**Оглавление**

[1. Вычисление значений выражений. 3](#_Toc182401315)

[1.1.Задание: 3](#_Toc182401316)

[1.2. Требование к приложению 3](#_Toc182401317)

[1.3. Описание проектируемого UI 3](#_Toc182401318)

[1.4.Код программы: 4](#_Toc182401319)

[1.5.Тестирование: 5](#_Toc182401320)

[2. Вычисление значения функции, заданной её графиком 6](#_Toc182401321)

[2.1.Задание: 7](#_Toc182401322)

[2.2. Требование к приложению 7](#_Toc182401323)

[2.3. Описание проектируемого UI 8](#_Toc182401324)

[2.4.Код программы: 8](#_Toc182401325)

[2.5.Тестирование: 10](#_Toc182401326)

[3. Проверка принадлежности точки заданной области на плоскости 12](#_Toc182401327)

[3.1.Задание: 12](#_Toc182401328)

[3.2. Требование к приложению 13](#_Toc182401329)

[3.3. Описание проектируемого UI 14](#_Toc182401330)

[3.4.Код программы: 14](#_Toc182401331)

[3.5.Тестирование: 16](#_Toc182401332)

[4. Вывод таблицы значений функции, заданной её графиком 18](#_Toc182401333)

[4.1.Задание 18](#_Toc182401334)

[4.2. Требование к приложению 19](#_Toc182401335)

[4.3. Описание проектируемого UI 19](#_Toc182401336)

[4.4. Код приложения 20](#_Toc182401337)

[4.5.Тестирование: 21](#_Toc182401338)

[5. Проверка серии «выстрелов по мишени» 22](#_Toc182401339)

[5.1.Задание: 22](#_Toc182401340)

[5.2. Требование к приложению 23](#_Toc182401341)

[5.3. Описание проектируемого UI 24](#_Toc182401342)

[5.4.Код программы: 24](#_Toc182401343)

[5.5.Тестирование: 26](#_Toc182401344)

[6. Вычисление суммы ряда с заданной точностью 27](#_Toc182401345)

[6.1.Задание: 27](#_Toc182401346)

[6.2. Требование к приложению 27](#_Toc182401347)

[6.3. Описание проектируемого UI 28](#_Toc182401348)

[6.4.Код программы: 28](#_Toc182401349)

[6.5.Тестирование: 29](#_Toc182401350)

[7. Вывод таблицы значений функции, вычисляемой с помощью ряда Тейлора 31](#_Toc182401351)

[7.1. Задание 31](#_Toc182401352)

[7.2. Требование к приложению 32](#_Toc182401353)

[7.3. Описание проектируемого UI 32](#_Toc182401354)

[7.4.Код приложения: 33](#_Toc182401355)

[7.5.Тестирование: 34](#_Toc182401356)

# 

# РЕШЕНИЕ СИСТЕМ ЛИНЕЙНЫХ АЛГЕБРАИЧЕСКИХ УРАВНЕНИЙ: ПРЯМЫЕ МЕТОДЫ

**Цели**:

1. Реализовать метод Гаусса решения системы линейных алгебраических уравнений.

2. Реализовать метод прогонки решения системы линейных алгебраических уравнений.

## 1. Реализация метода Гаусса.

1.1.Задание:Реализовать метод Гаусса решения системы линейных алгебраических уравнений (СЛАУ).

***Исходные данные:***

А – матрица;

В – вектор.

***Результаты:***

Решения системы линейных алгебраических уравнений методом Гаусса.

## 1.2. Логика решения задачи.

Прямой ход метода Гаусса — это алгоритм, используемый для решения систем линейных уравнений. Он состоит из последовательного преобразования системы уравнений в верхнетреугольный вид. Вот основные шаги:

**- Запись системы уравнений**: Начинаем с представления системы линейных уравнений в матричном виде, используя расширенную матрицу.

- **Выбор ведущего элемента**: На каждом шаге выбирается ведущий элемент (pivot), который обычно выбирается из первого столбца текущей подматрицы.

