МИНИСТЕРСТВО НАУКИ
И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное
образовательное учреждение высшего образования
«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

НГТУ
НЭТИ | **Факультет прикладной математики и информатики**

Кафедра прикладной математики

Практическое задание № 2

по дисциплине «Цифровые модели и оценивание параметров»

# НЕЛИНЕЙНЫЕ ОБРАТНЫЕ ЗАДАЧИ


Группа ПМ-05

Вариант 3

БОЛДЫРЕВ СЕРГЕЙ

ГРУШЕВ АНДРЕЙ

ПУЧКОВ ДМИТРИЙ

Преподаватели ВАГИН ДЕНИС ВЛАДИМИРОВИЧ


Новосибирск, 2023

## 1 Задание

Положение приёмников: M1(200,0,0), N1(300,0,0); M2(500,0,0), N2(600,0,0); M3(1000,0,0), N3(1100,0,0). Положение источников: A1(0,–500,0), B1(100,–500,0); A2(0,0,0), B2(100,0,0); A3(0,500,0), B3(100,500,0).

Однородное полупространство. Приёмники 1–3. Источник 2. Определить значение  полупространства. Добавить шум, равный 10 % от значения измерения.

## 2 Решение

Формула связи электрического тока в источнике (AB) и напряжения в приёмнике (MN):

$$V_{AB}^{MN} = \frac{I}{2\pi\sigma}\left(\left(\frac{1}{r_B^M} - \frac{1}{r_A^M}\right) - \left(\frac{1}{r_B^N} - \frac{1}{r_A^N}\right)\right)$$

Для решения нашей задачи необходимо собрать СЛАУ следующего вида:

$$A_{qs} = \sum_i^{N_i}\sum_j^{N_j}\left(w_k^i\right)^2 \frac{\partial\left(\delta\varepsilon_k^i(p^n)\right)}{\partial p_q}\frac{\partial\left(\delta\varepsilon_k^i(p^n)\right)}{\partial p_s}$$

$$b_q = -\sum_i^{N_i}\sum_j^{N_j}\left(w_k^i\right)^2 \delta\varepsilon_k^i(p^n)\frac{\partial\left(\delta\varepsilon_k^i(p^n)\right)}{\partial p_s}$$

В качестве **ε** мы рассматриваем сумму напряжений в каждом из приёмников от всех источников (в нашем случае от одного источника), $p^n$ – вектор токов.

Производные будут посчитаны аналитически. Формула:

$$V_{AB}^{MN} = \frac{-I}{2\pi\sigma^2}\left(\left(\frac{1}{r_B^M} - \frac{1}{r_A^M}\right) - \left(\frac{1}{r_B^N} - \frac{1}{r_A^N}\right)\right)$$

Продолжение итерационного процесса даст истинное значение искомой силы тока.

## 3 Результаты работы программы

Точность решения: 1e-10.

Максимум итераций: 1000.

$I = 1$.

$\sigma = 0{,}1$.

$\sigma_{\text{нач}} = 0{,}01$.

Тест 1 – Без зашумления

| Итерация | $\sigma$ |
|---|---|
| 1 | 1,900000E-002 |
| 2 | 3,439000E-002 |
| 3 | 5,695328E-002 |
| 4 | 8,146980E-002 |
| 5 | 9,656632E-002 |
| 6 | 9,988210E-002 |
| 7 | 9,999986E-002 |
| 8 | 1,000000E-001 |

Тест 2 - Зашумление по всем 3 приёмникам на плюс 10 процентов.

| **Итерация** | $\sigma$ |
|---|---|
| 1 | 1,890000E-002 |
| 2 | 3,387069E-002 |
| 3 | 5,512192E-002 |
| 4 | 7,682115E-002 |
| 5 | 8,872592E-002 |
| 6 | 9,085666E-002 |
| 7 | 9,090906E-002 |
| 8 | 9,090909E-002 |

Тест 3 - Зашумление по всем 3 приёмникам на минус 10 процентов.

