

# СОДЕРЖАНИЕ

<b>1. Математическая модель . . . . .</b>	<b>3</b>
1.1. Система уравнений Максвелла . . . . .	3
1.2. Вывод ротор-роторного уравнения . . . . .	4
1.3. Применение технологии выделения поля . . . . .	5
1.4. Задачи магнитотеллурических зондирований . . . . .	6
1.5. Вариационная постановка в форме уравнения Галеркина . . . . .	6
<b>2. Построение дискретного аналога . . . . .</b>	<b>7</b>
2.1. Конечноэлементная дискретизация . . . . .	7
2.2. Локальные матрицы и векторы прямоугольных EDGE-элементов	8
<b>3. Построение конечноэлементной сетки из прямоугольников с неравномерным шагом в горизонтально-слоистой среде с неоднородностями . . . . .</b>	<b>10</b>
3.1. Условные обозначения. Входные данные . . . . .	10
3.2. Выходные данные . . . . .	11
3.3. Алгоритм построения сетки . . . . .	14
3.4. Учет горизонтальных слоев . . . . .	18
3.5. Руководство по программе . . . . .	20
3.5.1. Используемые при разработке средства . . . . .	20
3.5.2. Описание пользовательского интерфейса . . . . .	20
3.5.3. Взаимодействие с Plot из пакета ScottPlot . . . . .	24
<b>4. Процедура решения СЛАУ с использованием библиотеки Intel OneMKL . . . . .</b>	<b>27</b>
4.1. Введение в Intel oneAPI Math Kernel Library . . . . .	27
4.2. Используемые при разработке средства . . . . .	27

4.3. Разреженный формат хранения матрицы CSR3 . . . . .	28
<b>5. Программный модуль для построения портрета матрицы .</b>	<b>30</b>
5.1. Руководство по плагину . . . . .	30
5.1.1. Входные и выходные данные . . . . .	30
5.1.2. Используемые при разработке средства . . . . .	31
<b>6. Результаты вычислительных экспериментов . . . . .</b>	<b>32</b>
6.1. Тестирование на линейных функциях . . . . .	32
6.1.1. Расчетная область . . . . .	32
6.1.2. Полином первой степени . . . . .	32
6.1.3. Полином второй степени . . . . .	33
6.1.4. Полином третьей степени . . . . .	33
6.1.5. Полином четвертой степени . . . . .	33
6.2. Оценка порядка аппроксимации . . . . .	33
6.3. Модель 2Д-1 из проекта СОММЕМІ . . . . .	34
6.4. Применение метода деревьев-кодереьев . . . . .	39
<b>СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ . . . . .</b>	<b>44</b>
<b>ПРИЛОЖЕНИЕ 1. Реализованные для задачи структуры . .</b>	<b>45</b>
<b>ПРИЛОЖЕНИЕ 2. Составление портрета и глобальной мат-</b>	
<b>рицы . . . . .</b>	<b>48</b>
<b>ПРИЛОЖЕНИЕ 3. Генерация сетки . . . . .</b>	<b>55</b>

# 1. Математическая модель

## 1.1. Система уравнений Максвелла

Уравнения Максвелла описывают связь между электромагнитным полем и зарядами и токами в сплошных средах. Система уравнений может быть представлена как в интегральной, так и в дифференциальной форме и является фундаментальной математической моделью для описания таких явлений, как электромагнитные волны, электромагнитная индукция, распространение света и многие другие. [5, 7]

Уравнения Максвелла представляют собой в векторной записи систему из четырёх фундаментальных математических выражений [11, стр. 26]:

$$\operatorname{rot} \vec{E} = -\frac{\partial \vec{B}}{\partial t}, \quad (1.1)$$

$$\operatorname{div} \vec{B} = 0, \quad (1.2)$$

$$\operatorname{rot} \vec{H} = \vec{J}^{\text{ст}} + \sigma \vec{E} + \frac{\partial(\varepsilon \vec{E})}{\partial t}, \quad (1.3)$$

$$\operatorname{div}(\varepsilon \vec{E}) = \rho, \quad (1.4)$$

где  $\vec{E}$  - напряженность электрического поля (В/м),

$\vec{B}$  - магнитная индукция (Тл),

$\vec{H}$  - напряжённость магнитного поля (А/м),

$t$  - время,

$\sigma$  - удельная электрическая проводимость среды (См/м),

$\varepsilon$  - диэлектрическая проницаемость среды (Ф/м),

$\vec{J}^{\text{ст}}$  - вектор плотностей сторонних токов (А/м<sup>2</sup>),

$\rho$  - объёмная плотность стороннего электрического заряда (Кл/м<sup>3</sup>).

## 1.2. Вывод ротор-роторного уравнения

Система уравнений (1.1)-(1.3) может быть преобразована к виду:

$$\operatorname{rot} \left( \frac{1}{\mu} \operatorname{rot} \vec{A} \right) + \sigma \frac{\partial \vec{A}}{\partial t} + \varepsilon \frac{\partial^2 \vec{A}}{\partial t^2} = \vec{J}^{\text{ст}}, \quad (1.5)$$

где  $\vec{A}$  – вектор-потенциал, через который определяются индукция магнитного поля  $\vec{B}$  и напряжённость электрического поля  $\vec{E}$  в виде:

$$\vec{B} = \operatorname{rot} \vec{A}, \quad (1.6)$$

$$\vec{E} = -\frac{\partial \vec{A}}{\partial t}. \quad (1.7)$$

А после подстановки (1.6) и (1.7) в уравнение (1.3) оно превращается в уравнение (1.5).

Магнитная индукция  $\vec{B}$  связана с напряженностью магнитного поля  $\vec{H}$  соотношением:

$$\vec{B} = \mu \vec{H}, \quad (1.8)$$

где  $\mu$  - магнитная проницаемость среды (Гн/м).

Когда токи смещения являются пренебрежимо малыми, т.е. можно считать, что  $\partial(\varepsilon \vec{A})/\partial t = 0$ , тогда уравнение (1.5) принимает вид:

$$\operatorname{rot} \left( \frac{1}{\mu} \operatorname{rot} \vec{A} \right) + \sigma \frac{\partial \vec{A}}{\partial t} = \vec{J}^{\text{ст}}. \quad (1.9)$$

Если источник является гармоническим по времени, тогда соотношение 1.7 можно записать следующим образом:

$$\vec{E} = -i\omega \vec{A}, \quad (1.10)$$

где  $i$  - мнимая единица,

$\omega = 2\pi\nu$  - круговая частота,

$\nu$  - частота электромагнитного поля.

И уравнение (1.9) примет следующий вид:

$$\operatorname{rot} \left( \frac{1}{\mu} \operatorname{rot} \vec{A} \right) + i\sigma\omega \vec{A} = \vec{J}^{\rightarrow \text{ст}}. \quad (1.11)$$

### 1.3. Применение технологии выделения поля

Для многих практических задач, которые описываются сложными математическими моделями, необходимо большое количество вычислительных ресурсов для приближенного решения. Можно сократить вычислительные затраты и получить достаточно точное решение, заменив сложную модель на более простую [11, стр. 418].

Рассмотрим краевую задачу (1.11) в области  $\Omega$ :

$$\frac{1}{\mu} \operatorname{rot} \operatorname{rot} \vec{A} + i\sigma\omega \vec{A} = \vec{J}^{\rightarrow \text{ст}}. \quad (1.12)$$

Будем считать, что область  $\Omega$  полностью включается в область  $\Omega'$  (или совпадает с ней), и в области  $\Omega'$  определена вектор-функция  $\vec{A}^{\rightarrow 1D}$ , являющаяся решением краевой задачи:

$$\frac{1}{\mu} \operatorname{rot} \operatorname{rot} \vec{A}^{\rightarrow 1D} + i\sigma^{1D}\omega \vec{A}^{\rightarrow 1D} = \vec{J}^{\rightarrow \text{ст}}. \quad (1.13)$$

Вектор-функция  $\vec{A}^{\rightarrow 1D}$  получена с гораздо меньшими вычислительными затратами численно, как решение задачи меньшей размерности, и при этом она достаточно хорошо приближает искомое решение  $\vec{A}$  задачи (1.12). Тогда решение  $\vec{A}$  исходной задачи (1.12) будем искать в виде суммы решений  $\vec{A}^{\rightarrow 1D}$ , как решение задачи (1.13) и задачи на «добавочное» поле  $\vec{A}^{\rightarrow a}$ :

$$\begin{aligned} \frac{1}{\mu} \operatorname{rot} \operatorname{rot} \left( \vec{A}^{\rightarrow 1D} + \vec{A}^{\rightarrow a} \right) + i\sigma\omega \left( \vec{A}^{\rightarrow 1D} + \vec{A}^{\rightarrow a} \right) = \\ \vec{J}^{\rightarrow \text{ст}} - \vec{J}^{\rightarrow \text{ст}} + \frac{1}{\mu} \operatorname{rot} \operatorname{rot} \vec{A}^{\rightarrow 1D} + i\sigma^{1D}\omega \vec{A}^{\rightarrow 1D}. \end{aligned} \quad (1.14)$$

Упростим выражение (1.14):

$$\frac{1}{\mu} \operatorname{rot} \operatorname{rot} \vec{A}^{\rightarrow a} + i\omega\sigma \vec{A}^{\rightarrow a} = -i\omega(\sigma - \sigma^{1D}) \vec{A}^{\rightarrow 1D}. \quad (1.15)$$

## 1.4. Задачи магнитотеллурических зондирований

Магнитотеллурическое зондирование - один из методов геофизического исследования глубинных структур Земли. Метод магнитотеллурического зондирования (МТЗ) основан на использовании естественного переменного магнитного поля Земли [6, 10].

Методика МТЗ сводится к длительным измерениям компонент электрического  $(E_x, E_y)$  и магнитного  $(H_x, H_y, H_z)$  полей. Расчеты этих полей позволяют оценить величину кажущегося сопротивления грунта  $(\rho_k)$ , что дает возможность получить информацию о структуре и параметрах геологических объектов на глубине. Эта информация может использоваться для решения различных задач, таких как поиск залежей полезных ископаемых и прочее.

## 1.5. Вариационная постановка в форме уравнения Галеркина

Запишем эквивалентную вариационную формулировку в форме Галеркина для уравнения (1.15) с однородными краевыми условиями 1-го рода:

$$\begin{aligned} \frac{1}{\mu} \int_{\Omega} \operatorname{rot} \vec{A}^{\rightarrow a} \cdot \operatorname{rot} \vec{\Psi} d\Omega + i\sigma\omega \int_{\Omega} \vec{A}^{\rightarrow a} \cdot \vec{\Psi} d\Omega = \\ = -i\omega(\sigma - \sigma^{1D}) \int_{\Omega} \vec{A}^{\rightarrow 1D} \cdot \vec{\Psi} d\Omega \end{aligned} \quad (1.16)$$

Уравнение (1.16) было получено в результате скалярного умножения (1.15) на вектор-функцию  $\vec{\Psi} \in \mathbb{H}_0(\operatorname{rot}, \Omega)$  вектор-функций  $\vec{\Phi}$ , таких что  $\int_{\Omega} \vec{\Phi} \cdot \vec{\Phi} d\Omega < \infty$ ,  $\int_{\Omega} \operatorname{rot} \vec{\Phi} \cdot \operatorname{rot} \vec{\Phi} d\Omega < \infty$ ,  $\left[ \vec{\Phi} \times \vec{n} \right] \Big|_{\partial\Omega} = 0$  и применения векторной формулы Грина.

## 2. Построение дискретного аналога

### 2.1. Конечноэлементная дискретизация

Конечноэлементная дискретизация уравнения (1.16) выполняется на сетке с прямоугольными ячейками с векторными базисными функциями из пространства  $\mathbb{H}_0(\text{rot}, \Omega)$  [11, стр. 668].

Рассмотрим прямоугольный конечный элемент  $\Omega_{rs} = [x_r, x_{r+1}] \times [y_s, y_{s+1}]$ , изображённый на рисунке 2.1. Определим на нем четыре локальные базисные вектор-функции:

$$\begin{aligned}\hat{\vec{\psi}}_1 &= \begin{pmatrix} 0 \\ \frac{x_{r+1}-x}{h_x} \end{pmatrix}, & \hat{\vec{\psi}}_2 &= \begin{pmatrix} 0 \\ \frac{x-x_r}{h_x} \end{pmatrix}, \\ \hat{\vec{\psi}}_3 &= \begin{pmatrix} \frac{y_{s+1}-y}{h_y} \\ 0 \end{pmatrix}, & \hat{\vec{\psi}}_4 &= \begin{pmatrix} \frac{y-y_s}{h_y} \\ 0 \end{pmatrix}.\end{aligned}\tag{2.1}$$

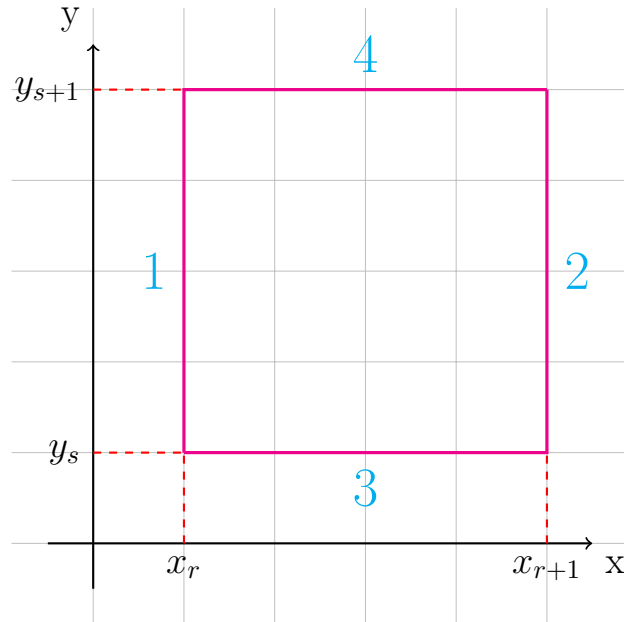


Рисунок 2.1 – Локальная нумерация рёбер и базисных вектор-функций на прямоугольном конечном элементе

Каждая из базисных вектор-функций 2.1 только на одном ребре прямоугольника  $\Omega_{rs}$  имеет ненулевую касательную составляющую:

$$\begin{aligned} \hat{\vec{\psi}}_1 &- \text{ на ребре 1,} \\ \hat{\vec{\psi}}_2 &- \text{ на ребре 2,} \\ \hat{\vec{\psi}}_3 &- \text{ на ребре 3,} \\ \hat{\vec{\psi}}_4 &- \text{ на ребре 4.} \end{aligned}$$

## 2.2. Локальные матрицы и векторы прямоугольных EDGE-элементов

Получим формулы для вычисления локальных матриц прямоугольного конечного элемента  $\Omega_{rs} = [x_r, x_{r+1}] \times [y_s, y_{s+1}]$  с базисом 2.1.

Сначала вычислим роторы локальных базисных вектор-функций на  $\Omega_{rs}$ :

$$\begin{aligned} \text{rot}_z \hat{\vec{\psi}}_1 &= \frac{\partial}{\partial x} \frac{x_{r+1} - x}{h_x} = -\frac{1}{h_x}, & \text{rot}_z \hat{\vec{\psi}}_2 &= \frac{\partial}{\partial x} \frac{x - x_r}{h_x} = \frac{1}{h_x}, \\ \text{rot}_z \hat{\vec{\psi}}_3 &= -\frac{\partial}{\partial y} \frac{y_{s+1} - y}{h_y} = \frac{1}{h_y}, & \text{rot}_z \hat{\vec{\psi}}_4 &= -\frac{\partial}{\partial y} \frac{y - y_s}{h_y} = -\frac{1}{h_y}. \end{aligned} \quad (2.2)$$

Тогда локальная матрица жёсткости этого элемента имеет вид:

$$G = \frac{1}{\mu_0} \begin{pmatrix} \frac{h_y}{h_x} & -\frac{h_y}{h_x} & -1 & 1 \\ -\frac{h_y}{h_x} & \frac{h_y}{h_x} & 1 & -1 \\ -1 & 1 & \frac{h_x}{h_y} & -\frac{h_x}{h_y} \\ 1 & -1 & -\frac{h_x}{h_y} & \frac{h_x}{h_y} \end{pmatrix}. \quad (2.3)$$

Локальная матрицы массы выглядит следующим образом:

$$M = \sigma\omega \frac{h_x h_y}{6} \begin{pmatrix} 2 & 1 & 0 & 0 \\ 1 & 2 & 0 & 0 \\ 0 & 0 & 2 & 1 \\ 0 & 0 & 1 & 2 \end{pmatrix}. \quad (2.4)$$



