged(object sender, SelectionChangedEventArgs e)

{

if (Enum.IsDefined(typeof(IntegrationMethodsWithConstantStep), ((ComboBox)sender).SelectedValue))

{

Integration\_CountNodesGrid.Visibility = Visibility.Collapsed;

Integration\_CountPointGrid.Visibility = Visibility.Collapsed;

}

if (Enum.IsDefined(typeof(IntegrationMethodsWithVariableStep), ((ComboBox)sender).SelectedValue))

{

Integration\_CountNodesGrid.Visibility = Visibility.Visible;

Integration\_CountPointGrid.Visibility = Visibility.Collapsed;

}

if (((ComboBox)sender).SelectedValue.ToString() == "MonteCarlo")

{

Integration\_CountNodesGrid.Visibility = Visibility.Collapsed;

Integration\_CountPointGrid.Visibility = Visibility.Visible;

}

Integration\_CountPointTextBox.Text = "";

Integration\_CountNodesTextBox.Text = "";

}

private void Integration\_CalculateButton\_Click(object sender, RoutedEventArgs e)

{

Integration\_MainChart.Plot.Clear();

if (String.IsNullOrEmpty(Integration\_FunctionTextBox.Text)||

String.IsNullOrEmpty(Integration\_StartXTextBox.Text)||

String.IsNullOrEmpty(Integration\_EndXTextBox.Text)||

String.IsNullOrEmpty(Integration\_StepTextBox.Text))

{

MessageBox.Show("Для начала рассчёта, заполните все поля");

return;

}

Function function = new Function("f(x) = " + Integration\_FunctionTextBox.Text.Trim());

double start\_x = new Expression(Integration\_StartXTextBox.Text.Trim()).calculate();

double end\_x = new Expression(Integration\_EndXTextBox.Text.Trim()).calculate();

double step = new Expression(Integration\_StepTextBox.Text.Trim()).calculate();

string function\_type\_string = Integration\_MethodComboBox.SelectedValue.ToString();

double integration\_result = 0;

if (Enum.IsDefined(typeof(IntegrationMethodsWithConstantStep), function\_type\_string))

{

IntegrationMethodsWithConstantStep method = (IntegrationMethodsWithConstantStep)Enum.Parse(typeof(IntegrationMethodsWithConstantStep), function\_type\_string);

IIntegratorWithConstantStep integrator = new IntegrationBuilder().Build(Integration\_FunctionTextBox.Text.Trim(), method);

integration\_result = integrator.Integrate(start\_x, end\_x, step);

}

if (Enum.IsDefined(typeof(IntegrationMethodsWithVariableStep), function\_type\_string))

{

if (String.IsNullOrEmpty(Integration\_CountNodesTextBox.Text))

{

MessageBox.Show("Для начала рассчёта, заполните поле count nodes");

return;

}

int countNodes = int.Parse(Integration\_CountNodesTextBox.Text);

IntegrationMethodsWithVariableStep method = (IntegrationMethodsWithVariableStep)Enum.Parse(typeof(IntegrationMethodsWithVariableStep), function\_type\_string);

IIntegratorWithVariableStep integrator = new IntegrationBuilder().Build(Integration\_FunctionTextBox.Text.Trim(), method);

integration\_result = integrator.Integrate(start\_x, end\_x, countNodes);

}

if (function\_type\_string == "MonteCarlo")

{

if (String.IsNullOrEmpty(Integration\_CountPointTextBox.Text))

{

MessageBox.Show("Для начала рассчёта, заполните поле count point");

return;

}

int countPoints = int.Parse(Integration\_CountPointTextBox.Text);

IIntegratorMonteCarloMethod integrator = new IntegrationBuilder().BuildMonteCarlo(Integration\_FunctionTextBox.Text.Trim());

integration\_result = integrator.Integrate(start\_x, end\_x, countPoints);

}

(double[] xs, double[] ys) = DataGenerate(function, start\_x - 5, end\_x + 5, step);

Integration\_MainChart.Plot.AddScatter(xs.ToArray(), ys.ToArray(), lineWidth: 4, markerSize: 0, label: "function");

double[] interval\_start\_x = { start\_x, start\_x };

double[] interval\_start\_y = { 0, function.calculate(start\_x) };

Integration\_MainChart.Plot.AddScatter(interval\_start\_x, interval\_start\_y, lineWidth: 4, markerSize: 0, label: "start integrate");

double[] interval\_end\_x = { end\_x, end\_x };

double[] interval\_end\_y = { 0, function.calculate(end\_x) };

Integration\_MainChart.Plot.AddScatter(interval\_end\_x, interval\_end\_y, lineWidth: 4, markerSize: 0, label: "end integrate");

(double[] fill\_xs, double[] fill\_ys) = DataGenerate(function, start\_x, end\_x, step);

Integration\_MainChart.Plot.AddFill(fill\_xs, fill\_ys, baseline: 0, color: System.Drawing.Color.FromArgb(100, System.Drawing.Color.Red));

Integration\_MainChart.Plot.AddHorizontalLine(0, color: System.Drawing.Color.Black);

Integration\_MainChart.Refresh();

MessageBox.Show($"Integration result: {integration\_result}");

}

}

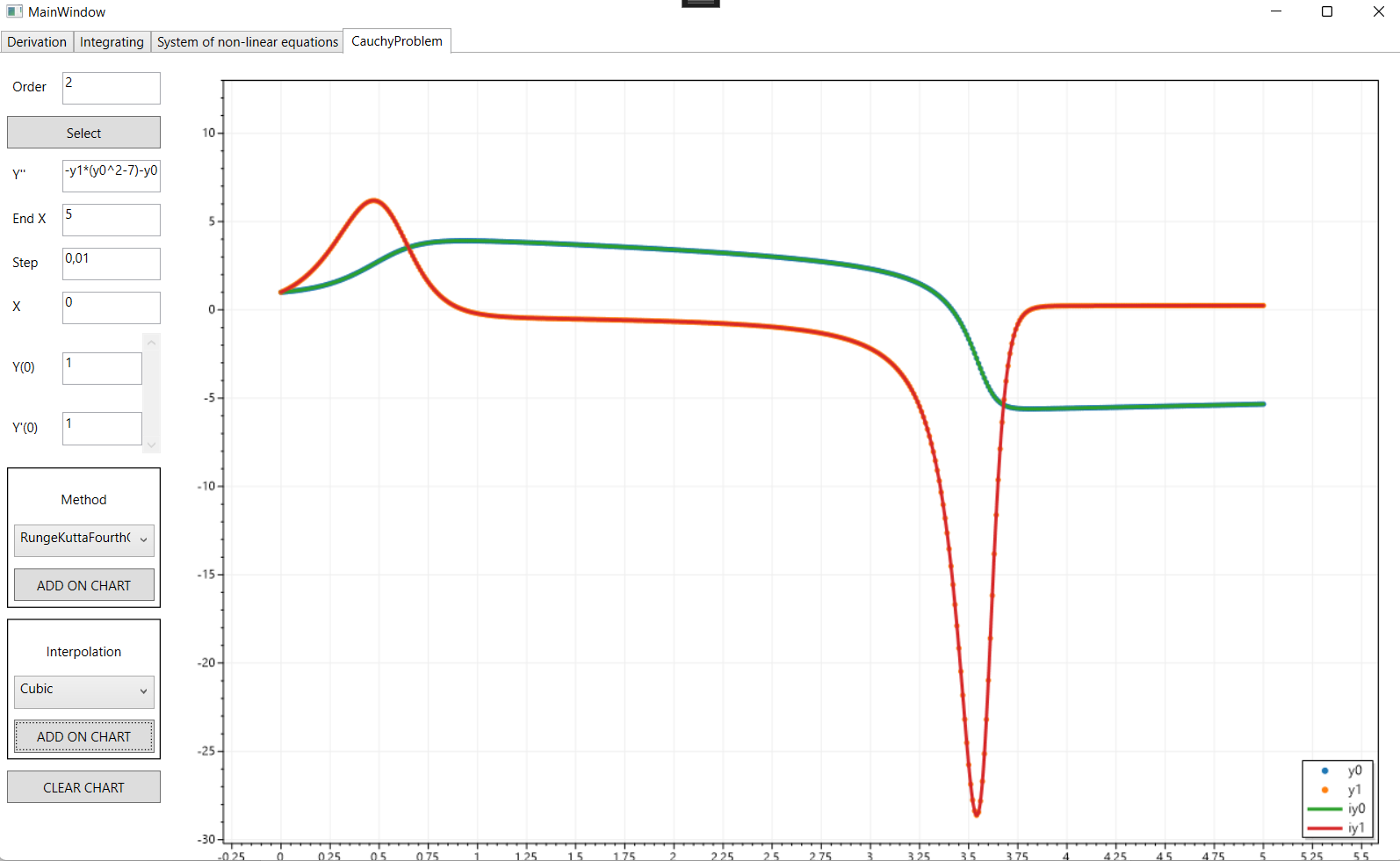
}

**Вывод**: в ходе выполнения лабораторной работы были получены практические навыки численного интегрирования с использованием основных методом: метод прямоугольников, методя трапеций, метод парабол (метод Симпсона), метода сплайнов, метод Монте-Карло, метод Гаусса, метода Чебышева.