| **Итерация** | $\sigma$ |
|---|---|
| 1 | 1,910000E-002 |
| 2 | 3,491671E-002 |
| 3 | 5,886083E-002 |
| 4 | 8,654028E-002 |
| 5 | 1,056776E-001 |
| 6 | 1,108454E-001 |
| 7 | 1,111105E-001 |
| 8 | 1,111111E-001 |

Тест 4 - Зашумление лишь одного приёмника на 1 процент.

| **Итерация** | $\sigma$ |
|---|---|
| 1 | 1,899671E-002 |
| 2 | 3,437280E-002 |
| 3 | 5,689185E-002 |
| 4 | 8,131041E-002 |
| 5 | 9,628953E-002 |
| 6 | 9,955735E-002 |
| 7 | 9,967202E-002 |
| 8 | 9,967215E-002 |
| 9 | 9,967215E-002 |
| … | … |
| 999 | 9,967215E-002 |
| 1000 | 9,967215E-002 |

## 4 Вывод

При одинаковом по силе и знаку зашумлении на всех приёмниках полученный результат отличается от истинного обратно пропорционально зашумлению. То есть при зашумлении на плюс 10%, то есть умножении разности потенциалов на 1,1, полученное значение удельной теплоёмкости будет равно истинному поделённому на 1,1.

При разном по силе и/или знаку зашумлении хотя бы на двух приёмниках наш алгоритм не выходит по невязке, а выходит только по количеству итераций. Однако он продолжает

давать близкие к истинным значениям результаты и при этом довольно быстро начинает стагнировать. Хорошим решением было бы реализовать выход из-за стагнации.

## 5 Код программы

**Program.cs**

```
using ConsoleApp1;

//Console.WriteLine("Hello, World!");
public class Program
{
    static void Main(string[] args)
    {
        Tests.CheckFirstTestFromTrainingManualBreaker();
    }
}
```

**Data.cs**

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Numerics;

using dVector = System.Collections.Generic.List<double>;
using iVector = System.Collections.Generic.List<int>;

namespace ConsoleApp1
{
    internal class Data
    {
        public static (Forward, dVector) FirstTestFromTrainingManual()
        {
            // Источники
            (dVector, dVector)[] sourcesCoor = new (dVector, dVector)[1];
            dVector A = new dVector { 0, 0, 0 };
            dVector B = new dVector { 100, 0, 0 };
            sourcesCoor[0] = (A, B);

            // Приёмники
            (dVector, dVector)[] receiversCoor = new (dVector, dVector)[3];
            dVector M1 = new dVector{ 200, 0, 0 };
            dVector N1 = new dVector { 300, 0, 0 };
            dVector M2 = new dVector { 500, 0, 0 };
            dVector N2 = new dVector { 600, 0, 0 };
            dVector M3 = new dVector { 1000, 0, 0 };
            dVector N3 = new dVector { 1100, 0, 0 };
            receiversCoor[0] = (N1, M1);
            receiversCoor[1] = (N2, M2);
            receiversCoor[2] = (N3, M3);

            dVector I = new dVector { 1 }; // Сила тока
            double sigma = 0.1;

            Forward syntheticDataGenerator = new(sourcesCoor, receiversCoor, I, sig-
ma, 3);

            return (syntheticDataGenerator,
syntheticDataGenerator.SolveForwardProblem());
        }

