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

Лабораторная работа №1
по дисциплине «Уравнения математической физики»

Решение эллиптических краевых задач методом конечных разностей. Дополнительное задание

Студенты	БЕГИЧЕВ АЛЕКСАНДР
	ШИШКИН НИКИТА
Группа	ПМ-92
Преподаватели	ЗАДОРОЖНЫЙ А. Г.
	ПАТРУШЕВ И. И.
	ПЕРСОВА М. Г.

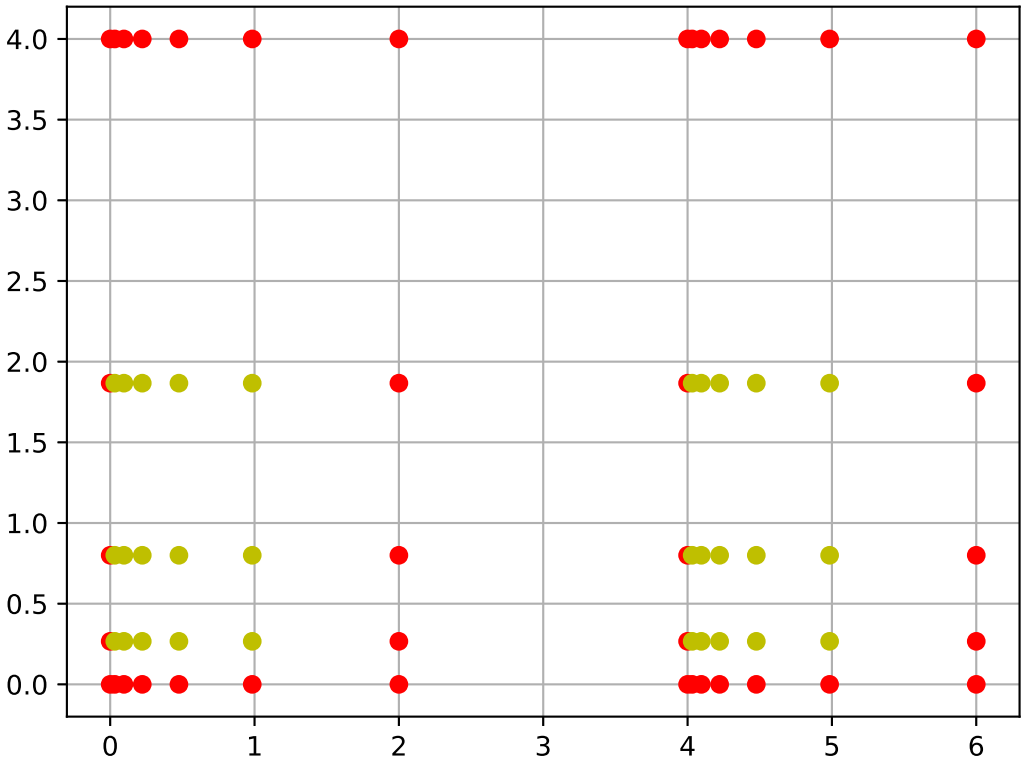
Новосибирск, 2022

Цель работы

Разработать программу для решения эллиптической краевой задачи методом конечных разностей, и при этом сетка может иметь любую прямоугольную форму.

Входные данные для сетки

Рассмотрим пример входных файлов для данной сетки:




```
You, now | 1 author (You)
1  0 2 4 6
2  0 4
3  6 6 6
4  4
5  2 2 2
6  2
7  0 1 0 0 1 0 1
8  1 1 0 2 3 0 1
```

Рис. 1: Входной файл для сетки

```
You, 1 second ago | 1 author (You)
1  20
2  1 0 1 0 0
3  1 0 1 1 1
4  1 2 3 0 0
5  1 2 3 1 1
6  1 0 0 0 1
7  1 1 1 0 1
8  1 2 2 0 1
9  1 3 3 0 1
```

Рис. 2: Входной файл для краевых

Входной файл для сетки

* x-линии
* y-линии
* количество разбиений каждого интервала по x
* количество разбиений каждого интервала по y
* коэффициент разрядки для каждого интервала по x
* коэффициент разрядки для каждого интервала по y

Далее задаются подобласти в виде:
* номер подобласти
* значение λ
* значение γ
* индекс в массиве x-линий (начало по x)
* индекс в массиве x-линий (конец по x)
* индекс в массиве y-линий (начало по y)
* индекс в массиве y-линий (конец по y)

В программе есть возможность задать вложенную сетку, для этого во входной файл добавляется параметр, отвечающий за количество вложенности сетки. Также, если в программе выбрана равномерная сетка, тогда коэффициент разрядки во входном файле не задается.

Входной файл для краевых условий

* значение β

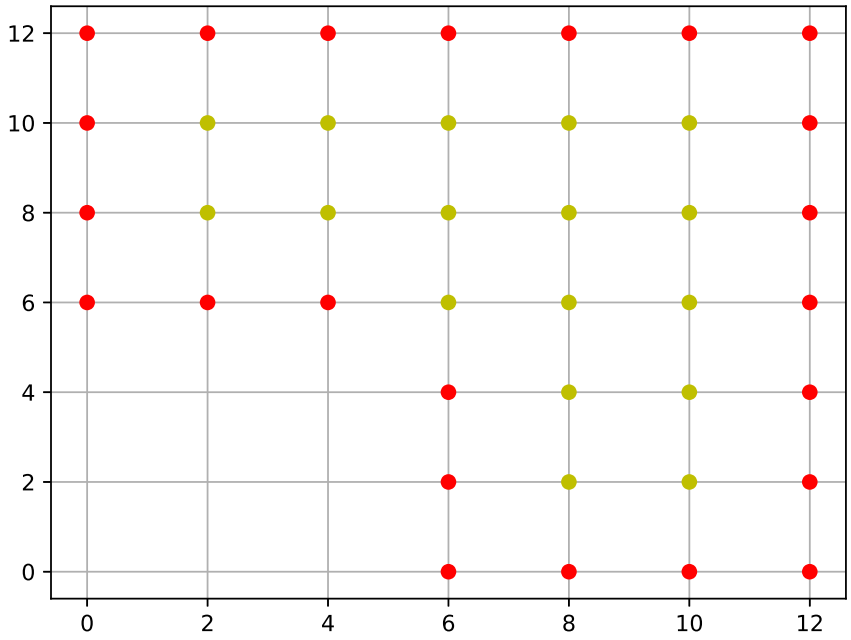
Далее задаются краевые условия в виде:
* тип краевого условия
* индекс в массиве x-линий (начало по x)
* индекс в массиве x-линий (конец по x)
* индекс в массиве y-линий (начало по y)
* индекс в массиве y-линий (конец по y)

Описание программы находится в основном отчете и не требует приведения изменений, так как архитектура программы осталась прежней.

