

# СОДЕРЖАНИЕ

<b>Задание . . . . .</b>	<b>3</b>
<b>1. Теоретическая часть . . . . .</b>	<b>4</b>
1.1. Векторные дифференциальные уравнения второго порядка с разрывными решениями . . . . .	4
1.2. Вариационная постановка . . . . .	5
1.3. Конечноэлементная дискретизация . . . . .	6
1.4. Построение матриц масс, жёсткости и вектора правой части на шестигранниках . . . . .	7
1.5. Учёт краевых условий . . . . .	8
<b>2. Практическая часть. . . . .</b>	<b>10</b>
2.1. Генерация трёхмерной сетки с ячейками в виде шестигранников	10
2.2. Численное интегрирование. . . . .	14
2.3. Процесс построения локальных матриц жёсткости и масс. . . .	16
2.4. Решение СЛАУ. . . . .	18
<b>Список используемых источников. . . . .</b>	<b>19</b>
<b>Приложение 3. Текст программы . . . . .</b>	<b>20</b>

# ЗАДАНИЕ

Разработка программы для моделирования трехмерного электромагнитного поля на шестигранниках с использованием векторного МКЭ.

# 1. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

## 1.1. ВЕКТОРНЫЕ ДИФФЕРЕНЦИАЛЬНЫЕ УРАВНЕНИЯ ВТОРОГО ПОРЯДКА С РАЗРЫВНЫМИ РЕШЕНИЯМИ

Математическая модель, служащая для описания электромагнитного поля в средах с изменяющимся коэффициентом магнитной проницаемости и в ситуациях, когда нельзя пренебрегать влиянием токов смещения, выглядит следующим образом (1.1):

$$\operatorname{rot} \left( \frac{1}{\mu} \operatorname{rot} \vec{A} \right) + \sigma \frac{\partial \vec{A}}{\partial t} + \epsilon \frac{\partial^2 \vec{A}}{\partial t^2} = \vec{J}^{\text{ст}}. \quad (1.1)$$

Математическая модель электромагнитного поля на основе уравнения (1.1) позволяет решать самые сложные задачи электромагнетизма. Она корректно описывает электромагнитные поля в ситуациях, когда среда содержит любые неоднородности с измененными электрическими и магнитными свойствами.

При решении задач с использованием схемы разделения полей, для описания осесимметричной горизонтально-слоистой среды используется следующее уравнение (1.2):

$$-\frac{1}{\mu_0} \Delta A_\varphi + \frac{A_\varphi}{\mu_0 r^2} + \sigma \frac{\partial A_\varphi}{\partial t} = J_\varphi. \quad (1.2)$$

В свою очередь, учёт от объектов, имеющих неоднородные значения удельной электропроводности, осуществляется за счёт математической модели, описываемой уравнением (1.3)

$$\operatorname{rot} \left( \frac{1}{\mu_0} \operatorname{rot} \vec{\mathbf{A}}^+ \right) + \sigma \frac{\partial \vec{\mathbf{A}}^+}{\partial t} = (\sigma - \sigma_n) \vec{\mathbf{E}}_n. \quad (1.3)$$

Для тестирования на правильность решения дифференциального уравнения (1.3) будем использовать уравнение (1.4), правая часть которого представляется в виде вектор-функции  $\vec{\mathbf{F}}$ , а также будет иметь место быть слагаемое  $\gamma \vec{\mathbf{A}}$  в левой части уравнения:

$$\operatorname{rot} \left( \frac{1}{\mu_0} \operatorname{rot} \vec{\mathbf{A}} \right) + \gamma \vec{\mathbf{A}} + \sigma \frac{\partial \vec{\mathbf{A}}}{\partial t} = \vec{\mathbf{F}}. \quad (1.4)$$

## 1.2. ВАРИАЦИОННАЯ ПОСТАНОВКА

Будем считать, что на границе  $S = S_1 \cup S_2$  расчётной области  $\Omega$ , в которой определено уравнение (1.4), заданы краевые условия двух типов:

$$\left( \vec{\mathbf{A}} \times \vec{\mathbf{n}} \right) \Big|_{S_1} = \vec{\mathbf{A}}^g \times \vec{\mathbf{n}}, \quad (1.5)$$

$$\left( \frac{1}{\mu} \operatorname{rot} \vec{\mathbf{A}} \times \vec{\mathbf{n}} \right) \Big|_{S_1} = \vec{\mathbf{H}}^\Theta \times \vec{\mathbf{n}}. \quad (1.6)$$

Тогда эквивалентная вариационная формулировка в форме Галёркина для уравнения (1.4) без производной по времени, и с учётом краевых условий (1.5) - (1.6) имеет вид:

$$\begin{aligned} \int_{\Omega} \frac{1}{\mu_0} \operatorname{rot} \vec{\mathbf{A}} \cdot \operatorname{rot} \vec{\Psi} \, d\Omega + \int_{\Omega} \gamma \vec{\mathbf{A}} \cdot \vec{\Psi} \, d\Omega &= \int_{\Omega} \vec{\mathbf{F}} \cdot \vec{\Psi} \, d\Omega + \\ &+ \int_{S_2} \left( \vec{\mathbf{H}}^\Theta \times \vec{\mathbf{n}} \right) \cdot \vec{\Psi} \, dS \quad \forall \vec{\Psi} \in H_0^{rot}. \end{aligned} \quad (1.7)$$

### 1.3. КОНЕЧНОЭЛЕМЕНТНАЯ ДИСКРЕТИЗАЦИЯ

На шестиграннике базисные вектор-функции удобнее строить с помощью шаблонного элемента. Обычно в качестве такого берут кубик  $[-1, 1] \times [-1, 1] \times [-1, 1]$  при использовании базиса лагранжева или иерархического типа.

