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

Курсовой проект по курсу

"Метод конечных элементов"

Группа: ПМ-02 ДАНЧЕНКО ИВАН

Новосибирск, 2024

Содержание

Формулировка задания

Реализация конечноэлементной аппроксимации нестационарного электромагнитного поля, возбуждаемого токовой петлей в трехмерной среде.

О конечноэлементной аппроксимации

Постановка нестационарной задачи

Начально-краевая задача определяется уравнением:

$$\operatorname{rot}\left(\frac{1}{\mu}\ \operatorname{rot}\ \vec{A}\right)+\sigma\frac{\partial\vec{A}}{\partial t}+\varepsilon\frac{\partial^2\vec{A}}{\partial t^2}=\vec{J^{\text{ст}}}$$

$$\left(\vec{A}\times\vec{n}\right)\Big|_{S_1}=\vec{A^g}\times\vec{n},$$

$$\vec{A}_{t=t_0}=\vec{A}_0.$$

Аппроксимация по времени

Обозначим через $\vec{A}=\vec{A}(x,y,z,t_j)$ значение вектор-потенциала на текущем временном слое, а через $A^{\vec{\Leftarrow 1}}=\vec{A}(x,y,z,t_{j-1})$, $A^{\Leftarrow 2}=\vec{A}(x,y,z,t_{j-2})$ и $A^{\vec{\Leftarrow 3}}=\vec{A}(x,y,z,t_{j-3})$ - значение вектор-потенциала на трёх предыдущих временных слоях. Тогда в результате аппроксимации по времени получим векторное уравнение:

$$\operatorname{rot}\left(\frac{1}{\mu}\ \operatorname{rot}\vec{A}\right)+\gamma\vec{A}=\vec{F}$$

где коэффициент γ и вектор-функция $\vec{F}$ определяются схемой аппроксимации по времени. В этом случае используется четырёхслойная неявная схема. Сделаем замену обозначения временных промежутков для более удобной записи:

$$\begin{aligned}
t_{01}&=t_j-t_{j-1}, \quad & t_{12}&=t_{j-1}-t_{j-2},\\
t_{02}&=t_j-t_{j-2}, \quad & t_{13}&=t_{j-1}-t_{j-3},\\
t_{03}&=t_j-t_{j-3}, \quad & t_{23}&=t_{j-2}-t_{j-3}.
\end{aligned}$$

Коэффициент γ и вектор-функция $\vec{F}$ имеют вид:

$$\gamma=\frac{\sigma(t_{01}t_{02}+t_{01}t_{03}+t_{02}t_{03})+2\varepsilon(t_{01}+t_{02}+t_{03})}{t_{01}t_{02}t_{03}},$$

$$\vec{F}=\vec{J^{\text{ст}}}+\frac{\sigma t_{02}t_{03}+2\varepsilon(t_{02}+t_{03})}{t_{01}t_{12}t_{13}}\vec{A^{\Leftarrow 1}}-\frac{\sigma t_{01}t_{03}+2\varepsilon(t_{01}+t_{03})}{t_{02}t_{12}t_{23}}\vec{A^{\Leftarrow 2}}+\frac{\sigma t_{01}t_{02}+2\varepsilon(t_{01}+t_{02})}{t_{03}t_{13}t_{23}}\vec{A^{\Leftarrow 3}}.$$

Локальные матрицы и вектор

В одномерном случае линейные базисные функции имеют вид:

$$\psi_1 = \mathrm{N}_1(\nu) = \frac{\nu_{r+1} - \nu}{h_\nu}, \psi_2 = \mathrm{N}_2(\nu) = \frac{\nu - \nu_r}{h_\nu}.$$

Перейдём в трехмерный случай и базисные функции получат вид:

$$\vec{\psi}_1 = \begin{pmatrix} Y_1(y)\cdot Z_1(z) \\ 0 \\ 0 \end{pmatrix}, \quad \vec{\psi}_2 = \begin{pmatrix} Y_2(y)\cdot Z_1(z) \\ 0 \\ 0 \end{pmatrix}, \quad \vec{\psi}_3 = \begin{pmatrix} 0 \\ X_1(x)\cdot Z_1(z) \\ 0 \end{pmatrix}, \quad \vec{\psi}_4 = \begin{pmatrix} 0 \\ X_2(x)\cdot Z_1(z) \\ 0 \end{pmatrix},$$

$$\vec{\psi}_5 = \begin{pmatrix} 0 \\ 0 \\ X_1(x)\cdot Y_1(y) \end{pmatrix}, \quad \vec{\psi}_6 = \begin{pmatrix} 0 \\ 0 \\ X_2(x)\cdot Y_1(y) \end{pmatrix}, \quad \vec{\psi}_7 = \begin{pmatrix} 0 \\ 0 \\ X_1(x)\cdot Y_2(y) \end{pmatrix}, \quad \vec{\psi}_8 = \begin{pmatrix} 0 \\ 0 \\ X_2(x)\cdot Y_2(y) \end{pmatrix},$$

$$\vec{\psi}_9 = \begin{pmatrix} Y_1(y)\cdot Z_2(z) \\ 0 \\ 0 \end{pmatrix}, \quad \vec{\psi}_{10} = \begin{pmatrix} Y_2(y)\cdot Z_2(z) \\ 0 \\ 0 \end{pmatrix}, \quad \vec{\psi}_{11} = \begin{pmatrix} 0 \\ X_1(x)\cdot Z_2(z) \\ 0 \end{pmatrix}, \quad \vec{\psi}_{12} = \begin{pmatrix} 0 \\ X_2(x)\cdot Z_2(z) \\ 0 \end{pmatrix}.$$

Локальные матрицы жёсткости и масс собираются по формулам:

$$G_{ij} = \int\limits_\Omega \mathrm{rot}\ \vec{\psi}_i \cdot \mathrm{rot}\ \vec{\psi}_j d\Omega, \quad M_{ij} = \int\limits_\Omega \gamma \vec{\psi}_i \cdot \vec{\psi}_j d\Omega.$$

Компоненты вектора b правой части определяются соотношением:

$$b_i = \int\limits_\Omega \vec{F} \cdot \vec{\psi}_i d\Omega$$

В программе используется численное интегрирование для вычисления локальных матриц. А вектор правой части вычисляется посредством умножения матрицы масс на вектор $\vec{F}$.

Входные файлы

GridParameters - файл для задания пространственной сетки. Файл состоит из 10 + n-строк (взависимости от количества заданных зон для коэффициента σ), каждая строка состоит из значений, где через пробел разделяются значения (в примере будет показано с использованием символа | для более понятного представления читателю):

1. количество разбиений по x 2-n. начало по x | конец по x | кол-во шагов | коэф. разрядки
3. разбиения слоёв по x
4. количество разбиений по y

5-n. начало по y | конец по y | кол-во шагов | коэф. разрядки
6. разбиения слоёв по y
7. количество разбиений по z

8-n. начало по z | конец по z | кол-во шагов | коэф. разрядки
9. разбиения слоёв по z
11. левая граница | правая граница | нижняя граница | верхняя граница | задняя граница | передняя граница
12. мю | эпсилон
13. количество разрывов n

14-n. значение сигмы | индекс начала зоны по x | индекс конца зоны по x | индекс начала зоны по y | индекс конца зоны по y | индекс начала зоны по z | индекс конца зоны по z

Пример задания области:

```
5
−5000 −200 4 1
−200 −100 4 1
−100 100 8 1
100 200 4 1
200 5000 4 1
−5000 −200 −100 100 200 5000
5
−5000 −200 4 1
−200 −100 4 1
−100 100 8 1
100 200 4 1
200 5000 4 1
−5000 −200 −100 100 200 5000
4
−5000 −100 6 1
−100 0 4 1
0 100 4 1
100 5000 6 1
−5000 −100 0 100 5000
1 1 1 1 1 1
1 0
2
1e-7 0 5 0 5 0 2
0.01 0 5 0 5 2 4
```

TimeGridParameters - файл для задания временной сетки. Во временной сетке нужно всего 4 параметра:

1. начало по t | конец по t | кол-во шагов по t | коэф. разрядки по t

Пример:

```
0 1e-2 100 1.2
```

Описание и возможности программы

1. В программе можно выбрать как будут собираться временные слои: физически (собирая из начальных условий двухслойную неявную схему, далее трёхслойную и потом четырёхслойную) или сгенерировать точные значения на всех трёх слоях.
2. СЛАУ можно решать с помощью BCGStab-LU или решение методом LU-разложения.

Немного про классы в программе:

1. Основным классом является **FEM**, в котором происходит генерация портрета глобальной матрицы, сборка локальных матриц и векторов, учёт краевых условий.
2. Генерация пространственной сетки происходит в классе **Grid**, а временной в **TimeGrid**
3. Для реализации численного интегрирования были созданы 2 record struct-а: **SegmentGaussOrder9**, который является методом Гаусса-9, и **TriLinearVectorBasis** для удобного вычисления базисных вектор-функций.
4. Само численное интегрирование реализовано в классе **Integration**, в который передаются все квадратуры.

5. Оставшиеся классы реализуют структуры для удобной работы с основными классами. Например есть 2 класса **SparseMatrix** и **Vector**, которые используются для хранения глобальной матрицы и вектора соответственно.

О физической задаче

В воздухе как-то располагается генератор, он большой, как такое может быть я не знаю. В первый момент времени в установившемся поле отключается генератор.

Сборка начального поля

Для сборки используется закон Био-Савара-Лапласа:

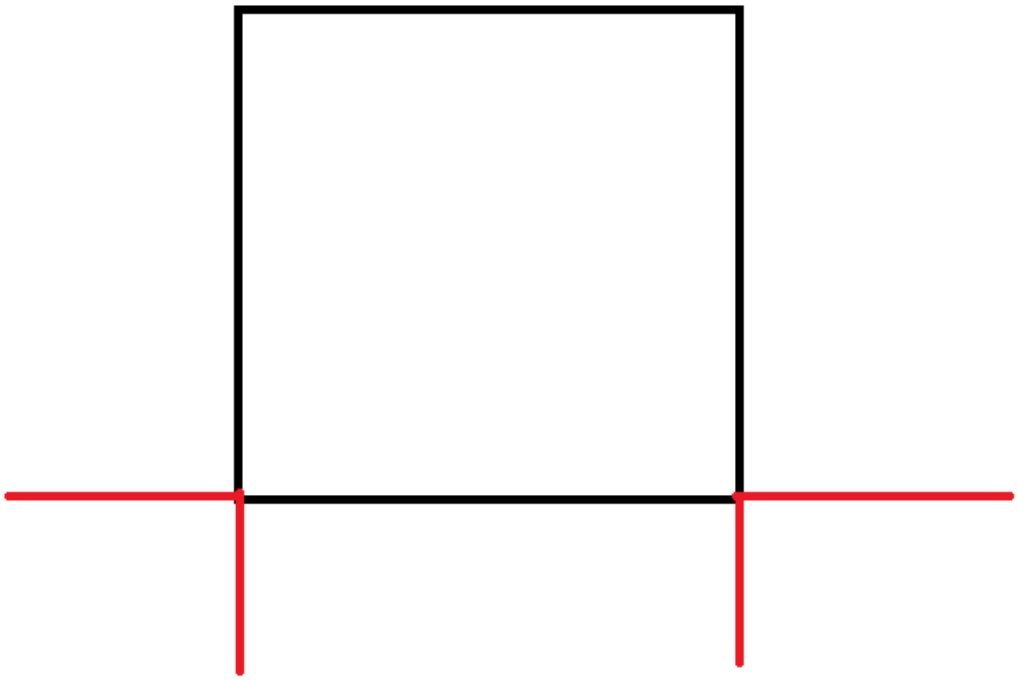
$$\vec{A}(r) = \frac{\mu_0}{4\pi} \int\limits_{\Omega} \frac{\vec{J}(r') \times (r - r')}{\parallel r - r' \parallel} \mathrm{dr}'$$

Для численного расчёта поля вся рамка разбивалась на маленькие отрезки (dh), на которых считалось значение поля. Далее перебирались все грани расчётной сетки, при этом как значение грани рассматривалась её середина. таким образом формула преобразовывается в вид:

$$\vec{A} = \sum_{i=1}^{n} \frac{\mu_0}{4\pi} \frac{\vec{J} \cdot \mathrm{dh}}{\sqrt{(x - x')^2 + (y - y')^2 + (z - z')^2}},$$

где n - количество граней.