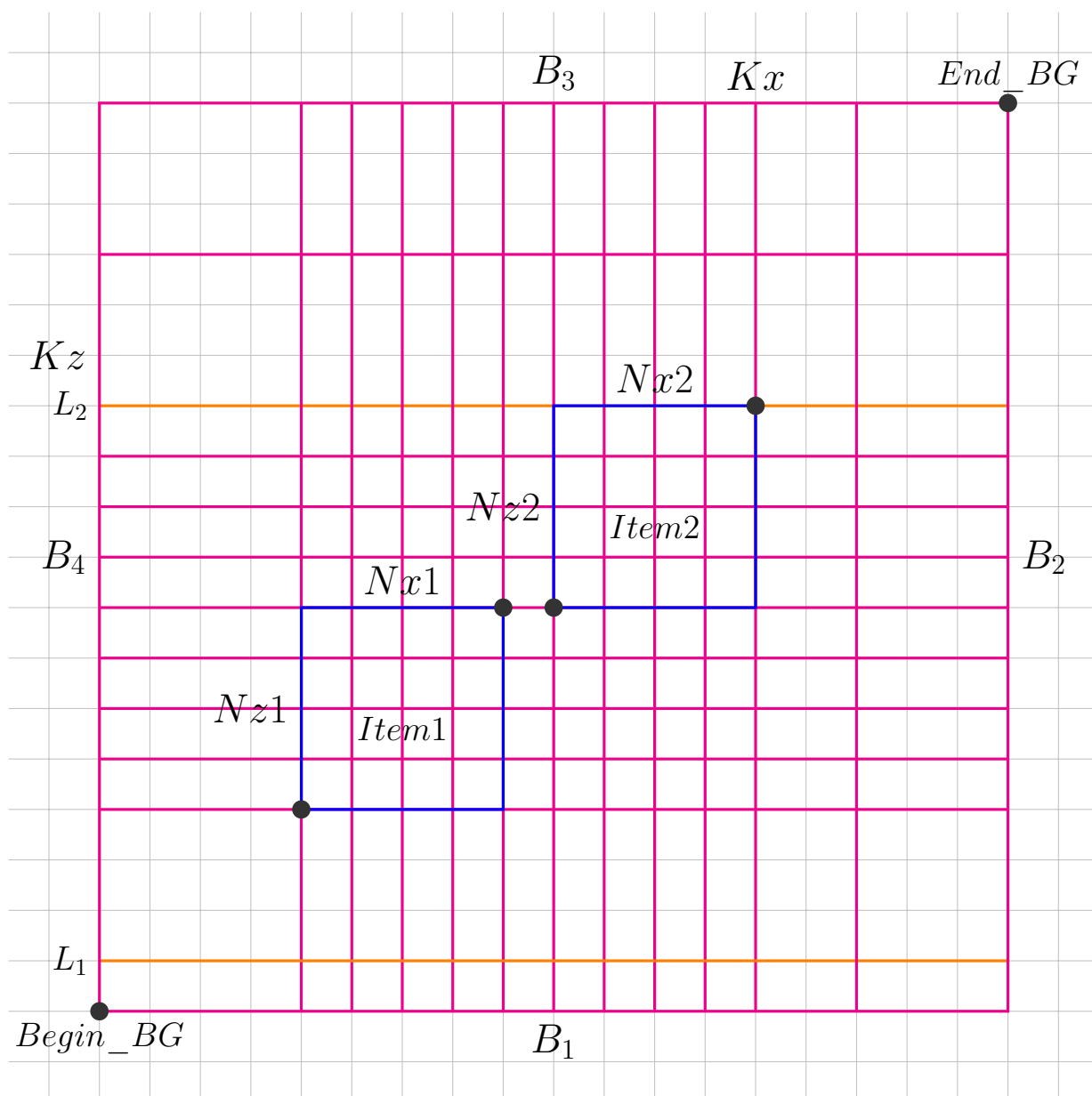
Локальный вектор правой части  $b$  может быть вычислен с помощью локальной матрицы  $M^1$ :

$$b = M^1 \begin{pmatrix} J_y \left( x_r, y_{s+\frac{1}{2}} \right) \\ J_y \left( x_{r+1}, y_{s+\frac{1}{2}} \right) \\ J_x \left( x_{r+\frac{1}{2}}, y_s \right) \\ J_x \left( x_{r+\frac{1}{2}}, y_{s+1} \right) \end{pmatrix}, \quad (2.5)$$

где  $M^1$  матрица массы, определяемая соотношением 2.4 при значении коэффициентов  $\sigma$  и  $\omega$ , равными единице.

### 3. Построение конечноэлементной сетки из прямоугольников с неравномерным шагом в горизонтально-слоистой среде с неоднородностями

#### 3.1. Условные обозначения. Входные данные



Данные для сетки вводятся пользователем через оконный интерфейс.

**Begin\_BG** и **End\_BG** - точки начала и конца большого бака.

**Item1** и **Item2** - объекты (точки объекта).

$N_x$  и  $N_z$  - количество разбиений по объекту.

$K_x$  и  $K_z$  - коэффициент разрядки от объекта.

$L_1, L_2$  - горизонтальные слои.

$\text{SideBound}(B_1, B_2, B_3, B_4)$  - номера краевых на границах.

### 3.2. Выходные данные

На выходе мы получаем файлы **nodes.txt**, **elems.txt**, **edges.txt**, **bounds.txt**, **interface.txt**, **items.txt**, **layers.txt**, **sigmas.txt**, а также с помощью пакета *ScottPlot.WPF* можно отобразить сетку в окне и сохранить в виде картинки формата (.png, .jpg, .bmp).

Структура файла **nodes.txt**:

Первое число (**int CountNodes**) - количество узлов.

Далее **CountNodes** строк - узлы (**double X**, **double Z**)

Структура файла **edges.txt**:

Первое число (**int CountEdge**) - количество ребер.

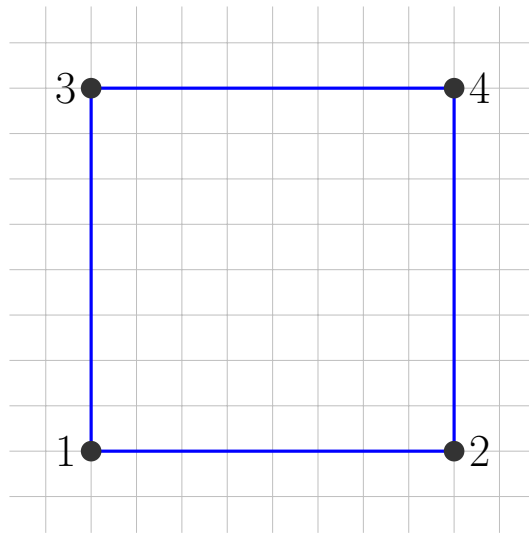
Далее **CountEdge** строк - координаты двух узлов ребра **double** ( $X_1, Z_1, X_2, Z_2$ ).

Структура файла **elems.txt**:

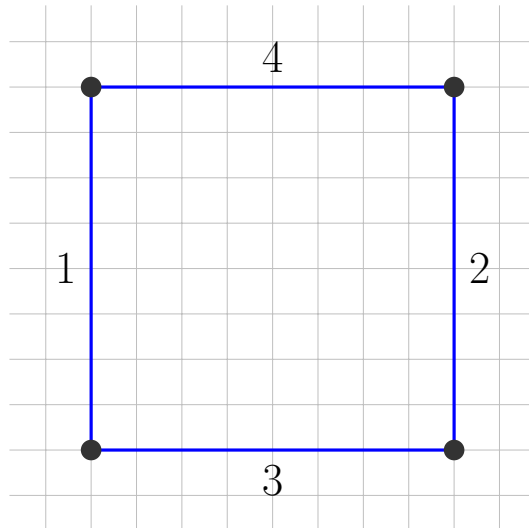
Первое число (**int CountElem**) - количество конечных элементов.

Далее **CountElem** строк - нумерация конечного элемента по узлам и ребрам.

Нумерация конечного элемента по узлам выглядит следующим образом:



Нумерация конечного элемента по ребрам выглядит следующим образом:



Структура файла **bounds.txt**:

Первое число (**int CountBound**) - количество ребер на которых задано краевое условие.

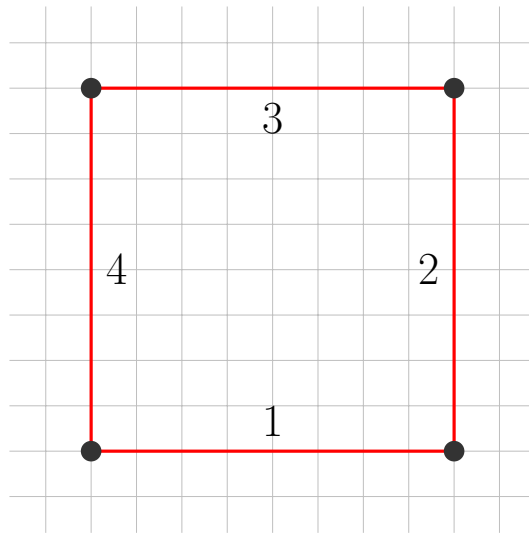
Далее **CountBound** строк - характеристика краевого условия. Характеристика краевого условия выглядит следующим образом:

Первое число (**int NumberBound**) - номер краевого условия.

Второе число (**int NumberSide**) - номер стороны сетки.

Третье число (**int NumberEdge**) - номер ребра.

Нумерация сторон сетки:



Структура файла **items.txt**:

Первое число (**int CountItems**) - количество объектов на сетке.

Далее **CountItems** строк - характеристика объекта. Характеристика объекта выглядит следующим образом:

Первые числа - координаты точек объекта **double** ( $X_1, Z_1, X_2, Z_2$ ).

Далее два **int** числа - количество разбиений по объекту ось X и ось Z соответственно.

Далее **double** число - значение проводимости объекта.

Последнее **string** слово - имя объекта.

Структура файла **layers.txt**:

Первое число (**int CountLayers**) - количество слоев на сетке.

Далее **CountLayers** строк по два **double** числа - координата **Z** и значение проводимости слоя.

Структура файла **sigmas.txt**:

Первое число (**int CountMaterial**) - количество материалов.

Далее **CountMaterial** строк по два **double** числа - значения одномерной и двумерной проводимости.

### Структура файла **interface.txt**:

Первая строка четыре **double** числа - координаты точек бака  $(X_1, Z_1, X_2, Z_2)$

Вторая строка два **int** числа - количество узлов ось X и ось Z соответственно.

Третья строка два **double** числа - коэффициенты разрядки ось X и ось Z соответственно.

Четвертая строка два числа - **double** число минимальный шаг по объекту и **int** число минимальное количество шагов по объекту.

Пятая строка **bool** значение - строгая сетка (true), не строгая сетка соответственно (false).

Шестая строка четыре **int** числа - номера краевых на сторонах сетки нижняя, правая, верхняя, левая соответственно.

### 3.3. Алгоритм построения сетки

1. Равномерно разбиваем объекты:
2. Считаем равномерные шаги  $(h_x, h_z)$  по объектам:

$$h_x = \frac{Item.End_x - Item.Begin_x}{N_x}, \quad h_z = \frac{Item.End_z - Item.Begin_z}{N_z}$$

3. Составляем список шагов  $(H_x, H_z)$  следующим образом:
  - (a) Добавляем значение шага  $h_x$  в список  $H_x$  ( $N_x$ ) раз ( $h_z$  в список  $H_z$  ( $N_z$ ) раз),
  - (b) Увеличиваем шаг  $h_x * K_x$  ( $h_z * K_z$ ) и добавляем в начало списка пока не пересечем  $Begin\_BG_x$  ( $Begin\_BG_z$ ),
  - (c) Увеличиваем шаг  $h_x * K_x$  ( $h_z * K_z$ ) и добавляем в конец списка пока не пересечем  $End\_BG_x$  ( $End\_BG_z$ ),
  - (d) В начало списка  $H_x$  ( $H_z$ ) добавлем 0.
4. Генерируем узлы по составленным спискам шагов  $(H_x, H_z)$ :

- (a) Берем начальные значения  $\mathbf{X} = \mathbf{Begin\_BG}_x$ ,  $\mathbf{Z} = \mathbf{Begin\_BG}_z$ ,
- (b) Берем шаг из списка  $\mathbf{H}_z$ , прибавляем к  $\mathbf{Z}$  и получаем  $\mathbf{Z\_new}$ ,
- (c) Берем шаг из списка  $\mathbf{H}_x$ , прибавляем к  $\mathbf{X}$  и получаем  $\mathbf{X\_new}$ ,
- (d) Создаем узел  $\mathbf{Node}(\mathbf{X\_new}, \mathbf{Z\_new})$ ,
- (e) Если шаги в списке  $\mathbf{H}_x$  остались, то возвращаемся к пункту (c),
- (f) Обновляем  $\mathbf{X} = \mathbf{Begin\_BG}_x$  и возвращаемся на пункт (b), пока шаги в списке  $\mathbf{H}_z$  не закончатся.

5. Генерируем нумерацию конечных элементов:

- (a) Генерируем номера узлов:
  - i. Берем начальные значения  $\mathbf{i} = 0$ ,  $\mathbf{j} = 0$ ,
  - ii. Генерируем номера конечного элемента
 
$$\mathbf{N}_1 = \mathbf{i} * \mathbf{CountX} + \mathbf{j},$$

$$\mathbf{N}_2 = \mathbf{i} * \mathbf{CountX} + \mathbf{j} + 1,$$

$$\mathbf{N}_3 = (\mathbf{i} + 1) * \mathbf{CountX} + \mathbf{j},$$

$$\mathbf{N}_4 = (\mathbf{i} + 1) * \mathbf{CountX} + \mathbf{j} + 1,$$
  - iii. Создаем конечный элемент  $\mathbf{Elem}(\mathbf{N}_1, \mathbf{N}_2, \mathbf{N}_3, \mathbf{N}_4)$ ,
  - iv. Прибавляем к  $\mathbf{j}$  единицу и повторяем шаг (b), пока  $\mathbf{j} < \mathbf{CountX} - 1$ ,
  - v. Обновляем  $\mathbf{j} = 0$ . Прибавляем к  $\mathbf{i}$  единицу и возвращаемся на шаг (b), пока  $\mathbf{i} < \mathbf{CountZ} - 1$ .
- (b) Генерируем ребра и номера ребер для конечного элемента:
  - i. Берем начальные значения  $\mathbf{i} = 0$ ,  $\mathbf{j} = 0$ ,
  - ii. Генерируем пять индексов:
 
$$\mathbf{left} = \mathbf{i} * ((\mathbf{CountX} - 1) + \mathbf{CountX}) + (\mathbf{CountX} - 1) + \mathbf{j},$$

$$\mathbf{right} = \mathbf{i} * ((\mathbf{CountX} - 1) + \mathbf{CountX}) + (\mathbf{CountX} - 1) + \mathbf{j} + 1,$$

$$\mathbf{bottom} = \mathbf{i} * ((\mathbf{CountX} - 1) + \mathbf{CountX}) + \mathbf{j},$$

$$\mathbf{top} = (\mathbf{i} + 1) * ((\mathbf{CountX} - 1) + \mathbf{CountX}) + \mathbf{j},$$

$$\mathbf{n\_elem} = \mathbf{i} * (\mathbf{CountX} - 1) + \mathbf{j},$$
  - iii. Создаем четыре ребра:

$\text{Edge}[\text{left}](\text{Elem}[\text{n\_elem}].\text{Node}[1], \text{Elem}[\text{n\_elem}].\text{Node}[3]),$   
 $\text{Edge}[\text{right}](\text{Elem}[\text{n\_elem}].\text{Node}[2], \text{Elem}[\text{n\_elem}].\text{Node}[4]),$   
 $\text{Edge}[\text{bottom}](\text{Elem}[\text{n\_elem}].\text{Node}[1], \text{Elem}[\text{n\_elem}].\text{Node}[2]),$   
 $\text{Edge}[\text{top}](\text{Elem}[\text{n\_elem}].\text{Node}[3], \text{Elem}[\text{n\_elem}].\text{Node}[4]),$

iv. Добавляем к элементу номера ребер:

$\text{Elem}[\text{n\_elem}](\text{left}, \text{right}, \text{bottom}, \text{top}),$

v. Прибавляем к  $j$  единицу и возвращаемся на шаг (ii), пока

$j < \text{CountX} - 1,$

vi. Обновляем  $j = 0$ . Прибавляем к  $i$  единицу и возвращаемся на шаг (ii), пока  $i < \text{CountZ} - 1$ .

6. Генерируем краевые условия:

(a) Берем из входных данных номера краевых условий **SideBound**,

(b) Генерируем краевые нижней границы:

i. Берем начальное значение  $i = 0$ ,

ii. Подсчитываем индекс ребра:

$\text{id} = i$

iii. Генерируем краевое:

$\text{Bound}(\text{SideBound}[0], 0, \text{id}),$

iv. Прибавляем к  $i$  единицу и возвращаемся на шаг (ii), пока

$i < \text{CountX} - 1$ .

(c) Генерируем краевые правой границы:

i. Берем начальное значение  $i = 1$ ,

ii. Подсчитываем индекс ребра:

$\text{id} = i * \text{CountX} + i * (\text{CountX} - 1) - 1)$

iii. Генерируем краевое:

$\text{Bound}(\text{SideBound}[1], 1, \text{id}),$

iv. Прибавляем к  $i$  единицу и возвращаемся на шаг (ii), пока

$i < \text{CountZ}$ .