Тестирование

Первый тест

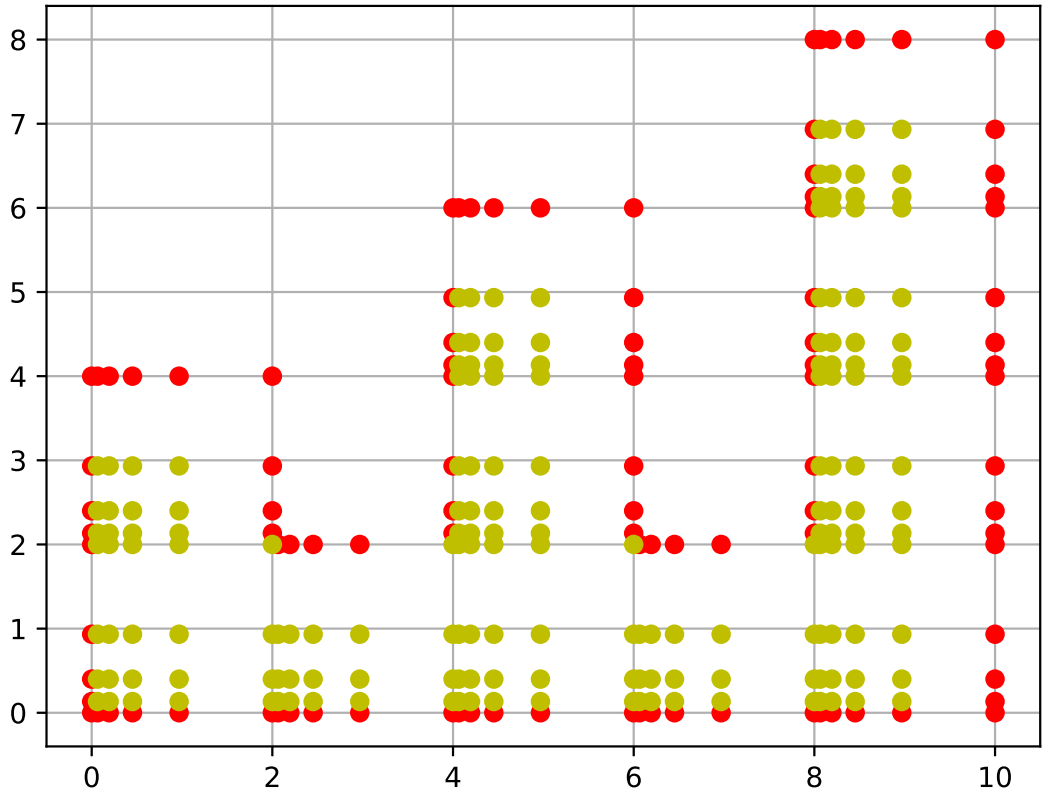
Функция: $x^3 + y$
Правая часть:$-6x$
Коэффициенты $\lambda = 1, \gamma = 0$
Сетка: равномерная
Заданы краевые условия 1-го рода и на левой верхней и правой границах краевое условие 2-го рода.



x_i	y_i	Точное	Численное	Вектор погрешности	Погрешность
2	8	16	15.99999999999405	$5.95 \cdot 10^{-12}$	$1.13 \cdot 10^{-14}$
2	10	18	17.99999999999534	$4.66 \cdot 10^{-12}$	–
4	8	72	71.99999999999245	$7.55 \cdot 10^{-12}$	–
4	10	74	73.99999999999461	$5.39 \cdot 10^{-12}$	–
6	6	222	221.99999999998843	$1.16 \cdot 10^{-11}$	–
6	8	224	223.999999999987	$1.3 \cdot 10^{-11}$	–
6	10	226	225.99999999999255	$7.45 \cdot 10^{-12}$	–
8	2	514	513.9999999999887	$1.13 \cdot 10^{-11}$	–
8	4	516	515.999999999982	$1.8 \cdot 10^{-11}$	–
8	6	518	517.9999999999784	$2.16 \cdot 10^{-11}$	–
8	8	520	519.999999999982	$1.8 \cdot 10^{-11}$	–
8	10	522	521.9999999999907	$9.32 \cdot 10^{-12}$	–
10	2	1,002	1,001.9999999999842	$1.58 \cdot 10^{-11}$	–
10	4	1,004	1,003.9999999999758	$2.42 \cdot 10^{-11}$	–
10	6	1,006	1,005.999999999974	$2.6 \cdot 10^{-11}$	–
10	8	1,008	1,007.9999999999799	$2.01 \cdot 10^{-11}$	–
10	10	1,010	1,009.9999999999899	$1.01 \cdot 10^{-11}$	–

Второй тест

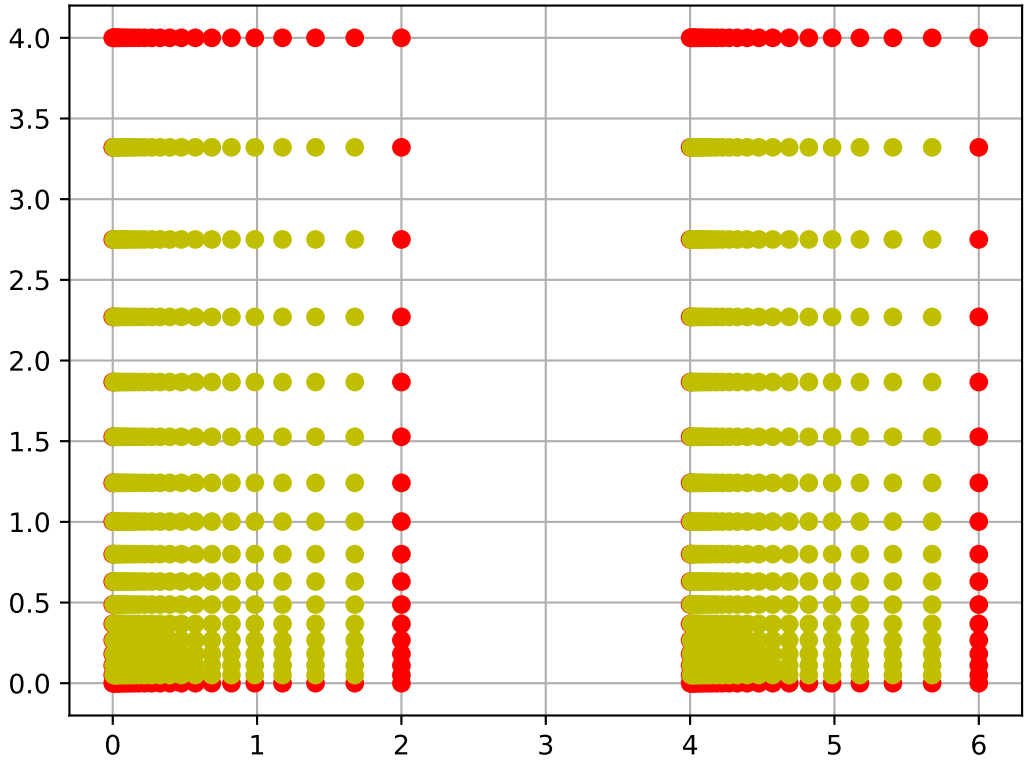
Функция: $4x^2 + y$
Правая часть:$-4 + 0.5 \cdot (4x^2 + y)$
Коэффициенты $\lambda = 0.5, \gamma = 0.5$
Сетка: неравномерная
Заданы краевые условия 1-го рода и на самых левой и правой границах краевые условия 2-го и 3-го рода.