Т.к. сетка симметричная относительно центра, то значения поля по модулю должны быть равные. На рисунке ниже представлен пример одинаковых значений. Прямоугольник с чёрной обводкой - генератор, а красные отрезки, выходящие из него, как раз и будут иметь одинаковые по модулю значения.



Также, чем дальше ребро находится от генератора, тем меньшее значение поля A имеет ребро. В таблице ниже представлен пример убывания поля от центра генератора. Для этого фиксируются значения по y и z и изменяется только значение по x.

x	A
-350	8.558631370619078E-08
-340	3.6786038161317225E-07
-330	3.8957377145217084E-07
-320	4.112871612911694E-07
-310	4.3300055113016796E-07
-300	4.5471394096916655E-07
-290	4.7642733080816513E-07
-280	4.981407206471637E-07
-270	5.198541104861623E-07
-260	5.415675003251609E-07
-250	5.632808901641594E-07
-240	5.84994280003158E-07
-230	6.067076698421566E-07
-220	6.284210596811551E-07
-210	6.501344495201538E-07
-200	6.718478393591523E-07
-190	6.935612291981508E-07
-180	7.152746190371495E-07
-170	7.36988008876148E-07
-160	7.587013987151467E-07
-150	7.804147885541452E-07
-140	8.021281783931437E-07
-130	8.238415682321424E-07
-120	8.455549580711409E-07
-110	8.672683479101394E-07
-100	8.889817377491381E-07
-90	9.106951275881366E-07
-80	9.324085174271353E-07
-70	9.541219072661338E-07
-60	9.758352971051323E-07
-50	9.975486869441309E-07
-40	1.0192620767831296E-06
-30	2.6563322281905395E-06
-20	2.435799837362655E-06
-10	1.4470282148126285E-06
0	0

Расчёт вектора индукции магнитного поля $\vec{B}$

Вектор индукции магнитного поля можно выразить через $\vec{A}$:

$$\vec{B} = \text{rot } \vec{A}$$

Для рассчёта применяется метод аппроксимации производной по базисным функциям, которые принимают вид:

$$\vec{B} = \sum_{i=0}^{n} \text{rot}\vec{\psi} \cdot A_i$$

Расчёт вектора напряжённости электрического поля $\vec{E}$

Вектор вектор напряжённости электрического поля можно выразить через $\vec{A}$:

$$\vec{E} = \frac{d\vec{A}}{\text{dt}}$$

Для рассчёта применяется метод аппроксимации по временной сетке, которые принимают вид:

$$\vec{E} = \frac{\vec{A}(t) - \vec{A}(t-\tau)}{t-\tau}$$

Листинг

FEM.cs

```
namespace VectorFEM3D;

public class FEM
{
    private SparseMatrix? _globalMatrix;
    private Vector? _globalVector;
    private Vector[]? _layers;
    private Vector? _solution;
    private Vector? _localVector;
    private Matrix? _stiffnessMatrix;
    private Matrix? _massMatrix;
    private Grid? _grid;
    private ITimeGrid _timeGrid;
    private Test? _test;
    private IBasis3D? _basis;
    private Integration? _integration;
    private SLAE? _slae;
    private Scheme _scheme;
    private Scheme _activeScheme;
    private Generator _generator = new Generator(-100, -100, 25, 100, 100, 25);

    private const double _mu0 = 1.25653706212 * 10e-6;
    //private const double _mu0 = 4 * Math.PI * 10e-7;

    public FEM(Grid grid, ITimeGrid timeGrid)
    {
        _grid = grid;
        _timeGrid = timeGrid;
        _basis = new TriLinearVectorBasis();
        _integration = new Integration(new SegmentGaussOrder9());
        _stiffnessMatrix = new(_basis.Size);
```

```
    _massMatrix = new(_basis.Size);
    _localVector = new(_basis.Size);
}

public void SetTest(Test test)
{
    _test = test;
}

public void SetSolver(SLAE slae)
{
    _slae = slae;
}

public void SetScheme(Scheme scheme)
{
    _scheme = scheme;
    _activeScheme = scheme;
}

public void Compute()
{
    BuildPortrait();
    PrepareLayers();

    int itime = 0;

    switch (_scheme)
    {
        case Scheme.Natural:
            itime = 0;

            _activeScheme = Scheme.Two_layer_Implicit;
            itime++;

            var hx = 5;
            var hy = 5;
            double delta = 1e-10;

            var stepsX = _generator.Length / hx;
            var stepsY = _generator.Width / hy;

            for (int i = 0; i < _grid.Edges.Length; i++)
            {
                double point;
                double len1;
                double len2;

                switch (_grid.Edges[i].GetAxis())
                {
                    case 0:
                        point = _generator.xStart + hx / 2.0;

                        for (int j = 0; j < stepsX; j++)
                        {
                         len1 = Math.Sqrt(Math.Pow(point - _grid.Edges[i].Point.X,
```

```
2) +
                                                    Math.Pow(_generator.yStart -
_grid.Edges[i].Point.Y, 2) +
                                                    Math.Pow(_generator.zStart -
_grid.Edges[i].Point.Z, 2));

                                if (len1 < 1e-10)
                                {
                                    len1 = delta;
                                }

                               len2 = Math.Sqrt(Math.Pow(point - _grid.Edges[i].Point.X,
2) +
                                                      Math.Pow(_generator.yEnd -
_grid.Edges[i].Point.Y, 2) +
                                                    Math.Pow(_generator.zStart -
_grid.Edges[i].Point.Z, 2));

                                if (len2 < 1e-10)
                                {
                                    len2 = delta;
                                }

                                _solution[i] += _mu0 / (4 * Math.PI) * hx / len1;
                                _solution[i] += -_mu0 / (4 * Math.PI) * hx / len2;

                                point += hx;
                            }

                            break;

                        case 1:
                            point = _generator.yStart + hy / 2.0;

                            for (int j = 0; j < stepsY; j++)
                            {
                                len1 = Math.Sqrt(
                                   Math.Pow(_generator.xEnd - _grid.Edges[i].Point.X,
2) +
                                    Math.Pow(point - _grid.Edges[i].Point.Y, 2) +
                                  Math.Pow(_generator.zStart - _grid.Edges[i].Point.Z,
2));

                                if (len1 < 1e-10)
                                {
                                    len1 = delta;
                                }

                                len2 = Math.Sqrt(
                                  Math.Pow(_generator.xStart - _grid.Edges[i].Point.X,
2) +
                                    Math.Pow(point - _grid.Edges[i].Point.Y, 2) +
                                  Math.Pow(_generator.zStart - _grid.Edges[i].Point.Z,
2));

                                if (len2 < 1e-10)
```

```
                        {
                            len2 = delta;
                        }

                        _solution[i] += _mu0 / (4 * Math.PI) * hy / len1;
                        _solution[i] += -_mu0 / (4 * Math.PI) * hy / len2;

                        point += hy;
                    }

                    break;

                case 2:
                    break;
            }

        }

        for (int i = 0; i < _grid.Edges.Length; i++)
        {
            if (_grid.DirichletBoundaries.Contains(i))
            {
                _solution[i] = 0;
            }
        }


        break;

    case Scheme.Two_layer_Implicit:
        itime = 1;
        break;

    case Scheme.Three_layer_Implicit:
        itime = 2;
        break;

    case Scheme.Four_layer_Implicit:
        itime = 3;
        break;
}


using (var sw = new StreamWriter("Tests/5.csv"))
{
    for ( ; itime < _timeGrid.TGrid.Length; itime++)
    {
        AssemblySLAE(itime);
        AccountDirichletBoundaries(itime);


        switch (_activeScheme)
        {
            case Scheme.Two_layer_Implicit:
                _slae.SetSLAE(_globalVector, _globalMatrix, _layers[0]);
```

```
            break;

        case Scheme.Three_layer_Implicit:
            _slae.SetSLAE(_globalVector, _globalMatrix, _layers[1]);

            break;

        case Scheme.Four_layer_Implicit:
            _slae.SetSLAE(_globalVector, _globalMatrix, _layers[2]);

            break;

    }

    _solution = _slae.Solve();

    List<List<double>> Bz = new List<List<double>>();
    List<List<double>> Ax = new List<List<double>>();
    List<List<double>> Ay = new List<List<double>>();

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

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

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

    var zo25 = new Point3D(-4975.0, 0.0, -25.0);
    var z0 = new Point3D(-4975.0, 0.0, 0);
    var z25 = new Point3D(-4975.0, 0.0, 25.0);

    var zo25y = new Point3D(0.0, -4975.0, -25.0);
    var z0y = new Point3D(0.0, -4975.0, 0);
    var z25y = new Point3D(0.0, -4975.0, 25.0);

    double h = 25;

    while (zo25.X < 5000)
    {
        var Ao25 = GetValue(zo25);
        var A0 = GetValue(z0);
        var A25 = GetValue(z25);

        Bz[0].Add(GetValueForRotAz(zo25));
        Bz[1].Add(GetValueForRotAz(z0));
        Bz[2].Add(GetValueForRotAz(z25));

        //Ax[0].Add(Ao25.X);
        //Ax[1].Add(A0.X);
        //Ax[2].Add(A25.X);
```

```
        Ay[0].Add(Ao25.Y);
        Ay[1].Add(A0.Y);
        Ay[2].Add(A25.Y);

        zo25.X += h;
        z0.X += h;
        z25.X += h;
    }

    while (zo25y.Y < 5000)
    {
        var Ao25 = GetValue(zo25y);
        var A0 = GetValue(z0y);
        var A25 = GetValue(z25y);

        //Bz[0].Add(GetValueForRotAz(zo25));
        //Bz[1].Add(GetValueForRotAz(z0));
        //Bz[2].Add(GetValueForRotAz(z25));

        Ax[0].Add(Ao25.X);
        Ax[1].Add(A0.X);
        Ax[2].Add(A25.X);

        //Ay[0].Add(Ao25.Y);
        //Ay[1].Add(A0.Y);
        //Ay[2].Add(A25.Y);

        zo25y.Y += h;
        z0y.Y += h;
        z25y.Y += h;
    }

    using (var anime1 = new StreamWriter("Tests/1Bz-25.txt"))
    {
        foreach (var kek in Bz[0])
        {
            anime1.WriteLine(kek);
        }
    }

    using (var anime1 = new StreamWriter("Tests/1Bz0.txt"))
    {
        foreach (var kek in Bz[1])
        {
            anime1.WriteLine(kek);
        }
    }

    using (var anime1 = new StreamWriter("Tests/1Bz25.txt"))
    {
        foreach (var kek in Bz[2])
        {
            anime1.WriteLine(kek);
        }
    }
```

```
                using (var anime1 = new StreamWriter("Tests/1Ax-25.txt"))
                {
                    foreach (var kek in Ax[0])
                    {
                        anime1.WriteLine(kek);
                    }
                }

                using (var anime1 = new StreamWriter("Tests/1Ax0.txt"))
                {
                    foreach (var kek in Ax[1])
                    {
                        anime1.WriteLine(kek);
                    }
                }

                using (var anime1 = new StreamWriter("Tests/1Ax25.txt"))
                {
                    foreach (var kek in Ax[2])
                    {
                        anime1.WriteLine(kek);
                    }
                }

                using (var anime1 = new StreamWriter("Tests/1Ay-25.txt"))
                {
                    foreach (var kek in Ay[0])
                    {
                        anime1.WriteLine(kek);
                    }
                }

                using (var anime1 = new StreamWriter("Tests/1Ay0.txt"))
                {
                    foreach (var kek in Ay[1])
                    {
                        anime1.WriteLine(kek);
                    }
                }

                using (var anime1 = new StreamWriter("Tests/1Ay25.txt"))
                {
                    foreach (var kek in Ay[2])
                    {
                        anime1.WriteLine(kek);
                    }
                }

                //var kek = GetValue(new Point3D(0.0, 0.0, 30));
                //var anime = CalculateEMF(new Point3D(0.25, 0.25, 30), itime);
                var A1 = GetValue(new Point3D(0.0, 0.0, 25.0));
                var A2 = GetValue(new Point3D(2.5, 0.0, 0.0));
                var A3 = GetValue(new Point3D(0.0, 2.5, 0.0));
                    //var modA1 = Math.Sqrt(Math.Pow(A1.X, 2) + Math.Pow(A1.Y, 2) +
Math.Pow(A1.Z, 2));
                    //var modA2 = Math.Sqrt(Math.Pow(A2.X, 2) + Math.Pow(A2.Y, 2) +
```

```
Math.Pow(A2.Z, 2));


            //Console.WriteLine($"{itime} = {_timeGrid[itime]} EMF = {CalculateEMF(new
Point3D(0.1, 0.1, 30), itime)}");
             Console.WriteLine($"{itime} = {_timeGrid[itime]} dBz = {CalculatedBz(new
Point3D(0, 0, 25), itime)}");
              Console.WriteLine($"modA1 = ({A1.X}, {A1.Y}, {A1.Z})\n" +
                          $"modA2 = ({A2.X}, {A2.Y}, {A2.Z})\n" +
                          $"modA3 = ({A3.X}, {A3.Y}, {A3.Z})\n");
              sw.WriteLine($"{itime} {CalculatedBz(new Point3D(0, 0, 25), itime)}");

              switch (_activeScheme)
              {
                  case Scheme.Two_layer_Implicit:
                      if (_scheme == Scheme.Natural)
                      {
                          if (itime == 2)
                          {
                              _activeScheme = Scheme.Three_layer_Implicit;

                              Vector.Copy(_solution, _layers[1]);

                              break;
                          }

                          Vector.Copy(_solution, _layers[0]);
                      }

                      else
                      {
                          Vector.Copy(_solution, _layers[0]);
                      }
                      break;

                  case Scheme.Three_layer_Implicit:
                      if (_scheme == Scheme.Natural)
                      {
                          //_activeScheme = Scheme.Four_layer_Implicit;

                          //Vector.Copy(_solution, _layers[2]);

                          Vector.Copy(_layers[1], _layers[0]);
                          Vector.Copy(_solution, _layers[1]);
                      }

                      else
                      {
                          Vector.Copy(_layers[1], _layers[0]);
                          Vector.Copy(_solution, _layers[1]);
                      }

                      break;

                  case Scheme.Four_layer_Implicit:
                      Vector.Copy(_layers[1], _layers[0]);
```

```
                        Vector.Copy(_layers[2], _layers[1]);
                        Vector.Copy(_solution, _layers[2]);
                        break;
                }

                //double error = 0;
                //for (int i = 0; i < _grid.Edges.Length; i++)
                //{
                //    error += Math.Pow(
                    //          _test.UValue(_grid.Edges[i].Point, _timeGrid[itime],
_grid.Edges[i].GetAxis()) - _solution[i], 2);
                //}
                //PrintError(itime);
                            //sw.WriteLine($"{_timeGrid[itime]},{Math.Sqrt(error  /
_grid.Edges.Length)}");

                //for (int i = 0; i < _grid.Edges.Length; i++)
                //{
                //    sw.WriteLine(
                                    //                  $"{i   +   1},{_solution[i]},
{_test.UValue(_grid.Edges[i].Point,     _timeGrid[0],     _grid.Edges[i].GetAxis())},
{_test.UValue(_grid.Edges[i].Point,    _timeGrid[0],    _grid.Edges[i].GetAxis())    -
_solution[i]}");
                //
                //    Console.WriteLine(
                                    //                  $"{i   +   1},{_solution[i]},
{_test.UValue(_grid.Edges[i].Point,     _timeGrid[0],     _grid.Edges[i].GetAxis())},
{_test.UValue(_grid.Edges[i].Point,    _timeGrid[0],    _grid.Edges[i].GetAxis())    -
_solution[i]}");
                //}
                //break;
            }
        }
    }

    private void AssemblySLAE(int itime)
    {
        _globalVector.Fill(0);
        _globalMatrix.Clear();

        for (int ielem = 0; ielem < _grid.Elements.Length; ielem++)
        {
            AssemblyLocalElement(ielem, itime);

            if (_activeScheme != Scheme.Natural)
            {
                    _stiffnessMatrix += SchemeUsage(ielem, itime, _activeScheme, 0) *
_massMatrix;
            }

            for (int i = 0; i < _basis.Size; i++)
            {
                for (int j = 0; j < _basis.Size; j++)
                {
                        AddElement(_grid.Elements[ielem][i], _grid.Elements[ielem][j],
_stiffnessMatrix[i, j]);
```

```
                }
            }

            AssemblyGlobalVector(ielem, itime);

            _stiffnessMatrix.Clear();
            _massMatrix.Clear();
            _localVector.Fill(0);
        }
    }

    private void AssemblyGlobalVector(int ielem, int itime)
    {
        double[] qj3 = new double[_basis.Size];
        double[] qj2 = new double[_basis.Size];
        double[] qj1 = new double[_basis.Size];

        switch (_activeScheme)
        {
            case Scheme.Natural:
              if (_grid.Edges[_grid.Elements[ielem][0]].Point0.X >= -51 &&
                  _grid.Edges[_grid.Elements[ielem][0]].Point0.X <= 51 &&
                  _grid.Edges[_grid.Elements[ielem][0]].Point0.Y >= -51 &&
                  _grid.Edges[_grid.Elements[ielem][0]].Point0.Y <= 51 &&
                  _grid.Edges[_grid.Elements[ielem][0]].Point0.Z <= 51 &&
                  _grid.Edges[_grid.Elements[ielem][11]].Point1.Z >= 30)
              {
                  for (int i = 0; i < _basis.Size; i++)
                  {
                      _globalVector[_grid.Elements[ielem][i]] = 1;
                  }
              }

              break;

            case Scheme.Two_layer_Implicit:
                for (int i = 0; i < _basis.Size; i++)
                {
                    for (int j = 0; j < _basis.Size; j++)
                    {
                        qj1[i] += _massMatrix[i, j] * _layers[0][_grid.Elements[ielem]
[j]];
                    }

                }

                for (int i = 0; i < _basis.Size; i++)
                {
                     _localVector[i] += SchemeUsage(ielem, itime, _activeScheme, 1) *
qj1[i];

                    _globalVector[_grid.Elements[ielem][i]] += _localVector[i];
                }

                break;
```

```
            case Scheme.Three_layer_Implicit:

                for (int i = 0; i < _basis.Size; i++)
                {
                    for (int j = 0; j < _basis.Size; j++)
                    {
                        qj2[i] += _massMatrix[i, j] * _layers[1][_grid.Elements[ielem]
[j]];
                        qj1[i] += _massMatrix[i, j] * _layers[0][_grid.Elements[ielem]
[j]];
                    }
                }

                for (int i = 0; i < _basis.Size; i++)
                {
                     _localVector[i] += SchemeUsage(ielem, itime, _activeScheme, 1) *
qj2[i];
                     _localVector[i] += SchemeUsage(ielem, itime, _activeScheme, 2) *
qj1[i];

                    _globalVector[_grid.Elements[ielem][i]] += _localVector[i];
                }

                break;

            case Scheme.Four_layer_Implicit:

                for (int i = 0; i < _basis.Size; i++)
                {
                    for (int j = 0; j < _basis.Size; j++)
                    {
                        qj3[i] += _massMatrix[i, j] * _layers[2][_grid.Elements[ielem]
[j]];
                        qj2[i] += _massMatrix[i, j] * _layers[1][_grid.Elements[ielem]
[j]];
                        qj1[i] += _massMatrix[i, j] * _layers[0][_grid.Elements[ielem]
[j]];
                    }
                }

                for (int i = 0; i < _basis.Size; i++)
                {
                     _localVector[i] += SchemeUsage(ielem, itime, _activeScheme, 1) *
qj3[i];
                     _localVector[i] += SchemeUsage(ielem, itime, _activeScheme, 2) *
qj2[i];
                     _localVector[i] += SchemeUsage(ielem, itime, _activeScheme, 3) *
qj1[i];

                    _globalVector[_grid.Elements[ielem][i]] += _localVector[i];
                }

                break;
        }
    }
```

```
    private void AddElement(int i, int j, double value)
    {
        if (i == j)
        {
            _globalMatrix.Di[i] += value;
            return;
        }

        if (i > j)
        {
            for (int icol = _globalMatrix.Ig[i]; icol < _globalMatrix.Ig[i + 1]; icol+
+)
            {
                if (_globalMatrix.Jg[icol] == j)
                {
                    _globalMatrix.Ggl[icol] += value;
                    return;
                }
            }
        }

        else
        {
            for (int icol = _globalMatrix.Ig[j]; icol < _globalMatrix.Ig[j + 1]; icol+
+)
            {
                if (_globalMatrix.Jg[icol] == i)
                {
                    _globalMatrix.Ggu[icol] += value;
                    return;
                }
            }
        }
    }

    private void AssemblyLocalElement(int ielem, int itime)
    {
        double hx = _grid.Edges[_grid.Elements[ielem][0]].Length;
        double hy = _grid.Edges[_grid.Elements[ielem][2]].Length;
        double hz = _grid.Edges[_grid.Elements[ielem][4]].Length;

        for (int i = 0; i < _basis.Size; i++)
        {
            for (int j = 0; j < _basis.Size; j++)
            {
                Func<Point3D, double> kek;
                Vector3D psi1 = new(0, 0, 0);
                Vector3D psi2 = new(0, 0, 0);
                Vector3D dPsi1 = new(0, 0, 0);
                Vector3D dPsi2 = new(0, 0, 0);

                int ik = i;
                int jk = j;