## Лабораторная работа №4

### «Методы решения дифференциальных уравнений»

**Цель работы**: изучение методов решения дифференциальных уравнений в частных производных и приобретение навыков решения практических задач с использованием программного средства Matlab.

**Скриншоты работы программы**

tk = [0:0.01:5];

x0 = [1 1];

[T X] = ode45('diffsys',tk,x0);

plot(T,X)

grid on

xlabel('x')

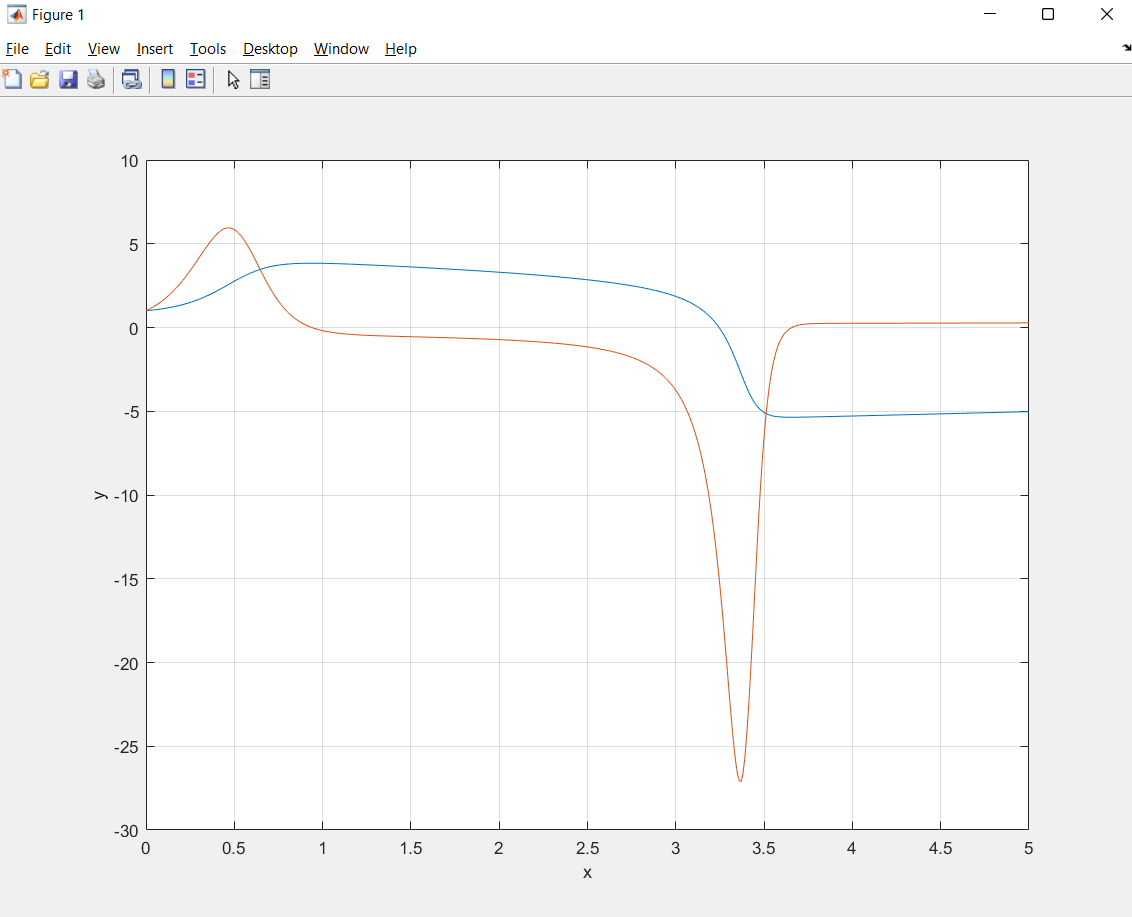
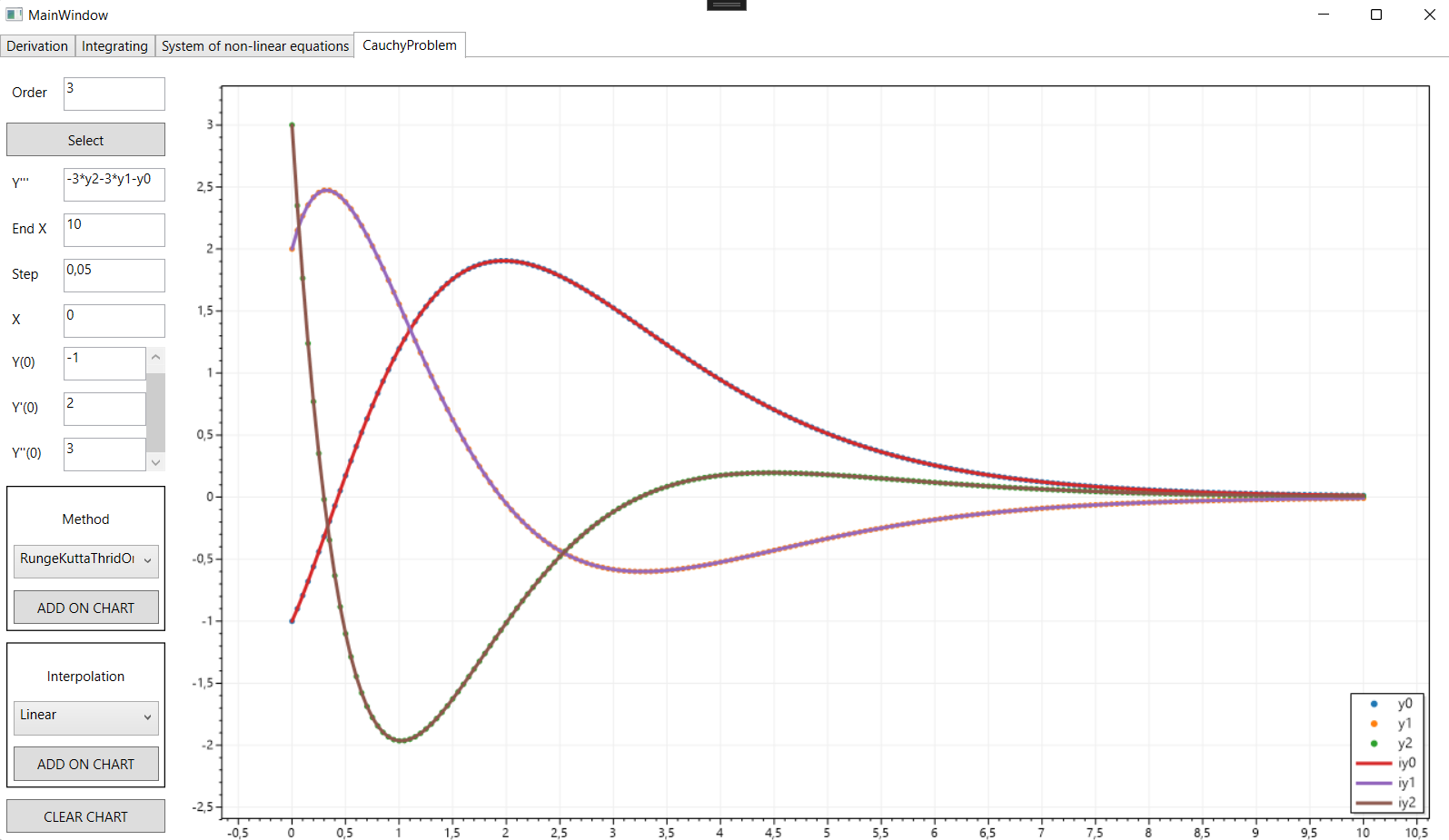
ylabel('y')

function dxdt = diffsys(t,x)

dxdt(1,1) = x(2);

dxdt(2,1) = -x(2)\*(x(1).^2-7)-x(1);

end

tk = [0:0.01:10];

x0 = [-1 2 3];

[T X] = ode45('diffsys',tk,x0);

plot(T,X)

grid on

xlabel('x')