Пусть у нас имеется произвольный шестигранник  $\Omega_m$  с вершинами  $(\hat{x}_i, \hat{y}_i, \hat{z}_i), i = 1 \dots 8$ . Тогда отображение шаблонного кубика  $\Omega^E$  в шестигранник  $\Omega_m$  будет задаваться соотношениями:

$$x = \sum_{i=1}^8 \hat{\varphi}_i(\xi, \eta, \zeta) \hat{x}_i, \quad y = \sum_{i=1}^8 \hat{\varphi}_i(\xi, \eta, \zeta) \hat{y}_i, \quad z = \sum_{i=1}^8 \hat{\varphi}_i(\xi, \eta, \zeta) \hat{z}_i, \quad (1.8)$$

где  $\hat{\varphi}_i(\xi, \eta, \zeta)$  - стандартные скалярные трилинейные базисные функции, определённые на шаблонном элементе  $\Omega^E$ .

Отображение базисных вектор-функций  $\hat{\varphi}_i(\xi, \eta, \zeta)$  шаблонного элемента  $\Omega^E$  на шестигранник  $\Omega_m$  можно определить следующим образом:

$$\hat{\psi}_i(x, y, z) = \mathbf{J}^{-1} \hat{\varphi}_i(\xi(x, y, z), \eta(x, y, z), \zeta(x, y, z)), \quad (1.9)$$

где

$$\mathbf{J} = \begin{pmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} & \frac{\partial z}{\partial \xi} \\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta} & \frac{\partial z}{\partial \eta} \\ \frac{\partial x}{\partial \zeta} & \frac{\partial y}{\partial \zeta} & \frac{\partial z}{\partial \zeta} \end{pmatrix} \quad (1.10)$$

- функциональная матрица преобразования координат, переводящего кубик  $\Omega^E$  в шестигранник  $\Omega_m$ .

## 1.4. ПОСТРОЕНИЕ МАТРИЦ МАСС, ЖЁСТКОСТИ И ВЕКТОРА ПРАВОЙ ЧАСТИ НА ШЕСТИГРАННИКАХ

В силу сложной геометрии выпуклых шестигранников, расчёт локальных матриц удобнее проводить на отображении конечного элемента  $\Omega_m$  в мастер-элемент  $\Omega_E$ , представляющий из себя куб размером  $\Omega_E = [-1, 1]_x \times [-1, 1]_y \times [-1, 1]_z$ . Тогда матрица жёсткости будет рассчитываться по формуле (1.11):

$$\begin{aligned} \hat{G}_{ij} &= \int_{\Omega_e} \frac{1}{\mu_0} \text{rot} \hat{\psi}_i \cdot \text{rot} \hat{\psi}_j d\Omega = \\ &= \int_{-1}^1 \int_{-1}^1 \int_{-1}^1 \frac{1}{\mu_0} \frac{1}{|J|} (\mathbf{J}^T \text{rot} \hat{\varphi}_i) \cdot (\mathbf{J}^T \text{rot} \hat{\varphi}_j) d\xi d\eta d\zeta \end{aligned} \quad (1.11)$$

матрица масс, в свою очередь, по формуле (1.12):

$$\hat{M}_{ij} = \int_{\Omega_e} \gamma \hat{\psi}_i \cdot \hat{\psi}_j d\Omega = \int_{-1}^1 \int_{-1}^1 \int_{-1}^1 \gamma (\mathbf{J}^{-1} \hat{\varphi}_i) \cdot (\mathbf{J}^{-1} \hat{\varphi}_j) |J| d\xi d\eta d\zeta \quad (1.12)$$

При расчёте локального вектора правой части будем использовать формулу:

$$\hat{\mathbf{b}}^{\mathbf{J}^{\text{CT}}} = \hat{\mathbf{M}} \hat{\mathbf{f}}. \quad (1.13)$$

она универсальна и применима для любой геометрии конечного элемента и любыми базисными функциями любого порядка на нём.