                kek = point =>
                {
                    psi1.Copy(_basis.GetPsi(ik, point));
```

```
                    psi2.Copy(_basis.GetPsi(jk, point));

                    return psi1 * psi2;
                };

                _massMatrix[i, j] += hx * hy * hz * _integration.Gauss3D(kek);

                kek = point =>
                {
                    dPsi1.Copy(_basis.GetDPsi(ik, point));
                    dPsi2.Copy(_basis.GetDPsi(jk, point));

                    return Vector3D.DotProductJacob(dPsi1, dPsi2, hx, hy, hz);
                };

                _stiffnessMatrix[i, j] += 1 / _grid.Mu * _integration.Gauss3D(kek);
            }

            _localVector[i] = _test.F(_grid.Edges[_grid.Elements[ielem][i]].Point,
_timeGrid[itime], i, _grid.GetSigma(
                new Point3D(_grid.Edges[_grid.Elements[ielem][0]].Point.X,
                    _grid.Edges[_grid.Elements[ielem][3]].Point.Y,
                    _grid.Edges[_grid.Elements[ielem][11]].Point.Z)));
        }

        _localVector = _massMatrix * _localVector;
    }

    private void AccountDirichletBoundaries(int itime)
    {
        foreach (var edge in _grid.DirichletBoundaries)
        {

            _globalMatrix.Di[edge] = 1;
          _globalVector[edge] = _test.UValue(_grid.Edges[edge].Point, _timeGrid[itime],
_grid.Edges[edge].GetAxis());

            for (int i = _globalMatrix.Ig[edge]; i < _globalMatrix.Ig[edge + 1]; i++)
                _globalMatrix.Ggl[i] = 0;

            for (int col = edge + 1; col < _globalMatrix.Size; col++)
               for (int j = _globalMatrix.Ig[col]; j < _globalMatrix.Ig[col + 1]; j++)
                    if (_globalMatrix.Jg[j] == edge)
                    {
                        _globalMatrix.Ggu[j] = 0;
                        break;
                    }
        }
    }

    private double SchemeUsage(int ielem, int itime, Scheme scheme, int i)
    {
       double t01;
       double t02;
       double t12;
```

```
        switch (scheme)
         {
             case Scheme.Two_layer_Implicit:
                 t01 = _timeGrid[itime] - _timeGrid[itime - 1];
                 switch (i)
                 {
                        // тут только параболическая, эпсилон для гиперболики придётся
добавить
                   case 0:
                   return (_grid.GetSigma(new Point3D(_grid.Edges[_grid.Elements[ielem]
[0]].Point.X,
                                _grid.Edges[_grid.Elements[ielem][3]].Point.Y,
                                _grid.Edges[_grid.Elements[ielem][11]].Point.Z)) / t01);

                   case 1:
                   return (_grid.GetSigma(new Point3D(_grid.Edges[_grid.Elements[ielem]
[0]].Point.X,
                                _grid.Edges[_grid.Elements[ielem][3]].Point.Y,
                                _grid.Edges[_grid.Elements[ielem][11]].Point.Z)) / t01);
                 }

                 break;

             case Scheme.Three_layer_Implicit:
                 t01 = _timeGrid[itime] - _timeGrid[itime - 1];
                 t02 = _timeGrid[itime] - _timeGrid[itime - 2];
                 t12 = _timeGrid[itime - 1] - _timeGrid[itime - 2];

                 switch (i)
                 {
                     case 0:
                   return (_grid.GetSigma(new Point3D(_grid.Edges[_grid.Elements[ielem]
[0]].Point.X,
                                     _grid.Edges[_grid.Elements[ielem][3]].Point.Y,
                                      _grid.Edges[_grid.Elements[ielem][11]].Point.Z)) *
(t01 + t02) + 2 * _grid.Epsilon) /
                                 (t01 * t02);

                   case 1:
                   return (_grid.GetSigma(new Point3D(_grid.Edges[_grid.Elements[ielem]
[0]].Point.X,
                                _grid.Edges[_grid.Elements[ielem][3]].Point.Y,
                                _grid.Edges[_grid.Elements[ielem][11]].Point.Z)) * t02 + 2
* _grid.Epsilon) / (t01 * t12);

                   case 2:
                  return -(_grid.GetSigma(new Point3D(_grid.Edges[_grid.Elements[ielem]
[0]].Point.X,
                                _grid.Edges[_grid.Elements[ielem][3]].Point.Y,
                                _grid.Edges[_grid.Elements[ielem][11]].Point.Z)) * t01 + 2
* _grid.Epsilon) / (t02 * t12);

                 }

                 break;
```

```
            case Scheme.Four_layer_Implicit:
                t01 = _timeGrid[itime] - _timeGrid[itime - 1];
                t02 = _timeGrid[itime] - _timeGrid[itime - 2];
                t12 = _timeGrid[itime - 1] - _timeGrid[itime - 2];
                double t03 = _timeGrid[itime] - _timeGrid[itime - 3];
                double t13 = _timeGrid[itime - 1] - _timeGrid[itime - 3];
                double t23 = _timeGrid[itime - 2] - _timeGrid[itime - 3];

                switch (i)
                {
                    case 0:
                  return (_grid.GetSigma(new Point3D(_grid.Edges[_grid.Elements[ielem]
[0]].Point.X,
                                        _grid.Edges[_grid.Elements[ielem][3]].Point.Y,
                                       _grid.Edges[_grid.Elements[ielem][11]].Point.Z))
                                          * (t01 * t02 + t01 * t03 + t02 * t03) + 2 *
_grid.Epsilon * (t01 + t02 + t03)) /
                                   (t01 * t02 * t03);

                    case 1:
                  return (_grid.GetSigma(new Point3D(_grid.Edges[_grid.Elements[ielem]
[0]].Point.X,
                                       _grid.Edges[_grid.Elements[ielem][3]].Point.Y,
                                      _grid.Edges[_grid.Elements[ielem][11]].Point.Z)) *
t02 * t03 +
                                    2 * _grid.Epsilon * (t02 + t03)) / (t01 * t12 * t13);

                    case 2:
                   return -(_grid.GetSigma(new Point3D(_grid.Edges[_grid.Elements[ielem]
[0]].Point.X,
                                        _grid.Edges[_grid.Elements[ielem][3]].Point.Y,
                                        _grid.Edges[_grid.Elements[ielem][11]].Point.Z))
* t01 * t03 +
                                     2 * _grid.Epsilon * (t01 + t03)) / (t02 * t12 * t23);

                    case 3:
                  return (_grid.GetSigma(new Point3D(_grid.Edges[_grid.Elements[ielem]
[0]].Point.X,
                                       _grid.Edges[_grid.Elements[ielem][3]].Point.Y,
                                      _grid.Edges[_grid.Elements[ielem][11]].Point.Z)) *
t01 * t02 +
                                    2 * _grid.Epsilon * (t01 + t02)) / (t03 * t13 * t23);
                }

                return 0;

            default:
                throw new Exception("Undefined scheme");
        }

        throw new Exception("SchemeUsage can't return a value");
    }

    private void BuildPortrait()
    {
          HashSet<int>[] list = new HashSet<int>[_grid.Edges.Length].Select(_ => new
```

```
HashSet<int>()).ToArray();
        foreach (var element in _grid.Elements)
            foreach (var pos in element)
                foreach (var node in element)
                    if (pos > node)
                        list[pos].Add(node);

        list = list.Select(childlist => childlist.Order().ToHashSet()).ToArray();
        int count = list.Sum(childlist => childlist.Count);

        _globalMatrix = new(_grid.Edges.Length, count);
        _globalVector = new(_grid.Edges.Length);
        _solution = new(_grid.Edges.Length);
        _layers = new Vector[3].Select(_ => new Vector(_grid.Edges.Length)).ToArray();

        _globalMatrix.Ig[0] = 0;

        for (int i = 0; i < list.Length; i++)
            _globalMatrix.Ig[i + 1] = _globalMatrix.Ig[i] + list[i].Count;

        int k = 0;

        foreach (var childlist in list)
            foreach (var value in childlist)
                _globalMatrix.Jg[k++] = value;
    }

    private void PrepareLayers()
    {
        switch (_scheme)
        {
            case Scheme.Two_layer_Implicit:
                for (int i = 0; i < _grid.Edges.Length; i++)
                {
                    _layers[0][i] = _test.UValue(_grid.Edges[i].Point, _timeGrid[0],
_grid.Edges[i].GetAxis());
                }
                break;

            case Scheme.Three_layer_Implicit:
                for (int i = 0; i < _grid.Edges.Length; i++)
                {
                    _layers[0][i] = _test.UValue(_grid.Edges[i].Point, _timeGrid[0],
_grid.Edges[i].GetAxis());
                    _layers[1][i] = _test.UValue(_grid.Edges[i].Point, _timeGrid[1],
_grid.Edges[i].GetAxis());
                }

                break;

            case Scheme.Four_layer_Implicit:
                for (int i = 0; i < _grid.Edges.Length; i++)
                {
                    _layers[0][i] = _test.UValue(_grid.Edges[i].Point, _timeGrid[0],
_grid.Edges[i].GetAxis());
                    _layers[1][i] = _test.UValue(_grid.Edges[i].Point, _timeGrid[1],
```

```
_grid.Edges[i].GetAxis());
                    _layers[2][i] = _test.UValue(_grid.Edges[i].Point, _timeGrid[2],
_grid.Edges[i].GetAxis());
                }

                break;
            }
        }

        private void PrintError(int itime)
        {
            double error = 0;
            for (int i = 0; i < _grid.Edges.Length; i++)
            {
                error += Math.Pow(
                                    _test.UValue(_grid.Edges[i].Point,  _timeGrid[itime],
_grid.Edges[i].GetAxis()) - _solution[i], 2);
            }
                    Console.WriteLine($"Layer  error  {itime}  =  {Math.Sqrt(error  /
_grid.Edges.Length)}");
        }

        public Vector3D GetValue(Point3D point)
        {
            var vector = new Vector3D(0, 0, 0);
            var kek = new Point3D(0, 0, 0);

            foreach (var elem in _grid.Elements)
            {
                        if  (point.X  >=  _grid.Edges[elem[0]].Point0.X  &&  point.X  <
_grid.Edges[elem[11]].Point1.X &&
                                point.Y  >=  _grid.Edges[elem[0]].Point0.Y  &&  point.Y  <
_grid.Edges[elem[11]].Point1.Y &&
                                point.Z  >=  _grid.Edges[elem[0]].Point0.Z  &&  point.Z  <
_grid.Edges[elem[11]].Point1.Z)
                {
                                    kek.X  =  (point.X  -  _grid.Edges[elem[0]].Point0.X)  /
_grid.Edges[elem[0]].Length;
                                    kek.Y  =  (point.Y  -  _grid.Edges[elem[2]].Point0.Y)  /
_grid.Edges[elem[2]].Length;
                                    kek.Z  =  (point.Z  -  _grid.Edges[elem[4]].Point0.Z)  /
_grid.Edges[elem[4]].Length;

                    for (int i = 0; i < _basis.Size; i++)
                    {
                        vector += _basis.GetPsi(i, kek) * _solution[elem[i]];
                    }

                    break;
                }
            }

            return vector;
        }

        private double CalculateEMF(Point3D point, int itime)
```

```
    {
        double result = 0;
        double height = 30;
        double pointX = 0;
        double pointY = 0;

        for (int ielem = 0; ielem < _grid.Elements.Length; ielem++)
        {
            if (point.X >= _grid.Edges[_grid.Elements[ielem][0]].Point0.X &&
                point.X < _grid.Edges[_grid.Elements[ielem][11]].Point1.X &&
                point.Y >= _grid.Edges[_grid.Elements[ielem][0]].Point0.Y &&
                point.Y < _grid.Edges[_grid.Elements[ielem][11]].Point1.Y &&
                point.Z >= _grid.Edges[_grid.Elements[ielem][0]].Point0.Z &&
                point.Z < _grid.Edges[_grid.Elements[ielem][11]].Point1.Z)
            {
                var hx = 1e-4;
                var hy = 1e-4;

                var stepsX = (_grid.Edges[_grid.Elements[ielem][11]].Point1.X -
                         _grid.Edges[_grid.Elements[ielem][0]].Point0.X) / hx;
                var stepsY = (_grid.Edges[_grid.Elements[ielem][11]].Point1.Y -
                         _grid.Edges[_grid.Elements[ielem][0]].Point0.Y) / hy;

                pointX = _grid.Edges[_grid.Elements[ielem][0]].Point0.X + hx / 2;
                pointY = _grid.Edges[_grid.Elements[ielem][0]].Point0.Y + hy / 2;

                for (int i = 0; i < stepsX; i++)
                {
                    result += hx *
                              GetValueFordAdt(
                                 new Point3D(pointX, _grid.Edges[_grid.Elements[ielem]
[0]].Point0.Y, point.Z), ielem,
                                  itime).X;

                    result += -hx *
                              GetValueFordAdt(
                                 new Point3D(pointX, _grid.Edges[_grid.Elements[ielem]
[11]].Point1.Y, point.Z), ielem,
                                  itime).X;

                    pointX += hx;
                }

                for (int i = 0; i < stepsY; i++)
                {
                    result += hy *
                              GetValueFordAdt(
                                       new Point3D( _grid.Edges[_grid.Elements[ielem]
[0]].Point0.X, pointY, point.Z), ielem,
                                  itime).Y;

                    result += -hy *
                              GetValueFordAdt(
                                        new Point3D(_grid.Edges[_grid.Elements[ielem]
[11]].Point1.X, pointY, point.Z), ielem,
                                  itime).Y;
```

```
                    pointY += hy;
                }

                break;
            }
        }

        return result;
    }

    private Vector3D GetValueFordAdt(Point3D point, int ielem, int itime)
    {
        var vector1 = new Vector3D(0, 0, 0);
        var vector2 = new Vector3D(0, 0, 0);
        var kek = new Point3D(0, 0, 0);
        var dt = _timeGrid[itime] - _timeGrid[itime - 1];

             kek.X  =  (point.X  -  _grid.Edges[_grid.Elements[ielem][0]].Point0.X)  /
_grid.Edges[_grid.Elements[ielem][0]].Length;
             kek.Y  =  (point.Y  -  _grid.Edges[_grid.Elements[ielem][2]].Point0.Y)  /
_grid.Edges[_grid.Elements[ielem][2]].Length;
             kek.Z  =  (point.Z  -  _grid.Edges[_grid.Elements[ielem][4]].Point0.Z)  /
_grid.Edges[_grid.Elements[ielem][4]].Length;

        for (int i = 0; i < _basis.Size; i++)
        {
            vector1 += _basis.GetPsi(i, kek) * _solution[_grid.Elements[ielem][i]];
            switch (_activeScheme)
            {
                case Scheme.Two_layer_Implicit:
                   vector2 += _basis.GetPsi(i, kek) * _layers[0][_grid.Elements[ielem]
[i]];
                    break;

                case Scheme.Three_layer_Implicit:
                   vector2 += _basis.GetPsi(i, kek) * _layers[1][_grid.Elements[ielem]
[i]];
                    break;

                case Scheme.Four_layer_Implicit:
                   vector2 += _basis.GetPsi(i, kek) * _layers[2][_grid.Elements[ielem]
[i]];
                    break;
            }
        }

        vector1 -= vector2;
        vector1 /= dt;

        return vector1;
    }

    private double GetValueForRotAz(Point3D point)
    {
        var vector = new Vector3D(0, 0, 0);
```

```
        var kek = new Point3D(0, 0, 0);

        foreach (var elem in _grid.Elements)
        {
                        if  (point.X  >=  _grid.Edges[elem[0]].Point0.X  &&  point.X  <
_grid.Edges[elem[11]].Point1.X &&
                            point.Y  >=  _grid.Edges[elem[0]].Point0.Y  &&  point.Y  <
_grid.Edges[elem[11]].Point1.Y &&
                            point.Z  >=  _grid.Edges[elem[0]].Point0.Z  &&  point.Z  <
_grid.Edges[elem[11]].Point1.Z)
            {
                                kek.X  =  (point.X  -  _grid.Edges[elem[0]].Point0.X)  /
_grid.Edges[elem[0]].Length;
                                kek.Y  =  (point.Y  -  _grid.Edges[elem[2]].Point0.Y)  /
_grid.Edges[elem[2]].Length;
                                kek.Z  =  (point.Z  -  _grid.Edges[elem[4]].Point0.Z)  /
_grid.Edges[elem[4]].Length;

                for (int i = 0; i < _basis.Size; i++)
                {
                    if (i is >= 4 and <= 7)
                    {
                        continue;
                    }

                    vector += _basis.GetDPsi(i, kek) * _solution[elem[i]];
                }

                break;
            }
        }

        return vector.Y - vector.X;
    }

    private Vector3D GetValueForRotAdt(Point3D point, int ielem, int itime)
    {
        var vector1 = new Vector3D(0, 0, 0);
        var vector2 = new Vector3D(0, 0, 0);
        var kek = new Point3D(0, 0, 0);
        var dt = _timeGrid[itime] - _timeGrid[itime - 1];

              kek.X  =  (point.X  -  _grid.Edges[_grid.Elements[ielem][0]].Point0.X)  /
_grid.Edges[_grid.Elements[ielem][0]].Length;
              kek.Y  =  (point.Y  -  _grid.Edges[_grid.Elements[ielem][2]].Point0.Y)  /
_grid.Edges[_grid.Elements[ielem][2]].Length;
              kek.Z  =  (point.Z  -  _grid.Edges[_grid.Elements[ielem][4]].Point0.Z)  /
_grid.Edges[_grid.Elements[ielem][4]].Length;

        for (int i = 0; i < _basis.Size; i++)
        {
            if (i is >= 4 and <= 7)
            {
                continue;
            }
```

```
            vector1 += _basis.GetDPsi(i, kek) * _solution[_grid.Elements[ielem][i]];

            switch (_activeScheme)
            {
                case Scheme.Two_layer_Implicit:
                  vector2 += _basis.GetDPsi(i, kek) * _layers[0][_grid.Elements[ielem]
[i]];
                    break;

                case Scheme.Three_layer_Implicit:
                  vector2 += _basis.GetDPsi(i, kek) * _layers[1][_grid.Elements[ielem]
[i]];
                    break;

                case Scheme.Four_layer_Implicit:
                  vector2 += _basis.GetDPsi(i, kek) * _layers[2][_grid.Elements[ielem]
[i]];
                    break;
            }
        }

        vector1 -= vector2;
        vector1 /= dt;

        return vector1;
    }

    private double CalculatedBz(Point3D point, int itime)
    {
        Vector3D vec1 = new(0.0, 0.0, 0.0);

        for (int ielem = 0; ielem < _grid.Elements.Length; ielem++)
        {
            if (point.X >= _grid.Edges[_grid.Elements[ielem][0]].Point0.X &&
                point.X < _grid.Edges[_grid.Elements[ielem][11]].Point1.X &&
                point.Y >= _grid.Edges[_grid.Elements[ielem][0]].Point0.Y &&
                point.Y < _grid.Edges[_grid.Elements[ielem][11]].Point1.Y &&
                point.Z >= _grid.Edges[_grid.Elements[ielem][0]].Point0.Z &&
                point.Z < _grid.Edges[_grid.Elements[ielem][11]].Point1.Z)
            {
                vec1 = GetValueForRotAdt(point, ielem, itime);

                break;
            }
        }

        //return Axy - Ayx;
        return vec1.Y - vec1.X;
    }

    private List<int> FindElementNumberForGenerator()
    {
        List<int> genNumbers = new List<int>();
        for (int i = 0; i < _grid.Elements.Length; i++)
        {
```

```
                if (!(_grid.Edges[_grid.Elements[i][0]].Point0.X > _generator.xEnd ||
_generator.xStart > _grid.Edges[_grid.Elements[i][0]].Point1.X ||
                      _grid.Edges[_grid.Elements[i][2]].Point0.Y > _generator.yEnd ||
_generator.yStart > _grid.Edges[_grid.Elements[i][2]].Point1.Y) &&
                    _grid.Edges[_grid.Elements[i][0]].Point0.Z <= _generator.zStart &&
_grid.Edges[_grid.Elements[i][11]].Point1.Z >= _generator.zEnd)
            {
                genNumbers.Add(i);
            }
        }

        return genNumbers;
    }
}
```