(d) Генерируем краевые верхней границы:



- i. Берем начальное значение  $\mathbf{i} = \mathbf{0}$ ,
  - ii. Подсчитываем индекс ребра:  

$$\mathbf{id} = \mathbf{CountX} * (\mathbf{CountZ} - 1) +$$

$$+ (\mathbf{CountX} - 1) * (\mathbf{CountY} - 1) + \mathbf{i}$$
  - iii. Генерируем краевое:  

$$\mathbf{Bound}(\mathbf{SideBound}[\mathbf{2}], \mathbf{2}, \mathbf{id}),$$
  - iv. Прибавляем к  $\mathbf{i}$  единицу и возвращаемся на шаг (ii), пока  

$$\mathbf{i} < \mathbf{CountX} - 1.$$
- (e) Генерируем краевые левой границы:
- i. Берем начальное значение  $\mathbf{i} = \mathbf{0}$ ,
  - ii. Подсчитываем индекс ребра:  

$$\mathbf{id} = (\mathbf{i} + 1) * (\mathbf{CountX} - 1) + \mathbf{i} * \mathbf{CountX}$$
  - iii. Генерируем краевое:  

$$\mathbf{Bound}(\mathbf{SideBound}[\mathbf{3}], \mathbf{3}, \mathbf{id}),$$
  - iv. Прибавляем к  $\mathbf{i}$  единицу и возвращаемся на шаг (ii), пока  

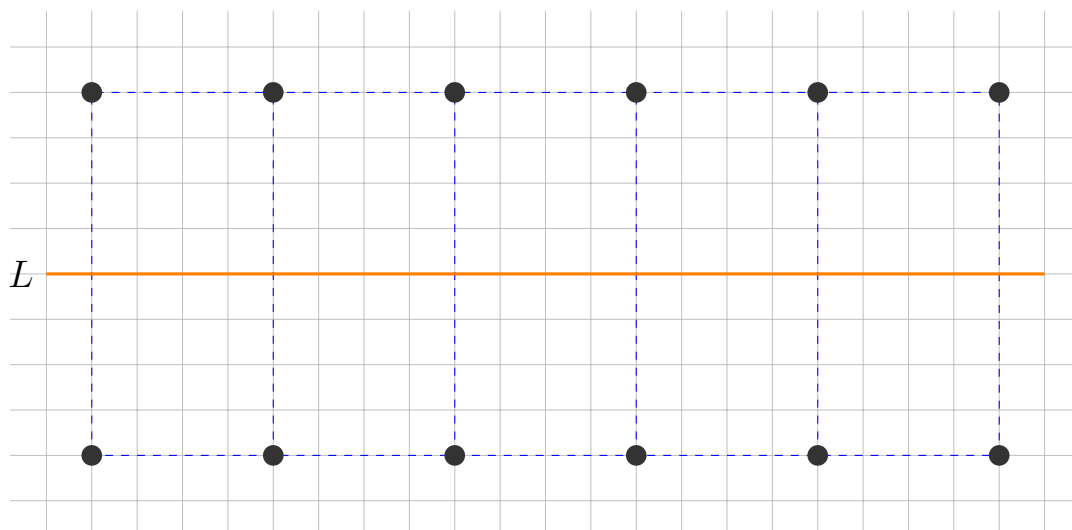
$$\mathbf{i} < \mathbf{CountZ} - 1.$$
- (f) Сортируем краевые в порядке убывания номера краевого.

### 3.4. Учет горизонтальных слоев

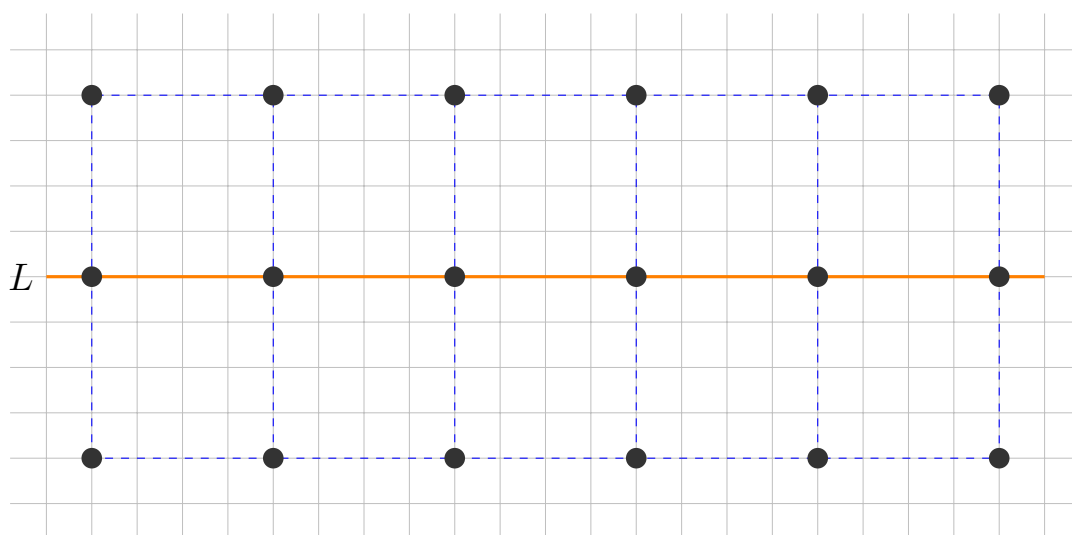
В сетке можно задавать горизонтальные слои ( $\mathbf{L}$ ).

**Важно:** слой должен проходить через узлы.

Неправильный учет слоя:



Правильный учет слоя:



Алгоритм учета горизонтальных слоев:

Учитывать слои будем после этапа построения списков шагов ( $\mathbf{H}_x, \mathbf{H}_z$ ).

1. Берем начальные значения  $\mathbf{Z} = \mathbf{Begin\_BG}_z$ ,
2. Берем шаг из списка  $\mathbf{H}_z$ , прибавляем к  $\mathbf{Z}$  и получаем  $\mathbf{Z\_new}$ ,

3. Если  $\mathbf{Z\_new} = \mathbf{L}$ , то на слое будут лежать узлы,  
Если  $\mathbf{Z\_new} > \mathbf{L}$ , добавим шаг, чтобы учесть слой,
4. Если добавленный шаг меньше минимального ( $\mathbf{min\_step}$ ), то значение шага прибавляем к предыдущему значению шага, а текущий шаг удаляем,
5. Если шаги в списке  $\mathbf{H_z}$  и непройденные слои остались, то возвращаемся к пункту (2).

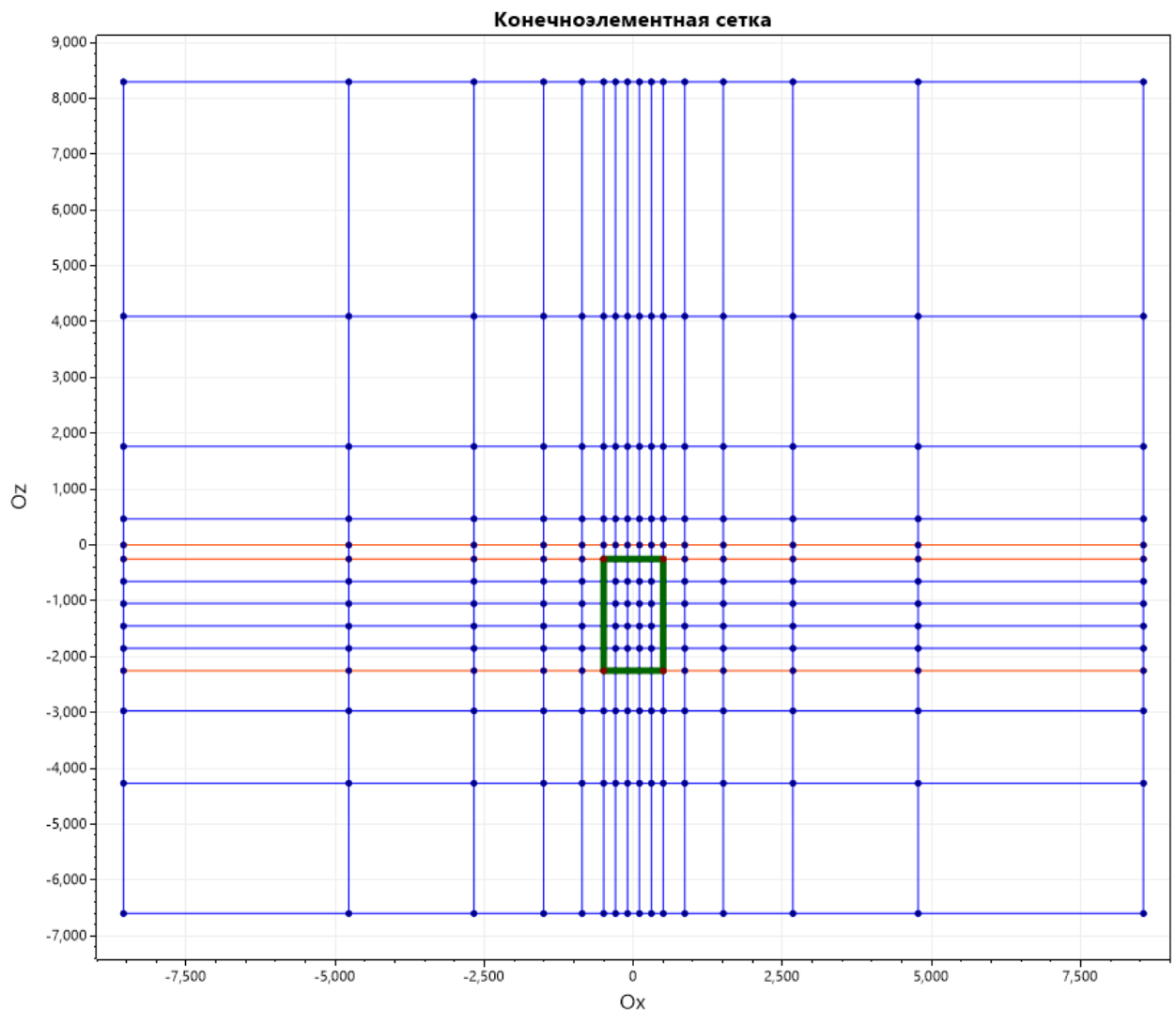


Рисунок 3.1 – Сетка в горизонтальной слоистой среде  $L=(0,-250,-500)$

## 3.5. Руководство по программе

### 3.5.1. Используемые при разработке средства

1. Visual Studio 2022 – основная среда разработки,
2. WPF – построение графического интерфейса,
3. C# – основной язык логики приложения,
4. ScottPlot - библиотека построения графиков,
5. WPFToggleSwitch - пакет для Toggle Button.

### 3.5.2. Описание пользовательского интерфейса

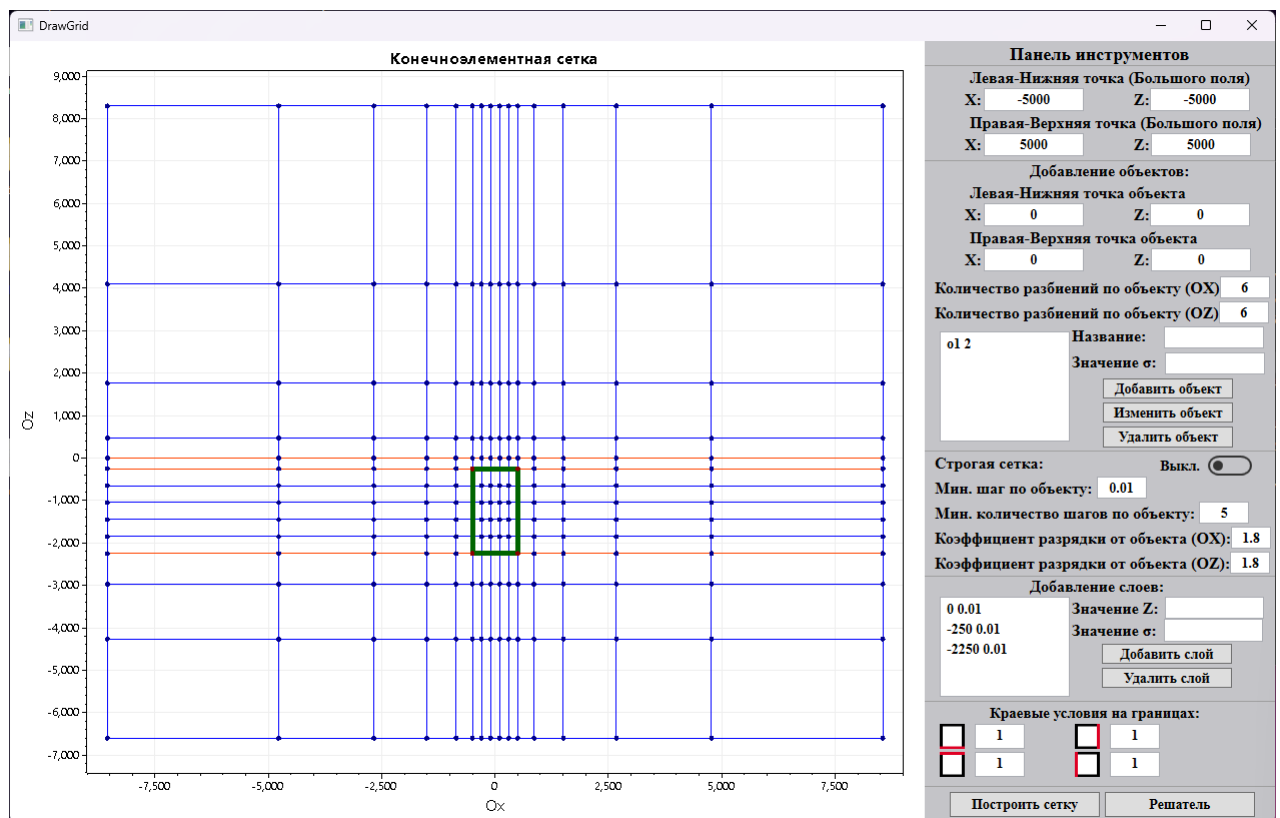


Рисунок 3.2 – Вид окна построения сетки


Таблица 3.1 – Описание бокового меню

Пункт меню	Назначение
Пункты меню для задания координат большого поля	

Продолжение таблицы 3.1

Пункт меню	Назначение
Левая-Нижняя точка (Большого поля)	Задает точку <b>Begin_BG</b>
Правая-Верхняя точка (Большого поля)	Задает точку <b>End_BG</b>
Пункты меню для задания объекта	
Левая-Нижняя точка объекта	Задает точку <b>Begin</b>
Правая-Верхняя точка объекта	Задает точку <b>End</b>
Количество разбиений по объекту (Ox)	Задает параметр объекта <b>Nx</b>
Количество разбиений по объекту (Oz)	Задает параметр объекта <b>Nz</b>
Название	Задает параметр объекта <b>Name</b>
Значение $\sigma$	Задает параметр объекта <b>Sigma</b>
Кнопка «Добавить объект»	Добавляет новый объект в список объектов
Кнопка «Изменить объект»	Изменяет выбранный в ListBox объект
Кнопка «Удалить объект»	Удаляет выбранный в ListBox объект
Пункты меню для задания общих параметров	
Строгая сетка	ВКЛ./ВЫКЛ. строгость сетки
Минимальный шаг по объекту	Задает параметр <b>min_step</b>
Минимальное количество шагов по объекту	Задает параметр <b>count_step</b>

Продолжение таблицы 3.1

Пункт меню	Назначение
Коэффициент разрядки от объекта (OX)	Задаёт параметр <b>Kx</b>
Коэффициент разрядки от объекта (OZ)	Задаёт параметр <b>Kz</b>
Пункты меню для задания слоя	
Значение Z	Задаёт значение слоя
Значение $\sigma$	Задаёт проводимость слоя
Кнопка «Добавить слой»	Добавляет значение слоя на сетку
Кнопка «Удалить слой»	Удаляет значение слоя из сетки
Пункты меню для задания краевых условий	
	Задаёт краевое условие на <u>нижней</u> стороне сетки
	Задаёт краевое условие на <u>правой</u> стороне сетки
	Задаёт краевое условие на <u>верхней</u> стороне сетки
	Задаёт краевое условие на <u>левой</u> стороне сетки
Кнопки построения сетки и запуска решателя	
Кнопка «Построить сетку»	Строит (перестраивает) сетку
Кнопка «Решатель»	Открывает окошко решателя