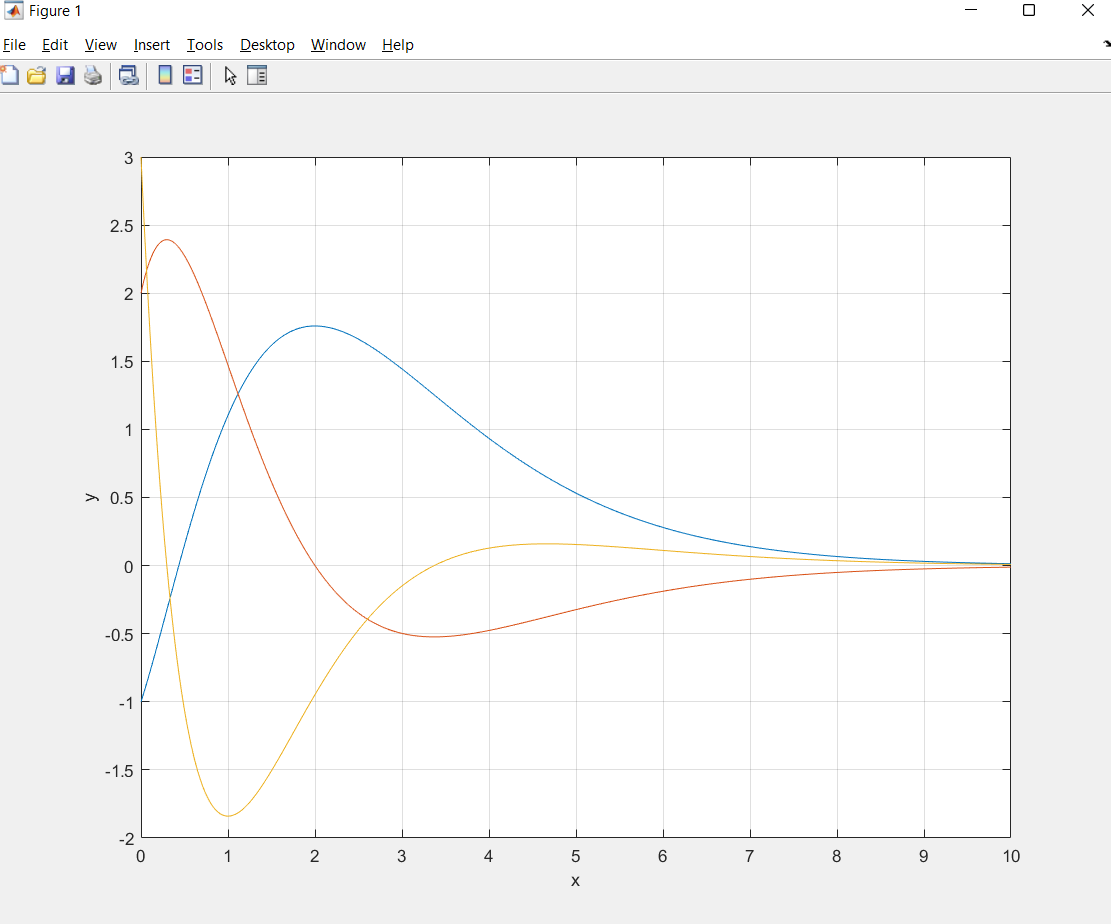
ylabel('y')

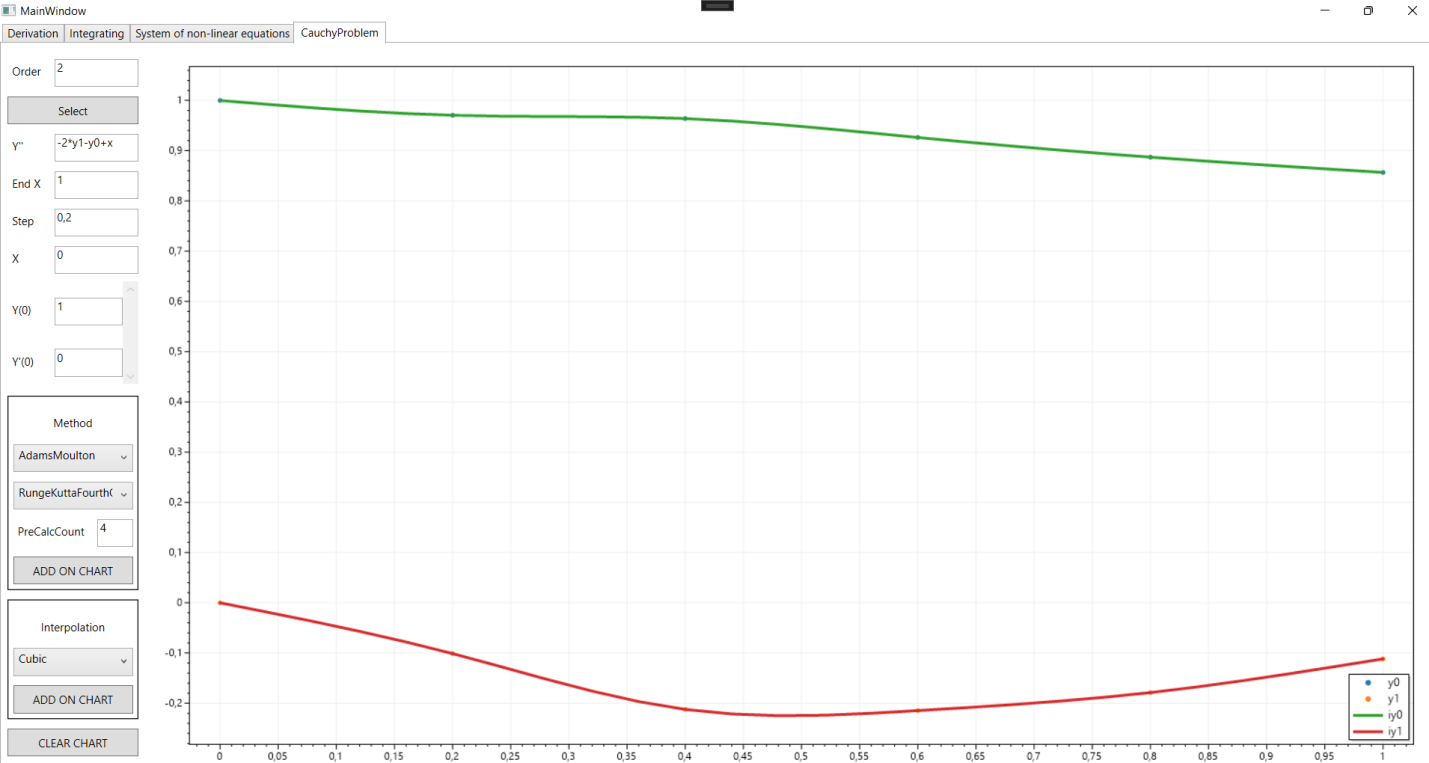
function dxdt = diffsys(t,x)

dxdt(1,1) = x(2);

dxdt(2,1) = x(3);

dxdt(3,1) = -3\*x(3)-3\*x(2)-x(1);

end



clear, clc

tk = [0:0.2:1];

x0 = [1 0];

[T X] = ode45('diffsys',tk,x0);

plot(T,X)

grid on

xlabel('x')

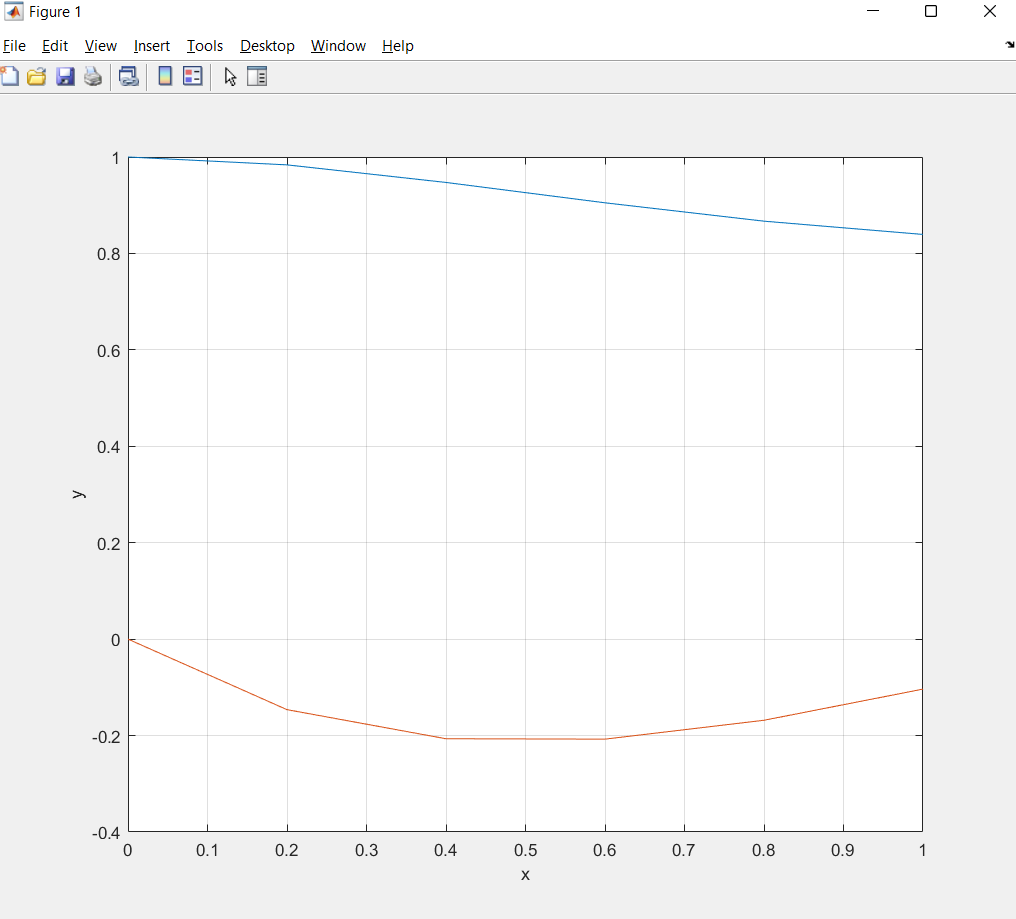
ylabel('y')

function dxdt = diffsys(t, x)