SLAE.cs

```
namespace VectorFEM3D;
public abstract class SLAE
{
   protected SparseMatrix matrix = default!;
   protected Vector vector = default!;
   public Vector solution = default!;
   public double time;
   protected double eps;
   protected int maxIters;
   public int lastIter;

   public SLAE()
   {
      eps = 1e-16;
      maxIters = 2000;
   }

   public SLAE(double eps, int maxIters)
   {
      this.eps = eps;
      this.maxIters = maxIters;
   }

   public void SetSLAE(Vector vector, SparseMatrix matrix, Vector solution)
   {
      this.vector = vector;
      this.matrix = matrix;
      this.solution = solution;
   }

   public abstract Vector Solve();


   protected void LU()
   {
      for (int i = 0; i < matrix.Size; i++)
      {

         for (int j = matrix.Ig[i]; j < matrix.Ig[i + 1]; j++)
         {
```

```
            int jCol = matrix.Jg[j];
            int jk = matrix.Ig[jCol];
            int k = matrix.Ig[i];

            int sdvig = matrix.Jg[matrix.Ig[i]] - matrix.Jg[matrix.Ig[jCol]];

            if (sdvig > 0)
               jk += sdvig;
            else
               k -= sdvig;

            double sumL = 0.0;
            double sumU = 0.0;

            for (; k < j && jk < matrix.Ig[jCol + 1]; k++, jk++)
            {
               sumL += matrix.Ggl[k] * matrix.Ggu[jk];
               sumU += matrix.Ggu[k] * matrix.Ggl[jk];
            }

            matrix.Ggl[j] -= sumL;
            matrix.Ggu[j] -= sumU;
            matrix.Ggu[j] /= matrix.Di[jCol];
         }

         double sumD = 0.0;
         for (int j = matrix.Ig[i]; j < matrix.Ig[i + 1]; j++)
            sumD += matrix.Ggl[j] * matrix.Ggu[j];

         matrix.Di[i] -= sumD;
      }
   }

   protected void ForwardElimination()
   {
      for (int i = 0; i < matrix.Size; i++)
      {
         for (int j = matrix.Ig[i]; j < matrix.Ig[i + 1]; j++)
         {
            solution[i] -= matrix.Ggl[j] * solution[matrix.Jg[j]];
         }

         solution[i] /= matrix.Di[i];
      }
   }

   protected void BackwardSubstitution()
   {
      for (int i = matrix.Size - 1; i >= 0; i--)
      {
         for (int j = matrix.Ig[i + 1] - 1; j >= matrix.Ig[i]; j--)
         {
            solution[matrix.Jg[j]] -= matrix.Ggu[j] * solution[i];
         }
      }
   }
```

```
    public void PrintSolution()
    {
        for(int i = 0; i < solution.Length; i++)
        {
            Console.WriteLine(solution[i]);
        }
    }
}


public class BCGSTABLUSolver : SLAE
{
    public BCGSTABLUSolver(double eps, int maxIters) : base(eps, maxIters) { }
    public override Vector Solve()
    {
        //solution = new(vector.Length);

        double vecNorm = vector.Norm();

        SparseMatrix matrixLU = new(matrix.Size, matrix.Jg.Length);
        SparseMatrix.Copy(matrix, matrixLU);

        Vector r = new(vector.Length);
        Vector p = new(vector.Length);
        Vector s;
        Vector t;
        Vector v = new(vector.Length);

        double alpha = 1.0;
        double beta;
        double omega = 1.0;
        double rho = 1.0;
        double rhoPrev;

        int i;

        LU(matrixLU);

        Vector r0 = DirElim(matrixLU, vector - matrix * solution);
        Vector.Copy(r0, r);

        for (i = 1; i <= maxIters && r.Norm() / vecNorm > eps; i++)
        {
            Console.WriteLine(r.Norm() / vecNorm);
            rhoPrev = rho;
            rho = (r0 * r);

            beta = rho / rhoPrev * alpha / omega;

            p = r + beta * (p - omega * v);

            v = DirElim(matrixLU, matrix * BackSub(matrixLU, p));

            alpha = rho / (r0 * v);
```

```
            s = r - alpha * v;

            t = DirElim(matrixLU, matrix * BackSub(matrixLU, s));

            omega = (t * s) / (t * t);

            solution = solution + omega * s + alpha * p;

            r = s - omega * t;
        }

        solution = BackSub(matrixLU, solution);

        return solution;
    }

    protected static void LU(SparseMatrix Matrix)
    {
        for (int i = 0; i < Matrix.Size; i++)
        {

            for (int j = Matrix.Ig[i]; j < Matrix.Ig[i + 1]; j++)
            {
                int jCol = Matrix.Jg[j];
                int jk = Matrix.Ig[jCol];
                int k = Matrix.Ig[i];

                int sdvig = Matrix.Jg[Matrix.Ig[i]] - Matrix.Jg[Matrix.Ig[jCol]];

                if (sdvig > 0)
                    jk += sdvig;
                else
                    k -= sdvig;

                double sumL = 0.0;
                double sumU = 0.0;

                for (; k < j && jk < Matrix.Ig[jCol + 1]; k++, jk++)
                {
                    sumL += Matrix.Ggl[k] * Matrix.Ggu[jk];
                    sumU += Matrix.Ggu[k] * Matrix.Ggl[jk];
                }

                Matrix.Ggl[j] -= sumL;
                Matrix.Ggu[j] -= sumU;
                Matrix.Ggu[j] /= Matrix.Di[jCol];
            }

            double sumD = 0.0;
            for (int j = Matrix.Ig[i]; j < Matrix.Ig[i + 1]; j++)
                sumD += Matrix.Ggl[j] * Matrix.Ggu[j];

            Matrix.Di[i] -= sumD;
        }
    }
```

```
    protected static Vector DirElim(SparseMatrix Matrix, Vector b)
    {
        Vector result = new Vector(b.Length);
        Vector.Copy(b, result);

        for (int i = 0; i < Matrix.Size; i++)
        {
            for (int j = Matrix.Ig[i]; j < Matrix.Ig[i + 1]; j++)
            {
                result[i] -= Matrix.Ggl[j] * result[Matrix.Jg[j]];
            }

            result[i] /= Matrix.Di[i];
        }

        return result;
    }

    protected static Vector BackSub(SparseMatrix Matrix, Vector b)
    {
        Vector result = new Vector(b.Length);
        Vector.Copy(b, result);

        for (int i = Matrix.Size - 1; i >= 0; i--)
        {
            for (int j = Matrix.Ig[i + 1] - 1; j >= Matrix.Ig[i]; j--)
            {
                result[Matrix.Jg[j]] -= Matrix.Ggu[j] * result[i];
            }
        }
        return result;
    }
}

public class LUSolver : SLAE
{
    public override Vector Solve()
    {
        solution = new(vector.Length);
        Vector.Copy(vector, solution);
        matrix = matrix.ConvertToProfile();

        LU();
        ForwardElimination();
        BackwardSubstitution();

        return solution;
    }
}

public class BCGSTABSolver : SLAE
{
    public BCGSTABSolver(double eps, int maxIters) : base(eps, maxIters) { }
    public override Vector Solve()
    {
        //solution = new(vector.Length);
```

```
    double vecNorm = vector.Norm();

    //SparseMatrix matrixLU = new(matrix.Size, matrix.Jg.Length);
    //SparseMatrix.Copy(matrix, matrixLU);

    Vector r = new(vector.Length);
    Vector p = new(vector.Length);
    Vector s;
    Vector t;
    Vector v = new(vector.Length);

    double alpha = 1.0;
    double beta;
    double omega = 1.0;
    double rho = 1.0;
    double rhoPrev;

    int i;

    //LU(matrixLU);

    Vector r0 = vector - matrix * solution;
    Vector.Copy(r0, r);

    for (i = 1; i <= maxIters && r.Norm() / vecNorm > eps; i++)
    {
        Console.WriteLine($"{i} {r.Norm() / vecNorm}");
        rhoPrev = rho;


        rho = (r0 * r);

        beta = rho / rhoPrev * alpha / omega;

        p = r + beta * (p - omega * v);

        v = matrix * p;

        alpha = rho / (r0 * v);

        s = r - alpha * v;

        t = matrix * s;

        omega = (t * s) / (t * t);

        solution = solution + omega * s + alpha * p;

        r = s - omega * t;
    }

    return solution;
}

protected static void LU(SparseMatrix Matrix)
```

```
{
   for (int i = 0; i < Matrix.Size; i++)
   {

      for (int j = Matrix.Ig[i]; j < Matrix.Ig[i + 1]; j++)
      {
         int jCol = Matrix.Jg[j];
         int jk = Matrix.Ig[jCol];
         int k = Matrix.Ig[i];

         int sdvig = Matrix.Jg[Matrix.Ig[i]] - Matrix.Jg[Matrix.Ig[jCol]];

         if (sdvig > 0)
            jk += sdvig;
         else
            k -= sdvig;

         double sumL = 0.0;
         double sumU = 0.0;

         for (; k < j && jk < Matrix.Ig[jCol + 1]; k++, jk++)
         {
            sumL += Matrix.Ggl[k] * Matrix.Ggu[jk];
            sumU += Matrix.Ggu[k] * Matrix.Ggl[jk];
         }

         Matrix.Ggl[j] -= sumL;
         Matrix.Ggu[j] -= sumU;
         Matrix.Ggu[j] /= Matrix.Di[jCol];
      }

      double sumD = 0.0;
      for (int j = Matrix.Ig[i]; j < Matrix.Ig[i + 1]; j++)
         sumD += Matrix.Ggl[j] * Matrix.Ggu[j];

      Matrix.Di[i] -= sumD;
   }
}

protected static Vector DirElim(SparseMatrix Matrix, Vector b)
{
   Vector result = new Vector(b.Length);
   Vector.Copy(b, result);

   for (int i = 0; i < Matrix.Size; i++)
   {
      for (int j = Matrix.Ig[i]; j < Matrix.Ig[i + 1]; j++)
      {
         result[i] -= Matrix.Ggl[j] * result[Matrix.Jg[j]];
      }

      result[i] /= Matrix.Di[i];
   }

   return result;
}
```

```
    protected static Vector BackSub(SparseMatrix Matrix, Vector b)
    {
        Vector result = new Vector(b.Length);
        Vector.Copy(b, result);

        for (int i = Matrix.Size - 1; i >= 0; i--)
        {
            for (int j = Matrix.Ig[i + 1] - 1; j >= Matrix.Ig[i]; j--)
            {
                result[Matrix.Jg[j]] -= Matrix.Ggu[j] * result[i];
            }
        }
        return result;
    }
}
```