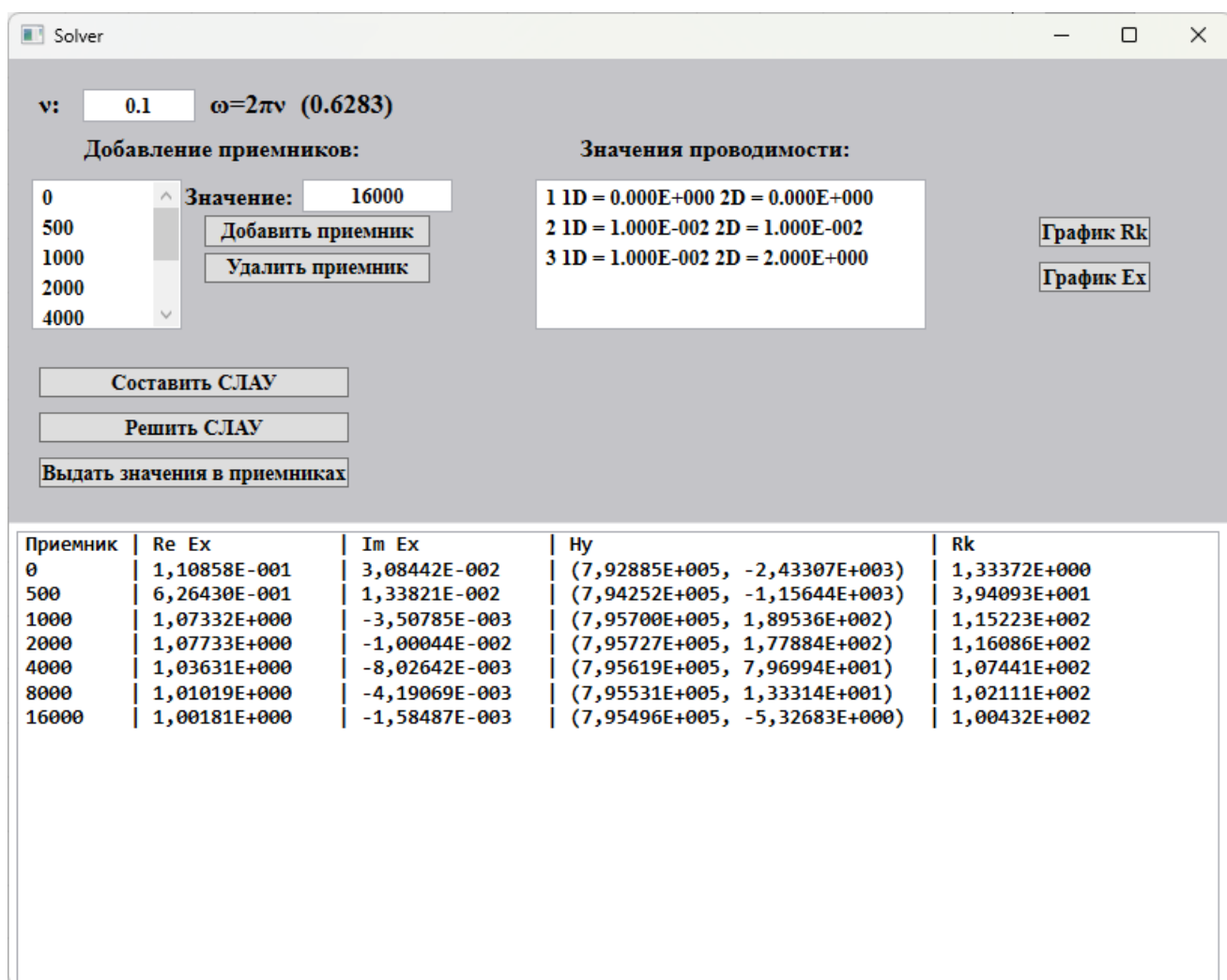


Рисунок 3.3 – Вид окна решателя

Таблица 3.2 – Описание окна 3.3

Пункт меню	Назначение
Значение $\nu$	Задаёт частоту электромагнитного поля
Добавление приемников	
Значение X	Задаёт координату X приемника на поверхности Земли
Кнопка «Добавить приемник»	Добавляет приемник
Кнопка «Удалить приемник»	Удаляет выбранный приемник

### Продолжение таблицы 3.2

Пункт меню	Назначение
Значения проводимости	Указаны материалы и их проводимости
Кнопка «Составить СЛАУ»	Составляет СЛАУ с заранее сгенерированной сеткой и введенной частотой
Кнопка «Решить СЛАУ»	Решает СЛАУ с помощью модуля PARDISO
Кнопка «Выдать значения в приемниках»	Выдает таблице с рассчитанными компонентами поля в приемниках
Кнопка «График $\rho_k$ »	Открывается окошко с графиком компоненты $\rho_k$
Кнопка «График $E_x$ »	Открывается окошко с графиком компоненты $E_x$

### 3.5.3. Взаимодействие с Plot из пакета ScottPlot

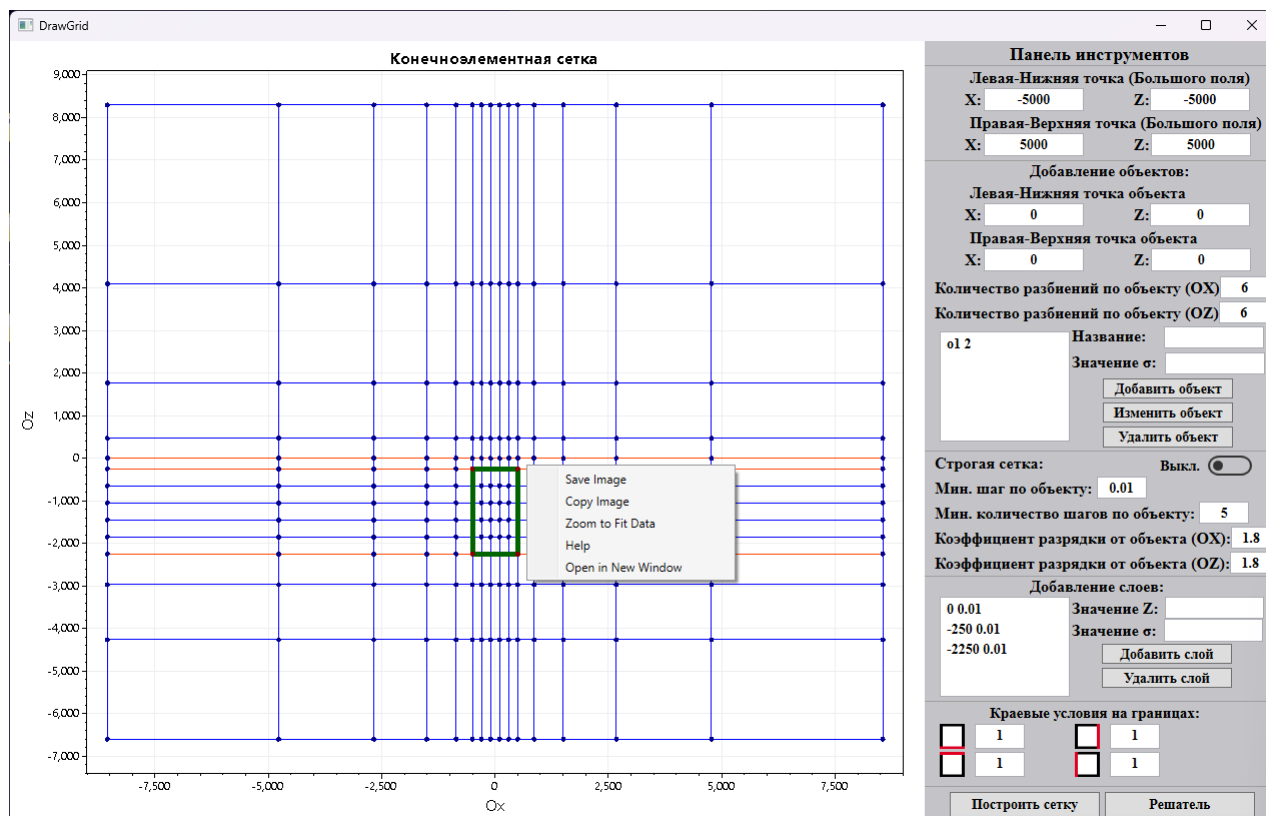




Таблица 3.3 – Описание инструментария пакета

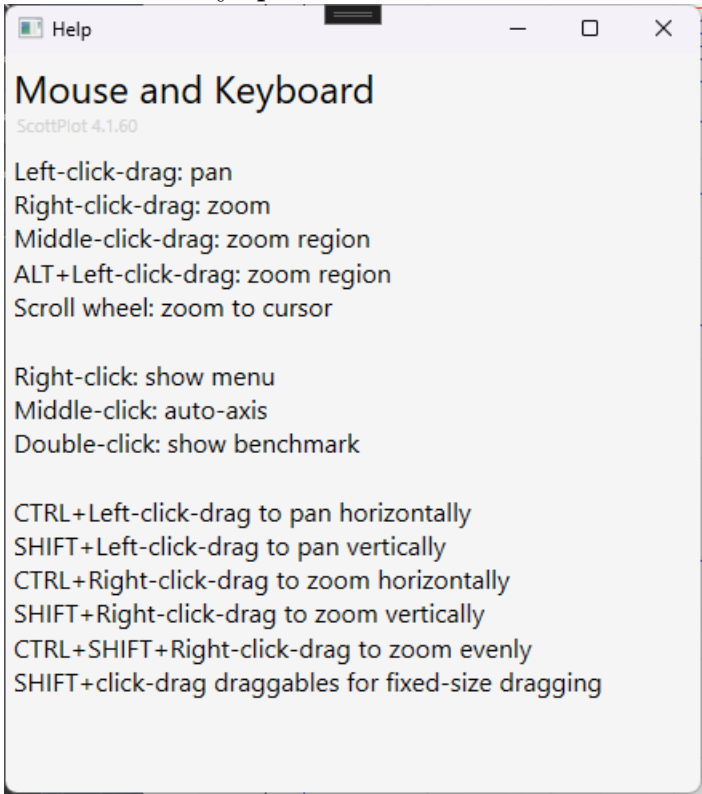
Пункт меню	Назначение
Save Image	Открывает диалоговое окно для сохранения Plot
Copy Image	Сохраняет картинку в буфер
Zoom to Fit Data	Увеличивает масштаб, чтобы соответствовать данным
Help	<p>Открывает окошко с инструкцией для управления Plot</p>  <p>The screenshot shows a help window with the title 'Mouse and Keyboard' for ScottPlot 4.1.60. It lists the following actions:</p> <ul style="list-style-type: none"> <li>Left-click-drag: pan</li> <li>Right-click-drag: zoom</li> <li>Middle-click-drag: zoom region</li> <li>ALT+Left-click-drag: zoom region</li> <li>Scroll wheel: zoom to cursor</li> <li>Right-click: show menu</li> <li>Middle-click: auto-axis</li> <li>Double-click: show benchmark</li> <li>CTRL+Left-click-drag to pan horizontally</li> <li>SHIFT+Left-click-drag to pan vertically</li> <li>CTRL+Right-click-drag to zoom horizontally</li> <li>SHIFT+Right-click-drag to zoom vertically</li> <li>CTRL+SHIFT+Right-click-drag to zoom evenly</li> <li>SHIFT+click-drag draggables for fixed-size dragging</li> </ul>
Open in New Window	Открывает Plot в новом окошке

Таблица 3.4 – Описание окошка с инструкцией

Действие	Назначение
ЛКМ + перетаскивание	Панорамирование (движение по сетке)
ПКМ + перетаскивание	Увеличение масштаба
СКМ + перетаскивание	Выделение области масштабирования

Продолжение таблицы 3.4

Действие	Назначение
ALT + ЛКМ + перетаскивание	Выделение области масштабирования
Колесо прокрутки	Масштабирование к курсору
Щелчок ПКМ	Открывает окошко помощи
Щелчок СКМ	Увеличивает масштаб, чтобы соответствовать данным
Двойной щелчок ЛКМ	Показывает benchmark
CTRL + ЛКМ + перетаскивание	Горизонтальное перемещение
SHIFT + ЛКМ + перетаскивание	Вертикальное перемещение
CTRL + ПКМ + перетаскивание	Горизонтальное увеличение
SHIFT + ПКМ + перетаскивание	Вертикальное увеличение
CTRL + SHIFT + ПКМ + перетаскивание	Равномерное увеличение

## **4. Процедура решения СЛАУ с использованием библиотеки Intel OneMKL**

### **4.1. Введение в Intel oneAPI Math Kernel Library**

Intel oneAPI Math Kernel Library (oneMKL) - это вычислительная математическая библиотека, состоящая из высокооптимизированных многопоточных подпрограмм для приложений, требующих максимальной производительности. Библиотека предоставляет интерфейсы языков программирования Fortan и C [1, 2].

Из математической библиотеки были использованы процедуры OneMKL PARDISO, который поддерживает прямой разреженный решатель, итеративный разреженный решатель и поддерживающие разреженные процедуры BLAS для решения разреженных систем уравнения.

### **4.2. Используемые при разработке средства**

1. Visual Studio 2022 – основная среда разработки,
2. C, C++ – основные языки логики приложения,
3. Intel OneMKL - библиотека оптимизированных математических процедур для научных приложений.

### 4.3. Разреженный формат хранения матрицы CSR3

Формат CSR3 (с 3 массивами), принятый для прямых разреженных решателей, является разновидностью формата CSR (с 4 массивами) [3].

Для симметричных матриц необходимо сохранить только верхнюю треугольную половину матрицы (верхний треугольный формат) или нижнюю треугольную половину матрицы (нижний треугольный формат).

Формат хранения разреженных матриц Intel one API MKL для прямых разреженных решателей определяется тремя массивами:  $a$ ,  $ja$  и  $ia$ .

$a$  - комплексный массив, содержащий ненулевые элементы разреженной матрицы.

$ja$  -  $i$ -й элемент целочисленного массива  $ja$  - это номер столбца, который содержит  $i$ -й элемент в массиве  $a$ .

$ia$  -  $j$ -й элемент целочисленного массива  $ia$  дает индекс элемента в массиве  $a$ , который является первым ненулевым элементом в строке  $j$ .

Длина массивов  $a$  и  $ja$  равна количеству ненулевых элементов в матрице.

$$B = \begin{bmatrix} 1 & -1 & * & -3 & * \\ -1 & 5 & * & * & * \\ * & * & 4 & 6 & 4 \\ -3 & * & 6 & 7 & * \\ * & * & 4 & * & -5 \end{bmatrix} \quad (4.1)$$

Матрица  $\mathbf{B}$  в разреженном строчно-столбцовом формате:

---


$$di = [1 \quad 5 \quad 4 \quad 7 \quad -5]$$

$$gg = [-1 \quad -3 \quad 6 \quad 4]$$

$$ig = [0 \quad 0 \quad 1 \quad 1 \quad 3 \quad 4]$$

$$\text{jg} = [0 \quad 0 \quad 2 \quad 2]$$

---

Матрица **B** в формате CSR3:

---

$$\text{a} = [1 \quad -1 \quad -3 \quad 5 \quad 4 \quad 6 \quad 4 \quad 7 \quad -5]$$

$$\text{ja} = [0 \quad 1 \quad 3 \quad 1 \quad 2 \quad 3 \quad 4 \quad 3 \quad 4]$$

$$\text{ia} = [0 \quad 3 \quad 4 \quad 7 \quad 8 \quad 9]$$

---

## 5. Программный модуль для построения портрета матрицы

### 5.1. Руководство по плагину

В моделировании физических процессов и явлений часто возникают СЛАУ с разреженными матрицами. Однако, для эффективного решения таких СЛАУ прямыми методами необходимо учитывать расположение ненулевых элементов в строках и столбцах матрицы. Для улучшения методов решения конечноэлементных СЛАУ, необходимы удобные инструменты для отображения информации и визуализации структуры матриц СЛАУ.

Плагин написан на языке C++, который позволяет выдавать информацию о разреженных матрицах и визуализировать их структуру в `Fig`. Это позволяет увеличить эффективность работы с СЛАУ с разреженными матрицами и повысить качество разработки новых методов решения.

#### 5.1.1. Входные и выходные данные

Входными данными для плагина является матрица хранящаяся в разреженном формате:

- `di` - массив вещественных чисел для хранения диагональных элементов,
- `gg` - массив вещественных чисел для хранения ненулевых элементов,
- `ig` - массив целых чисел для хранения номеров строк соответствующих элементов массива `gg`,
- `jg` - массив целых чисел для хранения номеров столбцов соответствующих элементов массива `gg`,
- `kuslau` - хранит одно целое число, размерность матрицы (необязательный файл).

Выходными данными плагина будет картинка формата (.png или .bmp) на выбор и файл **info.txt**, который содержит подробную информацию о матрице.

### 5.1.2. Используемые при разработке средства

1. Visual Studio 2019 - основная среда разработки плагина,
2. C++ – основной язык логики приложения,
3. <https://github.com/FarGroup/FarManager> - репозиторий исходного кода Far Manager,
4. <https://api.farmanager.com/ru> - основной источник документации для разработчиков плагинов для Far Manager,
5. <https://farplugins.sourceforge.io/> - исходный код для плагина Image Viewer,
6. <https://github.com/ArashPartow/bitmap> - исходный код библиотеки Bitmap.

Репозиторий с исходным кодом плагина - <https://github.com/desmond60/PortraitMatrix>

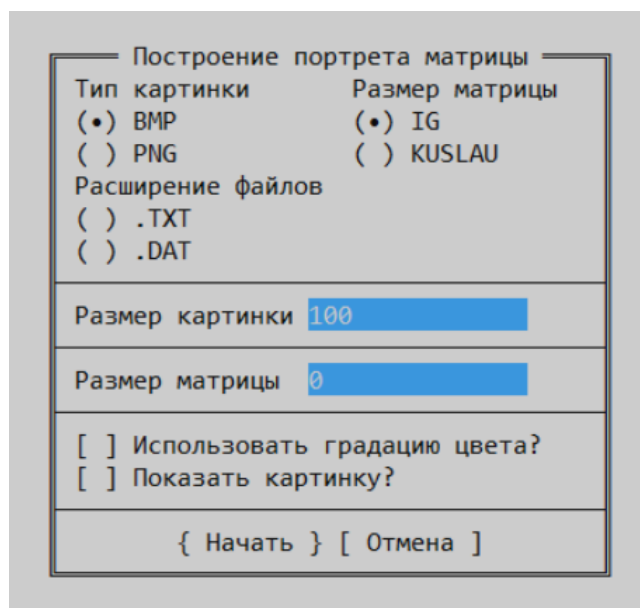


Рисунок 5.1 – Интерфейс плагина

## 6. Результаты вычислительных экспериментов

### 6.1. Тестирование на линейных функциях

#### 6.1.1. Расчетная область

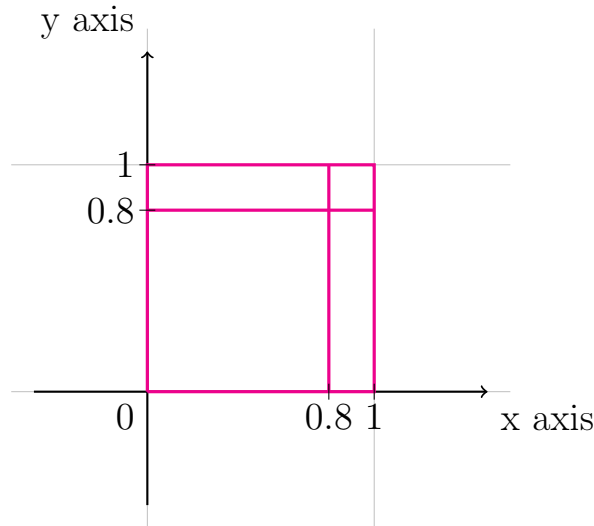


Рисунок 6.1 – Расчетная область для тестирования линейных функций

Расчетная область представляет собой квадрат со следующими параметрами:  $x \in [0, 1]$ ,  $y \in [0, 1]$ , состоит из 12 ребер и 4 прямоугольников, изображенная на рис. 6.1.