Для произвольных шестигранников интегралы в соотношениях (1.11) - (1.12) берутся численно. Соответствующая схема вычисления значения инте-

грала от функции  $f(\xi, \eta, \zeta)$  по единичной области  $\Omega_E$  выглядит следующим образом:

$$\int_{-1}^1 \int_{-1}^1 \int_{-1}^1 f(\xi, \eta, \zeta) d\xi d\eta d\zeta \approx \sum_{k=1}^n \sum_{l=1}^n \sum_{r=1}^n f(t_k, t_l, t_r). \quad (1.14)$$

## 1.5. УЧЁТ КРАЕВЫХ УСЛОВИЙ

При решении уравнения (1.2) с использованием векторного МКЭ базисные вектор-функции  $\vec{\psi}_i$  строятся так, что все базисные функции конечномерного пространства  $\mathbf{V}^{\text{rot}}$  с индексами  $i \in N_0$  имеют нулевые касательные к  $S_1$  составляющие (они образуют базис подпространства  $\mathbf{V}_0^{\text{rot}}$ ). Поэтому для выполнения однородных главных краевых условий достаточно к  $n_0$  уравнениям системы (1.7) добавить  $n - n_0$  уравнений вида:

$$q_i = 0, \quad i \in N \setminus N_0. \quad (1.15)$$

Для учёта неоднородного краевого условия (1.5), как и в случае однородного, к  $n_0$  уравнениям добавляется ещё  $n - n_0$  линейных (относительно весов  $q_i$ ) уравнений, которые и должны обеспечить необходимую близость на  $S_1$  касательных составляющих приближённого решения  $\vec{\mathbf{A}}^h = \sum_{j \in N} q_j^n \vec{\psi}_j$  к касательным составляющим вектора  $\vec{\mathbf{A}}^g$ .

Однако для учёта неоднородных краевых условий в векторном МКЭ можно использовать гораздо более простую и удобную для реализации процедуру. Основная её идея заключается в том, что вектор-функция  $\vec{\mathbf{A}}^g$  из правой части краевых условий (1.5) заменяется некоторым интерполянтном  $\vec{\mathbf{A}}^{g,h}$ , представленными в виде линейной комбинации базисных вектор-функций:

$$\vec{\mathbf{A}}^{g,h} = \sum_j q_j^g \vec{\psi}_j. \quad (1.16)$$

Веса  $q_j^g$  в разложении (1.16) можно найти следующим образом. Поскольку в векторном МКЭ базисные функции  $\vec{\psi}_i$  строятся так, что на поверхности  $S_1$  ненулевые касательные имеют только  $n - n_0$  базисных вектор-функций с номерами  $i \in N \setminus N_0$ , при этом для каждой из таких вектор-функций на поверхность  $S_1$  существует точка  $(x_i, y_i, z_i)$  и проходящий через эту точку касательный к поверхности  $S_1$  вектор  $\vec{\tau}_i$  такой, что

$$\vec{\psi}_i(x_i, y_i, z_i) \cdot \vec{\tau}_i \neq 0, \quad \vec{\psi}_j(x_i, y_i, z_i) \cdot \vec{\tau}_i = 0, \quad \forall j \neq i. \quad (1.17)$$

Домножим левую и правую части уравнения (1.16) скалярно на вектор  $\vec{\tau}_i$  в точке  $(x_i, y_i, z_i)$  с учётом (1.17) получим

$$\vec{\mathbf{A}}^{g,h}(x_i, y_i, z_i) \cdot \vec{\tau}_i = q_i^g \vec{\psi}_i(x_i, y_i, z_i) \cdot \vec{\tau}_i. \quad (1.18)$$

Полагая, что в точках  $(x_i, y_i, z_i)$  значения проекции на  $\vec{\tau}_i$  интерполянта  $\vec{\mathbf{A}}^{g,h}$  должны совпадать со значениями проекций на  $\vec{\tau}_i$  вектор-функции  $\vec{\mathbf{A}}^g$ , а также потребовав равенства касательных составляющих  $\vec{\mathbf{A}}^h \times \vec{\mathbf{n}}$  искомой вектор-функции  $\vec{\mathbf{A}}^h$  к касательным составляющим  $\vec{\mathbf{A}}^{g,h} \times \vec{\mathbf{n}}$  вектор-функции

$$\vec{\mathbf{A}}^{g,h} = \sum_{i \in N \setminus N_0} q_j^g \vec{\psi}_j, \quad (1.19)$$

получим следующее выражение для учёта неоднородного главного краевого условия:

$$q_i = \frac{\vec{\mathbf{A}}^g(x_i, y_i, z_i) \cdot \vec{\tau}_i}{\vec{\psi}_i(x_i, y_i, z_i) \cdot \vec{\tau}_i}, \quad i \in N \setminus N_0. \quad (1.20)$$



## 2. ПРАКТИЧЕСКАЯ ЧАСТЬ

### 2.1. ГЕНЕРАЦИЯ ТРЁХМЕРНОЙ СЕТКИ С ЯЧЕЙКАМИ В ВИДЕ ШЕСТИГРАННИКОВ

При написании программы был использован следующий подход к построению сетки на шестигранных элементах.