    }
```

```
}
```

**Algebra.cs**

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Numerics;
using System.Windows;

using dVector = System.Collections.Generic.List<double>;
using iVector = System.Collections.Generic.List<int>;


namespace ConsoleApp1
{
    public static class Algebra
    {
        public static dVector Add(dVector x, dVector y)
        {
            if (x.Count != y.Count)
            {
                throw new Exception("При сложении векторов было обнаружено, что век-
торы разной размерности");
            }
            dVector result = new dVector(new double[x.Count]);
            for (int i = 0; i < x.Count; i++)
            {
                result[i] = x[i] + y[i];
            }
            return result;
        }

        public static iVector Add(iVector x, iVector y)
        {
            if (x.Count != y.Count)
            {
                throw new Exception("При сложении векторов было обнаружено, что век-
торы разной размерности");
            }
            iVector result = new iVector(new int[x.Count]);
            for (int i = 0; i < x.Count; i++)
            {
                result[i] = x[i] + y[i];
            }
            return result;
        }

        public static double Norm(iVector x)
        {
            double sum = 0;
            foreach (var el in x)
            {
                sum += el * el;
            }
            return Math.Sqrt(sum);
        }

        public static double Norm(dVector x)
        {
            double sum = 0;
            foreach (var el in x)
```

```
            {
                sum += el * el;
            }
            return Math.Sqrt(sum);
        }

        public static void AddToLeft(dVector x, dVector y)
        {
            if (x.Count != y.Count)
            {
                throw new Exception("При сложении векторов было обнаружено, что век-
торы разной размерности");
            }
            for (int i = 0; i < x.Count; i++)
            {
                x[i] += y[i];
            }
        }

        public static void AddToLeft(iVector x, iVector y)
        {
            if (x.Count != y.Count)
            {
                throw new Exception("При сложении векторов было обнаружено, что век-
торы разной размерности");
            }
            for (int i = 0; i < x.Count; i++)
            {
                x[i] += y[i];
            }
        }

        public static void AddToRight(dVector x, dVector y)
        {
            if (x.Count != y.Count)
            {
                throw new Exception("При сложении векторов было обнаружено, что век-
торы разной размерности");
            }
            for (int i = 0; i < x.Count; i++)
            {
                y[i] += x[i];
            }
        }

        public static void AddToRight(iVector x, iVector y)
        {
            if (x.Count != y.Count)
            {
                throw new Exception("При сложении векторов было обнаружено, что век-
торы разной размерности");
            }
            for (int i = 0; i < x.Count; i++)
            {
                y[i] += x[i];
            }
        }

        public static void Scale(dVector x, double k)
        {
            for (int i = 0; i < x.Count; i++)
            {
                x[i] *= k;
            }
        }
```

```
        public static void Scale(iVector x, int k)
        {
            for (int i = 0; i < x.Count; i++)
            {
                x[i] *= k;
            }
        }

        public static dVector CreateAndScale(dVector x, double k)
        {
            dVector result = new dVector(x.Count);
            for (int i = 0; i < x.Count; i++)
            {
                result[i] = x[i] * k;
            }
            return result;
        }

        public static iVector CreateAndScale(iVector x, int k)
        {
            iVector result = new iVector(new int[x.Count]);
            for (int i = 0; i < x.Count; i++)
            {
                result[i] = x[i] * k;
            }
            return result;
        }

        public static void SubtrFromLeft(dVector x, dVector y)
        {
            if (x.Count != y.Count)
            {
                throw new Exception("При вычитании векторов было обнаружено, что
векторы разной размерности");
            }
            for (int i = 0; i < x.Count; i++)
            {
                x[i] -= y[i];
            }
        }

        public static void SubtrFromLeft(iVector x, iVector y)
        {
            if (x.Count != y.Count)
            {
                throw new Exception("При вычитании векторов было обнаружено, что
векторы разной размерности");
            }
            for (int i = 0; i < x.Count; i++)
            {
                x[i] -= y[i];
            }
        }

        public static void SubtrFromRight(dVector x, dVector y)
        {
            if (x.Count != y.Count)
            {
                throw new Exception("При вычитании векторов было обнаружено, что
векторы разной размерности");
            }
            for (int i = 0; i < x.Count; i++)
            {
                y[i] -= x[i];
```

```
            }
        }

        public static void SubtrFromRight(iVector x, iVector y)
        {
            if (x.Count != y.Count)
            {
                throw new Exception("При вычитании векторов было обнаружено, что
векторы разной размерности");
            }
            for (int i = 0; i < x.Count; i++)
            {
                y[i] -= x[i];
            }
        }

        public static void Swap(dVector x, dVector y)
        {
            var tmp = x;
            x = y;
            y = tmp;
        }

        public static void Swap(iVector x, iVector y)
        {
            var tmp = x;
            x = y;
            y = tmp;
        }

    }
}
```