Относительная погрешность	Относительная невязка
$1.52 \cdot 10^{-14}$	$5.81 \cdot 10^{-16}$

Третий тест

Функция: $x^2 - y$
Правая часть:$-1 + x^2 - y$
Коэффициенты $\lambda = 0.5, \gamma = 1$
Сетка: неравномерная
Заданы краевые условия 1-го рода на всех границах.



Относительная погрешность	Относительная невязка
$4.3 \cdot 10^{-16}$	$6.35 \cdot 10^{-13}$

Тексты основных модулей

Program.cs

```
using eMP_1;

GridFactory gridFactory = new();
MFD mfd = new(gridFactory.CreateGrid(GridType.Irregular,
↪  "InputParameters/grid(irregular).txt"), "InputParameters/boundaries.txt");
// MFD mfd = new(gridFactory.CreateGrid(GridType.Nested,
↪  "InputParameters/grid(nested).txt"), "InputParameters/boundaries.txt");
// MFD mfd = new(gridFactory.CreateGrid(GridType.Regular,
↪  "InputParameters/grid(regular).txt"), "InputParameters/boundaries.txt");
// mfd.SetTest(new FirstTest());  // x, λ = 1, γ = 0
// mfd.SetTest(new SecondTest()); // x^2 − y, λ = 0.5, γ = 1
// mfd.SetTest(new ThirdTest());  // 3x^3 + 2y^3, 1 area: λ = γ = 0.5, 2 area : λ = γ = 2
// mfd.SetTest(new FourthTest());    // ln(x + y), λ = 1, γ = 1
// mfd.SetTest(new FifthTest());  // 4x^4, λ = 1, γ = 0
// mfd.SetTest(new SixthTest());  // 4x^4 + 2y^4, λ = 1, γ = 0
// mfd.SetTest(new SeventhTest());// e^x + y, λ = 1, γ = 1
// mfd.SetTest(new EighthTest()); // x^3 + y, λ = 1, γ = 0
mfd.SetMethodSolvingSLAE(new GaussSeidel(1000, 1E-14, 1.17));
mfd.Compute();
```