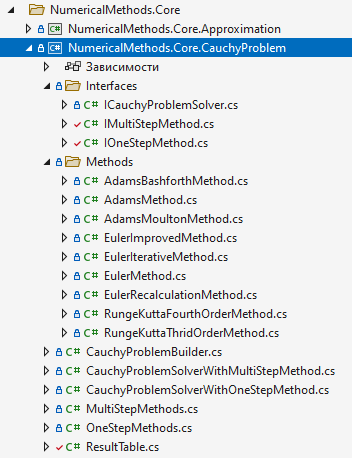
dxdt(1,1) = x(2);

dxdt(2,1) = -2\*x(2)-x(1) + t;

end



**Структура проекта**



**Листинг работы программы**

Листинг класса EulerMethod

using MathNet.Symbolics;

using NumericalMethods.Core.CauchyProblem.Interfaces;

namespace NumericalMethods.Core.CauchyProblem.Methods;

internal class EulerMethod : IOneStepMethod

{

public double Calculate(SymbolicExpression function, double h, double x, (string name, double value) yi, Dictionary<string, FloatingPoint> conditions)

{

return yi.value + h \* function.Evaluate(conditions).RealValue;

}

}

Листинг класса EulerRecalculationMethod

using MathNet.Symbolics;

using NumericalMethods.Core.CauchyProblem.Interfaces;

namespace NumericalMethods.Core.CauchyProblem.Methods;

internal class EulerRecalculationMethod : IOneStepMethod

{

public double Calculate(SymbolicExpression function, double h, double x, (string name, double value) yi, Dictionary<string, FloatingPoint> conditions)

{

Dictionary<string, FloatingPoint> conditionsCopy = conditions.ToDictionary(pair => pair.Key, pair => pair.Value);

double functionValue = function.Evaluate(conditionsCopy).RealValue;

double y\_next\_approximation = yi.value + h \* functionValue;

conditionsCopy["x"] = x + h;

conditionsCopy[yi.name] = y\_next\_approximation;

return yi.value + (h \* (functionValue + function.Evaluate(conditionsCopy).RealValue)) / 2;

}

}

Листинг класса EulerIterativeMethod

using MathNet.Symbolics;

using NumericalMethods.Core.CauchyProblem.Interfaces;

namespace NumericalMethods.Core.CauchyProblem.Methods;

internal class EulerIterativeMethod : IOneStepMethod

{

public double Calculate(SymbolicExpression function, double h, double x, (string name, double value) yi, Dictionary<string, FloatingPoint> conditions)

{

Dictionary<string, FloatingPoint> conditionsCopy = conditions.ToDictionary(pair => pair.Key, pair => pair.Value);

double functionValue = function.Evaluate(conditionsCopy).RealValue;

double y\_next = yi.value + h \* functionValue;

conditionsCopy["x"] = x + h;

for (int k = 0; k < 4; k++)

{

conditionsCopy[yi.name] = y\_next;

y\_next = yi.value + h \* (functionValue + function.Evaluate(conditionsCopy).RealValue) / 2;

}

return y\_next;

}

}

Листинг класса EulerImprovedMethod

using MathNet.Symbolics;

using NumericalMethods.Core.CauchyProblem.Interfaces;

namespace NumericalMethods.Core.CauchyProblem.Methods;

public class EulerImprovedMethod : IOneStepMethod

{

public double Calculate(SymbolicExpression function, double h, double x, (string name, double value) yi, Dictionary<string, FloatingPoint> conditions)

{

Dictionary<string, FloatingPoint> conditionsCopy = conditions.ToDictionary(pair => pair.Key, pair => pair.Value);

double y\_half = yi.value + h / 2 \* function.Evaluate(conditionsCopy).RealValue;

conditionsCopy["x"] = x + h / 2;

conditionsCopy[yi.name] = y\_half;

return yi.value + h \* function.Evaluate(conditionsCopy).RealValue;

}

}

Листинг класса RungeKuttaThridOrderMethod

using MathNet.Symbolics;

using NumericalMethods.Core.CauchyProblem.Interfaces;

namespace NumericalMethods.Core.CauchyProblem.Methods;

public class RungeKuttaThridOrderMethod : IOneStepMethod

{

public double Calculate(SymbolicExpression function, double h, double x, (string name, double value) yi, Dictionary<string, FloatingPoint> conditions)

{

Dictionary<string, FloatingPoint> conditionsCopy = conditions.ToDictionary(pair => pair.Key, pair => pair.Value);

double y\_delta, k\_one, k\_two, k\_three;

k\_one = h \* function.Evaluate(conditionsCopy).RealValue;

conditionsCopy["x"] = x + h / 3;

conditionsCopy[yi.name] = yi.value + k\_one / 3;

k\_two = h \* function.Evaluate(conditionsCopy).RealValue;

conditionsCopy["x"] = x + h \* 2.0 / 3.0;

conditionsCopy[yi.name] = yi.value + k\_two \* 2.0 / 3.0;

k\_three = h \* function.Evaluate(conditionsCopy).RealValue;

y\_delta = (k\_one + 3 \* k\_three) / 4;

return yi.value + y\_delta;

}

}

Листинг класса RungeKuttaFourthOrderMethod

using MathNet.Symbolics;

using NumericalMethods.Core.CauchyProblem.Interfaces;

namespace NumericalMethods.Core.CauchyProblem.Methods;

public class RungeKuttaFourthOrderMethod : IOneStepMethod

{

public double Calculate(SymbolicExpression function, double h, double x, (string name, double value) yi, Dictionary<string, FloatingPoint> conditions)

{

Dictionary<string, FloatingPoint> conditionsCopy = conditions.ToDictionary(pair => pair.Key, pair => pair.Value);

double y\_delta, k\_one, k\_two, k\_three, k\_four;

k\_one = h \* function.Evaluate(conditionsCopy).RealValue;

conditionsCopy["x"] = x + h / 2;

conditionsCopy[yi.name] = yi.value + k\_one / 2;

k\_two = h \* function.Evaluate(conditionsCopy).RealValue;

conditionsCopy[yi.name] = yi.value + k\_two / 2;

k\_three = h \* function.Evaluate(conditionsCopy).RealValue;

conditionsCopy["x"] = x + h;

conditionsCopy[yi.name] = yi.value + k\_three;

k\_four = h \* function.Evaluate(conditionsCopy).RealValue;

y\_delta = (k\_one + 2 \* k\_two + 2 \* k\_three + k\_four) / 6;

return yi.value + y\_delta;

}

}

Листинг класса AdamsMethod

using MathNet.Symbolics;

using NumericalMethods.Core.CauchyProblem.Interfaces;

namespace NumericalMethods.Core.CauchyProblem.Methods;

public class AdamsMethod : IMultiStepMethod

{

public ResultTable Calculate(List<SymbolicExpression> functions, double b, double h, ResultTable initialResult)

{

if (initialResult.CountRow < 4) throw new ArgumentException($"{nameof(initialResult)} count row less then 4");

for (double current\_x = initialResult["x"].Last().x + h; current\_x <= b; current\_x += h)

{

Dictionary<string, double> new\_ys = new Dictionary<string, double>();

List<Dictionary<string, FloatingPoint>> last\_four\_rows = initialResult.GetRows(^4..)