**Forward.cs**

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Numerics;

using dVector = System.Collections.Generic.List<double>;
using iVector = System.Collections.Generic.List<int>;

namespace ConsoleApp1
{
    public class Forward
    {
        public int ProblemDimension;

        public (dVector, dVector)[] ReceiversCoor;

        public (dVector, dVector)[] SourcesCoor;

        public double Sigma;

        public dVector SourcesCurrentStrength;

        public Forward((dVector, dVector)[] sourcesCoor, (dVector, dVector)[] re-
ceiversCoor,
                                dVector sourcesCurrentStrength, double sigma, int
problemDimension = 3)
        {
```

```
            ProblemDimension = problemDimension;
            Sigma = sigma;
            SourcesCoor = sourcesCoor;
            ReceiversCoor = receiversCoor;
            SourcesCurrentStrength = sourcesCurrentStrength;
        }

        public dVector SolveForwardProblem()
        {
            dVector receiversPotentialDifference = new dVector(new dou-
ble[ReceiversCoor.Length]);
            for (int i = 0; i < ProblemDimension; i++)
            {
                receiversPotentialDifference[i] = EP(ReceiversCoor[i]);
            }
            return receiversPotentialDifference;
        }

        public double R(dVector a, dVector b)
        {
            double sum = 0;
            for (int i = 0; i < ProblemDimension; i++)
            {
                sum += Math.Pow(Math.Abs(a[i] - b[i]), 2);
            }
            return Math.Sqrt(sum);
        }

        public double OneToOneEP((dVector, dVector) sourceCoor, (dVector, dVector)
receiverCoor, double currentStrength)
        {
            double forPotential1 = 1 / R(sourceCoor.Item1, receiverCoor.Item1) - 1 /
R(sourceCoor.Item2, receiverCoor.Item1);
            double forPotential2 = 1 / R(sourceCoor.Item1, receiverCoor.Item2) - 1 /
R(sourceCoor.Item2, receiverCoor.Item2);
            return (forPotential1 - forPotential2) * currentStrength / (2 * Math.PI
* Sigma);
        }

        public double EP((dVector, dVector) receiverCoor)
        {
            double sum = 0;
            for (int i = 0; i < SourcesCoor.Length; i++)
            {
                sum += OneToOneEP(SourcesCoor[i], receiverCoor, SourcesCurrent-
Strength[i]);
            }
            return sum;
        }