MFD.cs

```
namespace eMP_1;

public enum PointType {
    None,
    Boundary,
    Internal,
    Dummy
}

public enum BoundaryType {
    None,
    Dirichlet,
    Neumann,
    Mixed
}

public enum GridType {
    Regular,
    Irregular,
    Nested
}

public enum NormalType {
    None,
    LeftX,
    RightX,
    UpperY,
    BottomY
}

public class MFD {
    private readonly Grid _grid;
    private DiagMatrix _matrix;
    private ITest _test;
```

```
    private ISolver _solver;
    private readonly Boundary[] _boundaries;
    private double[] _q;
    private double[] _pr;
    private readonly double _beta;
    public double[] Weights
        => _q;

    public MFD(Grid grid, string boundaryPath) {
        try {
            using (var sr = new StreamReader(boundaryPath)) {
                _beta = double.Parse(sr.ReadLine());
                _boundaries = sr.ReadToEnd().Split("\n")
                .Select(str => Boundary.BoundaryParse(str)).ToArray();
            }

            _grid = grid;

        } catch (Exception ex) {
            Console.WriteLine(ex.Message);
        }
    }

    public void SetTest(ITest test)
        => _test = test;

    public void SetMethodSolvingSLAE(ISolver solver)
        => _solver = solver;

    public void Compute() {
        try {
            if (_test is null)
                throw new Exception("Set the test!");

            if (_solver is null)
                throw new Exception("Set the method solving SLAE!");

            _grid.Build();
            _grid.AssignBoundaryConditions(_boundaries);
            Init();
            BuildMatrix();
            _q = _solver.Compute(_matrix, _pr);

        } catch (Exception ex) {
            Console.WriteLine(ex.Message);
        }
    }

    private void Init() {
        _matrix = new(_grid.Points.Count, (_grid.AllLinesX.Count >
        ↪  _grid.AllLinesX.Count) ?
        _grid.AllLinesX.Count - 2 : _grid.AllLinesY.Count - 2);
        _pr = new double[_matrix.Size];
        _q = new double[_matrix.Size];
    }

    private void BuildMatrix() {
        double hx, hy, hix, hiy, hi;
        double lambda, gamma;
        double us, ubeta;
```

```
        double leftDerivative, rightDerivative;
        NormalType normalType;

        for (int i = 0; i < _grid.Points.Count; i++) {
            switch (_grid.Points[i].PointType) {
                case PointType.Boundary:

                    switch (_grid.Points[i].BoundaryType) {
                        case BoundaryType.Dirichlet:

                            _matrix.Diags[0][i] = 1;
                            _pr[i] = _test.U(_grid.Points[i]);

                            break;

                        case BoundaryType.Neumann:

                            lambda = _grid.Areas[_grid.Points[i].AreaNumber].Item2;

                            normalType = _grid.Normal(_grid.Points[i]);

                            switch (normalType) {
                                case NormalType.LeftX:

                                    hi = _grid.AllLinesX[_grid.Points[i].I + 1] -
                                    ↪  _grid.AllLinesX[_grid.Points[i].I];
                                    _matrix.Diags[0][i] = lambda / hi;
                                    _matrix.Diags[4][i] = -lambda / hi;
                                    _pr[i] = -lambda *
                                    ↪  RightDerivativeX(_grid.Points[i], hi);

                                    break;

                                case NormalType.BottomY:

                                    hi = _grid.AllLinesY[_grid.Points[i].J + 1] -
                                    ↪  _grid.AllLinesY[_grid.Points[i].J];
                                    _matrix.Diags[0][i] = lambda / hi;
                                    _matrix.Diags[3][i] = -lambda / hi;
                                    _pr[i] = -lambda *
                                    ↪  RightDerivativeY(_grid.Points[i], hi);

                                    break;

                                case NormalType.RightX:

                                    hi = _grid.AllLinesX[_grid.Points[i].I] -
                                    ↪  _grid.AllLinesX[_grid.Points[i].I - 1];
                                    _matrix.Diags[0][i] = lambda / hi;
                                    _matrix.Diags[2][i + _matrix.Indexes[2]] =
                                    ↪  -lambda / hi;
                                    _pr[i] = lambda *
                                    ↪  LeftDerivativeX(_grid.Points[i], hi);

                                    break;

                                case NormalType.UpperY:

                                    hi = _grid.AllLinesY[_grid.Points[i].J] -
                                    ↪  _grid.AllLinesY[_grid.Points[i].J - 1];
```

```
                                    _matrix.Diags[0][i] = lambda / hi;
                                    _matrix.Diags[1][i + _matrix.Indexes[1]] =
                                    ↪  -lambda / hi;
                                    _pr[i] = lambda *
                                    ↪  LeftDerivativeY(_grid.Points[i], hi);

                                    break;

                                default:
                                    throw new ArgumentOutOfRangeException(nameof(norm
                                    ↪  alType),
                                    $"This type of normal does not exist:
                                    ↪  {normalType}");
                            }

                            break;

                        case BoundaryType.Mixed:

                            lambda = _grid.Areas[_grid.Points[i].AreaNumber].Item2;

                            normalType = _grid.Normal(_grid.Points[i]);
                            us = _test.U(_grid.Points[i]);

                            switch (normalType) {
                                case NormalType.LeftX:

                                    hi = _grid.AllLinesX[_grid.Points[i].I + 1] -
                                    ↪  _grid.AllLinesX[_grid.Points[i].I];
                                    rightDerivative =
                                    ↪  RightDerivativeX(_grid.Points[i], hi);
                                    ubeta = -lambda * rightDerivative / _beta + us;
                                    _matrix.Diags[0][i] = lambda / hi + _beta;
                                    _matrix.Diags[4][i] = -lambda / hi;
                                    _pr[i] = -lambda * rightDerivative + _beta * (us
                                    ↪  - ubeta) + _beta * ubeta;

                                    break;

                                case NormalType.BottomY:

                                    hi = _grid.AllLinesY[_grid.Points[i].J + 1] -
                                    ↪  _grid.AllLinesY[_grid.Points[i].J];
                                    rightDerivative =
                                    ↪  RightDerivativeY(_grid.Points[i], hi);
                                    ubeta = -lambda * rightDerivative / _beta + us;
                                    _matrix.Diags[0][i] = lambda / hi + _beta;
                                    _matrix.Diags[3][i] = -lambda / hi;
                                    _pr[i] = -lambda * rightDerivative + _beta * (us
                                    ↪  - ubeta) + _beta * ubeta;

                                    break;

                                case NormalType.RightX:

                                    hi = _grid.AllLinesX[_grid.Points[i].I] -
                                    ↪  _grid.AllLinesX[_grid.Points[i].I - 1];
                                    leftDerivative = LeftDerivativeX(_grid.Points[i],
                                    ↪  hi);
                                    ubeta = lambda * leftDerivative / _beta + us;
```

```
                                    _matrix.Diags[0][i] = lambda / hi + _beta;
                                    _matrix.Diags[2][i + _matrix.Indexes[2]] =
                                    ↪  -lambda / hi;
                                    _pr[i] = lambda * leftDerivative + _beta * (us -
                                    ↪  ubeta) + _beta * ubeta;

                                    break;

                                case NormalType.UpperY:

                                    hi = _grid.AllLinesY[_grid.Points[i].J] -
                                    ↪  _grid.AllLinesY[_grid.Points[i].J - 1];
                                    leftDerivative = LeftDerivativeY(_grid.Points[i],
                                    ↪  hi);
                                    ubeta = lambda * leftDerivative / _beta + us;
                                    _matrix.Diags[0][i] = lambda / hi + _beta;
                                    _matrix.Diags[1][i + _matrix.Indexes[1]] =
                                    ↪  -lambda / hi;
                                    _pr[i] = lambda * leftDerivative + _beta * (us -
                                    ↪  ubeta) + _beta * ubeta;

                                    break;

                                default:
                                    throw new ArgumentOutOfRangeException(nameof(norm
                                    ↪  alType),
                                    $"This type of normal does not exist:
                                    ↪  {normalType}");
                            }

                            break;


                        default:
                            throw new
                            ↪  ArgumentOutOfRangeException(nameof(BoundaryType),
                            $"This type of boundary does not exist:
                            ↪  {_grid.Points[i].BoundaryType}");
                    }

                    break;

                case PointType.Internal:

                    hx = _grid.AllLinesX[_grid.Points[i].I + 1] -
                    ↪  _grid.AllLinesX[_grid.Points[i].I];
                    hy = _grid.AllLinesY[_grid.Points[i].J + 1] -
                    ↪  _grid.AllLinesY[_grid.Points[i].J];

                    (lambda, gamma) =
                    (_grid.Areas[_grid.Points[i].AreaNumber].Item2,
                    _grid.Areas[_grid.Points[i].AreaNumber].Item3);

                    _pr[i] = _test.F(_grid.Points[i]);

                    if (_grid is RegularGrid) {
                        _matrix.Diags[0][i] = lambda * (2.0 / (hx * hx) + 2.0 / (hy *
                        ↪  hy)) + gamma;
                        _matrix.Diags[3][i] = -lambda / (hy * hy);
                        _matrix.Diags[4][i] = -lambda / (hx * hx);
```

```
                        _matrix.Diags[1][i + _matrix.Indexes[1]] = -lambda / (hy *
                        ↪ hy);
                        _matrix.Diags[2][i + _matrix.Indexes[2]] = -lambda / (hx *
                        ↪ hx);
                    } else {
                        hix = _grid.AllLinesX[_grid.Points[i].I] -
                        ↪ _grid.AllLinesX[_grid.Points[i].I - 1];
                        hiy = _grid.AllLinesY[_grid.Points[i].J] -
                        ↪ _grid.AllLinesY[_grid.Points[i].J - 1];

                        _matrix.Diags[0][i] = lambda * (2.0 / (hix * hx) + 2.0 / (hiy
                        ↪ * hy)) + gamma;
                        _matrix.Diags[2][i + _matrix.Indexes[2]] = -lambda * 2.0 /
                        ↪ (hix * (hx + hix));
                        _matrix.Diags[1][i + _matrix.Indexes[1]] = -lambda * 2.0 /
                        ↪ (hiy * (hy + hiy));
                        _matrix.Diags[4][i] = -lambda * 2.0 / (hx * (hx + hix));
                        _matrix.Diags[3][i] = -lambda * 2.0 / (hy * (hy + hiy));
                    }

                    break;

                case PointType.Dummy:

                    _matrix.Diags[0][i] = 1;
                    _pr[i] = 0;

                    break;

                default:
                    throw new ArgumentOutOfRangeException(nameof(PointType),
                    $"This type of point does not exist:
                    ↪ {_grid.Points[i].PointType}");
            }
        }
    }

    private double LeftDerivativeX(Point2D point, double h)
        => (_test.U(point) - _test.U(point - (h, 0))) / h;

    private double LeftDerivativeY(Point2D point, double h)
        => (_test.U(point) - _test.U(point - (0, h))) / h;

    private double RightDerivativeX(Point2D point, double h)
        => (_test.U(point + (h, 0)) - _test.U(point)) / h;

    private double RightDerivativeY(Point2D point, double h)
        => (_test.U(point + (0, h)) - _test.U(point)) / h;
}
```