На внешних границах расчетной области заданы условия первого рода.

Физические параметры среды:  $\mu = \mu_0$ ,  $\sigma = 1$  См/м.

Частота источника поля  $\omega = 100$  Гц.

#### 6.1.2. Полином первой степени

Аналитическое решение:  $\vec{A} = \begin{bmatrix} 2x + 3y + i(6x + 7y) \\ 3x - 2y + i(x + y) \end{bmatrix}$ .

Правая часть:  $\vec{J} = i\sigma\omega \begin{bmatrix} 2x + 3y + i(6x + 7y) \\ 3x - 2y + i(x + y) \end{bmatrix}$ .

Относительная погрешность:  $2.573E - 011$ .



### 6.1.3. Полином второй степени

Аналитическое решение:  $\vec{A} = \begin{bmatrix} 2x^2 + 3y^2 + \iota (6x^2 + 7y^2) \\ 3x^2 - 2y^2 + \iota (x^2 + y^2) \end{bmatrix}.$

Правая часть:  $\vec{J} = i\sigma\omega \begin{bmatrix} -6 + i(-14) \\ -6 + i(-2) \end{bmatrix}.$

Относительная погрешность:  $3.026E - 012.$

### 6.1.4. Полином третьей степени

Аналитическое решение:  $\vec{A} = \begin{bmatrix} 2x^3 + 3y^3 + i(6x^3 + 7y^3) \\ 3x^3 - 2y^3 + i(x^3 + y^3) \end{bmatrix}.$

Правая часть:  $\vec{J} = i\sigma\omega \begin{bmatrix} -18y + i(-42y) \\ -18x + i(-6x) \end{bmatrix}.$

Относительная погрешность:  $2.614E - 012.$

### 6.1.5. Полином четвертой степени

Аналитическое решение:  $\vec{A} = \begin{bmatrix} 2x^4 + 3y^4 + i(6x^4 + 7y^4) \\ 3x^4 - 2y^4 + i(x^4 + y^4) \end{bmatrix}.$

Правая часть:  $\vec{J} = i\sigma\omega \begin{bmatrix} -36y^2 + i(-84y^2) \\ -36x^2 + i(-12x^2) \end{bmatrix}.$

Относительная погрешность:  $5.605E - 002.$

## 6.2. Оценка порядка аппроксимации

Проведем исследования на порядок аппроксимации. В качестве задачи будем использовать полином четвертой степени:

Аналитическое решение:  $\vec{A} = \begin{bmatrix} 2x^4 + 3y^4 + i(6x^4 + 7y^4) \\ 3x^4 - 2y^4 + i(x^4 + y^4) \end{bmatrix}.$

Правая часть:  $\vec{J} = i\sigma\omega \begin{bmatrix} -36y^2 + i(-84y^2) \\ -36x^2 + i(-12x^2) \end{bmatrix}.$

Таблица 6.1 – Оценка порядка аппроксимации

Кол. ребер	$\ A^* - A\ $	$\frac{(\ A^* - A\ )_{i-1}}{(\ A^* - A\ )_i}$	k
12	5,605E-002	-	-
24	2,974E-002	1,88E+00	0,91
60	1,166E-002	2,55E+00	1,35
220	3,280E-003	3,55E+00	1,83
840	8,761E-004	3,74E+00	1,90

Порядок аппроксимации приближается к 2.

### 6.3. Модель 2Д-1 из проекта СОММЕМІ

Для верификации решения двумерной задачи была выбрана модель 2Д-1, изображенная на рис. 6.2, приведенная в международном проекте СОММЕМІ [9, стр. 60]. Эта модель имеет самую простую геометрическую форму - содержит симметричную, прямоугольную вставку с высокой электропроводностью, размещенную в однородном проводящем полупространстве.

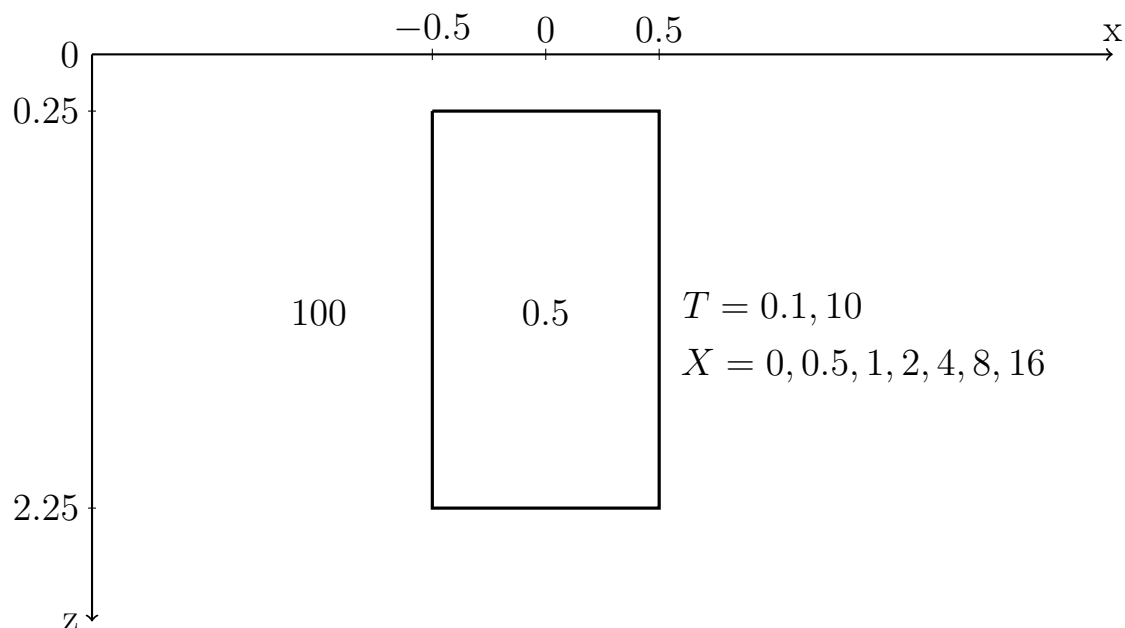


Рисунок 6.2 – Структура модели 2Д-1, координаты даны в км, периоды в с, сопротивления в Ом · м

Расчеты выполнялись для двух частот 0.1 Гц и 10 Гц. Ток направлен параллельно земной поверхности.

Кажущееся сопротивление рассчитывается по формуле:

$$\rho_k = \frac{|Z|^2}{\omega\mu_0}. \quad (6.1)$$

Значение импеданса можно найти по формуле:

$$Z = \frac{E_x}{H_y} = \frac{-i\omega \left( \vec{A}^{1D} + \vec{A}^a \right)}{H_y}. \quad (6.2)$$

Таблица 6.2 содержит результаты расчетов для частоты 0.1 Гц, которые сравниваются с данными, указанными для модели 2Д-1, представленной в материалах проекта COMMEMI.

Таблица 6.2 – Сравнение компонент  $E_x$  и  $\rho_k$  при частоте 0.1 Гц.

X	COMMEMI			Программа			Погрешность		
	Re $E_x$	Im $E_x$	$\rho_k$	Re $E_x$	Im $E_x$	$\rho_k$	Re $E_x$	Im $E_x$	$\rho_k$
0	0,127	-0,031	1,650	0,102	0,030	1,141	0,025	0,061	0,509
500	0,688	-0,012	48,660	0,618	0,014	38,574	0,070	0,026	10,086
1000	1,064	0,003	114,620	1,079	0,003	116,296	0,015	0,006	1,676
2000	1,078	0,009	118,030	1,078	-0,010	116,198	0,000	0,019	1,832
4000	1,038	0,007	109,440	1,037	-0,008	107,472	0,001	0,015	1,968
8000	1,015	0,004	104,340	1,010	-0,004	102,116	0,005	0,008	2,224
16000	1,008	0,002	102,870	1,002	-0,002	100,444	0,006	0,004	2,426

На рис. 6.3-6.4 приведены результаты расчеты для частоты 0.1 Гц, которые сравниваются с данными, полученными для модели 2Д-1 в материалах проекта COMMEMI.

На графиках видно, что кривая, полученная с помощью моего приложения, почти полностью совпадает со средней кривой, полученной авторами проекта COMMEMI.

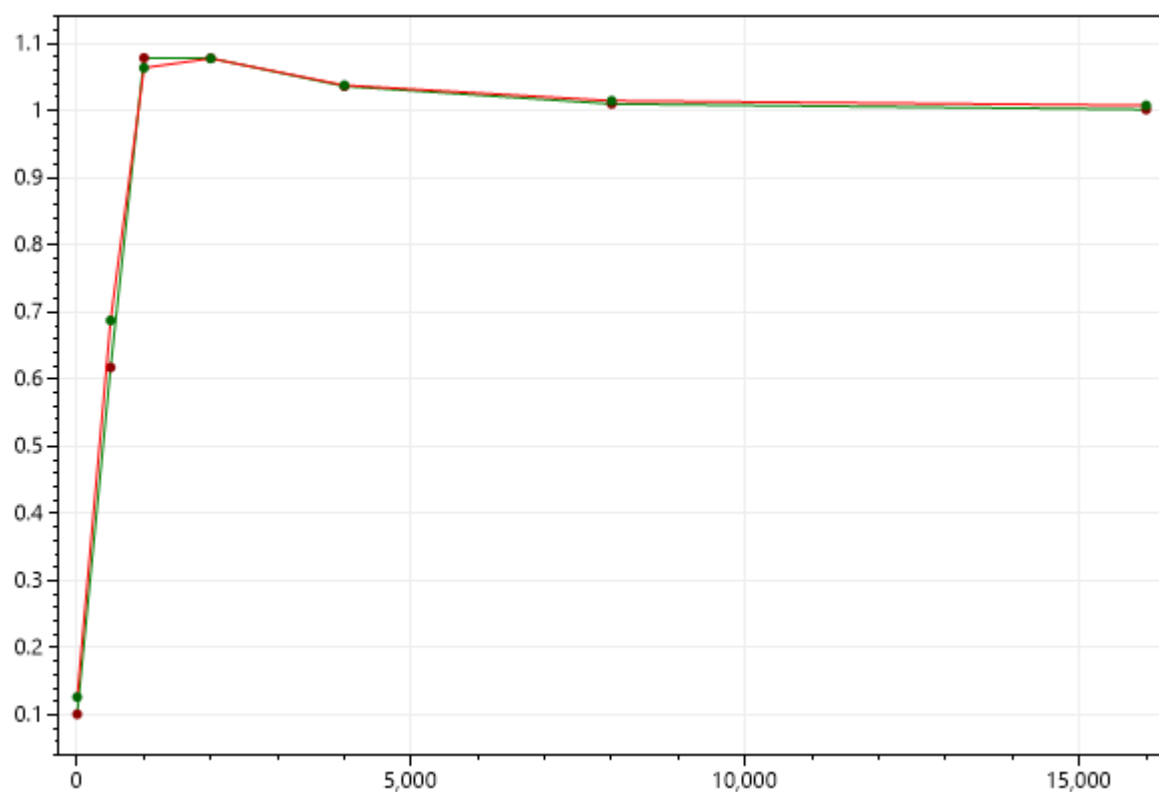


Рисунок 6.3 – График электрического поля (обозначен зеленым цветом) для частоты 0.1 Гц в сравнении с данными из проекта COMMEMI (обозначен красным цветом)

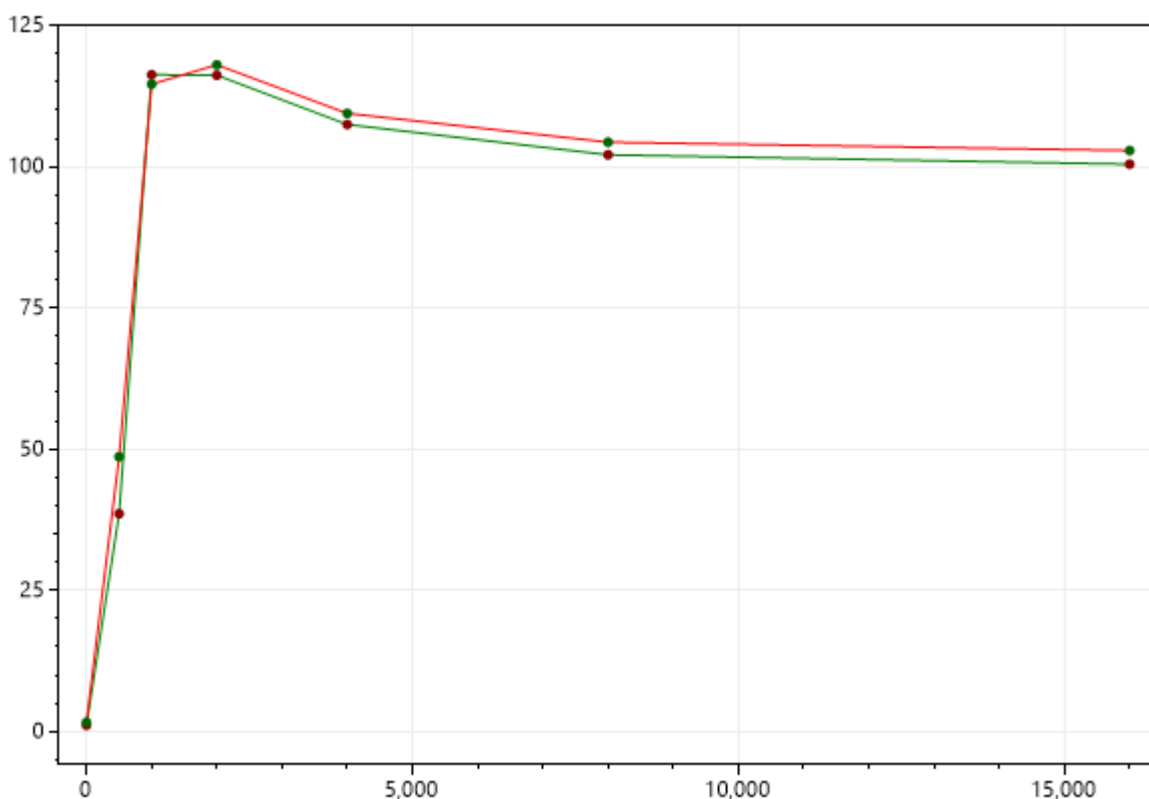


Рисунок 6.4 – График кажущегося сопротивления (обозначен зеленым цветом) для частоты 0.1 Гц в сравнении с данными из проекта COMMEMI (обозначен красным цветом)

Таблица 6.3 содержит результаты расчетов для частоты 10 Гц, которые сравниваются с данными, указанными для модели 2Д-1, представленной в материалах проекта COMMEMI.

Таблица 6.3 – Сравнение компонент  $E_x$  и  $\rho_k$  при частоте 10 Гц.

X	COMMEMI			Программа			Погрешность		
	Re $E_x$	Im $E_x$	$\rho_k$	Re $E_x$	Im $E_x$	$\rho_k$	Re $E_x$	Im $E_x$	$\rho_k$
0	0,289	-0,144	10,340	0,258	0,123	9,378	0,031	0,267	0,962
500	0,703	-0,053	49,500	0,643	0,052	45,524	0,060	0,105	3,976
1000	0,962	0,007	92,820	0,979	-0,010	95,897	0,017	0,017	3,077
2000	0,989	0,008	98,220	0,992	-0,004	98,570	0,003	0,012	0,350
4000	0,995	0,007	99,530	0,999	0,001	99,793	0,004	0,003	0,263
8000	0,996	0,006	99,840	1,000	0,000	100,048	0,004	0,006	0,208
16000	0,996	0,004	99,830	1,000	0,000	100,035	0,004	0,004	0,205

На рис. 6.5-6.6 приведены результаты расчетов для частоты 10 Гц, которые сравниваются с данными, полученными для модели 2Д-1 в материалах проекта COMMEMI.

На графиках также видно, что кривая, полученная с помощью моего приложения, почти полностью совпадает со средней кривой, полученной авторами проекта COMMEMI.