- **Элиминация**: С помощью элементарных преобразований строк (замена строки, умножение строки на число, сложение строк) приводим элементы ниже ведущего элемента в текущем столбце к нулю. Это делается для каждого столбца, начиная с первого и продвигаясь вправо и вниз.

- Повторение процесса: Процесс продолжается для следующего столбца и последующих строк, пока не будет достигнута последняя строка.

- **Получение верхнетреугольной формы**: В результате этих преобразований система уравнений преобразуется в верхнетреугольный вид, что позволяет легко найти значения переменных.

- **Обратный ход**: После того как система приведена к верхнетреугольному виду, можно использовать обратный ход (back substitution) для нахождения значений переменных, начиная с последнего уравнения.

## 1.3.1.Код программы Lab05\_01:

***Листинг 1 – Код программы Lab05\_01***

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using PuzanovVE.NM;

namespace PuzanovVE.NM

{

//14 вариант

internal class Program

{

static void Main(string[] args)

{

double[] Array1 = new double[] { 16, -4, -28 };

double[] Array2 = new double[] { -4, 10, 1 };

double[] Array3 = new double[] { -28, 1, 69 };

double[][] A = new double[][] { Array1, Array2, Array3 };

double[] B = new double[]

{

52,

-25,

-99

};

GaousseSolver solver = new GaousseSolver();

double [] result = solver.Solve(A, B);

Console.WriteLine("Решение СЛАУ:\n");

Console.Write("{");

foreach (var value in result)

{

Console.Write(value);

Console.Write("\t");

}

Console.Write("}");

Console.ReadLine();

}

}

}

## 1.3.2.Код класса GaousseSolver:

***Листинг 2 – Код класса*** GaousseSolver:

using System;

namespace PuzanovVE.NM

{

/// <summary>

/// Метод сопряженных градиентов для решения СЛАУ.

/// </summary>

public class ConjugateGradientSolver : SlauSolver

{

/// <summary>

/// Решает СЛАУ Ax = b методом сопряженных градиентов.

/// </summary>

/// <param name="A">Матрица коэффициентов.</param>

/// <param name="b">Правая часть.</param>

/// <param name="epsilon">Требуемая точность вычислений.</param>

/// <returns>Вектор неизвестных - решение СЛАУ.</returns>

public override double[] Solve(double[][] A, double[] b)

{

double epsilon = 0.0001;

int n = b.Length; // Определяем размерность системы

double[] x = new double[n]; // Начальное приближение x = 0

double[] r = new double[n]; // Вектор-невязка r = b - Ax

double[] p = new double[n]; // Вектор-градиент p = r

// Шаг 0: Вычисляем начальную невязку

r = Subtract(b, Multiply(A, x)); // r(0) = b - Ax(0)

p = (double[])r.Clone(); // p(0) = r(0)

double rsOld = Dot(r, r); // ||r(0)||^2

double rsNew;

// Итерационный процесс

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

{

// 3.1 Определяем величину шага

double[] Ap = Multiply(A, p); // A \* p(k)

double alpha = rsOld / Dot(p, Ap); // λ(k)

// 3.2 Вычисляем k-е приближение

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

{

x[i] += alpha \* p[i]; // x(k+1) = x(k) + λ(k) \* p(k)

}

// 3.3 Корректируем невязку

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

{

r[i] -= alpha \* Ap[i]; // r(k+1) = r(k) - λ(k) \* A \* p(k)

}

// 3.4 Проверка относительной невязки

rsNew = Dot(r, r); // ||r(k+1)||^2

if (Math.Sqrt(rsNew) <= epsilon)

{

break; // Условие выполнено, решение получено

}

// 3.5 Вычисляем коэффициент

double beta = rsNew / rsOld; // ω(k)

// 3.6 Задаем новый вектор-градиент

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

{

p[i] = r[i] + beta \* p[i]; // p(k+1) = r(k+1) + ω(k) \* p(k)

}

rsOld = rsNew; // Обновляем значение для следующей итерации

}

return x; // Возвращаем найденное решение

}

/// <summary>

/// Умножает матрицу на вектор.