.Select(row => row.Select(pair => new KeyValuePair<string, FloatingPoint>(pair.Key, pair.Value))

.ToDictionary(pair => pair.Key, pair => pair.Value))

.ToList();

for (int i = 0; i < functions.Count; i++)

{

string current\_y\_name = last\_four\_rows.Last().ElementAt(i + 1).Key;

double current\_y\_value = last\_four\_rows.Last()[current\_y\_name].RealValue;

double fi = functions[i].Evaluate(last\_four\_rows.Last()).RealValue;

double fi\_minus\_one = functions[i].Evaluate(last\_four\_rows.SkipLast(1).Last()).RealValue;

double fi\_minus\_two = functions[i].Evaluate(last\_four\_rows.SkipLast(2).Last()).RealValue;

double fi\_minus\_three = functions[i].Evaluate(last\_four\_rows.SkipLast(3).Last()).RealValue;

double delta\_f = fi - fi\_minus\_one;

double delta\_f\_two = fi - 2 \* fi\_minus\_one + fi\_minus\_two;

double delta\_f\_three = fi - 3 \* fi\_minus\_one + 3 \* fi\_minus\_two - fi\_minus\_three;

double new\_y\_value = current\_y\_value + fi \* h + Math.Pow(h, 2) / 2.0 \* delta\_f + 5 \* Math.Pow(h, 3) / 12.0 \* delta\_f\_two + 3 \* Math.Pow(h, 4) / 8.0 \* delta\_f\_three;

new\_ys.Add(current\_y\_name, new\_y\_value);

}

initialResult.Add(current\_x, new\_ys);

}

return initialResult;

}

}

Листинг класса AdamsBashforthMethod

using MathNet.Symbolics;

using NumericalMethods.Core.CauchyProblem.Interfaces;

namespace NumericalMethods.Core.CauchyProblem.Methods;

public class AdamsBashforthMethod : IMultiStepMethod

{

public ResultTable Calculate(List<SymbolicExpression> functions, double b, double h, ResultTable initialResult)

{

if (initialResult.CountRow < 1) throw new ArgumentException($"{nameof(initialResult)} count row less then 1");

Func<Dictionary<string, double>, Dictionary<string, FloatingPoint>> convert\_with\_floating\_point =

(row) => row

.Select(pair => new KeyValuePair<string, FloatingPoint>(pair.Key, pair.Value))

.ToDictionary(pair => pair.Key, pair => pair.Value);

double current\_x = initialResult["x"].Last().x + h;

if (initialResult.CountRow == 1)

{

Dictionary<string, double> new\_ys = new Dictionary<string, double>();

Dictionary<string, FloatingPoint> first\_row = convert\_with\_floating\_point(initialResult.GetRow(0));

for (int i = 0; i < functions.Count; i++)

{

string current\_y\_name = first\_row.ElementAt(i + 1).Key;

double current\_y\_value = first\_row[current\_y\_name].RealValue;

double new\_y\_value = current\_y\_value + h \* functions[i].Evaluate(first\_row).RealValue;

new\_ys.Add(current\_y\_name, new\_y\_value);

}

initialResult.Add(current\_x, new\_ys);

current\_x += h;

if (current\_x > b) return initialResult;

}

if (initialResult.CountRow == 2)

{

Dictionary<string, double> new\_ys = new Dictionary<string, double>();

Dictionary<string, FloatingPoint> first\_row = convert\_with\_floating\_point(initialResult.GetRow(0));

Dictionary<string, FloatingPoint> second\_row = convert\_with\_floating\_point(initialResult.GetRow(1));

for (int i = 0; i < functions.Count; i++)

{

string current\_y\_name = second\_row.ElementAt(i + 1).Key;

double current\_y\_value = second\_row[current\_y\_name].RealValue;

double new\_y\_value = current\_y\_value + h \* (3.0 / 2.0 \* functions[i].Evaluate(second\_row).RealValue

- 1.0 / 2.0 \* functions[i].Evaluate(first\_row).RealValue);

new\_ys.Add(current\_y\_name, new\_y\_value);

}

initialResult.Add(current\_x, new\_ys);

current\_x += h;

if (current\_x > b) return initialResult;

}

if (initialResult.CountRow == 3)

{

Dictionary<string, double> new\_ys = new Dictionary<string, double>();

Dictionary<string, FloatingPoint> first\_row = convert\_with\_floating\_point(initialResult.GetRow(0));

Dictionary<string, FloatingPoint> second\_row = convert\_with\_floating\_point(initialResult.GetRow(1));

Dictionary<string, FloatingPoint> third\_row = convert\_with\_floating\_point(initialResult.GetRow(2));

for (int i = 0; i < functions.Count; i++)

{

string current\_y\_name = third\_row.ElementAt(i + 1).Key;

double current\_y\_value = third\_row[current\_y\_name].RealValue;

double new\_y\_value = current\_y\_value + h \* (23.0 / 12.0 \* functions[i].Evaluate(third\_row).RealValue

- 4.0 / 3.0 \* functions[i].Evaluate(second\_row).RealValue

+ 5.0 / 12.0 \* functions[i].Evaluate(first\_row).RealValue);

new\_ys.Add(current\_y\_name, new\_y\_value);

}

initialResult.Add(current\_x, new\_ys);

current\_x += h;

if (current\_x > b) return initialResult;

}

if (initialResult.CountRow == 4)

{

Dictionary<string, double> new\_ys = new Dictionary<string, double>();

Dictionary<string, FloatingPoint> first\_row = convert\_with\_floating\_point(initialResult.GetRow(0));

Dictionary<string, FloatingPoint> second\_row = convert\_with\_floating\_point(initialResult.GetRow(1));

Dictionary<string, FloatingPoint> third\_row = convert\_with\_floating\_point(initialResult.GetRow(2));

Dictionary<string, FloatingPoint> four\_row = convert\_with\_floating\_point(initialResult.GetRow(3));

for (int i = 0; i < functions.Count; i++)

{

string current\_y\_name = four\_row.ElementAt(i + 1).Key;

double current\_y\_value = four\_row[current\_y\_name].RealValue;

double new\_y\_value = current\_y\_value + h \* (55.0 / 24.0 \* functions[i].Evaluate(four\_row).RealValue

- 59.0 / 24.0 \* functions[i].Evaluate(third\_row).RealValue

+ 37.0 / 24.0 \* functions[i].Evaluate(second\_row).RealValue

- 3.0 / 8.0 \* functions[i].Evaluate(first\_row).RealValue);

new\_ys.Add(current\_y\_name, new\_y\_value);

}

initialResult.Add(current\_x, new\_ys);

current\_x += h;

if (current\_x > b) return initialResult;

}

for (; current\_x <= b; current\_x += h)

{

Dictionary<string, double> new\_ys = new Dictionary<string, double>();

Dictionary<string, FloatingPoint> last\_five\_row = convert\_with\_floating\_point(initialResult.GetRow(initialResult.CountRow - 5));

Dictionary<string, FloatingPoint> last\_four\_row = convert\_with\_floating\_point(initialResult.GetRow(initialResult.CountRow - 4));

Dictionary<string, FloatingPoint> last\_third\_row = convert\_with\_floating\_point(initialResult.GetRow(initialResult.CountRow - 3));

Dictionary<string, FloatingPoint> last\_second\_row = convert\_with\_floating\_point(initialResult.GetRow(initialResult.CountRow - 2));

Dictionary<string, FloatingPoint> last\_first\_row = convert\_with\_floating\_point(initialResult.GetRow(initialResult.CountRow - 1));

for (int i = 0; i < functions.Count; i++)

{

string current\_y\_name = last\_first\_row.ElementAt(i + 1).Key;

double current\_y\_value = last\_first\_row[current\_y\_name].RealValue;

double new\_y\_value = current\_y\_value + h \* (1901.0 / 720.0 \* functions[i].Evaluate(last\_first\_row).RealValue

- 1387.0 / 360.0 \* functions[i].Evaluate(last\_second\_row).RealValue

+ 109.0 / 30.0 \* functions[i].Evaluate(last\_third\_row).RealValue

- 637.0 / 360.0 \* functions[i].Evaluate(last\_four\_row).RealValue

+ 251.0 / 720.0 \* functions[i].Evaluate(last\_five\_row).RealValue);

new\_ys.Add(current\_y\_name, new\_y\_value);

}

initialResult.Add(current\_x, new\_ys);

}

return initialResult;

}

}

Листинг класса AdamsMoultonMethod

using MathNet.Symbolics;

using NumericalMethods.Core.CauchyProblem.Interfaces;

namespace NumericalMethods.Core.CauchyProblem.Methods;

public class AdamsMoultonMethod : IMultiStepMethod

{

public ResultTable Calculate(List<SymbolicExpression> functions, double b, double h, ResultTable initialResult)

{

Func<Dictionary<string, double>, Dictionary<string, FloatingPoint>> convert\_with\_floating\_point =

(row) => row

.Select(pair => new KeyValuePair<string, FloatingPoint>(pair.Key, pair.Value))

.ToDictionary(pair => pair.Key, pair => pair.Value);

ResultTable filledResultTable = new AdamsBashforthMethod().Calculate(functions, b, h, initialResult);

Dictionary<string, FloatingPoint> second\_row = convert\_with\_floating\_point(filledResultTable.GetRow(1));

for (int i = 0; i < functions.Count; i++)

{

KeyValuePair<string, double> previous\_y = filledResultTable.GetRow(0).ElementAt(i + 1);

double new\_y\_value = previous\_y.Value + h \* functions[i].Evaluate(second\_row).RealValue;

filledResultTable.EditRowItem(1, previous\_y.Key, new\_y\_value);