1. [lines amount x] 2 [lines amount y] 2 [lines amount z] 2
2. [field description of points]
3. 0.0 0.0 0.0      1.0 0.0 0.0
4. 0.0 1.0 0.0      1.0 1.0 0.0
5. 0.0 0.0 1.0      1.0 0.0 1.0
6. 0.0 1.0 1.0      1.0 1.0 1.0
7. [unique areas amount] 1
8. [unique areas description]
9. 1 0 1 0 1 0 1
10. [unique areas coefficients description]
11. 1 1.0 1.0
12. [delimiters above X description] 1 1.0
13. [delimiters above Y description] 3 1.1
14. [delimiters above Z description] 4 0.8
15. [borders amount] 6
16. [borders description]
17. 1 1 0 1 0 0 0 1
18. 1 1 0 1 1 1 0 1
19. 1 1 0 0 0 1 0 1
20. 1 1 1 1 0 1 0 1

21. 1 1 0 1 0 1 0 0

22. 1 1 0 1 0 1 1 1

В первой строке заданы количество опорных узлов  $N_x^W$ ,  $N_y^W$ ,  $N_z^W$ , базовой плоскости по осям  $X$ ,  $Y$ ,  $Z$  соответственно. С третьей по шестую строки перечислены тройки чисел  $(x_i, y_i, z_i)$  - как раз и определяющие эти опорные узлы.

В седьмой строке указано количество уникальных областей в расчётной области, которые имеют определённые уникальные значения физических параметров  $\mu$  и  $\sigma$ . Начиная с девятой строки (в общем случае должен быть построчный перечень каждой области) описывается геометрическое расположение  $i$  - ой области. В одиннадцатой строке указаны уникальные значения параметров  $\mu$  и  $\sigma$  для  $i$  - ой области.

В строках с двенадцатой по четырнадцатую описывается количество и характер необходимых разбиений для осей  $X$ ,  $Y$ ,  $Z$  соответственно.

В пятнадцатой строке целочисленным значением задаётся количество границ. Далее с семнадцатой по двадцать вторую строки описывается расположение и характер этих границ. Первым числом задаётся тип краевого условия (т.е. принимает значения 1 или 2), вторым числом задаётся номер формулы, третьим первая координатная линия по оси  $X$ , четвёртым вторая координатная линия по оси  $X$ , пятым и шестым аналогично по оси  $Y$  и седьмым и восьмым по оси  $Z$ .

Пример расчётной области этой фигуры изображён на рисунке (2.1).

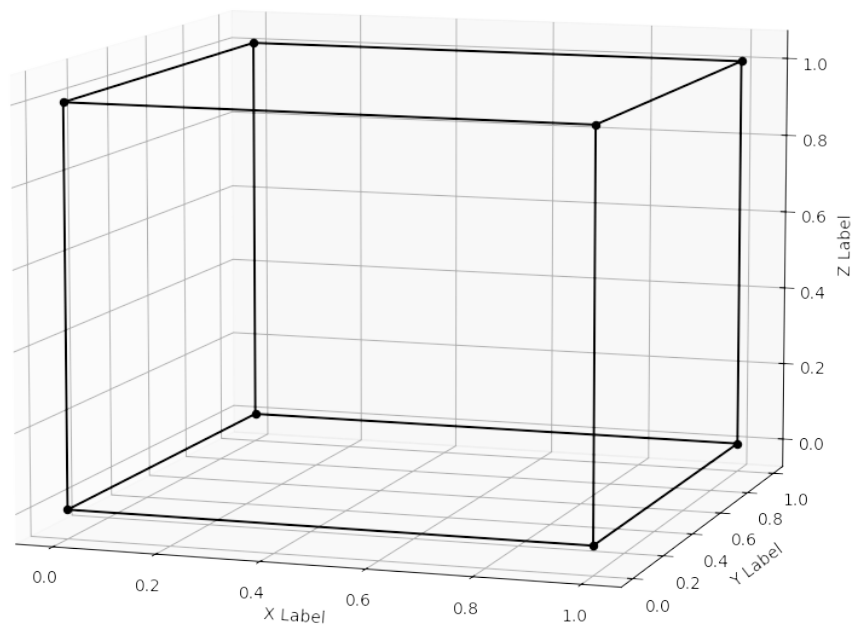


Рисунок 2.1 – Расчетная область для кубика

Попробуем подробить расчётную область (2.1) на несколько частей. Получим сетку изображённую на рисунке (2.2).