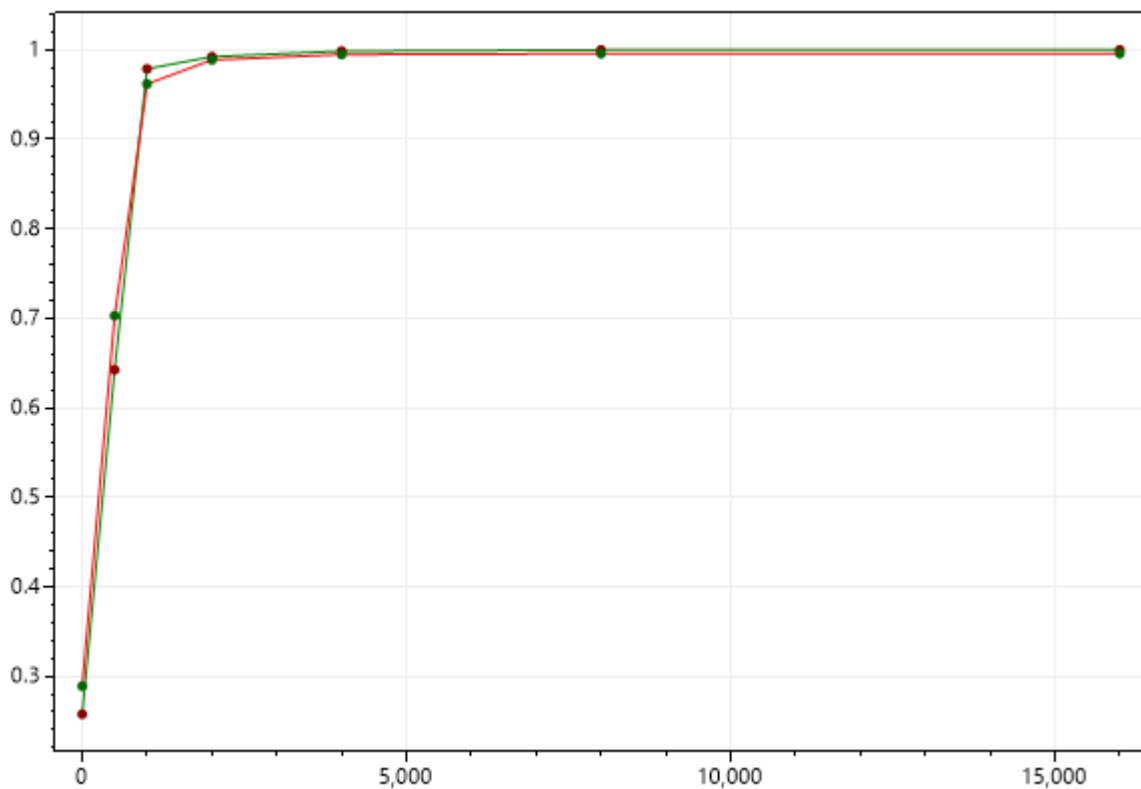


Рисунок 6.5 – График электрического поля (обозначен зеленым цветом) для частоты 10 Гц в сравнении с данными из проекта COMMEMI (обозначен красным цветом)

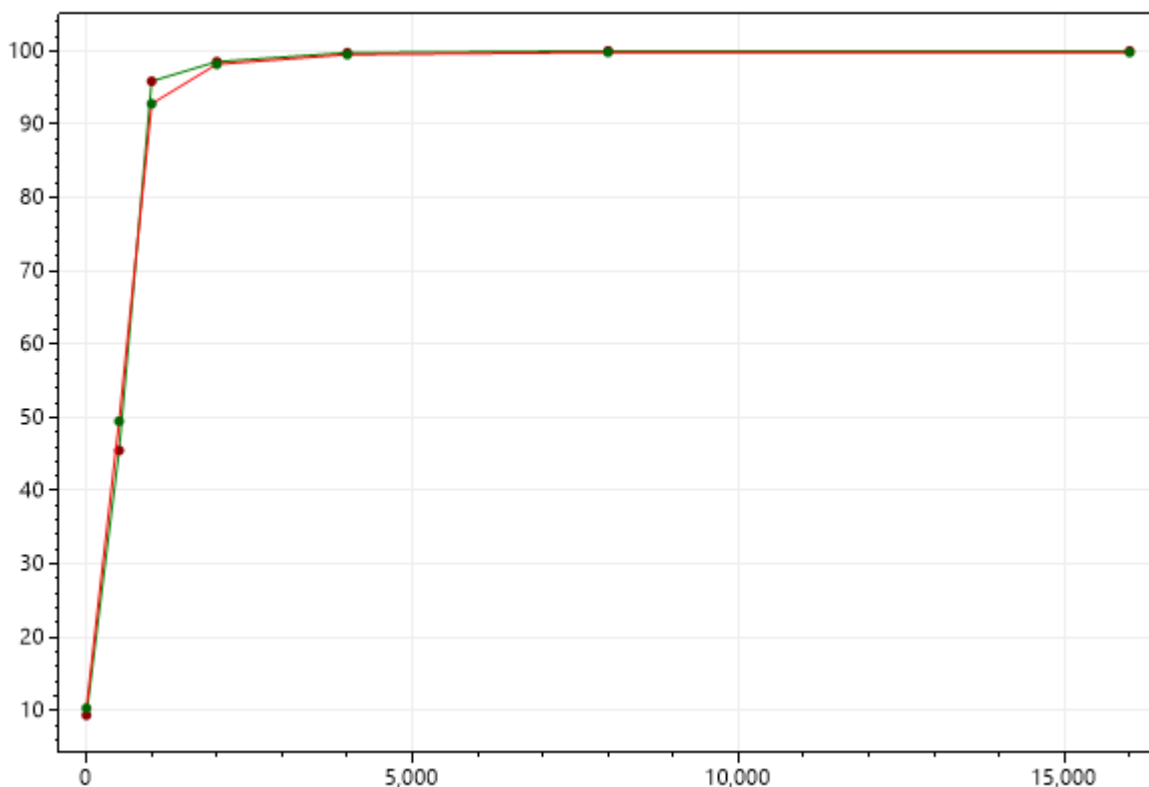


Рисунок 6.6 – График кажущегося сопротивления (обозначен зеленым цветом) для частоты 10 Гц в сравнении с данными из проекта COMMEMI (обозначен красным цветом)

#### 6.4. Применение метода деревьев-кодереьев

Основные вычислительные затраты в векторном МКЭ расходуются на решение системы линейных алгебраических уравнений (СЛАУ). При решении задач магнитостатики с использованием векторного МКЭ получится СЛАУ с большим нулевым ядром. Метод деревьев-кодереьев был разработан для устранения нулевого ядра матрицы СЛАУ, его теория описана в [4, 8].

Будем использовать дерево, изображенное на рис. 6.7 на прямоугольной сетке, которое приводит к трехдиагональной матрице.

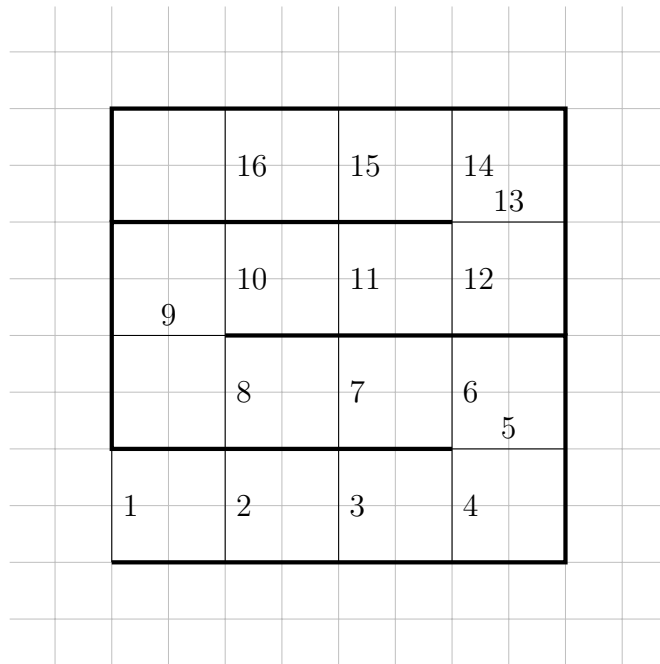


Рисунок 6.7 – Дерево на прямоугольной сетке, которое приводит к трехдиагональной матрице

Неизвестные, связанные с ребрами дерева, находящимися в среде с  $\sigma = 0$ , были исключены после сборки матрицы и вектора правой части СЛАУ.



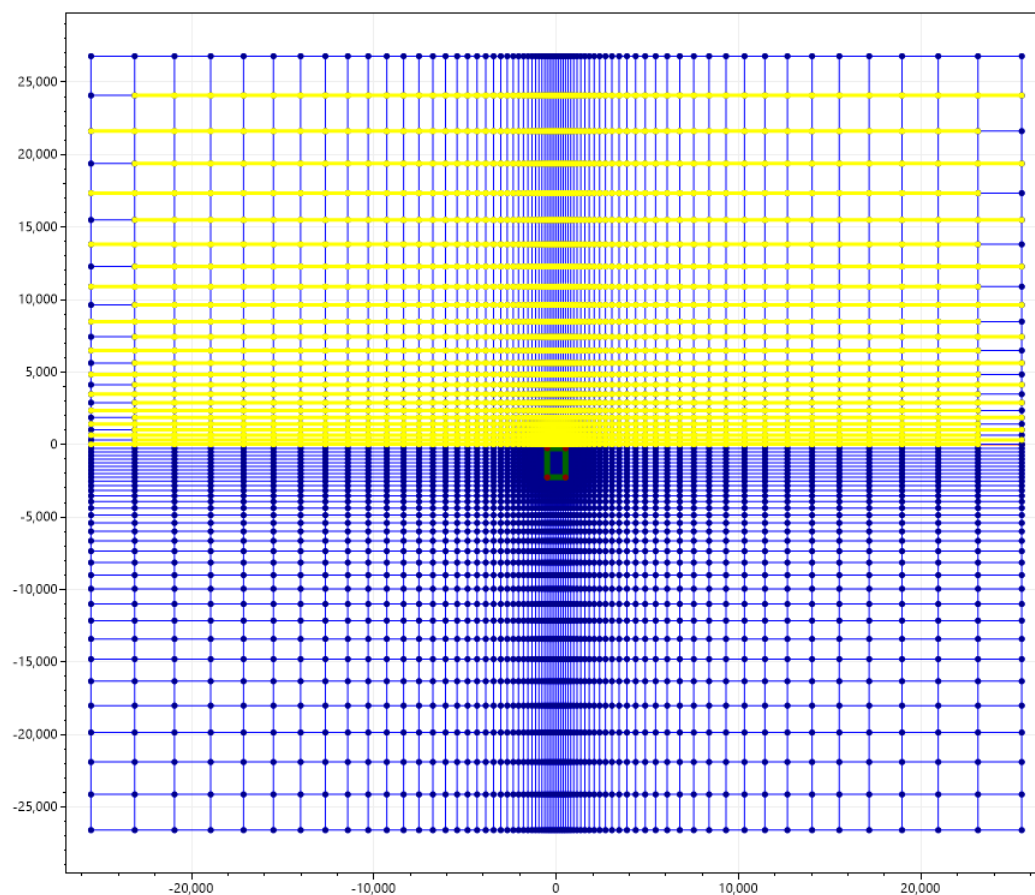


Рисунок 6.8 – Сетка с объектом и выделенными ребрами в воздухе, которые будут исключены после сборки матрицы

Были проведены вычислительные эксперименты без дерева и с деревом. Результаты приведены в таблице 6.4.

Таблица 6.4 – Заполненность и время решения СЛАУ с методом деревьев-кодереьев и без него

Число неизвестных	Число ненулевых элементов в нижнем треугольнике матрицы	время без дерева	время с деревом
881	3401	0.009 с.	0.011 с.
307 243	1 226 623	2.267 с.	2.241 с.
860 587	3 438 415	8.523 с.	8.080 с.

Продолжение таблицы 6.4

Число неизвестных	Число ненулевых элементов в нижнем треугольнике матрицы	время без дерева	время с деревом
1 489 425	5 952 525	14.963 с.	14.681 с.
1 866 227	7 459 115	21.549 с.	21.282 с.
2 287 562	9 143 834	31.681 с.	29.740 с.
2 744 726	10 971 878	37.542 с.	35.575 с.
4 362 945	17 442 921	70.183 с.	66.578 с.

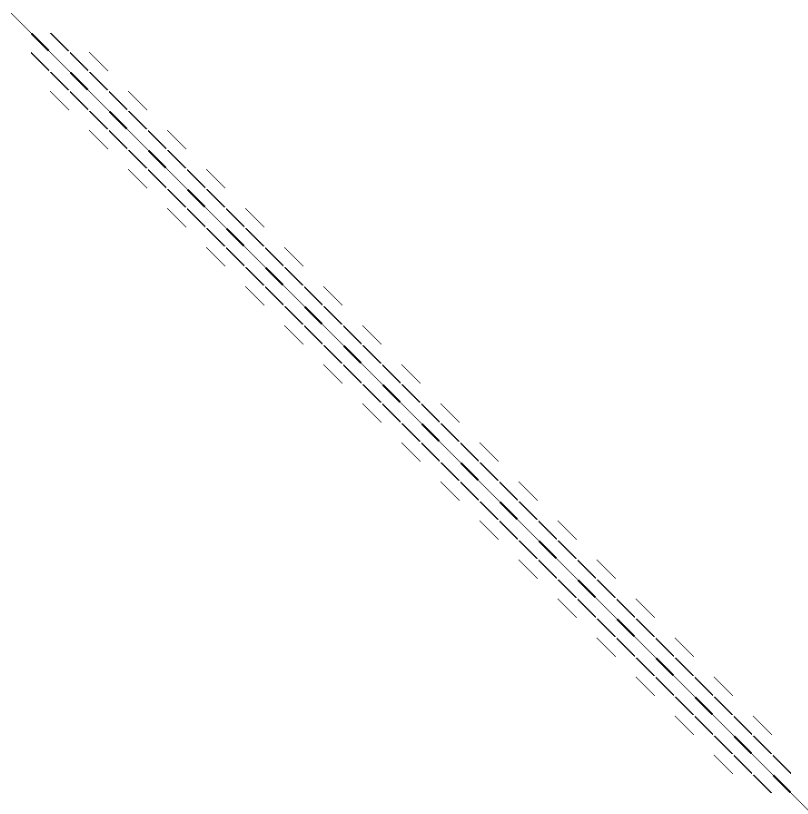


Рисунок 6.9 – Портрет СЛАУ (881 неизвестных) без применения дерева

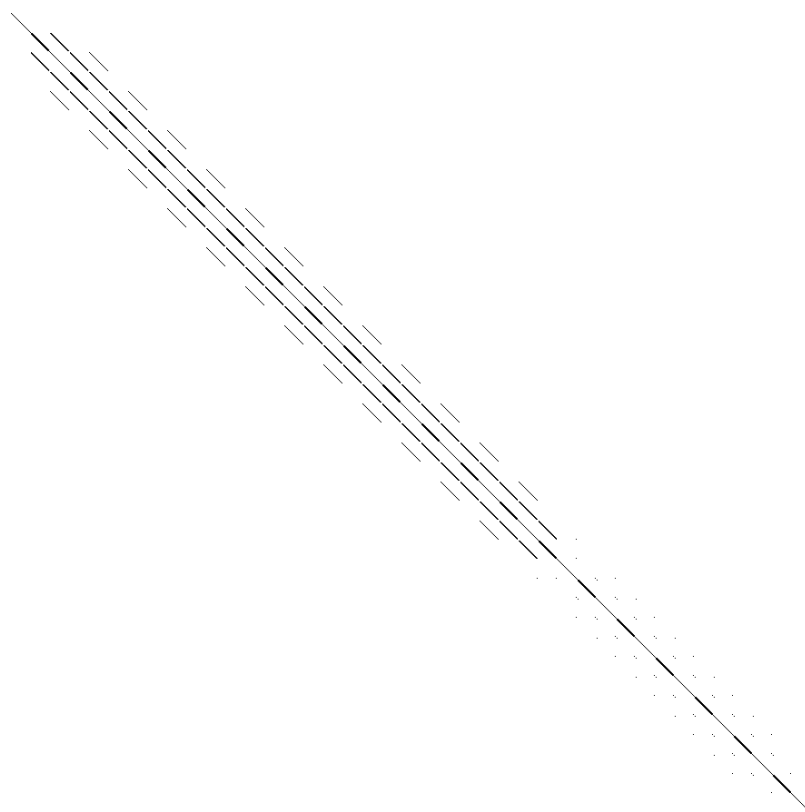


Рисунок 6.10 – Портрет СЛАУ (881 неизвестных) с применением дерева

При количестве неизвестных меньше 2 000 000 метод деревьев-кодереьев не дает никакого эффекта. А при количестве неизвестных больше 2 000 000 уже чувствуется результат, время сокращается на 1-2 секунды.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

- [1] O. Schenk and K. Gartner. Solving unsymmetric sparse systems of linear equations with PARDISO. J. of Future Generation Computer Systems, 20(3):475-487, 2004.
- [2] Intel(R) MKL Reference Manual [Электронный ресурс] – 3254 p. – Режим доступа: [http://software.intel.com/sites/products/documentation/hpc/composerxe/en-us/mklxe/mkl\\_manual\\_win\\_mac/index.htm](http://software.intel.com/sites/products/documentation/hpc/composerxe/en-us/mklxe/mkl_manual_win_mac/index.htm)
- [3] Saad Y., Iterative methods for sparse linear systems – 2nd ed. – SIAM, Philadelphia, – 2003, – 528 p.
- [4] P. Domnikov Reducing the curl-curl matrix bandwidth in 2D magnetostatics using tree-cotree method. Novosibirsk: NSTU, 2019.
- [5] J. Jackson Classical Electrodynamics. University of Illinois, 1962.
- [6] Electromagnetic methods in applied geophysics: Volume 2, Application, Parts A and B. – SEG, 1991.– 988 p.
- [7] P. Monk Finite Element Methods for Maxwell's Equations. Oxford science publications - Newark, 2003.
- [8] A. Bossavit, Computational electromagnetism: variational formulations, complementarity, edge elements. Academic Press, 1998.
- [9] М.С. Жданов, И.М. Варенцов, Н.Г. Голубев, В.А. Крылов. Методы моделирования электромагнитных полей (Материалы международного проекта COMMEMI). Наука, 1990.
- [10] Жданов М.С. Электроразведка / М.С.Жданов – М.: Недра, 1986.
- [11] Ю.Г. Соловейчик, М.Э. Рояк, М.Г. Персова. Метод конечных элементов для решения скалярных и векторных задач. Учебное пособие. - Новосибирск: НГТУ, 2007.