}

Dictionary<string, FloatingPoint> third\_row = convert\_with\_floating\_point(filledResultTable.GetRow(2));

for (int i = 0; i < functions.Count; i++)

{

KeyValuePair<string, FloatingPoint> previous\_y = second\_row.ElementAt(i + 1);

double new\_y\_value = previous\_y.Value.RealValue + 1.0 / 2.0 \* h

\* (functions[i].Evaluate(third\_row).RealValue + functions[i].Evaluate(second\_row).RealValue);

filledResultTable.EditRowItem(2, previous\_y.Key, new\_y\_value);

}

Dictionary<string, FloatingPoint> four\_row = convert\_with\_floating\_point(filledResultTable.GetRow(3));

for (int i = 0; i < functions.Count; i++)

{

KeyValuePair<string, FloatingPoint> previous\_y = third\_row.ElementAt(i + 1);

double new\_y\_value = previous\_y.Value.RealValue + h

\* (5.0 / 12.0 \* functions[i].Evaluate(four\_row).RealValue

+ 2.0 / 3.0 \* functions[i].Evaluate(third\_row).RealValue

- 1.0 / 12.0 \* functions[i].Evaluate(second\_row).RealValue);

filledResultTable.EditRowItem(3, previous\_y.Key, new\_y\_value);

}

Dictionary<string, FloatingPoint> five\_row = convert\_with\_floating\_point(filledResultTable.GetRow(4));

for (int i = 0; i < functions.Count; i++)

{

KeyValuePair<string, FloatingPoint> previous\_y = four\_row.ElementAt(i + 1);

double new\_y\_value = previous\_y.Value.RealValue + h

\* (3.0 / 8.0 \* functions[i].Evaluate(five\_row).RealValue

+ 19.0 / 24.0 \* functions[i].Evaluate(four\_row).RealValue

- 5.0 / 24.0 \* functions[i].Evaluate(third\_row).RealValue

+ 1.0 / 24.0 \* functions[i].Evaluate(second\_row).RealValue);

filledResultTable.EditRowItem(4, previous\_y.Key, new\_y\_value);

}

for (int row\_index = 5; row\_index < filledResultTable.CountRow; row\_index++)

{

Dictionary<string, FloatingPoint> previous\_four\_row = convert\_with\_floating\_point(filledResultTable.GetRow(row\_index - 4));

Dictionary<string, FloatingPoint> previous\_third\_row = convert\_with\_floating\_point(filledResultTable.GetRow(row\_index - 3));

Dictionary<string, FloatingPoint> previous\_two\_row = convert\_with\_floating\_point(filledResultTable.GetRow(row\_index - 2));

Dictionary<string, FloatingPoint> previous\_one\_row = convert\_with\_floating\_point(filledResultTable.GetRow(row\_index - 1));

Dictionary<string, FloatingPoint> current\_row = convert\_with\_floating\_point(filledResultTable.GetRow(row\_index));

for (int i = 0; i < functions.Count; i++)

{

KeyValuePair<string, FloatingPoint> previous\_y = previous\_one\_row.ElementAt(i + 1);

double new\_y\_value = previous\_y.Value.RealValue + h

\* (251.0 / 720.0 \* functions[i].Evaluate(current\_row).RealValue

+ 646.0 / 720.0 \* functions[i].Evaluate(previous\_one\_row).RealValue

- 264.0 / 720.0 \* functions[i].Evaluate(previous\_two\_row).RealValue

+ 106.0 / 720.0 \* functions[i].Evaluate(previous\_third\_row).RealValue

- 19.0 / 720.0 \* functions[i].Evaluate(previous\_four\_row).RealValue);

filledResultTable.EditRowItem(row\_index, previous\_y.Key, new\_y\_value);

}

}

return filledResultTable;

}

}

Листинг класса IOneStepMethod

using MathNet.Symbolics;

namespace NumericalMethods.Core.CauchyProblem.Interfaces

{

public interface IOneStepMethod

{

double Calculate(SymbolicExpression function, double h, double x, (string name, double value) yi, Dictionary<string, FloatingPoint> conditions);

}

}

Листинг класса IMultiStepMethod

using MathNet.Symbolics;

namespace NumericalMethods.Core.CauchyProblem.Interfaces

{

public interface IMultiStepMethod

{

ResultTable Calculate(List<SymbolicExpression> functions, double b, double h, ResultTable initialResult);

}

}

Листинг класса ICauchyProblemSolver

namespace NumericalMethods.Core.CauchyProblem.Interfaces;

public interface ICauchyProblemSolver

{

ResultTable Calculate(double b, double h, (double x, Dictionary<string, double> ys) initialGuess);

}

Листинг класса CauchyProblemBuilder

using NumericalMethods.Core.CauchyProblem.Interfaces;

using NumericalMethods.Core.CauchyProblem.Methods;

namespace NumericalMethods.Core.CauchyProblem;

public static class CauchyProblemBuilder

{

public static ICauchyProblemSolver BuildWithOneStep(string function, OneStepMethods methodType)

{

return new CauchyProblemSolverWithOneStepMethod(GetChoice(methodType), function);

}

public static ICauchyProblemSolver BuildWithMultiStep(string function, MultiStepMethods multiStepMethodType, OneStepMethods oneStepMethodType, int preCalculatedPointsNumber)

{

IOneStepMethod one\_step\_method = GetChoice(oneStepMethodType);

return multiStepMethodType switch

{

MultiStepMethods.AdamsBashforth => new CauchyProblemSolverWithMultiStepMethod(new AdamsBashforthMethod(), one\_step\_method, preCalculatedPointsNumber, function),

MultiStepMethods.AdamsMoulton => new CauchyProblemSolverWithMultiStepMethod(new AdamsMoultonMethod(), one\_step\_method, preCalculatedPointsNumber, function)

};

}

public static ICauchyProblemSolver CreateAdams(string function, OneStepMethods oneStepMethodType)

{

IOneStepMethod one\_step\_method = GetChoice(oneStepMethodType);

return new CauchyProblemSolverWithMultiStepMethod(new AdamsMethod(), one\_step\_method, 4, function);

}

private static IOneStepMethod GetChoice(OneStepMethods methodType)

{

return methodType switch

{

OneStepMethods.Euler => new EulerMethod(),

OneStepMethods.EulerIterative => new EulerIterativeMethod(),

OneStepMethods.EulerImproved => new EulerImprovedMethod(),

OneStepMethods.EulerRecalculation => new EulerRecalculationMethod(),

OneStepMethods.RungeKuttaThridOrder => new RungeKuttaThridOrderMethod(),

OneStepMethods.RungeKuttaFourthOrder => new RungeKuttaFourthOrderMethod()

};

}

}

Листинг класса CauchyProblemSolverWithMultiStepMethod

using MathNet.Symbolics;

using NumericalMethods.Core.CauchyProblem.Interfaces;

namespace NumericalMethods.Core.CauchyProblem;

internal record class CauchyProblemSolverWithMultiStepMethod(IMultiStepMethod MultiStepMethod, IOneStepMethod OneStepMethod, int PreCalculatedPointsNumber, string function) : ICauchyProblemSolver

{

public ResultTable Calculate(double b, double h, (double x, Dictionary<string, double> ys) initialGuess)

{

Dictionary<string, double> sortedYs = initialGuess.ys

.OrderBy(pair => pair.Key)

.ToDictionary(pair => pair.Key, pair => pair.Value);

KeyValuePair<string, double> maxOrderY = sortedYs.Last();

var order = int.Parse(maxOrderY.Key[1..]);

Dictionary<string, string> functions = new Dictionary<string, string>();

for (int i = 0; i < order; i++)

{

functions.Add(sortedYs.ElementAt(i).Key, sortedYs.ElementAt(i + 1).Key);

}

functions.Add(maxOrderY.Key, function);

ResultTable result = new ResultTable(order);

result.Add(initialGuess.x, sortedYs);

int precalculated\_points\_count = 0;

for (double current\_x = initialGuess.x + h; precalculated\_points\_count <= PreCalculatedPointsNumber; current\_x += h)

{

Dictionary<string, double> ys = new Dictionary<string, double>();

Dictionary<string, FloatingPoint> conditions = new Dictionary<string, FloatingPoint>() { { "x", current\_x - h } };

foreach (var yName in initialGuess.ys.Keys)

{

conditions.Add(yName, result[yName].Last().yi);

}

foreach (KeyValuePair<string, double> y in initialGuess.ys)

{

var newValueY = OneStepMethod.Calculate(SymbolicExpression.Parse(functions[y.Key]), h, current\_x - h, (y.Key, conditions[y.Key].RealValue), conditions);

ys.Add(y.Key, newValueY);

}

result.Add(current\_x, ys);

precalculated\_points\_count++;

}

return MultiStepMethod.Calculate(

functions: functions.Select(pair => SymbolicExpression.Parse(pair.Value)).ToList(),

b: b,

h: h,

result);