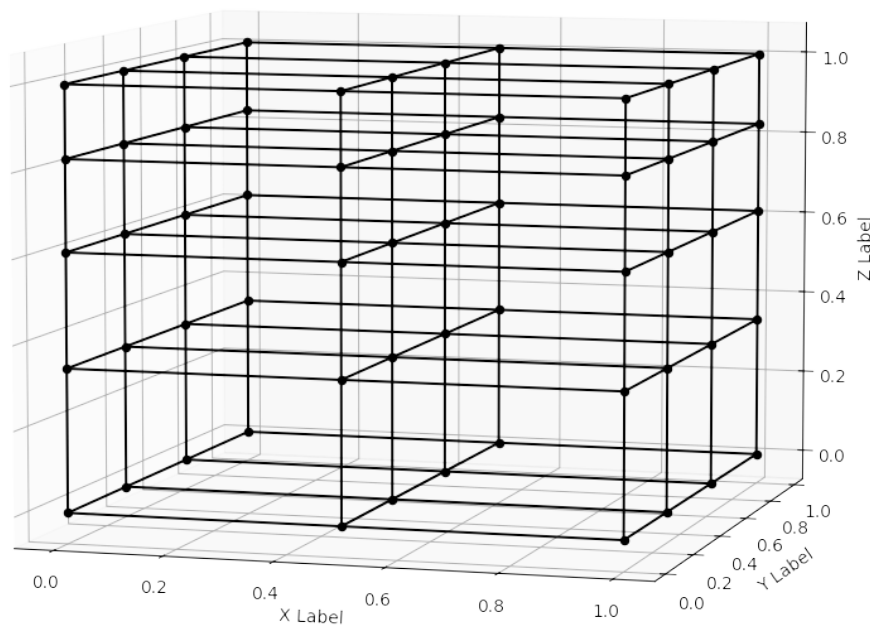
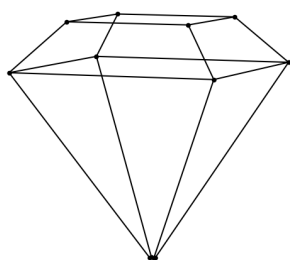
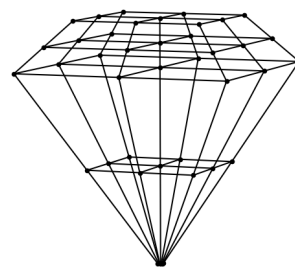


Рисунок 2.2 – Секта для кубика

Приведём ещё несколько примеров для построения сеток на шестигранниках, изображённых на рисунках 2.3 - 2.7.

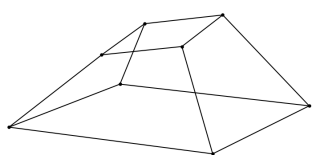


а)

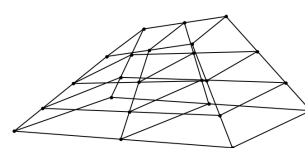


б)

Рисунок 2.3 – Расчётная область в форме изумруда (а) и сетка к ней (б).

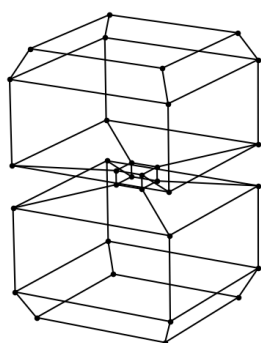


а)

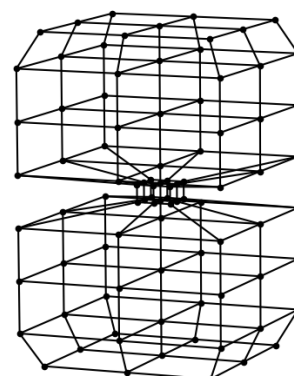


б)

Рисунок 2.4 – Расчётная область в форме скошенной пирамиды (а) и сетка к ней (б).

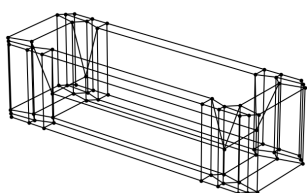


а)

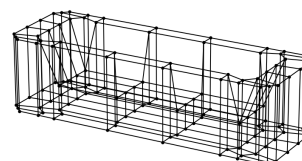


б)

Рисунок 2.5 – Расчётная область в форме песочных часов (а) и сетка к ней (б).

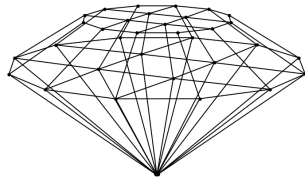


а)

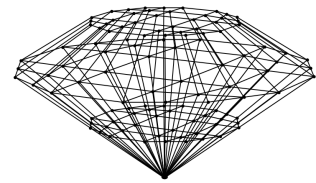


б)

Рисунок 2.6 – Расчётная область в форме ванной (а) и сетка к ней (б).



а)



б)

Рисунок 2.7 – Расчётная область в форме детализированного изумруда (а) и сетка к ней (б).

Таким образом, программа для построения сетки может строить достаточно геометрически сложные фигуры.

## 2.2. ЧИСЛЕННОЕ ИНТЕГРИРОВАНИЕ

При расчёте элементов локальных матриц жёсткости (1.11) и масс (1.12) будем использовать численное интегрирование методами Гаусса разных порядков (2, 3, 4, 5). Результаты численного интегрирования на некоторых функциях приведены в таблицах 2.1 - 2.7. Область интегрирования для всех функций единый:  $\Omega_E \in [-1; 1]_x \times [-1; 1]_y \times [-1; 1]_z$ .

Таблица 2.1 – Тестирование численного интегрирования на функции  $u = 2$ .