GridFactory.cs

```
namespace eMP_1;

public interface IFactory {
    public Grid CreateGrid(GridType gridType, string path);
}

public class GridFactory : IFactory {
    public Grid CreateGrid(GridType gridType, string path) {
```

```
        return gridType switch {
            GridType.Regular => new RegularGrid(path),

            GridType.Irregular => new IrregularGrid(path),

            GridType.Nested => new NestedGrid(path),

            _ => throw new ArgumentOutOfRangeException(nameof(gridType),
            $"This type of grid does not exist: {gridType}")
        };
    }
}
```

Grid.cs

```
namespace eMP_1;

public abstract class Grid {
    public abstract ImmutableArray<double> LinesX { get; init; }
    public abstract ImmutableArray<double> LinesY { get; init; }
    public abstract ImmutableList<double> AllLinesX { get; }
    public abstract ImmutableList<double> AllLinesY { get; }
    public abstract ImmutableList<Point2D> Points { get; }
    public abstract ImmutableArray<(int, double, double, int, int, int, int)> Areas {
    ↪  get; }

    public abstract void Build();

    public NormalType Normal(Point2D point) {
        if (point.PointType != PointType.Boundary) {
            throw new Exception("To determine the normal to the boundary, the point
            ↪  needs the boundary!");
        }

        var normalType = NormalType.None;

        foreach (var area in Areas) {
            normalType = point switch {
                _ when point.X == LinesX[area.Item4] => NormalType.LeftX,
                _ when point.Y == LinesY[area.Item6] => NormalType.BottomY,
                _ when point.X == LinesX[area.Item5] => NormalType.RightX,
                _ when point.Y == LinesY[area.Item7] => NormalType.UpperY,

                _ => NormalType.None,
            };

            if (normalType != NormalType.None) {
                break;
            }
        }

        return normalType;
    }

    protected PointType PointTypesWithoutInternalCheck(double x, double y) {
        foreach (var area in Areas) {
            if (x >= LinesX[area.Item4] && x <= LinesX[area.Item5] && y >=
            ↪  LinesY[area.Item6] && y <= LinesY[area.Item7]) {
                return PointType.Boundary;
            }
```

```
        }

        return PointType.Dummy;
    }

    protected void InternalCheck() {
        foreach (var point in Points.Where(point => point.PointType !=
        ↪  PointType.Dummy)) {
            if (Points.Any(lstPoint => lstPoint.I == point.I - 1 && lstPoint.J ==
            ↪  point.J && lstPoint.PointType != PointType.Dummy) &&
                Points.Any(lstPoint => lstPoint.I == point.I + 1 && lstPoint.J ==
                ↪  point.J && lstPoint.PointType != PointType.Dummy) &&
                Points.Any(lstPoint => lstPoint.J == point.J - 1 && lstPoint.I ==
                ↪  point.I && lstPoint.PointType != PointType.Dummy) &&
                Points.Any(lstPoint => lstPoint.J == point.J + 1 && lstPoint.I ==
                ↪  point.I && lstPoint.PointType != PointType.Dummy)) {
                point.PointType = PointType.Internal;
            }
        }
    }

    protected void SetAreaNumber() {
        for (int i = 0; i < Points.Count; i++) {
            for (int iArea = 0; iArea < Areas.Length; iArea++) {
                if (Points[i].X >= Areas[iArea].Item4 && Points[i].X <=
                ↪  Areas[iArea].Item5 &&
                    Points[i].Y >= Areas[iArea].Item6 && Points[i].Y <=
                    ↪  Areas[iArea].Item7) {
                    Points[i].AreaNumber = Areas[iArea].Item1;
                }
            }
        }
    }

    public void AssignBoundaryConditions(Boundary[] boundaries) {
        foreach (var point in Points.Where(point => point.PointType ==
        ↪  PointType.Boundary)) {
            for (int k = 0; k < boundaries.Length; k++) {
                if (point.X >= LinesX[boundaries[k].X1] && point.X <=
                ↪  LinesX[boundaries[k].X2]
                && point.Y >= LinesY[boundaries[k].Y1] && point.Y <=
                ↪  LinesY[boundaries[k].Y2]) {
                    point.BoundaryType = boundaries[k].BoundaryType;
                    break;
                }
            }
        }
    }

    protected void WriteToFilePoints() {
        using (var sw = new StreamWriter("points/boundaryPoints.txt")) {
            Points.ForEach(x => { if (x.PointType == PointType.Boundary)
            ↪  sw.WriteLine(x); });
        }

        using (var sw = new StreamWriter("points/internalPoints.txt")) {
            Points.ForEach(x => { if (x.PointType == PointType.Internal)
            ↪  sw.WriteLine(x); });
        }