# ПРИЛОЖЕНИЕ 1. Реализованные для задачи структуры

```
1 // % ***** Структура объекта ***** % //
2 public struct Item
3 {
4     public Vector<double> Begin { get; set; } // Нижняя-Левая точка объекта
5     public Vector<double> End { get; set; } // Верхняя-Правая точка объекта
6     public int Nx { get; set; } // Количество разбиений по Оси X
7     public int Nz { get; set; } // Количество разбиений по Оси Z
8     public string Name { get; set; } // Имя объекта
9     public double Sigma { get; set; } // Значение проводимости
10
11     //: Конструктор
12     public Item(Vector<double> begin, Vector<double> end, int nx, int nz,
13         double sigma, string name) { }
14 }
15
16 // % ***** Структура узла ***** % //
17 public struct Node
18 {
19     public double X { get; set; } // Координата X
20     public double Z { get; set; } // Координата Z
21
22     public Node(double _X, double _Z) {
23         (X, Z) = (_X, _Z);
24     }
25
26     public void Deconstruct(out double x,
27         out double z) {
28         (x, z) = (X, Z);
29     }
30 }
31
32 // % ***** Структура ребра ***** % //
33 public struct Edge
34 {
35     public Node NodeBegin { get; set; } // Начальный узел ребра
36     public Node NodeEnd { get; set; } // Конечный узел ребра
37
38     public Edge(Node _begin, Node _end) {
39         NodeBegin = _begin;
40         NodeEnd = _end;
41     }
42
43     public void Deconstruct(out Node begin,
```

```

44         out Node end) {
45             (begin, end) = (NodeBegin, NodeEnd);
46         }
47     }
48
49     // % ***** Структура конечного элемента ***** % //
50     public struct Elem
51     {
52         public int[] Node;           // Номера узлов конечного элемента
53         public int[] Edge;           // Номера ребер конечного элемента
54         public int Material { get; set; } // Номер материала
55
56         public Elem(params int[] node) {
57             Node = node;
58         }
59
60         public void Deconstruct(out int[] nodes,
61                                 out int[] edges) {
62             nodes = Node;
63             edges = Edge;
64         }
65     }
66
67     // % ***** Структура краевого ***** % //
68     public struct Bound
69     {
70         public int Edge { get; set; } // Номер ребра
71         public int NumBound { get; set; } // Номер краевого
72         public int NumSide { get; set; } // Номер стороны
73
74         public Bound(int num, int side, int edge) {
75             NumBound = num;
76             NumSide = side;
77             Edge = edge;
78         }
79
80         public void Deconstruct(out int num, out int side, out int edge) {
81             num = NumBound;
82             side = NumSide;
83             edge = Edge;
84         }
85     }
86
87     // % ***** Структура слоя ***** % //
88     public struct Layer
89     {
90         public double Z { get; set; } // Координата слоя
91         public double Sigma { get; set; } // Значение проводимости
92
93         public Layer(double z, double sigma) {

```

```
94         this.Z = z;  
95         this.Sigma = sigma;  
96     }  
97 }
```

---

## ПРИЛОЖЕНИЕ 2. Составление портрета и глобальной матрицы

```
1  /// Класс МКЭ
2  public class FEM
3  {
4      /// Поля и свойства
5      private List<Node>                Nodes;    /// Узлы
6      private List<Edge>                Edges;    /// Ребра
7      private List<Elem>                Elems;    /// КЭ
8      private List<Bound>               Bounds;   /// Краевые
9      private List<Item>                Items;    /// Объекты
10     private List<Layer>               Layers;    /// Слои
11     private List<(double Sigma1D, double Sigma2D)> Sigmas; /// Значения проводимости
12     private SLAU slau;                /// Структура СЛАУ
13
14     public double                      Nu          { get; set; }    /// Частота тока
15     public double                      W => 2.0 * PI * Nu;          /// Круговая частота
16     public Harm1D                      harm1D      { get; set; }    /// Структура одномерной задачи
17
18     /// Конструктор FEM
19     public FEM(Grid grid, Harm1D harm1D) {
20         (Nodes, Edges, Elems, Bounds, Items, Layers, Sigmas) = grid;
21         this.harm1D = harm1D;
22         this.slau = new SLAU();
23     }
24
25     /// Метод составления СЛАУ
26     public SLAU CreateSLAU() {
27         portrait();          /// Составление портрета
28         global();            /// Составление глобальной матрицы
29         return slau;
30     }
31
32     /// Составление портрета
33     private void portrait() {
34
35         /// Генерируем массивы ig и jg и размерность
36         GenPortrait(ref slau.ig, ref slau.jg, Elems.ToArray());
37         slau.N = Edges.Count;
38
39         /// Выделяем память
40         slau.gg      = new ComplexVector(slau.ig[slau.N]);
41         slau.di      = new ComplexVector(slau.N);
42         slau.pr      = new ComplexVector(slau.N);
43         slau.q        = new ComplexVector(slau.N);
```



```

44     }
45
46     ///Составление глобальной матрицы
47     private void global() {
48
49         ///Обходим конечные элементы
50         for (int index_fin_el = 0; index_fin_el < Elems.Count; index_fin_el++) {
51
52             ///Составляем локальную матрицу и локальный вектор
53             (ComplexMatrix loc_mat, ComplexVector local_f) = local(index_fin_el);
54
55             ///Заносим в глобальную матрицу
56             EntryMatInGlobalMatrix(loc_mat, Elems[index_fin_el].Edge);
57             EntryVecInGlobalMatrix(local_f, Elems[index_fin_el].Edge);
58         }
59
60         ///Обходим краевые
61         for (int index_kraev = 0; index_kraev < Bounds.Count; index_kraev++) {
62             Bound kraev = Bounds[index_kraev];
63             if (kraev.NumBound == 1)
64                 MainKraev(kraev); ///главное краевое
65             else if (kraev.NumBound == 2)
66                 NaturalKraev(kraev); ///естественное краевое
67         }
68     }
69
70     ///Построение локальной матрицы и вектора
71     private (ComplexMatrix, ComplexVector) local(int index_fin_el) {
72
73         ///Подсчет компонент
74         double hx = Nodes[Elems[index_fin_el].Node[1]].X - Nodes[Elems[index_fin_el].Node[0]].X;
75         double hy = Nodes[Elems[index_fin_el].Node[2]].Y - Nodes[Elems[index_fin_el].Node[0]].Y;
76
77         ///Построение матрицы жесткости (G)
78         Matrix<double> G = build_G(index_fin_el, hx, hy);
79
80         ///Построение матрицы массы (M)
81         ComplexMatrix M = build_M(index_fin_el, hx, hy);
82
83         ///Построение локальной правой части
84         ComplexVector local_f = build_F(index_fin_el, hx, hy);
85
86         ///Построение локальной матрицы
87         ComplexMatrix local_matrix = G + new Complex(0, 1) * M;
88
89         return (local_matrix, local_f);
90     }
91
92     ///Построение матрицы жесткости (G)
93     private Matrix<double> build_G(int index_fin_el, double hx, double hy) {

```

```

94
95 // Подсчет коэффициентов для основной задачи
96 double coef_y_on_x = hy / hx;
97 double coef_x_on_y = hx / hy;
98 double coef_nu = 1.0 / Nu0;
99
100 // Матрица жесткости
101 var G_matrix = new Matrix<double>(new double[4, 4]{
102     { 1, -1, -1, 1},
103     {-1, 1, 1, -1},
104     {-1, 1, 1, -1},
105     { 1, -1, -1, 1}
106 });
107
108 // Умножение на coef_y_on_x
109 for (int i = 0; i < 2; i++)
110     for (int j = 0; j < 2; j++)
111         G_matrix[i, j] *= coef_y_on_x;
112
113 // Умножение на coef_x_on_y
114 for (int i = 2; i < 4; i++)
115     for (int j = 2; j < 4; j++)
116         G_matrix[i, j] *= coef_x_on_y;
117
118 return coef_nu * G_matrix;
119 }
120
121 //: Построение матрицы масс (M)
122 private ComplexMatrix build_M(int index_fin_el, double hx, double hy) {
123
124     // Подсчет коэффициента для основной задачи
125     double coef = (W * Sigmas[Elements[index_fin_el].Material - 1].Sigma2D * hx * hy) / 6.0;
126
127     // Матрица масс
128     var M_matrix = new ComplexMatrix(new Complex[4, 4]{
129         {2, 1, 0, 0},
130         {1, 2, 0, 0},
131         {0, 0, 2, 1},
132         {0, 0, 1, 2}
133     });
134
135     return coef * M_matrix;
136 }
137
138 //: Построение вектора правой части (F)
139 private ComplexVector build_F(int index_fin_el, double hx, double hy) {
140
141     // Подсчет коэффициента
142     double coef = (hx * hy) / 6.0;
143

```

```

144 // Матрица масс
145 var M_matrix = new ComplexMatrix(new Complex[4, 4]{
146     {2, 1, 0, 0},
147     {1, 2, 0, 0},
148     {0, 0, 2, 1},
149     {0, 0, 1, 2}
150 });
151 M_matrix = coef * M_matrix;
152
153 // ***** Для задачи основной ***** //
154
155 // Вычисление вектора-потенциала
156 Complex Fcoef = -new Complex(0, 1) * W *
157 (Sigmas[Elemes[index_fin_el].Material - 1].Sigma2D -
158 Sigmas[Elemes[index_fin_el].Material - 1].Sigma1D);
159
160 var f = new ComplexVector(4);
161 ComplexVector.Fill(f, new Complex(0, 0));
162
163 if (Abs(Sigmas[Elemes[index_fin_el].Material - 1].Sigma2D -
164     Sigmas[Elemes[index_fin_el].Material - 1].Sigma1D) <= 1e-10) return f;
165
166 // Боковые ребра = (0,0)
167
168 // Находим значение нижнего ребра
169 f[2] = F(Edges[Elemes[index_fin_el].Edge[2]]);
170
171 // Находим значение верхнего ребра
172 f[3] = F(Edges[Elemes[index_fin_el].Edge[3]]);
173
174 // Умножение на коэффициент
175 f = Fcoef * f;
176
177 // ***** //
178
179 return M_matrix * f;
180 }
181
182 private Complex F(Edge edge) {
183
184     // Строим узел
185     Node node = edge.NodeBegin.Y == edge.NodeEnd.Y ?
186         new Node((edge.NodeBegin.X + edge.NodeEnd.X) / 2.0, edge.NodeBegin.Y) :
187         new Node(edge.NodeBegin.X, (edge.NodeBegin.Y + edge.NodeEnd.Y) / 2.0);
188
189     // Находим узлы из одномерной задачи
190     int id1 = 0, id2 = 0;
191     for (int i = 0; i < harm1D.Nodes.Count - 1; i++) {
192         if (node.Y >= harm1D.Nodes[i].Y && node.Y <= harm1D.Nodes[i + 1].Y) {
193             id1 = i;

```

```

194         id2 = i + 1;
195     }
196 }
197
198 if (id1 == id2)
199     return harm1D.U[1];
200
201 // Находим коэффициенты прямой
202 Complex k = (harm1D.U[id2] - harm1D.U[id1]) /
203             (harm1D.Nodes[id2].Y - harm1D.Nodes[id1].Y);
204 Complex b = harm1D.U[id2] - k * harm1D.Nodes[id2].Y;
205
206 return k * node.Y + b;
207 }
208
209 //: Занесение матрицы в глобальную матрицу
210 private void EntryMatInGlobalMatrix(ComplexMatrix mat, int[] index) {
211     for (int i = 0, h = 0; i < mat.Rows; i++) {
212         int ibeg = index[i];
213         //if (!Bounds.Exists(n => n.Edge == ibeg)) {
214             for (int j = i + 1; j < mat.Columns; j++)
215             {
216                 int iend = index[j];
217                 int temp = ibeg;
218
219                 if (temp < iend)
220                     (iend, temp) = (temp, iend);
221
222                 h = slau.ig[temp];
223                 while (slau.jg[h++] - iend != 0) ;
224                 --h;
225                 slau.gg[h] += mat[i, j];
226             }
227             slau.di[ibeg] += mat[i, i];
228         //}
229     }
230 }
231
232 //: Занесение вектора в глобальный вектор
233 private void EntryVecInGlobalMatrix(ComplexVector vec, int[] index) {
234     for (int i = 0; i < vec.Length; i++)
235         slau.pr[index[i]] += vec[i];
236 }
237
238 //: Учет главного краевого условия
239 private void MainKraev(Bound bound) {
240
241     // Номер ребра и значение краевого основной задачи
242     (int row, Complex value) = (bound.Edge, new Complex(0, 0));
243

```

```

244 // Учет краевого
245 slau.di[row] = new Complex(1, 0);
246 slau.pr[row] = value;
247
248 // Зануляем в треугольнике (столбцы)
249 for (int i = slau.ig[row]; i < slau.ig[row + 1]; i++) {
250     slau.pr[slau.jg[i]] -= slau.gg[i] * value;
251     slau.gg[i] = 0;
252 }
253
254 // Зануляем в треугольнике (строки)
255 for (int i = row + 1; i < slau.N; i++) {
256     for (int j = slau.ig[i]; j < slau.ig[i + 1]; j++) {
257         if (slau.jg[j] == row) {
258             slau.pr[i] -= slau.gg[j] * value;
259             slau.gg[j] = 0;
260         }
261     }
262 }
263 }
264
265 //: Учет естественного краевого условия
266 private void NaturalKraev(Bound kraev) { }
267
268 //: Установка параметров для МКЭ
269 public bool TrySetParameter(string name, double value) {
270     if (name == nameof(Nu)) {
271         Nu = value;
272         return true;
273     }
274     return false;
275 }
276
277 //: Генерация ig, jg (портрета)
278 public void GenPortrait(ref Vector<int> ig, ref Vector<int> jg, Elem[] elems) {
279
280     var connectivityList = new List<HashSet<int>>();
281
282     for (int i = 0; i < countEdge; i++)
283         connectivityList.Add(new());
284
285     int localSize = elems[0].Edge.Count();
286
287     foreach (var element in elems.Select(element => element.Edge.OrderBy(n => n).ToArray()))
288         for (int i = 0; i < localSize - 1; i++) {
289             int nodeToInsert = element[i];
290             for (int j = i + 1; j < localSize; j++) {
291                 int posToInsert = element[j];
292                 connectivityList[posToInsert].Add(nodeToInsert);
293             }

```

```
294     }
295
296     var orderedList = connectivityList.Select(list => list.OrderBy(val => val)).ToList();
297
298     ig = new int[connectivityList.Count + 1];
299
300     ig[0] = 0;
301     ig[1] = 0;
302
303     for (int i = 1; i < connectivityList.Count; i++)
304         ig[i + 1] = ig[i] + connectivityList[i].Count;
305
306     jg = new int[ig[^1]];
307
308     for (int i = 1, j = 0; i < connectivityList.Count; i++)
309         foreach (var it in orderedList[i])
310             jg[j++] = it;
311     }
312 }
```

## ПРИЛОЖЕНИЕ 3. Генерация сетки

```
1 public partial class MainWindow : Window
2 {
3
4     ///Генерация узлов сетки
5     private List<Node> generate_node() {
6
7         ///Слой границ объектов
8         List<double> border_X = new List<double>(2 * items.Count)
9         { items[0].Begin[0], items[0].End[0] };
10
11         List<double> border_Y = new List<double>(2 * items.Count)
12         { items[0].Begin[1], items[0].End[1] };
13
14         double X_min = items[0].Begin[0], Y_min = items[0].Begin[1];
15         double X_max = items[0].End[0], Y_max = items[0].End[1];
16         for (int i = 1; i < items.Count; i++) {
17             if (items[i].Begin[0] < X_min)
18                 X_min = items[i].Begin[0];
19             if (items[i].Begin[1] < Y_min)
20                 Y_min = items[i].Begin[1];
21             if (items[i].End[0] > X_max)
22                 X_max = items[i].End[0];
23             if (items[i].End[1] > Y_max)
24                 Y_max = items[i].End[1];
25
26             border_X.Add(items[i].Begin[0]);
27             border_X.Add(items[i].End[0]);
28             border_Y.Add(items[i].Begin[1]);
29             border_Y.Add(items[i].End[1]);
30         }
31         double copy_Y_min = Y_min;
32         double copy_X_min = X_min;
33
34         border_X = border_X.OrderByDescending(n => n).Distinct().ToList();
35         border_Y = border_Y.OrderByDescending(n => n).Distinct().ToList();
36
37         ///Сортировка по X
38         items = items.OrderBy(n => n.Begin[0]).ToList();
39
40         ///Список индексов объектов (нужен чтобы пропускать пройденные объекты)
41         List<int> index_list = new List<int>() { 0 };
42
43         ///Составляем листы шагов X
44         List<double> H_Axe_X = new List<double>();
45
```

```

46 // Считаем шаг по объекту
47 double Hx = (items[0].End[0] - items[0].Begin[0]) / items[0].Nx;
48 double X_temp_max = items[0].End[0];
49
50 // Генерация шагов по оси X
51 while (X_min < X_max) {
52
53     while (X_min < X_temp_max && Abs(X_temp_max - X_min) > 1e-6) {
54         X_min += Hx;
55         H_Axe_X.Add(Hx);
56     }
57
58     X_min -= Hx;
59     H_Axe_X.RemoveAt(H_Axe_X.Count - 1);
60     var step = Abs(X_temp_max - X_min);
61     H_Axe_X.Add(step);
62     X_min += step;
63
64     // Проверяем входит ли он в другой объект
65     bool isFound = false;
66     for (int i = 0; i < items.Count; i++) {
67         if (X_min > items[i].Begin[0] && X_min < items[i].End[0] &&
68             !index_list.Contains(i)) {
69             index_list.Add(i);
70             isFound = true;
71             Hx = (items[i].End[0] - items[i].Begin[0]) / items[i].Nx;
72             X_temp_max = items[i].End[0];
73             break;
74         }
75     }
76
77     // Если наступили на границу
78     if (Abs(X_min - X_max) <= 1e-6)
79         break;
80
81     // Если перескочили границу
82     if (X_min > X_max) {
83         X_min -= Hx;
84         H_Axe_X.RemoveAt(H_Axe_X.Count - 1);
85         H_Axe_X.Add(Abs(items[^1].End[0] - X_min));
86         break;
87     }
88
89     while (!isFound) {
90         X_min += Hx;
91         H_Axe_X.Add(Hx);
92
93         for (int i = 0; i < items.Count; i++) {
94             if (X_min > items[i].Begin[0] && X_min < items[i].End[0] &&
95                 !index_list.Contains(i)) {