Аналитический результат	Гаусс 2	Гаусс 3	Гаусс 4	Гаусс 5
16.0	1.6000000e+01	1.6000000e+01	1.6000000e+01	1.6000000e+01

Таблица 2.2 – Тестирование численного интегрирования на функции

$$u = x + y + z.$$

Аналитический результат	Гаусс 2	Гаусс 3	Гаусс 4	Гаусс 5
0.0	0.0000000e+00	-2.2204460e-16	5.6898930e-16	-6.5225603e-16

Таблица 2.3 – Тестирование численного интегрирования на функции

$$u = x^2 + y^2 + z^2.$$

Аналитический результат	Гаусс 2	Гаусс 3	Гаусс 4	Гаусс 5
8.0	8.0000000e+00	8.0000000e+00	8.0000000e+00	8.0000000e+00

Таблица 2.4 – Тестирование численного интегрирования на функции

$$u = x \cdot y \cdot z.$$

Аналитический результат	Гаусс 2	Гаусс 3	Гаусс 4	Гаусс 5
0.0	8.0000000e+00	0.0000000e+00	0.0000000e+00	8.6736174e-18

Таблица 2.5 – Тестирование численного интегрирования на функции

$$u = x^2 \cdot y^2 \cdot z^2.$$

Аналитический результат	Гаусс 2	Гаусс 3	Гаусс 4	Гаусс 5
$\frac{8}{27} \approx 0.29630$	2.9629630e-01	2.9629630e-01	2.9629630e-01	2.9629630e-01

Таблица 2.6 – Тестирование численного интегрирования на функции

$$u = \cos(x + y + z).$$

Аналитический результат	Гаусс 2	Гаусс 3	Гаусс 4	Гаусс 5
4.7666...	4.7063579e+00	4.7671091e+00	4.7665835e+00	4.7665859e+00

Таблица 2.7 – Тестирование численного интегрирования на функции

$$u = e^{x+y+z}.$$

Аналитический результат	Гаусс 2	Гаусс 3	Гаусс 4	Гаусс 5
12.9845...	1.2857243e+01	1.2983458e+01	1.2984538e+01	1.2984543e+01

## 2.3. ПРОЦЕСС ПОСТРОЕНИЯ ЛОКАЛЬНЫХ МАТРИЦ ЖЁСТКОСТИ И МАСС

Рассмотрим процесс построения локальных матриц жёсткости и масс. При генерации локальной матрицы масс на параллелепипеде в векторном МКЭ используется следующая локальная матрица

$$\hat{\mathbf{M}} = \gamma \frac{h_x h_y h_z}{36} \begin{pmatrix} \mathbf{D} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{D} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{D} \end{pmatrix}, \quad (2.1)$$

где  $\mathbf{0}$  - матрица размером  $4 \times 4$ , полностью заполненная нулями, а

$$\hat{\mathbf{D}} = \begin{pmatrix} 4 & 2 & 2 & 1 \\ 2 & 4 & 1 & 2 \\ 2 & 1 & 4 & 2 \\ 1 & 2 & 2 & 4 \end{pmatrix}.$$

Тогда на единичном элементе  $\Omega_E \in [-1; 1]_x \times [-1; 1]_y \times [-1; 1]_z$  и при  $\gamma = 1$  матрица (2.1) примет вид:

$$\hat{\mathbf{M}} = \frac{2}{9} \begin{pmatrix} \mathbf{D} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{D} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{D} \end{pmatrix}.$$

Попробуем сделать ту же самую процедуру при  $\gamma = 1$  и на элемента  $\Omega_E \in [-1; 1]_x \times [-1; 1]_y \times [-1; 1]_z$ , но используя интегралы из (1.12). Элементы сгенерированной матрицы в виде консольного вывода программы изображены на рисунке 2.8.





```

1.333333e+00 -3.333333e-01 -3.333333e-01 -6.666667e-01 -6.666667e-01 6.666667e-01 -3.333333e-01 3.333333e-01 -6.666667e-01 6.666667e-01 -3.333333e-01 3.333333e-01
-3.333333e-01 1.333333e+00 -6.666667e-01 -3.333333e-01 6.666667e-01 -6.666667e-01 3.333333e-01 -3.333333e-01 -3.333333e-01 3.333333e-01 -6.666667e-01 6.666667e-01
-3.333333e-01 -6.666667e-01 1.333333e+00 -3.333333e-01 -3.333333e-01 3.333333e-01 -6.666667e-01 6.666667e-01 6.666667e-01 -6.666667e-01 3.333333e-01 -3.333333e-01
-6.666667e-01 -3.333333e-01 -3.333333e-01 1.333333e+00 3.333333e-01 -3.333333e-01 6.666667e-01 -6.666667e-01 3.333333e-01 -3.333333e-01 6.666667e-01 -6.666667e-01
-6.666667e-01 6.666667e-01 -3.333333e-01 3.333333e-01 1.333333e+00 -3.333333e-01 -3.333333e-01 -6.666667e-01 -6.666667e-01 -3.333333e-01 6.666667e-01 3.333333e-01
6.666667e-01 -6.666667e-01 3.333333e-01 -3.333333e-01 -3.333333e-01 1.333333e+00 -6.666667e-01 -3.333333e-01 -3.333333e-01 -6.666667e-01 3.333333e-01 6.666667e-01
-3.333333e-01 3.333333e-01 -6.666667e-01 6.666667e-01 -3.333333e-01 -6.666667e-01 1.333333e+00 -3.333333e-01 6.666667e-01 3.333333e-01 -6.666667e-01 -3.333333e-01
3.333333e-01 -3.333333e-01 6.666667e-01 -6.666667e-01 -6.666667e-01 -3.333333e-01 -3.333333e-01 1.333333e+00 3.333333e-01 6.666667e-01 -3.333333e-01 -6.666667e-01
-6.666667e-01 -3.333333e-01 6.666667e-01 3.333333e-01 -6.666667e-01 -3.333333e-01 6.666667e-01 3.333333e-01 1.333333e+00 -3.333333e-01 -3.333333e-01 -6.666667e-01
6.666667e-01 3.333333e-01 -6.666667e-01 -3.333333e-01 -3.333333e-01 -6.666667e-01 3.333333e-01 6.666667e-01 -3.333333e-01 1.333333e+00 -6.666667e-01 -3.333333e-01
-3.333333e-01 -6.666667e-01 3.333333e-01 6.666667e-01 6.666667e-01 3.333333e-01 -6.666667e-01 -3.333333e-01 1.333333e+00 -3.333333e-01 -3.333333e-01
3.333333e-01 6.666667e-01 -3.333333e-01 -6.666667e-01 3.333333e-01 6.666667e-01 -3.333333e-01 -6.666667e-01 -3.333333e-01 1.333333e+00