```

```
        using (var sw = new StreamWriter("points/dummyPoints.txt")) {
            Points.ForEach(x => { if (x.PointType == PointType.Dummy)
            ↪  sw.WriteLine(x); });
        }
    }
}
```

RegularGrid.cs

```
namespace eMP_1;

public class RegularGrid : Grid {
    private List<double> _allLinesX;
    private List<double> _allLinesY;
    private readonly List<Point2D> _points;
    private readonly (int, double, double, int, int, int, int)[] _areas;
    public override ImmutableArray<double> LinesX { get; init; }
    public override ImmutableArray<double> LinesY { get; init; }
    public override ImmutableList<double> AllLinesX
        => _allLinesX.ToImmutableList();
    public override ImmutableList<double> AllLinesY
        => _allLinesY.ToImmutableList();
    public override ImmutableList<Point2D> Points
        => _points.ToImmutableList();
    public override ImmutableArray<(int, double, double, int, int, int, int)> Areas
        => _areas.ToImmutableArray();
    public int SplitsX { get; init; }
    public int SplitsY { get; init; }

    public RegularGrid(string path) {
        try {
            using (var sr = new StreamReader(path)) {
                LinesX = sr.ReadLine().Split().Select(value =>
                ↪  double.Parse(value)).ToImmutableArray();
                LinesY = sr.ReadLine().Split().Select(value =>
                ↪  double.Parse(value)).ToImmutableArray();
                SplitsX = int.Parse(sr.ReadLine());
                SplitsY = int.Parse(sr.ReadLine());
                _areas = sr.ReadToEnd().Split("\n").Select(row => row.Split())
                .Select(value => (int.Parse(value[0]), double.Parse(value[1]),
                ↪  double.Parse(value[2]),
                int.Parse(value[3]), int.Parse(value[4]), int.Parse(value[5]),
                ↪  int.Parse(value[6]))).ToArray();
            }

            _allLinesX = new();
            _allLinesY = new();
            _points = new();
        } catch (Exception ex) {
            Console.WriteLine(ex.Message);
        }
    }

    public override void Build() {
        double h;
        double lenght = LinesX.Last() - LinesX.First();

        h = lenght / SplitsX;

        _allLinesX.Add(LinesX.First());
```

```
48
49          while (Math.Round(_allLinesX.Last() + h, 1) < LinesX.Last())
50              _allLinesX.Add(_allLinesX.Last() + h);
51
52          _allLinesX = _allLinesX.Union(LinesX).OrderBy(value => value).ToList();
53
54          lenght = LinesY.Last() - LinesY.First();
55
56          h = lenght / SplitsY;
57
58          _allLinesY.Add(LinesY.First());
59
60          while (Math.Round(_allLinesY.Last() + h, 1) < LinesY.Last())
61              _allLinesY.Add(_allLinesY.Last() + h);
62
63          _allLinesY = _allLinesY.Union(LinesY).OrderBy(value => value).ToList();
64
65          for (int i = 0; i < _allLinesX.Count; i++)
66              for (int j = 0; j < _allLinesY.Count; j++)
67                  _points.Add(new(_allLinesX[i], _allLinesY[j], i, j,
68                  PointTypesWithoutInternalCheck(_allLinesX[i], _allLinesY[j])));
69
70          InternalCheck();
71          SetAreaNumber();
72          WriteToFilePoints();
73      }
74  }
```

IrregularGrid.cs

```
1   namespace eMP_1;
2
3   public class IrregularGrid : Grid {
4       private readonly List<double> _allLinesX;
5       private readonly List<double> _allLinesY;
6       private readonly List<Point2D> _points;
7       private readonly int[] _splitsX;
8       private readonly int[] _splitsY;
9       private readonly double[] _kX;
10      private readonly double[] _kY;
11      private readonly (int, double, double, int, int, int, int)[] _areas;
12      public override ImmutableArray<double> LinesX { get; init; }
13      public override ImmutableArray<double> LinesY { get; init; }
14      public override ImmutableList<Point2D> Points
15       => _points.ToImmutableList();
16      public override ImmutableList<double> AllLinesX
17          => _allLinesX.ToImmutableList();
18      public override ImmutableList<double> AllLinesY
19          => _allLinesY.ToImmutableList();
20      public override ImmutableArray<(int, double, double, int, int, int, int)> Areas
21          => _areas.ToImmutableArray();
22      public ImmutableArray<int> SplitsX
23          => _splitsX.ToImmutableArray();
24      public ImmutableArray<int> SplitsY
25          => _splitsY.ToImmutableArray();
26      public ImmutableArray<double> KX
27          => _kX.ToImmutableArray();
28      public ImmutableArray<double> KY
29          => _kY.ToImmutableArray();
30
```

```
    public IrregularGrid(string path) {
        try {
            using (var sr = new StreamReader(path)) {
                LinesX = sr.ReadLine().Split().Select(value =>
                ↪  double.Parse(value)).ToImmutableArray();
                LinesY = sr.ReadLine().Split().Select(value =>
                ↪  double.Parse(value)).ToImmutableArray();
                _splitsX = sr.ReadLine().Split().Select(value =>
                ↪  int.Parse(value)).ToArray();
                _splitsY = sr.ReadLine().Split().Select(value =>
                ↪  int.Parse(value)).ToArray();
                _kX = sr.ReadLine().Split().Select(value =>
                ↪  double.Parse(value)).ToArray();
                _kY = sr.ReadLine().Split().Select(value =>
                ↪  double.Parse(value)).ToArray();
                _areas = sr.ReadToEnd().Split("\n").Select(row => row.Split())
                .Select(value => (int.Parse(value[0]), double.Parse(value[1]),
                ↪  double.Parse(value[2]),
                int.Parse(value[3]), int.Parse(value[4]), int.Parse(value[5]),
                ↪  int.Parse(value[6]))).ToArray();

            }

            _allLinesX = new();
            _allLinesY = new();
            _points = new();

        } catch (Exception ex) {
            Console.WriteLine(ex.Message);
        }
    }

    public override void Build() {
        for (int i = 0; i < LinesX.Length - 1; i++) {
            double h;
            double sum = 0;
            double lenght = LinesX[i + 1] - LinesX[i];

            for (int k = 0; k < _splitsX[i]; k++)
                sum += Math.Pow(_kX[i], k);

            h = lenght / sum;

            _allLinesX.Add(LinesX[i]);

            while (Math.Round(_allLinesX.Last() + h, 1) < LinesX[i + 1]) {
                _allLinesX.Add(_allLinesX.Last() + h);

                h *= _kX[i];
            }
        }

        _allLinesX.Add(LinesX.Last());

        for (int i = 0; i < LinesY.Length - 1; i++) {
            double h;
            double sum = 0;
            double lenght = LinesY[i + 1] - LinesY[i];

            for (int k = 0; k < _splitsY[i]; k++)
```

```
                sum += Math.Pow(_kY[i], k);

            h = lenght / sum;

            _allLinesY.Add(LinesY[i]);

            while (Math.Round(_allLinesY.Last() + h, 1) < LinesY[i + 1]) {
                _allLinesY.Add(_allLinesY.Last() + h);

                h *= _kY[i];
            }
        }

        _allLinesY.Add(LinesY.Last());

        for (int i = 0; i < _allLinesX.Count; i++)
            for (int j = 0; j < _allLinesY.Count; j++)
                _points.Add(new(_allLinesX[i], _allLinesY[j], i, j,
                PointTypesWithoutInternalCheck(_allLinesX[i], _allLinesY[j])));

        InternalCheck();
        SetAreaNumber();
        WriteToFilePoints();
    }
}
```