/// </summary>

public static double[] Multiply(double[][] A, double[] x)

{

int n = A.Length; // Размерность матрицы

double[] result = new double[n]; // Результирующий вектор

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

{

result[i] = 0; // Инициализация элемента

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

{

result[i] += A[i][j] \* x[j]; // Суммирование произведений

}

}

return result; // Возвращаем результат

}

/// <summary>

/// Вычисляет скалярное произведение двух векторов.

/// </summary>

public static double Dot(double[] x, double[] y)

{

double result = 0; // Инициализация результата

for (int i = 0; i < x.Length; i++)

{

result += x[i] \* y[i]; // Суммирование произведений

}

return result; // Возвращаем скалярное произведение

}

/// <summary>

/// Вычитает один вектор из другого.

/// </summary>

public static double[] Subtract(double[] a, double[] b)

{

int n = a.Length; // Размерность векторов

double[] result = new double[n]; // Результирующий вектор

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

{

result[i] = a[i] - b[i]; // Вычитание соответствующих элементов

}

return result; // Возвращаем результат

}

}

}

## 1.4.Тестирование:

Контрольный пример 1:

Исходные данные:

А – матрица;

В – вектор.

Результат работы программы для указанных исходных данных приведён на рисунке 1

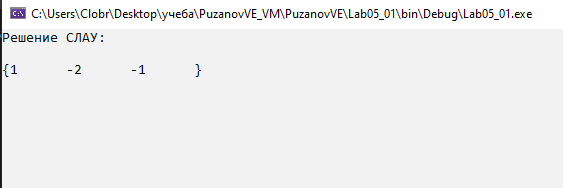


Рисунок 1 – Результат работы приложения Lab05\_01

## 2. Реализация метода прогонки.

2.1.Задание:Реализовать метод прогонки решения системы линейных алгебраических уравнений с трехдиагональной матрицей.

***Исходные данные:***

А – матрица;

В – вектор.

***Результаты:***

Решения системы линейных алгебраических уравнений методом Гаусса.

## 1.2. Логика решения задачи.

Прямой ход метода Гаусса — это алгоритм, используемый для решения систем линейных уравнений. Он состоит из последовательного преобразования системы уравнений в верхнетреугольный вид. Вот основные шаги:

**- Запись системы уравнений**: Начинаем с представления системы линейных уравнений в матричном виде, используя расширенную матрицу.

- **Выбор ведущего элемента**: На каждом шаге выбирается ведущий элемент (pivot), который обычно выбирается из первого столбца текущей подматрицы.

- **Элиминация**: С помощью элементарных преобразований строк (замена строки, умножение строки на число, сложение строк) приводим элементы ниже ведущего элемента в текущем столбце к нулю. Это делается для каждого столбца, начиная с первого и продвигаясь вправо и вниз.

- Повторение процесса: Процесс продолжается для следующего столбца и последующих строк, пока не будет достигнута последняя строка.

- **Получение верхнетреугольной формы**: В результате этих преобразований система уравнений преобразуется в верхнетреугольный вид, что позволяет легко найти значения переменных.

- **Обратный ход**: После того как система приведена к верхнетреугольному виду, можно использовать обратный ход (back substitution) для нахождения значений переменных, начиная с последнего уравнения.

## 1.3.1.Код программы Lab05\_01:

***Листинг 1 – Код программы Lab05\_01***

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using PuzanovVE.NM;

namespace PuzanovVE.NM

{

//14 вариант

internal class Program

{

static void Main(string[] args)

{

double[] Array1 = new double[] { 16, -4, -28 };

double[] Array2 = new double[] { -4, 10, 1 };

double[] Array3 = new double[] { -28, 1, 69 };

double[][] A = new double[][] { Array1, Array2, Array3 };

double[] B = new double[]

{

52,

-25,

-99

};

GaousseSolver solver = new GaousseSolver();

double [] result = solver.Solve(A, B);

Console.WriteLine("Решение СЛАУ:\n");

Console.Write("{");

foreach (var value in result)

{

Console.Write(value);

Console.Write("\t");

}

Console.Write("}");

Console.ReadLine();

}

}

}

## 1.3.2.Код класса GaousseSolver:

***Листинг 2 – Код класса*** GaousseSolver:

using System;

namespace PuzanovVE.NM

{

/// <summary>

/// Метод сопряженных градиентов для решения СЛАУ.