```

Рисунок 2.9 – Консольный вывод сгенерированной матрицы масс на  $\Omega_E$

## 2.4. РЕШЕНИЕ СЛАУ

Через LOS или Pardiso.

# СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ

1. А.Н. Тихонов, А.А. Самарский Уравнения математической физики: Учеб.пособие. / А.Н. Тихонов, А.А. Самарский — 6-е изд., — М: Изд-во МГУ, 1999 — 799 с.
2. Ю.Г. Соловейчик, М.Э. Рояк, М.Г. Персова Метод конечных элементов для скалярных и векторных задач Учеб. пособие. — Новосибирск: Изд-во НГТУ, 2007 — 896 с.
3. М.Ю.Баландин, Э.П.Шурина Векторный метод конечных элементов: Учеб. пособие. - Новосибирск: Изд-во НГТУ, 2001 — 69 с.
4. М.Ю.Баландин, Э.П.Шурина Методы решения СЛАУ большой размерности: Учеб. пособие. - Новосибирск: Изд-во НГТУ, 2000 — 70 с.
5. М.Г. Персова, Ю.Г. Соловейчик, Д.В. Вагин, П.А. Домников, Ю.И. Кошкина Численные методы в уравнениях математической физики. - Новосибирск: Изд-во НГТУ, 2016 — 60 с.

# ПРИЛОЖЕНИЕ 3. ТЕКСТ ПРОГРАММЫ

## FEM.h

```
1  #pragma once
2
3  #include <vector>
4  #include <array>
5  #include <fstream>
6
7  #include "../Mathematical_objects/MathematicalHeader.h"
8  #include "../Logger/Logger.h"
9  #include "../Mesh/Mesh.h"
10 #include "../Solvers/MainSolverHeader.h"
11
12 typedef std::array<double, 3> vector;
13
14 enum class EquationType {
15     Hyperbolical, Parabolical,
16     Elliptical, NotStated };
17
18 enum class InputExtension { Txt, Bin };
19
20 class FEM {
21     bool _isDataCommitted = false;
22     bool isDataCommitted() const { return _isDataCommitted; }
23     bool _isSolved = false;
24     bool isSolved() const { return _isSolved; }
25     // Based.
26     std::vector<std::array<size_t, 13>> _areas{};
27     std::vector<std::array<size_t, 3>> _borderRibs{};
28     std::vector<std::array<size_t, 6>> _newBorderRibs{};
29     std::vector<std::array<double, 3>> _points{};
30     std::vector<std::pair<size_t, size_t>> _generatedRibs{};
31     std::vector<std::pair<size_t, std::pair<double, double>>> _areasInfo{};
32     std::vector<double> _time{};
33     void SolveElliptical();
34     void SolveParabolical();
35     void SolveHyperbolical();
36     GlobalMatrix* A;
37     GlobalVector* b;
38     GlobalVector x;
39     Solver* _s;
40 public:
41     void ReadMeshData(InputExtension ie);
42     void GetMeshData(const Mesh* mesh);
43     void BuildMatrixAndVector();
44     vector GetSolutionAtPoint(double x, double y, double z);
45     void SetSolver(Solver* s);
46     void Solve();
47     void WriteAnswer();
48     EquationType Type = EquationType::NotStated;
49     __declspec(property(get = isDataCommitted)) bool IsDataCommitted;
50     __declspec(property(get = isSolved)) bool IsSolved;
51     FEM();
52     FEM(std::vector<std::array<size_t, 13>> areas,
53         std::vector<std::array<size_t, 3>> borderRibs,
54         std::vector<std::array<double, 3>> points,
55         std::vector<std::pair<size_t, size_t>> generatedRibs) :
56         _areas(areas), _borderRibs(borderRibs),
57         _points(points), _generatedRibs(generatedRibs) {}
58     ~FEM();
59 };
```