Grid.cs

```
namespace VectorFEM3D;

public class Grid
{
    private readonly List<double> _xStart = new List<double>();
    private readonly List<double> _xEnd = new List<double>();
    private readonly List<int> _xSteps = new List<int>();
    private readonly List<double> _xRaz = new List<double>();
    private readonly List<double> _yStart = new List<double>();
    private readonly List<double> _yEnd = new List<double>();
    private readonly List<int> _ySteps = new List<int>();
    private readonly List<double> _yRaz = new List<double>();
    private readonly List<double> _zStart = new List<double>();
    private readonly List<double> _zEnd = new List<double>();
    private readonly List<int> _zSteps = new List<int>();
    private readonly List<double> _zRaz = new List<double>();
    private readonly int[] _boundaries;

    private readonly double[] _xZones;
    private readonly double[] _yZones;
    private readonly double[] _zZones;
    private readonly int[][] _zones;
    private readonly double[] _sigmaValues;

    private readonly List<double> _xValues = new List<double>();
    private readonly List<double> _yValues = new List<double>();
    private readonly List<double> _zValues = new List<double>();

    private List<int> _sumSteps;

    private List<List<Point3D>> NodeList = new List<List<Point3D>>();
    private Edge3D[][] EdgeList;
   private List<(int, HashSet<int>)> _identicalPoints = new List<(int, HashSet<int>)>();
    public Point3D[] Nodes { get; private set; }
    public Edge3D[] Edges { get; private set; }
    public HashSet<int> DirichletBoundaries { get; private set; }
     public List<(HashSet<(int, int)>, ElementSide)> NewmanBoundaries { get; private
set; }
```

```
public int[][] Elements { get; private set; }
public double Mu { get; set; }
public double Sigma { get; set; }
public double Epsilon { get; set; }

public Grid(string path)
{
    using (var sr = new StreamReader(path))
    {
        string[] data;
        data = sr.ReadLine()!.Split(" ").ToArray();
        int kek = Convert.ToInt32(data[0]);
        for (int i = 0; i < kek; i++)
        {
            data = sr.ReadLine()!.Split(" ").ToArray();
            _xStart.Add(Convert.ToDouble(data[0]));
            _xEnd.Add(Convert.ToDouble(data[1]));
            _xSteps.Add(Convert.ToInt32(data[2]));
            _xRaz.Add(Convert.ToDouble(data[3]));
        }

    _xZones = sr.ReadLine()!.Split(" ").Select(x => Convert.ToDouble(x)).ToArray();

        data = sr.ReadLine()!.Split(" ").ToArray();
        kek = Convert.ToInt32(data[0]);
        for (int i = 0; i < kek; i++)
        {
            data = sr.ReadLine()!.Split(" ").ToArray();
            _yStart.Add(Convert.ToDouble(data[0]));
            _yEnd.Add(Convert.ToDouble(data[1]));
            _ySteps.Add(Convert.ToInt32(data[2]));
            _yRaz.Add(Convert.ToDouble(data[3]));
        }

    _yZones = sr.ReadLine()!.Split(" ").Select(x => Convert.ToDouble(x)).ToArray();

        data = sr.ReadLine()!.Split(" ").ToArray();
        kek = Convert.ToInt32(data[0]);
        for (int i = 0; i < kek; i++)
        {
            data = sr.ReadLine()!.Split(" ").ToArray();
            _zStart.Add(Convert.ToDouble(data[0]));
            _zEnd.Add(Convert.ToDouble(data[1]));
            _zSteps.Add(Convert.ToInt32(data[2]));
            _zRaz.Add(Convert.ToDouble(data[3]));
        }

    _zZones = sr.ReadLine()!.Split(" ").Select(x => Convert.ToDouble(x)).ToArray();

        data = sr.ReadLine()!.Split(" ").ToArray();
        _boundaries = new int[6];
        _boundaries[0] = Convert.ToInt32(data[0]);
        _boundaries[1] = Convert.ToInt32(data[1]);
        _boundaries[2] = Convert.ToInt32(data[2]);
        _boundaries[3] = Convert.ToInt32(data[3]);
        _boundaries[4] = Convert.ToInt32(data[4]);
```

```
            _boundaries[5] = Convert.ToInt32(data[5]);

            data = sr.ReadLine()!.Split(" ").ToArray();
            Mu = Convert.ToDouble(data[0]);
            //Sigma = Convert.ToDouble(data[1]);
            Epsilon = Convert.ToDouble(data[1]);

            kek = Convert.ToInt32(sr.ReadLine());
            _zones = new int[kek].Select(_ => new int[6]).ToArray();
            _sigmaValues = new double[kek];

            for (int i = 0; i < kek; i++)
            {
                data = sr.ReadLine()!.Split(" ").ToArray();
                _sigmaValues[i] = Convert.ToDouble(data[0]);
                _zones[i][0] = Convert.ToInt32(data[1]);
                _zones[i][1] = Convert.ToInt32(data[2]);
                _zones[i][2] = Convert.ToInt32(data[3]);
                _zones[i][3] = Convert.ToInt32(data[4]);
                _zones[i][4] = Convert.ToInt32(data[5]);
                _zones[i][5] = Convert.ToInt32(data[6]);
            }
        }
    }

    public void BuildGrid()
    {
        _sumSteps = new List<int>();
        int count = 1;
        int nodeCount = 1;
        _sumSteps.Add(0);
        for (int i = 0; i < _xSteps.Count; i++)
        {
            _sumSteps[0] += _xSteps[i];
        }
        count *= _sumSteps[0];
        nodeCount *= _sumSteps[0] + 1;
        int nodesInRow = _sumSteps[0] + 1;

        _sumSteps.Add(0);
        for (int i = 0; i < _ySteps.Count; i++)
        {
            _sumSteps[1] += _ySteps[i];
        }
        count *= _sumSteps[1];
        nodeCount *= _sumSteps[1] + 1;
        int nodesInSlice = nodesInRow * (_sumSteps[1] + 1);

        _sumSteps.Add(0);
        for (int i = 0; i < _zSteps.Count; i++)
        {
            _sumSteps[2] += _zSteps[i];
        }
        count *= _sumSteps[2];
        nodeCount *= _sumSteps[2] + 1;
```

```
        Elements = new int[count].Select(_ => new int[12]).ToArray();
        Nodes = new Point3D[nodeCount];
        //for (int i = 0; i < 8; i++)
        //{
        //    NodeList.Add(new List<Point3D>());
        //}

        List<double> sumRazX = new(), sumRazY = new(), sumRazZ = new();
        for (int i = 0; i < _xSteps.Count; i++)
        {
            sumRazX.Add(0);
            for (int j = 0; j < _xSteps[i]; j++)
            {
                sumRazX[i] += Math.Pow(_xRaz[i], j);
            }
        }

        for (int i = 0; i < _ySteps.Count; i++)
        {
            sumRazY.Add(0);
            for (int j = 0; j < _ySteps[i]; j++)
            {
                sumRazY[i] += Math.Pow(_yRaz[i], j);
            }
        }

        for (int i = 0; i < _zSteps.Count; i++)
        {
            sumRazZ.Add(0);
            for (int j = 0; j < _zSteps[i]; j++)
            {
                sumRazZ[i] += Math.Pow(_zRaz[i], j);
            }
        }

        int xEdges = _sumSteps[0];
        int yEdges = 1 + _sumSteps[0];
        int zEdges = _sumSteps[2] * nodesInSlice;
        int edgesInSlice = xEdges * (1 + _sumSteps[1]) + yEdges * _sumSteps[1];

        Edges = new Edge3D[edgesInSlice * (_sumSteps[2] + 1) + zEdges];
        //EdgeList = new Edge3D[8].Select(_ => new Edge3D[edgesInSlice * (_zSteps + 1)
+ zEdges]).ToArray();

        DirichletBoundaries = new();

        double x = 0, y = 0, z = 0;
        double xStep = 0, yStep = 0, zStep = 0;
        //double xStep = (_xEnd - _xStart) / sumRazX;
        //double yStep = (_yEnd - _yStart) / sumRazY;
        //double zStep = (_zEnd - _zStart) / sumRazZ;

        for (int i = 0; i < _xSteps.Count; i++)
        {
            x = _xStart[i];
            xStep = (_xEnd[i] - _xStart[i]) / sumRazX[i];
```

```
            for (int j = 0; j < _xSteps[i]; j++)
            {
                _xValues.Add(x);
                x += xStep;
                xStep *= _xRaz[i];
            }

        }
        _xValues.Add(_xEnd[^1]);

        for (int i = 0; i < _ySteps.Count; i++)
        {
            y = _yStart[i];
            yStep = (_yEnd[i] - _yStart[i]) / sumRazY[i];

            for (int j = 0; j < _ySteps[i]; j++)
            {
                _yValues.Add(y);
                y += yStep;
                yStep *= _yRaz[i];
            }

        }
        _yValues.Add(_yEnd[^1]);

        for (int i = 0; i < _zSteps.Count; i++)
        {
            z = _zStart[i];
            zStep = (_zEnd[i] - _zStart[i]) / sumRazZ[i];

            for (int j = 0; j < _zSteps[i]; j++)
            {
                _zValues.Add(z);
                z += zStep;
                zStep *= _zRaz[i];
            }

        }
        _zValues.Add(_zEnd[^1]);

        for (int i = 0; i < _zValues.Count; i++)
        {
            for (int j = 0; j < _yValues.Count; j++)
            {
                for (int k = 0; k < _xValues.Count; k++)
                {
                 Nodes[i * nodesInSlice + j * nodesInRow + k] = new Point3D(_xValues[k],
_yValues[j], _zValues[i]);
                }
            }
        }

        int index = 0;

        for (int j = 0; j < _sumSteps[2] + 1; j++)
```

```
        {
            int xLocal = 0;
            int yLocal = 0;
            int zLocal = 0;


            for (int i = 0; i < nodesInSlice / nodesInRow; i++)
            {
                for (int k = 0; k < nodesInRow - 1; k++)
                {
                        Edges[index++] = new Edge3D(Nodes[xLocal + nodesInSlice * j],
Nodes[xLocal + nodesInSlice * j + 1]);
                    xLocal++;
                }

                xLocal++;

                if (i != nodesInSlice / nodesInRow - 1)
                {
                    for (int k = 0; k < nodesInRow; k++)
                    {
                        Edges[index++] = new Edge3D(Nodes[yLocal + nodesInSlice * j],
                            Nodes[yLocal + nodesInSlice * j + nodesInRow]);
                        yLocal++;
                    }
                }
            }

            if (j != _sumSteps[2])
            {
                for (int k = 0; k < nodesInSlice; k++)
                {
                    Edges[index++] = new Edge3D(Nodes[zLocal + nodesInSlice * j],
                        Nodes[zLocal + nodesInSlice * j + nodesInSlice]);
                    zLocal++;
                }
            }
        }

        index = 0;

        for (int k = 0; k < _sumSteps[2]; k++)
        {
            for (int i = 0; i < _sumSteps[1]; i++)
            {
                for (int j = 0; j < _sumSteps[0]; j++)
                {
                    // x
                  Elements[index][0] = j + (nodesInSlice + edgesInSlice) * k + (xEdges
+ yEdges) * i;
                    Elements[index][1] = j + (nodesInSlice + edgesInSlice) * k + (xEdges
+ yEdges) * (i + 1);
                    Elements[index][2] = j + (nodesInSlice + edgesInSlice) * k + (xEdges
+ yEdges) * i + xEdges;
                    Elements[index][3] = j + (nodesInSlice + edgesInSlice) * k + (xEdges
+ yEdges) * i + xEdges + 1;
```

```
                    Elements[index][4] =
                        j + (nodesInSlice + edgesInSlice) * k + (xEdges + yEdges) * i
+ edgesInSlice - _sumSteps[0] * i;
                    Elements[index][5] =
                        j + (nodesInSlice + edgesInSlice) * k + (xEdges + yEdges) * i
+ edgesInSlice + 1 - _sumSteps[0] * i;
                    Elements[index][6] =
                        j + (nodesInSlice + edgesInSlice) * k + (xEdges + yEdges) * i
+ edgesInSlice + nodesInRow -
                        _sumSteps[0] * i;
                    Elements[index][7] =
                        j + (nodesInSlice + edgesInSlice) * k + (xEdges + yEdges) * i
+ edgesInSlice + 1 +
                        nodesInRow - _sumSteps[0] * i;

                    Elements[index][8] = j + (nodesInSlice + edgesInSlice) * (k + 1) +
(xEdges + yEdges) * i;
                    Elements[index][9] = j + (nodesInSlice + edgesInSlice) * (k + 1) +
(xEdges + yEdges) * (i + 1);
                     Elements[index][10] = j + (nodesInSlice + edgesInSlice) * (k + 1)
+ (xEdges + yEdges) * i + xEdges;
                      Elements[index++][11] = j + (nodesInSlice + edgesInSlice) * (k +
1) + (xEdges + yEdges) * i + xEdges + 1;
                }
            }
        }
    }

    public double GetSigma(Point3D point)
    {
        for (int i = 0; i < _zones.Length; i++)
        {
             if (point.X <= _xZones[_zones[i][1]] && point.Y <= _yZones[_zones[i][3]]
&& point.Z <= _zZones[_zones[i][5]] &&
                 point.X > _xZones[_zones[i][0]] && point.Y > _yZones[_zones[i][2]] &&
point.Z > _zZones[_zones[i][4]])
            {
                return _sigmaValues[i];
            }
        }

        throw new Exception("Can't find eligible zone for sigma");
    }

    public void AccountBoundaryConditions()
    {
        for (int ielem = 0; ielem < Elements.Length; ielem++)
        {
            if (ielem < _sumSteps[0] * _sumSteps[1])
            {
                if (_boundaries[2] == 1) DirichletBoundary(ElementSide.Bottom, ielem);
            }

             if (ielem >= _sumSteps[0] * _sumSteps[1] * _sumSteps[2] - _sumSteps[0] *
_sumSteps[1] || _sumSteps[2] == 1)
```

```
            {
                if (_boundaries[3] == 1) DirichletBoundary(ElementSide.Upper, ielem);
            }

            if (ielem % _sumSteps[0] == 0)
            {
                if (_boundaries[0] == 1) DirichletBoundary(ElementSide.Left, ielem);
            }

            if ((ielem + 1) % _sumSteps[0] == 0)
            {
                if (_boundaries[1] == 1) DirichletBoundary(ElementSide.Right, ielem);
            }

            if (ielem % (_sumSteps[0] * _sumSteps[1]) < _sumSteps[0])
            {
                if (_boundaries[5] == 1) DirichletBoundary(ElementSide.Front, ielem);
            }

            if (ielem % (_sumSteps[0] * _sumSteps[1]) >= _sumSteps[0] * _sumSteps[1] -
_sumSteps[0])
            {
                if (_boundaries[4] == 1) DirichletBoundary(ElementSide.Rear, ielem);
            }
        }
    }

    private void DirichletBoundary(ElementSide elementSide, int ielem)
    {
        switch (elementSide)
        {
            case ElementSide.Bottom:
                DirichletBoundaries.Add(Elements[ielem][0]);
                DirichletBoundaries.Add(Elements[ielem][1]);
                DirichletBoundaries.Add(Elements[ielem][2]);
                DirichletBoundaries.Add(Elements[ielem][3]);
                break;

            case ElementSide.Upper:
                DirichletBoundaries.Add(Elements[ielem][8]);
                DirichletBoundaries.Add(Elements[ielem][9]);
                DirichletBoundaries.Add(Elements[ielem][10]);
                DirichletBoundaries.Add(Elements[ielem][11]);
                break;

            case ElementSide.Left:
                DirichletBoundaries.Add(Elements[ielem][2]);
                DirichletBoundaries.Add(Elements[ielem][4]);
                DirichletBoundaries.Add(Elements[ielem][6]);
                DirichletBoundaries.Add(Elements[ielem][10]);
                break;

            case ElementSide.Right:
                DirichletBoundaries.Add(Elements[ielem][3]);
                DirichletBoundaries.Add(Elements[ielem][5]);
                DirichletBoundaries.Add(Elements[ielem][7]);
```

```
                DirichletBoundaries.Add(Elements[ielem][11]);
                break;

            case ElementSide.Front:
                DirichletBoundaries.Add(Elements[ielem][0]);
                DirichletBoundaries.Add(Elements[ielem][4]);
                DirichletBoundaries.Add(Elements[ielem][5]);
                DirichletBoundaries.Add(Elements[ielem][8]);
                break;

            case ElementSide.Rear:
                DirichletBoundaries.Add(Elements[ielem][1]);
                DirichletBoundaries.Add(Elements[ielem][6]);
                DirichletBoundaries.Add(Elements[ielem][7]);
                DirichletBoundaries.Add(Elements[ielem][9]);
                break;
        }
    }
}

public interface ITimeGrid
{
    public double[] TGrid { get; set;  }

    public double this[int index]
    {
        get => TGrid[index];
        set => TGrid[index] = value;
    }
}

public class TimeGrid : ITimeGrid
{
    private readonly double _tStart;
    private readonly double _tEnd;
    private readonly int _tSteps;
    private readonly double _tRaz;
    public double[] TGrid { get; set;  }

    public TimeGrid(string path)
    {
        using (var sr = new StreamReader(path))
        {
            string[] data;
            data = sr.ReadLine()!.Split(" ").ToArray();
            _tStart = Convert.ToDouble(data[0]);
            _tEnd = Convert.ToDouble(data[1]);
            _tSteps = Convert.ToInt32(data[2]);
            _tRaz = Convert.ToDouble(data[3]);
            TGrid = new double[_tSteps + 1];
        }
    }

    public double this[int index]
    {
        get => TGrid[index];
```

```
        set => TGrid[index] = value;
    }

    public void BuildTimeGrid()
    {
        double sumRaz = 0;
        for (int i = 0; i < _tSteps; i++)
            sumRaz += Math.Pow(_tRaz, i);

        double t = _tStart;
        double tStep = (_tEnd - _tStart) / sumRaz;

        for (int i = 0; i < _tSteps; i++)
        {
            TGrid[i] = t;
            t += tStep;
            tStep *= _tRaz;
        }

        TGrid[_tSteps] = _tEnd;
    }
}

public class GeneratedTimeGrid : ITimeGrid
{
    public double[] TGrid { get; set;  }

    public GeneratedTimeGrid(string path)
    {
        using (var sr = new StreamReader(path))
        {
            string[] data;
          data = sr.ReadToEnd()!.Split("\r\n", StringSplitOptions.RemoveEmptyEntries);
            TGrid = new double[(data.Length - 1) / 2];
            TGrid = data.Select(Convert.ToDouble).ToArray();
        }
    }

    public double this[int index]
    {
        get => TGrid[index];
        set => TGrid[index] = value;
    }

}
```