NestedGrid.cs

```
namespace eMP_1;

public class NestedGrid : Grid {
    private readonly List<double> _allLinesX;
    private readonly List<double> _allLinesY;
    private readonly List<Point2D> _points;
    private readonly int[] _splitsX;
    private readonly int[] _splitsY;
    private readonly double[] _kX;
    private readonly double[] _kY;
    private readonly (int, double, double, int, int, int, int)[] _areas;
    public override ImmutableArray<double> LinesX { get; init; }
    public override ImmutableArray<double> LinesY { get; init; }
    public override ImmutableList<Point2D> Points
     => _points.ToImmutableList();
    public override ImmutableList<double> AllLinesX
        => _allLinesX.ToImmutableList();
    public override ImmutableList<double> AllLinesY
        => _allLinesY.ToImmutableList();
    public override ImmutableArray<(int, double, double, int, int, int, int)> Areas
     => _areas.ToImmutableArray();
    public ImmutableArray<int> SplitsX => _splitsX.ToImmutableArray();
    public ImmutableArray<int> SplitsY => _splitsY.ToImmutableArray();
    public ImmutableArray<double> KX => _kX.ToImmutableArray();
    public ImmutableArray<double> KY => _kY.ToImmutableArray();
    public int NumberNesting { get; init; }

    public NestedGrid(string path) {
        try {
            using (var sr = new StreamReader(path)) {
                LinesX = sr.ReadLine().Split().Select(value =>
                 double.Parse(value)).ToImmutableArray();
```

```
                LinesY = sr.ReadLine().Split().Select(value =>
                ↪  double.Parse(value)).ToImmutableArray();
                _splitsX = sr.ReadLine().Split().Select(value =>
                ↪  int.Parse(value)).ToArray();
                _splitsY = sr.ReadLine().Split().Select(value =>
                ↪  int.Parse(value)).ToArray();
                _kX = sr.ReadLine().Split().Select(value =>
                ↪  double.Parse(value)).ToArray();
                _kY = sr.ReadLine().Split().Select(value =>
                ↪  double.Parse(value)).ToArray();
                NumberNesting = int.Parse(sr.ReadLine());
                _areas = sr.ReadToEnd().Split("\n").Select(row => row.Split())
                .Select(value => (int.Parse(value[0]), double.Parse(value[1]),
                ↪  double.Parse(value[2]),
                int.Parse(value[3]), int.Parse(value[4]), int.Parse(value[5]),
                ↪  int.Parse(value[6]))).ToArray();

            }

            _allLinesX = new();
            _allLinesY = new();
            _points = new();
        } catch (Exception ex) {
            Console.WriteLine(ex.Message);
        }
    }

    public override void Build() {
        for (int i = 0; i < NumberNesting; i++)
            for (int j = 0; j < _kX.Length; j++)
                _kX[j] = Math.Sqrt(_kX[j]);

        for (int i = 0; i < LinesX.Length - 1; i++) {
            double h;
            double sum = 0;
            double lenght = LinesX[i + 1] - LinesX[i];

            for (int k = 0; k < _splitsX[i]; k++)
                sum += Math.Pow(_kX[i], k);

            h = lenght / sum;

            _allLinesX.Add(LinesX[i]);

            while (Math.Round(_allLinesX.Last() + h, 1) < LinesX[i + 1]) {
                _allLinesX.Add(_allLinesX.Last() + h);

                h *= _kX[i];
            }
        }

        _allLinesX.Add(LinesX.Last());

        for (int i = 0; i < NumberNesting; i++)
            for (int j = 0; j < _kY.Length; j++)
                _kY[j] = Math.Sqrt(_kY[j]);

        for (int i = 0; i < LinesY.Length - 1; i++) {
            double h;
            double sum = 0;
```

```
            double lenght = LinesY[i + 1] - LinesY[i];

            for (int k = 0; k < _splitsY[i]; k++)
                sum += Math.Pow(_kY[i], k);

            h = lenght / sum;

            _allLinesY.Add(LinesY[i]);

            while (Math.Round(_allLinesY.Last() + h, 1) < LinesY[i + 1]) {
                _allLinesY.Add(_allLinesY.Last() + h);

                h *= _kY[i];
            }
        }

        _allLinesY.Add(LinesY.Last());

        for (int i = 0; i < _allLinesX.Count; i++)
            for (int j = 0; j < _allLinesY.Count; j++)
                _points.Add(new(_allLinesX[i], _allLinesY[j], i, j,
                PointTypesWithoutInternalCheck(_allLinesX[i], _allLinesY[j])));

        InternalCheck();
        SetAreaNumber();
        WriteToFilePoints();
    }
}
```

DiagMatrix.cs

```
namespace eMP_1;

public class DiagMatrix {
    public double[][] Diags { get; set; }
    public int[] Indexes { get; init; }
    public int Size { get; init; }
    public int ZeroDiags { get; init; }

    public DiagMatrix(int countPoints, int zeroDiags) {
        Size = countPoints;
        ZeroDiags = zeroDiags;
        Diags = new double[5][];
        Diags[0] = new double[countPoints];
        Diags[1] = new double[countPoints - 1];
        Diags[2] = new double[countPoints - zeroDiags - 2];
        Diags[3] = new double[countPoints - 1];
        Diags[4] = new double[countPoints - zeroDiags - 2];
        Indexes = new int[] { 0, -1, -2 - zeroDiags, 1, 2 + zeroDiags };
    }
}
```