}

}

Листинг класса CauchyProblemSolverWithOneStepMethod

using MathNet.Symbolics;

using NumericalMethods.Core.CauchyProblem.Interfaces;

namespace NumericalMethods.Core.CauchyProblem;

public record class CauchyProblemSolverWithOneStepMethod(IOneStepMethod Method, string function) : ICauchyProblemSolver

{

public ResultTable Calculate(double b, double h, (double x, Dictionary<string, double> ys) initialGuess)

{

Dictionary<string, double> sortedYs = initialGuess.ys

.OrderBy(pair => pair.Key)

.ToDictionary(pair => pair.Key, pair => pair.Value);

KeyValuePair<string, double> maxOrderY = sortedYs.Last();

var order = int.Parse(maxOrderY.Key[1..]);

Dictionary<string, string> functions = new Dictionary<string, string>();

for (int i = 0; i < order; i++)

{

functions.Add(sortedYs.ElementAt(i).Key, sortedYs.ElementAt(i + 1).Key);

}

functions.Add(maxOrderY.Key, function);

ResultTable result = new ResultTable(order);

result.Add(initialGuess.x, sortedYs);

for (double current\_x = initialGuess.x + h; Math.Round(current\_x, 7) <= Math.Round(b, 7); current\_x += h)

{

Dictionary<string, double> ys = new Dictionary<string, double>();

Dictionary<string, FloatingPoint> conditions = new Dictionary<string, FloatingPoint>() { { "x", current\_x - h } };

foreach (var yName in initialGuess.ys.Keys)

{

conditions.Add(yName, result[yName].Last().yi);

}

foreach (KeyValuePair<string, double> y in initialGuess.ys)

{

var newValueY = Method.Calculate(SymbolicExpression.Parse(functions[y.Key]), h, current\_x - h, (y.Key, conditions[y.Key].RealValue), conditions);

ys.Add(y.Key, newValueY);

}

result.Add(current\_x, ys);

}

return result;

}

}

Листинг класса MultiStepMethods

namespace NumericalMethods.Core.CauchyProblem;

public enum MultiStepMethods

{

AdamsBashforth,

AdamsMoulton

}

Листинг класса OneStepMethods

namespace NumericalMethods.Core.CauchyProblem;

public enum OneStepMethods

{

Euler,

EulerRecalculation,

EulerIterative,

EulerImproved,

RungeKuttaThridOrder,

RungeKuttaFourthOrder

}

Листинг класса ResultTable

namespace NumericalMethods.Core.CauchyProblem

{

public class ResultTable

{

public List<string> Header { get; }

private List<Dictionary<string, double>> Body { get; }

public int CountRow => Body.Count;

public ResultTable(int differentialOrder)

{

Header = new List<string>(differentialOrder + 1);

Header.Add("x");

for (int i = 0; i < differentialOrder + 1; i++)

{

Header.Add($"y{i}");

}

Body = new List<Dictionary<string, double>>();

}

public List<(double x, double yi)> this[string variableName]

{

get

{

int index = Header.IndexOf(variableName);

if (index == -1) throw new ArgumentException($"Invalid {nameof(variableName)}");

List<(double x, double yi)> result = new List<(double x, double yi)>();

foreach (Dictionary<string, double> row in Body)

{

result.Add((row["x"], row[variableName]));

}

return result;

}

}

public ResultTable Add(double x, Dictionary<string, double> ys)

{

Dictionary<string, double> sortedYs = ys

.OrderBy(y => y.Key)

.Prepend(new KeyValuePair<string, double>("x", x))

.ToDictionary(pair => pair.Key, pair => pair.Value);

Body.Add(sortedYs);

return this;

}

public ResultTable EditRowItem(int row\_index, string item\_key, double new\_value)

{

Body[row\_index][item\_key] = new\_value;

return this;

}

public List<Dictionary<string, double>> GetRows(Range range)

{

var (start, length) = range.GetOffsetAndLength(CountRow);

return Body.GetRange(start, length);

}

public Dictionary<string, double> GetRow(int index)

{

return Body[index];

}

}

}

Листинг класса MainWindow.xaml.cs

namespace NumericalMethods.WPFApplication

{

/// <summary>

/// Interaction logic for MainWindow.xaml

/// </summary>

public partial class MainWindow : Window

{

private readonly List<Point> \_points = new List<Point>();

private Function \_current\_function;

private double \_start\_x;

private double \_end\_x;

private double \_step;

private int \_order;

private ResultTable \_resultTable;

public MainWindow()

{

InitializeComponent();

Width = 1300;

Height = 750;

InitializeCauchyProblem();

}

private void InitializeCauchyProblem()

{

foreach (var item in Enum.GetValues(typeof(OneStepMethods)))

{

CauchyProblem\_FunctionTypeComboBox.Items.Add(item);

}

foreach (var item in Enum.GetValues(typeof(MultiStepMethods)))

{

CauchyProblem\_FunctionTypeComboBox.Items.Add(item);

}

CauchyProblem\_FunctionTypeComboBox.Items.Add("Adams");

foreach (var item in Enum.GetValues(typeof(InterpolationFunctionType)))

{

CauchyProblem\_InterpolationFunctionTypeComboBox.Items.Add(item);

}

foreach (var item in Enum.GetValues(typeof(OneStepMethods)))

{

CauchyProblem\_OneStepMethodComboBox.Items.Add(item);

}

CauchyProblem\_InterpolationFunctionTypeComboBox.SelectedItem = CauchyProblem\_InterpolationFunctionTypeComboBox.Items[0];

CauchyProblem\_FunctionTypeComboBox.SelectedItem = CauchyProblem\_FunctionTypeComboBox.Items[0];

CauchyProblem\_OneStepMethodComboBox.SelectedItem = CauchyProblem\_OneStepMethodComboBox.Items[CauchyProblem\_OneStepMethodComboBox.Items.Count -1];

CauchyProblem\_OrderTextBox.Text = "3";

CauchyProblem\_MaxOrderFunctionTextBox.Text = "-3\*y2-3\*y1-y0";

CauchyProblem\_EndXTextBox.Text = "10";

CauchyProblem\_StepTextBox.Text = "0,05";

CauchyProblem\_InitXTextBox.Text = "0";

CauchyProblem\_MainChart.Plot.Legend(enable: true);

}

private void CauchyProblem\_OrderSelectButton\_Click(object sender, RoutedEventArgs e)

{

if (String.IsNullOrEmpty(CauchyProblem\_OrderTextBox.Text))

{

MessageBox.Show("Заполните поле order");

CauchyProblem\_Properties.Visibility = Visibility.Collapsed;

return;

}

\_order = int.Parse(CauchyProblem\_OrderTextBox.Text);

CauchyProblem\_FillPropertiesGrid();

CauchyProblem\_Properties.Visibility = Visibility.Visible;

}

private string FormingContenLabelOrderFunction(int order)

{

if (order <= 3)

return new StringBuilder("Y").Append('\'', order).ToString();

else

return $"Y^({order})";

}

private void CauchyProblem\_FillPropertiesGrid()

{

CauchyProblem\_MaxOrderFunctionLabel.Content = FormingContenLabelOrderFunction(\_order);

}

private void CauchyProblem\_EditLabelsInitFunctions(double x)

{

foreach(var children in CauchyProblem\_InitFunctionsGrid.Children)

{

if(children is Grid)

{

foreach(var gridChildren in ((Grid)children).Children)

{

if(gridChildren is Label)

{

string content = ((Label)gridChildren).Content.ToString();

int f = content.IndexOf('(');

((Label)gridChildren).Content = content.Remove(content.LastIndexOf('(')) + $"({x})";

break;

}

}

}

}

}

private void CauchyProblem\_FunctionTypeComboBox\_SelectionChanged(object sender, SelectionChangedEventArgs e)

{

if(((ComboBox)sender).SelectedValue.ToString() == "Adams")

{

CauchyProblem\_PreCalcCountGrid.Visibility = Visibility.Collapsed;

CauchyProblem\_OneStepMethodComboBox.Visibility = Visibility.Visible;

return;

}

if (Enum.IsDefined(typeof(MultiStepMethods), ((ComboBox)sender).SelectedValue.ToString())){

CauchyProblem\_OneStepMethodComboBox.Visibility = Visibility.Visible;

CauchyProblem\_PreCalcCountGrid.Visibility = Visibility.Visible;

}

else

{

CauchyProblem\_OneStepMethodComboBox.Visibility = Visibility.Collapsed;

CauchyProblem\_PreCalcCountGrid.Visibility = Visibility.Collapsed;

}

}

private void CauchyProblem\_FillInitFunctionsGrid()

{

CauchyProblem\_InitFunctionsGrid.RowDefinitions.Clear();

CauchyProblem\_InitFunctionsGrid.Children.Clear();

Grid grid;

Label label;

TextBox textBox;

for (int i = 0; i < \_order; i++)

{

CauchyProblem\_InitFunctionsGrid.RowDefinitions.Add(new RowDefinition());

grid = new Grid()