Basis.cs

```
namespace VectorFEM3D;

public interface IBasis3D
{
    int Size { get; }
    Vector3D GetPsi(int number, Point3D point);
    Vector3D GetDPsi(int number, Point3D point);
}
```

```
public readonly record struct TriLinearVectorBasis : IBasis3D
{
    public int Size => 12;
    private readonly Vector3D _vector = new Vector3D(0, 0, 0);

    public TriLinearVectorBasis() { }

    public Vector3D GetPsi(int number, Point3D point)
        => number switch
        {
            0 => _vector.UpdateVector(GetXi(0, point.Y) * GetXi(0, point.Z), 0, 0),
            1 => _vector.UpdateVector(GetXi(1, point.Y) * GetXi(0, point.Z), 0, 0),
            2 => _vector.UpdateVector(0, GetXi(0, point.X) * GetXi(0, point.Z), 0),
            3 => _vector.UpdateVector(0, GetXi(1, point.X) * GetXi(0, point.Z), 0),
            4 => _vector.UpdateVector(0, 0, GetXi(0, point.X) * GetXi(0, point.Y)),
            5 => _vector.UpdateVector(0, 0, GetXi(1, point.X) * GetXi(0, point.Y)),
            6 => _vector.UpdateVector(0, 0, GetXi(0, point.X) * GetXi(1, point.Y)),
            7 => _vector.UpdateVector(0, 0, GetXi(1, point.X) * GetXi(1, point.Y)),
            8 => _vector.UpdateVector(GetXi(0, point.Y) * GetXi(1, point.Z), 0, 0),
            9 => _vector.UpdateVector(GetXi(1, point.Y) * GetXi(1, point.Z), 0, 0),
            10 => _vector.UpdateVector(0, GetXi(0, point.X) * GetXi(1, point.Z), 0),
            11 => _vector.UpdateVector(0, GetXi(1, point.X) * GetXi(1, point.Z), 0),
              _ => throw new ArgumentOutOfRangeException(nameof(number), number, "Not
expected function number")
        };

    public Vector3D GetDPsi(int number, Point3D point)
        => number switch
        {
            0 => _vector.UpdateVector(0, -GetXi(0, point.Y), GetXi(0, point.Z)),
            1 => _vector.UpdateVector(0, -GetXi(1, point.Y), -GetXi(0, point.Z)),
            2 => _vector.UpdateVector(GetXi(0, point.X), 0, -GetXi(0, point.Z)),
            3 => _vector.UpdateVector(GetXi(1, point.X), 0, GetXi(0, point.Z)),
            4 => _vector.UpdateVector(-GetXi(0, point.X), GetXi(0, point.Y), 0),
            5 => _vector.UpdateVector(-GetXi(1, point.X), -GetXi(0, point.Y), 0),
            6 => _vector.UpdateVector(GetXi(0, point.X), GetXi(1, point.Y), 0),
            7 => _vector.UpdateVector(GetXi(1, point.X), -GetXi(1, point.Y), 0),
            8 => _vector.UpdateVector(0,  GetXi(0, point.Y), GetXi(1, point.Z)),
            9 => _vector.UpdateVector(0, GetXi(1, point.Y), -GetXi(1, point.Z)),
            10 => _vector.UpdateVector(-GetXi(0, point.X), 0, -GetXi(1, point.Z)),
            11 => _vector.UpdateVector(-GetXi(1, point.X), 0, GetXi(1, point.Z)),
              _ => throw new ArgumentOutOfRangeException(nameof(number), number, "Not
expected function number")
        };

    private double GetXi(int number, double value)
        => number switch
        {
            0 => 1 - value,
            1 => value,
              _ => throw new ArgumentOutOfRangeException(nameof(number), number, "Not
expected Xi member")
        };
}
```