    }
}
```

**Inverse.cs**

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Numerics;

using dVector = System.Collections.Generic.List<double>;
using iVector = System.Collections.Generic.List<int>;
```

```
namespace ConsoleApp1
{
    public class Inverse : Forward
    {
        //private dVector _actualReceiversPotentialDiff;
        public dVector ActualReceiversPotentialDiff;

        public Inverse((dVector, dVector)[] sourcesCoor, (dVector, dVector)[] re-
ceiversCoor, dVector sourcesCurrentStrength,
                                dVector receiversPotentialDiff, double sigma, int
problemDimension = 3) :
                                base(sourcesCoor, receiversCoor, sourcesCurrent-
Strength, sigma, problemDimension)
        {
            ActualReceiversPotentialDiff = receiversPotentialDiff;
        }

        public Inverse(Forward forwardProblem, double sigma, dVector receiversPoten-
tialDiff) :
                                base(forwardProblem.SourcesCoor, forwardProb-
lem.ReceiversCoor,
                                forwardProblem.SourcesCurrentStrength, sigma, forward-
Problem.ProblemDimension)
        {
            ActualReceiversPotentialDiff = receiversPotentialDiff;
        }

        public double DeltaEP(int i)
        {
            var hypotheticalValues = EP(ReceiversCoor[i]);
            var realValues = ActualReceiversPotentialDiff[i];

            return hypotheticalValues - realValues;
        }

        public double DEP(int q, int k)
        {
            double forPotential1 = 1 / R(SourcesCoor[q].Item1, Receiv-
ersCoor[k].Item1) - 1 / R(SourcesCoor[q].Item2, ReceiversCoor[k].Item1);
            double forPotential2 = 1 / R(SourcesCoor[q].Item1, Receiv-
ersCoor[k].Item2) - 1 / R(SourcesCoor[q].Item2, ReceiversCoor[k].Item2);
            return -SourcesCurrentStrength[q] * (forPotential1 - forPotential2) / (2
* Math.PI * Sigma * Sigma);
        }

        public double W(int k)
        {
            return 1 / ActualReceiversPotentialDiff[k];
        }

        public double AQS(int q, int s)
        {
            double sum = 0;
            for (int i = 0; i < ActualReceiversPotentialDiff.Count; i++)
            {
                sum += W(i) * W(i) * DEP(q, i) * DEP(s, i);
            }
            return sum;
        }

        public double BQ(int q)
        {
            double sum = 0;
            for (int i = 0; i < ActualReceiversPotentialDiff.Count; i++)
```

```
            {
                sum -= W(i) * W(i) * DeltaEP(i) * DEP(q, i);
            }
            return sum;
        }

        public dVector Iter(SLAU SLAE)
        {
            for (int i = 0; i < 1; i++)
            {
                for (int j = 0; j < 1; j++)
                {
                    SLAE.A[i][j] = AQS(i, j);
                }
            }

            for (int i = 0; i < 1; i++)
            {
                SLAE.B[i] = BQ(i);
            }

            return SLAE.SolveSLAEGauss();
        }

        public double SolveInverseProblem(double acc, int maxIter)
        {
            //dVector a_ = new dVector(1);
            dVector[] A = new dVector[100];
            A[1] = new dVector(new double[1]);
            for (int i = 0; i < 1; i++)
            {
                A[i] = new dVector(new double[1]);
            }
            dVector B = new dVector(new double[1]);
            SLAU SLAE = new(A, B, 1);

            dVector errorVector = CalcErrorVector();

            for (int i = 0; i < maxIter && Algebra.Norm(errorVector) > acc; i++)
            {
                Sigma += Iter(SLAE)[0];
                errorVector = CalcErrorVector();
            }

            return Sigma;
        }

        public dVector CalcErrorVector()
        {
            dVector errorVector = new dVector(new dou-
ble[ActualReceiversPotentialDiff.Count]);
            for (int i = 0; i < ActualReceiversPotentialDiff.Count; i++)
            {
                errorVector[i] = EP(ReceiversCoor[i]);
            }
            Algebra.SubtrFromLeft(errorVector, ActualReceiversPotentialDiff);
            return errorVector;
        }

        public double SolveInverseProblemWithLog(double acc, int maxIter)
        {
            dVector[] A = new dVector[100];
            A[1] = new dVector(new double[1]);
            for (int i = 0; i < 1; i++)
            {
```

```
                A[i] = new dVector(new double[1]);
            }
            dVector B = new dVector(new double[1]);
            SLAU SLAE = new(A, B, 1);

            dVector errorVector = CalcErrorVector();

            Console.Write("Изначальный: ");
            WriteLineSigma();

            for (int i = 0; i < maxIter && Algebra.Norm(errorVector) > acc; i++)
            {
                Sigma += Iter(SLAE)[0];
                Console.Write($"{i + 1} итерация: ");
                WriteLineSigma();
                errorVector = CalcErrorVector();
            }
            Console.WriteLine();
            Console.Write("Вывод: ");
            WriteLineSigma();


            return Sigma;
        }

        public void WriteLineSigma()
        {
            Console.WriteLine($"{Sigma.ToString("E")}  ");
        }

    }
}
```