/// </summary>

public class ConjugateGradientSolver : SlauSolver

{

/// <summary>

/// Решает СЛАУ Ax = b методом сопряженных градиентов.

/// </summary>

/// <param name="A">Матрица коэффициентов.</param>

/// <param name="b">Правая часть.</param>

/// <param name="epsilon">Требуемая точность вычислений.</param>

/// <returns>Вектор неизвестных - решение СЛАУ.</returns>

public override double[] Solve(double[][] A, double[] b)

{

double epsilon = 0.0001;

int n = b.Length; // Определяем размерность системы

double[] x = new double[n]; // Начальное приближение x = 0

double[] r = new double[n]; // Вектор-невязка r = b - Ax

double[] p = new double[n]; // Вектор-градиент p = r

// Шаг 0: Вычисляем начальную невязку

r = Subtract(b, Multiply(A, x)); // r(0) = b - Ax(0)

p = (double[])r.Clone(); // p(0) = r(0)

double rsOld = Dot(r, r); // ||r(0)||^2

double rsNew;

// Итерационный процесс

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

{

// 3.1 Определяем величину шага

double[] Ap = Multiply(A, p); // A \* p(k)

double alpha = rsOld / Dot(p, Ap); // λ(k)

// 3.2 Вычисляем k-е приближение

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

{

x[i] += alpha \* p[i]; // x(k+1) = x(k) + λ(k) \* p(k)

}

// 3.3 Корректируем невязку

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

{

r[i] -= alpha \* Ap[i]; // r(k+1) = r(k) - λ(k) \* A \* p(k)

}

// 3.4 Проверка относительной невязки

rsNew = Dot(r, r); // ||r(k+1)||^2

if (Math.Sqrt(rsNew) <= epsilon)

{

break; // Условие выполнено, решение получено

}

// 3.5 Вычисляем коэффициент

double beta = rsNew / rsOld; // ω(k)

// 3.6 Задаем новый вектор-градиент

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

{

p[i] = r[i] + beta \* p[i]; // p(k+1) = r(k+1) + ω(k) \* p(k)

}

rsOld = rsNew; // Обновляем значение для следующей итерации

}

return x; // Возвращаем найденное решение

}

/// <summary>

/// Умножает матрицу на вектор.

/// </summary>

public static double[] Multiply(double[][] A, double[] x)

{

int n = A.Length; // Размерность матрицы

double[] result = new double[n]; // Результирующий вектор

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

{

result[i] = 0; // Инициализация элемента

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

{

result[i] += A[i][j] \* x[j]; // Суммирование произведений

}

}

return result; // Возвращаем результат

}

/// <summary>

/// Вычисляет скалярное произведение двух векторов.

/// </summary>

public static double Dot(double[] x, double[] y)

{

double result = 0; // Инициализация результата

for (int i = 0; i < x.Length; i++)

{

result += x[i] \* y[i]; // Суммирование произведений

}

return result; // Возвращаем скалярное произведение

}

/// <summary>

/// Вычитает один вектор из другого.

/// </summary>

public static double[] Subtract(double[] a, double[] b)

{

int n = a.Length; // Размерность векторов

double[] result = new double[n]; // Результирующий вектор

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

{

result[i] = a[i] - b[i]; // Вычитание соответствующих элементов

}

return result; // Возвращаем результат

}

}

}

## 1.4.Тестирование:

Контрольный пример 1:

Исходные данные:

А – матрица;

В – вектор.

Результат работы программы для указанных исходных данных приведён на рисунке 1

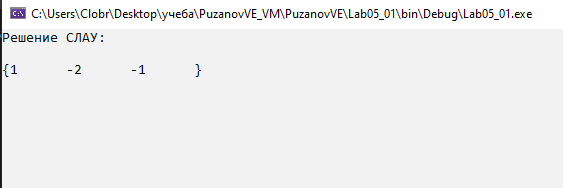


Рисунок 1 – Результат работы приложения Lab05\_01