Integration.cs

```
namespace VectorFEM3D;

public class Integration
{
    private readonly SegmentGaussOrder9 _quadratures;

    public Integration(SegmentGaussOrder9 quadratures)
    {
        _quadratures = quadratures;
    }

    public double Gauss3D(Func<Point3D, double> psi)
    {
        double result = 0;
        Point3D point = new(0, 0, 0);

        for (int i = 0; i < _quadratures.Size; i++)
        {
            point.X = (_quadratures.GetPoint(i) + 1) / 2.0;

            for (int j = 0; j < _quadratures.Size; j++)
            {
                point.Y = (_quadratures.GetPoint(j) + 1) / 2.0;

                for (int k = 0; k < _quadratures.Size; k++)
                {
                    point.Z = (_quadratures.GetPoint(k) + 1) / 2.0;

                                result += psi(point) * _quadratures.GetWeight(i) *
_quadratures.GetWeight(j) *
                                _quadratures.GetWeight(k);
                }
            }
        }

        return result / 8.0;
    }
}
```

QuadratureNode.cs

```
namespace VectorFEM3D;

public interface IQuadrature
{
    int Size { get; }
    double GetPoint(int number);
    double GetWeight(int number);
}

public readonly record struct SegmentGaussOrder9 : IQuadrature
{
    public int Size => 5;

    public SegmentGaussOrder9() { }
```

```
    public double GetPoint(int number)
        => number switch
        {
            0 => 0.0,
            1 => 1.0 / 3.0 * Math.Sqrt(5 - 2 * Math.Sqrt(10.0 / 7.0)),
            2 => -1.0 / 3.0 * Math.Sqrt(5 - 2 * Math.Sqrt(10.0 / 7.0)),
            3 => 1.0 / 3.0 * Math.Sqrt(5 + 2 * Math.Sqrt(10.0 / 7.0)),
            4 => -1.0 / 3.0 * Math.Sqrt(5 + 2 * Math.Sqrt(10.0 / 7.0)),
              _ => throw new ArgumentOutOfRangeException(nameof(number), number, "Not
expected point number")
        };

    public double GetWeight(int number)
        => number switch
        {
            0 => 128.0 / 225.0,
            1 => (322.0 + 13.0 * Math.Sqrt(70.0)) / 900.0,
            2 => (322.0 + 13.0 * Math.Sqrt(70.0)) / 900.0,
            3 => (322.0 - 13.0 * Math.Sqrt(70.0)) / 900.0,
            4 => (322.0 - 13.0 * Math.Sqrt(70.0)) / 900.0,
              _ => throw new ArgumentOutOfRangeException(nameof(number), number, "Not
expected weight number")
        };
}
```