**SLAU.cs**

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Numerics;

using dVector = System.Collections.Generic.List<double>;
using iVector = System.Collections.Generic.List<int>;


namespace ConsoleApp1
{
    public class SLAU

    {
        public dVector[] A;

        public dVector B;

        public int MatrixDimension;
        public double NearZero = 1e-14;

        public SLAU(dVector[] a, dVector b, int matrixDimension)
        {
            MatrixDimension = matrixDimension;
            A = a;
            B = b;
        }
```

```
        private bool IsNearZero(double x)
        {
            if (x < NearZero && x > -NearZero)
            {
                return true;
            }
            return false;
        }

        public dVector SolveSLAEGauss()
        {
            for (int i = 0; i < MatrixDimension; i++)
            {
                if (IsNearZero(A[i][i]))
                {
                    bool foundStringForSwap = false;
                    for (int j = i + 1; j < MatrixDimension; j++)
                    {
                        if (IsNearZero(A[j][i]))
                        {
                            continue;
                        }
                        Algebra.Swap(A[j], A[i]);
                        foundStringForSwap = true;
                        break;
                    }
                    if (foundStringForSwap == false)
                    {
                        throw new UnsolvableMatrixException("СЛАУ не решается");
                    }
                }
                for (int j = i + 1; j < MatrixDimension; j++)
                {
                    double coef = A[j][i] / A[i][i];
                    Algebra.SubtrFromLeft(A[j], Algebra.CreateAndScale(A[i], coef));
                    B[j] -= B[i] * coef;
                }
            }
            dVector result = new dVector(new double[MatrixDimension]);
            for (int i = MatrixDimension - 1; i >= 0; i--)
            {
                double sum = 0;
                for (int j = i + 1; j < MatrixDimension; j++)
                {
                    sum += result[j] * A[i][j];
                }
                result[i] = (B[i] - sum) / A[i][i];
            }
            return result;
        }

    }
}
```

**DimensionGuarantor.cs**

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Numerics;
```

```
using dVector = System.Collections.Generic.List<double>;
using iVector = System.Collections.Generic.List<int>;


namespace ConsoleApp1
{
    internal class DimensionGuarantor
    {
        public static bool IsDimensionsCoincideArrOfPairCoorsInside((dVector, dVec-
tor)[] coordinates, int dimension)
        {
            foreach (var vectorPair in coordinates)
            {
                if (vectorPair.Item1.Count != dimension || vectorPair.Item2.Count !=
dimension)
                {
                    return false;
                }
            }
            return true;
        }

        public static bool IsDimensionsCoincideArrOfPairCoorsOutside((dVector, dVec-
tor)[] coordinates, dVector vector)
        {
            if (coordinates.Length != vector.Count)
            {
                return false;
            }
            return true;
        }

        public static bool IsDimensionsCoincide(dVector vector, int dimension)
        {
            if (vector.Count != dimension)
            {
                return false;
            }
            return true;
        }

        public static bool IsDimensionsCoincide(iVector vector, int dimension)
        {
            if (vector.Count != dimension)
            {
                return false;
            }
            return true;
        }

        public static bool IsDimensionsCoincideMatrixFull(dVector[] matrix, int di-
mension)
        {
            if (matrix.Length != dimension)
            {
                return false;
            }
            foreach (var str in matrix)
            {
                if (str.Count != dimension)
                {
                    return false;
                }
            }
            return true;
```

```
        }

    }
}
```