```



```

96
97         X_min -= Hx;
98         H_Axe_X.RemoveAt(H_Axe_X.Count - 1);
99         step = Abs(items[i].Begin[0] - X_min);
100         if (step > min_step)
101             H_Axe_X.Add(step);
102         X_min += step;
103
104         index_list.Add(i);
105         isFound = true;
106         Hx = (items[i].End[0] - items[i].Begin[0]) / items[i].Nx;
107         X_temp_max = items[i].End[0];
108         break;
109     }
110 }
111 }
112 }
113
114 // Сортировка по Y
115 items = items.OrderBy(n => n.Begin[1]).ToList();
116
117 // Составляем листы шагов Y
118 List<double> H_Axe_Y = new List<double>();
119
120 // Находим объект
121 index_list.Clear();
122 index_list.Add(0);
123
124 // Считаем шаг по объекту
125 double Hy = (items[0].End[1] - items[0].Begin[1]) / items[0].Ny;
126 double Y_temp_max = items[0].End[1];
127
128 // Генерация шагов по оси Y
129 while (Y_min < Y_max) {
130
131     while (Y_min < Y_temp_max && Abs(Y_temp_max - Y_min) > 1e-6) {
132         Y_min += Hy;
133         H_Axe_Y.Add(Hy);
134     }
135
136     Y_min -= Hy;
137     H_Axe_Y.RemoveAt(H_Axe_Y.Count - 1);
138     var step = Abs(Y_temp_max - Y_min);
139     H_Axe_Y.Add(step);
140     Y_min += step;
141
142     // Проверяем входит ли он в другой объект
143     bool isFound = false;
144     for (int i = 0; i < items.Count; i++) {
145         if (Y_min > items[i].Begin[1] && Y_min < items[i].End[1] &&

```

```

146         !index_list.Contains(i)) {
147             index_list.Add(i);
148             isFound = true;
149             Hy = (items[i].End[1] - items[i].Begin[1]) / items[i].Ny;
150             Y_temp_max = items[i].End[1];
151             break;
152         }
153     }
154
155     // Если наступили на границу
156     if (Abs(Y_min - Y_max) <= 1e-6)
157         break;
158
159     // Если перескочили границу
160     if (Y_min > Y_max) {
161         Y_min -= Hy;
162         H_Axe_Y.RemoveAt(H_Axe_Y.Count - 1);
163         H_Axe_Y.Add(Abs(items[^1].End[1] - Y_min));
164         break;
165     }
166
167     while (!isFound) {
168         Y_min += Hy;
169         H_Axe_Y.Add(Hy);
170
171         for (int i = 0; i < items.Count; i++) {
172             if (Y_min > items[i].Begin[1] && Y_min < items[i].End[1] &&
173                 !index_list.Contains(i)) {
174
175                 Y_min -= Hy;
176                 H_Axe_Y.RemoveAt(H_Axe_Y.Count - 1);
177                 step = Abs(items[i].Begin[1] - Y_min);
178                 if (step > min_step)
179                     H_Axe_Y.Add(step);
180                 Y_min += step;
181
182                 index_list.Add(i);
183                 isFound = true;
184                 Hy = (items[i].End[1] - items[i].Begin[1]) / items[i].Ny;
185                 Y_temp_max = items[i].End[1];
186                 break;
187             }
188         }
189     }
190 }
191
192 // Учет границ Y
193 H_Axe_Y.Insert(0, 0);
194 double Y_temp = copy_Y_min;
195 int id = 1;

```

```

196 List<(int index, double val_layer, double val)> new_Axe =
197         new List<(int, double, double)>();
198 for (int i = 0; i < H_Axe_Y.Count && id != border_Y.Count + 1; i++) {
199
200     // Прибавляем шаг
201     Y_temp += H_Axe_Y[i];
202
203     // Если равно значит узлы будут стоять на узлы, переходим у следующему слою
204     if (Abs(Y_temp - border_Y[id]) <= 1e-6) {
205         id++;
206         continue;
207     }
208
209     // Если перескочили слой, значит добавим шаг в чтобы задеть слой
210     if (Y_temp > border_Y[id]) {
211         new_Axe.Add((i + new_Axe.Count, border_Y[id], Y_temp - H_Axe_Y[i]));
212         id++;
213     }
214 }
215
216 // Корректировка шагов по Оси Y
217 for (int i = 0; i < new_Axe.Count; i++) {
218     double step = Abs(new_Axe[i].val_layer - new_Axe[i].val);
219     H_Axe_Y.Insert(new_Axe[i].index, step);
220     H_Axe_Y[new_Axe[i].index + 1] = Abs(H_Axe_Y[new_Axe[i].index + 1] - step);
221 }
222
223 // Корректировка шагов, чтобы не было узких прямоугольников
224 List<int> index_rem = new List<int>();
225 for (int i = 1; i < H_Axe_Y.Count - 1; i++) {
226     if (H_Axe_Y[i] <= min_step) {
227         H_Axe_Y[i - 1] += H_Axe_Y[i];
228         index_rem.Add(i);
229     }
230 }
231
232 // Удлаение не подходящих шагов
233 for (int i = 0; i < index_rem.Count; i++) {
234     H_Axe_Y.RemoveAt(index_rem[i]);
235     index_rem = index_rem.Select(n => n - 1).ToList();
236 }
237 H_Axe_Y.RemoveAt(0);
238
239 // Учет границ X
240 H_Axe_X.Insert(0, 0);
241 double X_temp = copy_X_min;
242 id = 1;
243 new_Axe = new List<(int, double, double)>();
244 for (int i = 0; i < H_Axe_X.Count && id != border_X.Count + 1; i++) {
245

```

```

246         // Прибавляем шаг
247         X_temp += H_Axe_X[i];
248
249         // Если равно значит узлы будут стоять на узлы, переходим у следующему слою
250         if (Abs(X_temp - border_X[id]) <= 1e-6) {
251             id++;
252             continue;
253         }
254
255         // Если перескочили слой, значит добавим шаг в чтобы задеть слой
256         if (X_temp > border_X[id]) {
257             new_Axe.Add((i + new_Axe.Count, border_X[id], X_temp - H_Axe_X[i]));
258             id++;
259         }
260     }
261
262     // Корректировка шагов по Оси Y
263     for (int i = 0; i < new_Axe.Count; i++) {
264         double step = Abs(new_Axe[i].val_layer - new_Axe[i].val);
265         H_Axe_X.Insert(new_Axe[i].index, step);
266         H_Axe_X[new_Axe[i].index + 1] = Abs(H_Axe_X[new_Axe[i].index + 1] - step);
267     }
268
269     // Корректировка шагов, чтобы не было узких прямоугольников
270     index_rem = new List<int>();
271     for (int i = 1; i < H_Axe_X.Count - 1; i++){
272         if (H_Axe_X[i] < min_step){
273             H_Axe_X[i - 1] += H_Axe_X[i];
274             index_rem.Add(i);
275         }
276     }
277
278     // Удлаение не подходящих шагов
279     for (int i = 0; i < index_rem.Count; i++) {
280         H_Axe_X.RemoveAt(index_rem[i]);
281         index_rem = index_rem.Select(n => n - 1).ToList();
282     }
283     H_Axe_X.RemoveAt(0);
284
285     // Генерация шагов по оси X от объекта
286     double temp_V = copy_X_min;
287     double temp_H = H_Axe_X[0];
288     while (temp_V >= Begin_BG[0]) {
289         temp_H *= Kx;
290         H_Axe_X.Insert(0, temp_H);
291         temp_V -= temp_H;
292         if (temp_V < Begin_BG[0] && IsStrictGrid) {
293             temp_H -= (Begin_BG[0] - temp_V);
294             H_Axe_X[0] = temp_H;
295         }

```

```

296     }
297     double X = IsStrictGrid ? Begin_BG[1] : temp_V;
298
299     temp_V = X_max;
300     temp_H = H_Axe_X[^1];
301     while (temp_V <= End_BG[0]) {
302         temp_H *= Kx;
303         H_Axe_X.Add(temp_H);
304         temp_V += temp_H;
305         if (temp_V > End_BG[0] && IsStrictGrid) {
306             temp_H -= (temp_V - End_BG[0]);
307             H_Axe_X[^1] = temp_H;
308         }
309     }
310
311     // Генерация шагов по оси Y от объекта
312     temp_V = copy_Y_min;
313     temp_H = H_Axe_Y[0];
314     while (temp_V >= Begin_BG[1]) {
315         temp_H *= Ky;
316         H_Axe_Y.Insert(0, temp_H);
317         temp_V -= temp_H;
318         if (temp_V < Begin_BG[1] && IsStrictGrid) {
319             temp_H -= (Begin_BG[1] - temp_V);
320             H_Axe_Y[0] = temp_H;
321         }
322     }
323     double Y = IsStrictGrid ? Begin_BG[0] : temp_V;
324
325     temp_V = Y_max;
326     temp_H = H_Axe_Y[^1];
327     while (temp_V <= End_BG[1]) {
328         temp_H *= Ky;
329         H_Axe_Y.Add(temp_H);
330         temp_V += temp_H;
331         if (temp_V > End_BG[1] && IsStrictGrid) {
332             temp_H -= (temp_V - End_BG[1]);
333             H_Axe_Y[^1] = temp_H;
334         }
335     }
336
337     H_Axe_X.Insert(0, 0);
338     H_Axe_Y.Insert(0, 0);
339
340     // Учет горизонтальных слоев
341     if (layers.Count > 0) {
342         Y_temp = Y;
343         id = 1;
344         new_Axe = new List<(int, double, double)>();
345         for (int i = 0; i < H_Axe_Y.Count && id != layers.Count + 1; i++) {

```

```

346
347 // Прибавляем шаг
348 Y_temp += H_Axe_Y[i];
349
350 // Если равно значит узлы будут стоять на узлы, переходим у следующему слою
351 if (Abs(Y_temp - layers[~id].Y) <= 1e-6) {
352     id++;
353     continue;
354 }
355
356 // Если перескочили слой, значит добавим шаг в чтобы задеть слой
357 if (Y_temp > layers[~id].Y) {
358     new_Axe.Add((i + new_Axe.Count, layers[~id].Y, Y_temp - H_Axe_Y[i]));
359     id++;
360 }
361 }
362
363 // Корректировка шагов по Оси Y
364 for (int i = 0; i < new_Axe.Count; i++) {
365     double step = Abs(new_Axe[i].val_layer - new_Axe[i].val);
366     H_Axe_Y.Insert(new_Axe[i].index, step);
367     H_Axe_Y[new_Axe[i].index + 1] = Abs(H_Axe_Y[new_Axe[i].index + 1] - step);
368 }
369 }
370
371 // Корректировка шагов, чтобы не было узких прямоугольников
372 index_rem = new List<int>();
373 for (int i = 1; i < H_Axe_Y.Count - 1; i++) {
374     if (H_Axe_Y[i] < e) {
375         H_Axe_Y[i - 1] += H_Axe_Y[i];
376         index_rem.Add(i);
377     }
378 }
379
380 // Удлаение не подходящих шагов
381 for (int i = 0; i < index_rem.Count; i++) {
382     H_Axe_Y.RemoveAt(index_rem[i]);
383     index_rem = index_rem.Select(n => n - 1).ToList();
384 }
385
386 // Сколько будет линий на графике (Количество узлов на Осях)
387 CountX = H_Axe_X.Count;
388 CountY = H_Axe_Y.Count;
389
390 // Генерация узлов
391 Node[] nodes = new Node[CountX * CountY];
392 double Y_new = Y, X_new;
393 id = 0;
394 for (int i = 0; i < H_Axe_Y.Count; i++) {
395     Y_new += H_Axe_Y[i];

```

```

396         X_new = X;
397         for (int j = 0; j < H_Axe_X.Count; j++, id++) {
398             X_new += H_Axe_X[j];
399             nodes[id] = new Node(X_new, Y_new);
400         }
401     }
402
403     return new List<Node>(nodes);
404 }
405
406 //: Генерация конечных элементов
407 private List<Elem> generate_elem() {
408
409     Elem[] elems = new Elem[(CountX - 1) * (CountY - 1)];
410
411     for (int i = 0, id = 0; i < CountY - 1; i++)
412         for (int j = 0; j < CountX - 1; j++, id++) {
413             elems[id] = new Elem(
414                 i * CountX + j,
415                 i * CountX + j + 1,
416                 (i + 1) * CountX + j,
417                 (i + 1) * CountX + j + 1
418             );
419         }
420
421     return new List<Elem>(elems);
422 }
423
424 //: Генерация ребер
425 private List<Edge> generate_edge(List<Elem> elems, List<Node> nodes) {
426
427     Edge[] edges = new Edge[CountX * (CountY - 1) + CountY * (CountX - 1)];
428
429     for (int i = 0; i < CountY - 1; i++)
430         for (int j = 0; j < CountX - 1; j++) {
431             int left = i * ((CountX - 1) + CountX) + (CountX - 1) + j;
432             int right = i * ((CountX - 1) + CountX) + (CountX - 1) + j + 1;
433             int bottom = i * ((CountX - 1) + CountX) + j;
434             int top = (i + 1) * ((CountX - 1) + CountX) + j;
435             int n_elem = i * (CountX - 1) + j;
436
437             edges[left] = new Edge(nodes[elems[n_elem].Node[0]], nodes[elems[n_elem].Node[2]]);
438             edges[right] = new Edge(nodes[elems[n_elem].Node[1]], nodes[elems[n_elem].Node[3]]);
439             edges[bottom] = new Edge(nodes[elems[n_elem].Node[0]], nodes[elems[n_elem].Node[1]]);
440             edges[top] = new Edge(nodes[elems[n_elem].Node[2]], nodes[elems[n_elem].Node[3]]);
441
442             elems[n_elem] = elems[n_elem] with { Edge = new[] { left, right, bottom, top },
443                 Material = Helper.GetMaterial(layers, items, nodes, elems[n_elem]) };
444         }
445

```

```

446         return new List<Edge>(edges);
447     }
448
449     ///Генерация краевых
450     private List<Bound> generate_bound(List<Edge> edge) {
451
452         Bound[] bounds = new Bound[2 * (CountX - 1) + 2 * (CountY - 1)];
453         int id = 0;
454
455         ///Нижняя сторона
456         for (int i = 0; i < CountX - 1; i++, id++)
457             bounds[id] = new Bound(
458                 SideBound![0],
459                 0,
460                 i
461             );
462
463         ///Правая сторона
464         for (int i = 1; i < CountY; i++, id++)
465             bounds[id] = new Bound(
466                 SideBound![1],
467                 1,
468                 i * CountX + i * (CountX - 1) - 1
469             );
470
471         ///Верхняя сторона
472         for (int i = 0; i < CountX - 1; i++, id++)
473             bounds[id] = new Bound(
474                 SideBound![2],
475                 2,
476                 CountX * (CountY - 1) + (CountX - 1) * (CountY - 1) + i
477             );
478
479         ///Левая сторона
480         for (int i = 0; i < CountY - 1; i++, id++)
481             bounds[id] = new Bound(
482                 SideBound![3],
483                 3,
484                 (i + 1) * (CountX - 1) + i * CountX
485             );
486
487         ///Сортируем по номеру краевого
488         bounds = bounds.OrderByDescending(n => n.NumBound).ToArray();
489
490         return new List<Bound>(bounds);
491     }
492
493     ///Генерация проводимости
494     private List<(double Sigma1D, double Sigma2D)> generate_sigma() {
495

```



```

496         // Воздух
497         List<(double Sigma1D, double Sigma2D)> sigmas =
498             new List<(double Sigma1D, double Sigma2D)>(new(double, double)[] { (0, 0) });
499
500         for (int i = 0; i < layers.Count; i++) {
501             sigmas.Add((layers[i].Sigma, layers[i].Sigma));
502
503             // Ищем индексы объектов которые находятся в слое
504             int[] id = GetIdItems(items, layers, i);
505             for (int j = 0; j < id.Length; j++)
506                 sigmas.Add((layers[i].Sigma, items[id[j]].Sigma));
507         }
508
509         return sigmas;
510     }
511 }

```