Solver.cs

```
namespace eMP_1;

public interface ISolver {
    public int MaxIters { get; init; }
    public double Eps { get; init; }
```

```
    public double W { get; init; }

    public double[] Compute(DiagMatrix diagMatrix, double[] pr);
}

public record GaussSeidel(int MaxIters, double Eps, double W) : ISolver {
    public double[] Compute(DiagMatrix diagMatrix, double[] pr) {
        double[] qk = new double[diagMatrix.Size];
        double[] qk1 = new double[diagMatrix.Size];
        double[] residual = new double[diagMatrix.Size];
        double prNorm = pr.Norm();

        for (int i = 0; i < MaxIters; i++) {
            for (int k = 0; k < diagMatrix.Size; k++) {
                double fstSum = MultLine(diagMatrix, k, qk1, 1);
                double scdSum = MultLine(diagMatrix, k, qk, 2);

                residual[k] = pr[k] - (fstSum + scdSum);
                qk1[k] = qk[k] + W * residual[k] / diagMatrix.Diags[0][k];
            }

            qk1.Copy(qk);
            qk1.Fill(0);

            if (residual.Norm() / prNorm < Eps)
                break;
        }

        return qk;
    }

    private static double MultLine(DiagMatrix diagMatrix, int i, double[] vector, int
    ↪  method) {
        double sum = 0;

        if (method == 0 || method == 1) {
            if (i > 0) {
                sum += diagMatrix.Diags[1][i - 1] * vector[i - 1];

                if (i > diagMatrix.ZeroDiags + 1)
                    sum += diagMatrix.Diags[2][i - diagMatrix.ZeroDiags - 2] *
                    ↪  vector[i - diagMatrix.ZeroDiags - 2];
            }
        }

        if (method == 0 || method == 2) {
            sum += diagMatrix.Diags[0][i] * vector[i];

            if (i < diagMatrix.Size - 1) {
                sum += diagMatrix.Diags[3][i] * vector[i + 1];

                if (i < diagMatrix.Size - diagMatrix.ZeroDiags - 2)
                    sum += diagMatrix.Diags[4][i] * vector[i + diagMatrix.ZeroDiags +
                    ↪  2];
            }
        }

        return sum;
    }
}
```

Boundary.cs

```cs
namespace eMP_1;

public readonly record struct Boundary(BoundaryType BoundaryType, int X1, int X2, int Y1, int Y2) {
    public static Boundary BoundaryParse(string boundaryStr) {
        var data = boundaryStr.Split();
        Boundary boundary = new((BoundaryType)Enum.Parse(typeof(BoundaryType), data[0]),
        int.Parse(data[1]), int.Parse(data[2]), int.Parse(data[3]), int.Parse(data[4]));

        return boundary;
    }
}
```

Point2D.cs

```cs
namespace eMP_1;

public class Point2D {
    public double X { get; init; }
    public double Y { get; init; }
    public int I { get; init; }
    public int J { get; init; }
    public PointType PointType { get; set; }
    public BoundaryType BoundaryType { get; set; } = BoundaryType.None;
    public int AreaNumber { get; set; }

    public Point2D(double x, double y, int i, int j, PointType pointType) {
        X = x;
        Y = y;
        I = i;
        J = j;
        PointType = pointType;
    }

    public static Point2D Parse(string pointStr) {
        var data = pointStr.Split();
        Point2D point = new(double.Parse(data[0]), double.Parse(data[1]),
        int.Parse(data[2]), int.Parse(data[3]), (PointType)Enum.Parse(typeof(PointType), data[4]));

        return point;
    }

    public static Point2D operator +(Point2D point, (double, double) value)
        => new(point.X + value.Item1, point.Y + value.Item2, point.I, point.J, point.PointType);

    public static Point2D operator -(Point2D point, (double, double) value)
        => new(point.X - value.Item1, point.Y - value.Item2, point.I, point.J, point.PointType);

    public override string ToString()
        => $"{X} {Y}";
}
```

Tests.cs

```
namespace eMP_1;

public interface ITest {
    public double U(Point2D point);
    public double F(Point2D point);
}

public class FirstTest : ITest {
    public double U(Point2D point)
        => point.X;

    public double F(Point2D point)
        => 0;
}

public class SecondTest : ITest {
    public double U(Point2D point)
        => point.X * point.X - point.Y;

    public double F(Point2D point)
        => -1 + point.X * point.X - point.Y;
}

public class ThirdTest : ITest {
    public double U(Point2D point)
        => 3 * point.X * point.X * point.X + 2 * point.Y * point.Y * point.Y;

    public double F(Point2D point)
        => (point.AreaNumber == 0) ? -9 * point.X - 6 * point.Y + 0.5 *
        (3 * point.X * point.X * point.X + 2 * point.Y * point.Y * point.Y) :
        -36 * point.X - 24 * point.Y + 2 * (3 * point.X * point.X * point.X + 2 *
        ↪  point.Y * point.Y * point.Y);
}

public class FourthTest : ITest {
    public double U(Point2D point)
        => Math.Log(point.X + point.Y);

    public double F(Point2D point)
        => 2 / ((point.X + point.Y) * (point.X + point.Y));
}

public class FifthTest : ITest {
    public double U(Point2D point)
        => 4 * point.X * point.X * point.X * point.X;
    public double F(Point2D point)
        => -48 * point.X * point.X;
}

public class SixthTest : ITest {
    public double U(Point2D point)
        => 4 * point.X * point.X * point.X * point.X + 2 * point.Y * point.Y *
        ↪  point.Y * point.Y;

    public double F(Point2D point)
        => -48 * point.X * point.X - 24 * point.Y * point.Y;
}

```

```
public class SeventhTest : ITest {
    public double U(Point2D point)
        => Math.Exp(point.X) + point.Y;

    public double F(Point2D point)
        => point.Y;
}

public class EighthTest : ITest {
    public double U(Point2D point)
        => point.X * point.X * point.X + point.Y;

    public double F(Point2D point)
        => -6 * point.X;
}
```

Array1DExtension.cs

```
namespace eMP_1;

public static class Array1DExtension {
    public static double Norm(this double[] array) {
        double result = 0;

        for (int i = 0; i < array.Length; i++)
            result += array[i] * array[i];

        return Math.Sqrt(result);
    }

    public static void Fill(this double[] array, double value) {
        for (int i = 0; i < array.Length; i++)
            array[i] = value;
    }

    public static void Copy(this double[] source, double[] destination) {
        for (int i = 0; i < source.Length; i++)
            destination[i] = source[i];
    }
}
```