**UnsolvableMatrixException.cs**

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Numerics;

using dVector = System.Collections.Generic.List<double>;
using iVector = System.Collections.Generic.List<int>;


namespace ConsoleApp1
{
    internal class DimensionGuarantor
    {
        public static bool IsDimensionsCoincideArrOfPairCoorsInside((dVector, dVec-
tor)[] coordinates, int dimension)
        {
            foreach (var vectorPair in coordinates)
            {
                if (vectorPair.Item1.Count != dimension || vectorPair.Item2.Count !=
dimension)
                {
                    return false;
                }
            }
            return true;
        }

        public static bool IsDimensionsCoincideArrOfPairCoorsOutside((dVector, dVec-
tor)[] coordinates, dVector vector)
        {
            if (coordinates.Length != vector.Count)
            {
                return false;
            }
            return true;
        }

        public static bool IsDimensionsCoincide(dVector vector, int dimension)
        {
            if (vector.Count != dimension)
            {
                return false;
            }
            return true;
        }

        public static bool IsDimensionsCoincide(iVector vector, int dimension)
        {
            if (vector.Count != dimension)
            {
                return false;
            }
            return true;
        }
```

```
        public static bool IsDimensionsCoincideMatrixFull(dVector[] matrix, int di-
mension)
        {
            if (matrix.Length != dimension)
            {
                return false;
            }
            foreach (var str in matrix)
            {
                if (str.Count != dimension)
                {
                    return false;
                }
            }
            return true;
        }

    }
}
```

**Tests.cs**

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Numerics;

using dVector = System.Collections.Generic.List<double>;
using iVector = System.Collections.Generic.List<int>;


namespace ConsoleApp1
{
    internal class Tests
    {
        /// <summary>
        /// 1 источник,
        /// 3 приёмник, sigma = 0.1,
        /// начальная апроксимация = 0.01,
        /// точность = 1e-10,
        /// максимальное кол-во итераций = 1000
        /// </summary>
        public static void CheckFirstTestFromTrainingManual()
        {
            (Forward problemFromTrainingManual, dVector ReceiversPotentialDiff) =
                                                                Da-
ta.FirstTestFromTrainingManual();

            double new_sigma = 0.01;

            Inverse inverseProblem = new(problemFromTrainingManual, new_sigma, Re-
ceiversPotentialDiff);
            inverseProblem.SolveInverseProblemWithLog(1e-10, 1000);
        }

        public static void CheckFirstTestFromTrainingManualPlus10Percent()
        {
            (Forward problemFromTrainingManual, dVector ReceiversPotentialDiff) =
                                                                Da-
ta.FirstTestFromTrainingManual();

            double new_sigma = 0.01;
```

```
            for (int i = 0; i < ReceiversPotentialDiff.Count; i++)
            {
                ReceiversPotentialDiff[i] *= 1.1;
            }

            Inverse inverseProblem = new(problemFromTrainingManual, new_sigma, Re-
ceiversPotentialDiff);
            inverseProblem.SolveInverseProblemWithLog(1e-10, 1000);
        }

        public static void CheckFirstTestFromTrainingManualMinus10Percent()
        {
            (Forward problemFromTrainingManual, dVector ReceiversPotentialDiff) =
                                                                Da-
ta.FirstTestFromTrainingManual();

            double new_sigma = 0.01;

            for (int i = 0; i < ReceiversPotentialDiff.Count; i++)
            {
                ReceiversPotentialDiff[i] *= 0.9;
            }

            Inverse inverseProblem = new(problemFromTrainingManual, new_sigma, Re-
ceiversPotentialDiff);
            inverseProblem.SolveInverseProblemWithLog(1e-10, 1000);
        }

        public static void CheckFirstTestFromTrainingManualBreaker()
        {
            (Forward problemFromTrainingManual, dVector ReceiversPotentialDiff) =
                                                                Da-
ta.FirstTestFromTrainingManual();

            double new_sigma = 0.01;

            ReceiversPotentialDiff[0] *= 1.01;
            ReceiversPotentialDiff[1] *= 1;
            ReceiversPotentialDiff[2] *= 1;

            Inverse inverseProblem = new(problemFromTrainingManual, new_sigma, Re-
ceiversPotentialDiff);
            inverseProblem.SolveInverseProblemWithLog(1e-10, 1000);
        }

    }
}
```