{

Height = 30,

Margin = new Thickness(0, 10, 0, 0)

};

grid.ColumnDefinitions.Add(new ColumnDefinition()

{

Width = new GridLength(50)

});

grid.ColumnDefinitions.Add(new ColumnDefinition());

label = new Label()

{

Content = FormingContenLabelOrderFunction(i) + "()"

};

textBox = new TextBox();

grid.Children.Add(label);

Grid.SetColumn(grid.Children[grid.Children.Count - 1], 0);

grid.Children.Add(textBox);

Grid.SetColumn(grid.Children[grid.Children.Count - 1], 1);

CauchyProblem\_InitFunctionsGrid.Children.Add(grid);

Grid.SetRow(CauchyProblem\_InitFunctionsGrid.Children[CauchyProblem\_InitFunctionsGrid.Children.Count - 1], i);

}

}

private void CauchyProblem\_InitXTextBox\_TextChanged(object sender, TextChangedEventArgs e)

{

if (String.IsNullOrEmpty(((TextBox)sender).Text))

{

CauchyProblem\_ParentInitFunctionsGrid.Visibility = Visibility.Collapsed;

CauchyProblem\_InitFunctionsGrid.RowDefinitions.Clear();

CauchyProblem\_InitFunctionsGrid.Children.Clear();

}

else

{

if (((TextBox)sender).Text.Last() is '.' or '-') return;

if (CauchyProblem\_InitFunctionsGrid.Children.Count == 0)

{

CauchyProblem\_FillInitFunctionsGrid();

CauchyProblem\_ParentInitFunctionsGrid.Visibility = Visibility.Visible;

}

CauchyProblem\_EditLabelsInitFunctions(double.Parse(((TextBox)sender).Text));

}

}

private void CauchyProblem\_AddOnChartButton\_Click(object sender, RoutedEventArgs e)

{

if(String.IsNullOrEmpty(CauchyProblem\_OrderTextBox.Text)||

String.IsNullOrEmpty(CauchyProblem\_MaxOrderFunctionTextBox.Text)||

String.IsNullOrEmpty(CauchyProblem\_EndXTextBox.Text) ||

String.IsNullOrEmpty(CauchyProblem\_StepTextBox.Text)||

String.IsNullOrEmpty(CauchyProblem\_InitXTextBox.Text))

{

MessageBox.Show("заполните все поля");

return;

}

if(CauchyProblem\_InitFunctionsGrid.Children.Count == 0)

{

MessageBox.Show("заполните все поля");

return;

}

foreach (var children in CauchyProblem\_InitFunctionsGrid.Children)

{

if (children is Grid)

{

foreach (var gridChildren in ((Grid)children).Children)

{

if (gridChildren is TextBox)

{

if(String.IsNullOrEmpty((gridChildren as TextBox).Text))

{

MessageBox.Show("заполните все поля");

return;

}

}

}

}

}

Dictionary<string, double> initialGuess = CauchyProblem\_CreateInitialGuess();

if (CauchyProblem\_FunctionTypeComboBox.SelectedValue.ToString() == "Adams")

{

\_resultTable = CauchyProblemBuilder.CreateAdams(

function: CauchyProblem\_MaxOrderFunctionTextBox.Text,

oneStepMethodType: Enum.Parse<OneStepMethods>(CauchyProblem\_OneStepMethodComboBox.SelectedValue.ToString()))

.Calculate(

b: double.Parse(CauchyProblem\_EndXTextBox.Text),

h: double.Parse(CauchyProblem\_StepTextBox.Text),

initialGuess: (x: double.Parse(CauchyProblem\_InitXTextBox.Text), ys: initialGuess)

);

return;

}

if (Enum.IsDefined(typeof(MultiStepMethods), CauchyProblem\_FunctionTypeComboBox.SelectedValue.ToString()))

{

if (String.IsNullOrEmpty(CauchyProblem\_PreCalcCount.Text))

{

MessageBox.Show("заполните все поля");

return;

}

\_resultTable = CauchyProblemBuilder.BuildWithMultiStep(

function: CauchyProblem\_MaxOrderFunctionTextBox.Text,

multiStepMethodType: Enum.Parse<MultiStepMethods>(CauchyProblem\_FunctionTypeComboBox.SelectedValue.ToString()),

oneStepMethodType: Enum.Parse<OneStepMethods>(CauchyProblem\_OneStepMethodComboBox.SelectedValue.ToString()),

preCalculatedPointsNumber: int.Parse(CauchyProblem\_PreCalcCount.Text))

.Calculate(

b: double.Parse(CauchyProblem\_EndXTextBox.Text),

h: double.Parse(CauchyProblem\_StepTextBox.Text),

initialGuess: (x: double.Parse(CauchyProblem\_InitXTextBox.Text), ys: initialGuess)

);

}

if(Enum.IsDefined(typeof(OneStepMethods), CauchyProblem\_FunctionTypeComboBox.SelectedValue.ToString()))

{

\_resultTable = CauchyProblemBuilder.BuildWithOneStep(

function: CauchyProblem\_MaxOrderFunctionTextBox.Text,

methodType: Enum.Parse<OneStepMethods>(CauchyProblem\_FunctionTypeComboBox.SelectedValue.ToString()))

.Calculate(

b: double.Parse(CauchyProblem\_EndXTextBox.Text),

h: double.Parse(CauchyProblem\_StepTextBox.Text),

initialGuess: (x: double.Parse(CauchyProblem\_InitXTextBox.Text), ys: initialGuess)

);

}

foreach(var y in initialGuess)

{

List<(double x, double yi)> points = \_resultTable[y.Key];

CauchyProblem\_MainChart.Plot.AddScatter(points.Select(point=> point.x).ToArray(), points.Select(point => point.yi).ToArray(), lineWidth: 0, markerSize: 5, label: y.Key);

}

CauchyProblem\_MainChart.Refresh();

}

private new Dictionary<string, double> CauchyProblem\_CreateInitialGuess()

{

Dictionary<string, double> result = new Dictionary<string, double>();

for (int i = 0; i < CauchyProblem\_InitFunctionsGrid.Children.Count; i++)

{

if (CauchyProblem\_InitFunctionsGrid.Children[i] is Grid)

{

foreach (var gridChildren in ((Grid)CauchyProblem\_InitFunctionsGrid.Children[i]).Children)

{

if (gridChildren is TextBox)

{

result.Add($"y{i}", double.Parse(((TextBox)gridChildren).Text));

break;

}

}

}

}

return result;

}

private void CauchyProblem\_ClearChartButton\_Click(object sender, RoutedEventArgs e)

{

CauchyProblem\_MainChart.Plot.Clear();

CauchyProblem\_MainChart.Refresh();

}

private void CauchyProblem\_AddOnChartInterpolationButton\_Click(object sender, RoutedEventArgs e)

{

string? function\_type\_string = CauchyProblem\_InterpolationFunctionTypeComboBox.SelectedValue.ToString();

if (function\_type\_string is null) return;

Dictionary<string, double> initialGuess = CauchyProblem\_CreateInitialGuess();

InterpolationFunctionType interpolation\_type = Enum.Parse<InterpolationFunctionType>(function\_type\_string);

foreach (KeyValuePair<string, double> y in initialGuess)

{

IEnumerable<Point> calculated\_ys = \_resultTable[y.Key].Select(pair => new Point(pair.x, pair.yi));

IInterpolationFunction? interpolation\_function = InterpolationBuilder.Build(calculated\_ys, interpolation\_type);

if (interpolation\_function is null) return;

double start\_x = new Expression(CauchyProblem\_InitXTextBox.Text.Trim().Replace(',', '.')).calculate();

double end\_x = new Expression(CauchyProblem\_EndXTextBox.Text.Trim().Replace(',', '.')).calculate();

double step = new Expression(CauchyProblem\_StepTextBox.Text.Trim().Replace(',', '.')).calculate() / 5.0;

int roundNumbers = step.ToString().Contains(',') ? step.ToString().Split(',')[1].Length % 16 : 1;

var xs = new List<double>();

var ys = new List<double>();

for (double x = start\_x; Math.Round(x, roundNumbers) <= end\_x; x += step)

{

double? current\_y = interpolation\_function.Calculate(x);

if (current\_y is null) continue;

xs.Add(x);

ys.Add((double)current\_y);

}

double? last\_y = interpolation\_function.Calculate(end\_x);

if (last\_y is not null)

{

xs.Add(end\_x);

ys.Add(last\_y.Value);

}

CauchyProblem\_MainChart.Plot.AddScatter(xs.ToArray(), ys.ToArray(), lineWidth: 3, markerSize: 0, label: "i" + y.Key);

}

CauchyProblem\_MainChart.Refresh();

}

}

}

**Вывод**: в ходе выполнения лабораторной работы были изучены методы решения дифференциальных уравнений в частных производных и приобретены навыки решения практических задач с использованием программного средства Matlab.