## FEM.cpp

```

1  #include "FEM.h"
2
3  void FEM::SolveElliptical() {
4      A = new GlobalMatrix(); b = new GlobalVector(_generatedRibs.size());
5      x = GlobalVector(_generatedRibs.size());
6      A->GeneratePortrait(_areas, _generatedRibs.size());
7      A->Fill(_areas, _points, _generatedRibs, _areasInfo);
8      b->Fill(_areas, _points, _generatedRibs);
9      A->CommitBoundaryConditions(_newBorderRibs);
10     b->CommitBoundaryConditions(_newBorderRibs, _points, _generatedRibs); }
11
12 void FEM::SolveParabolical() {
13     Logger::ConsoleOutput("Couldn't read Parabolic problem", NotificationColor::Alert);
14     exit(-1); }
15
16 void FEM::SolveHyperbolical() {
17     Logger::ConsoleOutput("Couldn't read Hyperbolic problem", NotificationColor::Alert);
18     exit(-1); }
19
20 void FEM::ReadMeshData(InputExtension ie) {
21     Logger::ConsoleOutput("Couldn't read from file", NotificationColor::Alert);
22     exit(-1); }
23
24 void FEM::GetMeshData(const Mesh* mesh) {
25     for (const auto& point : mesh->Points) {
26         std::array<double, 3> _point = { point.x, point.y, point.z };
27         _points.push_back(_point); }
28     for (const auto& area : mesh->getAreasAsRibs()) {
29         std::array<size_t, 13> _area = { area.subdomainNum_,
30             area.refs_[0], area.refs_[1], area.refs_[2], area.refs_[3],
31             area.refs_[4], area.refs_[5], area.refs_[6], area.refs_[7],
32             area.refs_[8], area.refs_[9], area.refs_[10], area.refs_[11] };
33         _areas.push_back(_area); }
34     for (const auto& rib : mesh->getRibsRefs()) {
35         std::pair<size_t, size_t> _rib{ rib.p1, rib.p2 };
36         _generatedRibs.push_back(_rib); }
37     for (const auto& border : mesh->getNewBorderRibs()) {
38         std::array<size_t, 6> rwe{border[0], border[1], border[2], border[3], border[4],
39             ↪ border[5]};
40         _newBorderRibs.push_back(rwe); }
41     for (const auto& areasInfos : mesh->AreasInfo) {
42         std::pair<size_t, std::pair<double, double>> _area{ areasInfos.subdomainNum_,
43             ↪ {areasInfos.mu_, areasInfos.sigma_} };
44         _areasInfo.push_back(_area); }
45     _isDataCommited = true; }
46
47 void FEM::BuildMatrixAndVector() {
48     switch (Type) {
49     case EquationType::Hyperbolical: SolveHyperbolical(); break;
50     case EquationType::Parabolical: SolveParabolical(); break;
51     case EquationType::Elliptical: SolveElliptical(); break;
52     case EquationType::NotStated: default:
53         Logger::ConsoleOutput("Equation type didn't stated. Exit program.",
54             ↪ NotificationColor::Alert);
55         exit(-1); break; } }
56
57 vector FEM::GetSolutionAtPoint(double x, double y, double z) {
58     Logger::ConsoleOutput("Can't get solution at point.", NotificationColor::Alert);
59     exit(-1);
60     return vector(); }
61
62 void FEM::SetSolver(Solver* s) { _s = s; }
63
64 void FEM::Solve() { x = _s->Solve(*A, *b); }
65
66 void FEM::WriteAnswer() {
67     std::ofstream fout("Data/Output/solution.txt");
68     for (size_t i(0); i < x.getSize(); ++i)
69         fout << i << ". " << std::setprecision(15) << std::scientific << x(i) <<
70             ↪ std::endl;
71     fout.close(); }

```

```

69 FEM::FEM() { Logger::ConsoleOutput("FEM declared, but it's empty",
   ↳ NotificationColor::Warning); }
70
71 FEM::~FEM() {}

```

## Integration.h

```

1  #pragma once
2
3  #include <array>
4  #include <functional>
5
6  typedef std::function<double(double, double, double)> function;
7
8  class Integration {
9  private:
10     static const std::array<double, 2> t2; static const std::array<double, 2> tau2;
11     static const std::array<double, 3> t3; static const std::array<double, 3> tau3;
12     static const std::array<double, 4> t4; static const std::array<double, 4> tau4;
13     static const std::array<double, 5> t5; static const std::array<double, 5> tau5;
14 public:
15     Integration() = delete;
16     static double Gauss2(function f); static double Gauss3(function f);
17     static double Gauss4(function f); static double Gauss5(function f);
18 };
19

```

## Integration.cpp

```

1  #include "Integration.h"
2
3  const std::array<double, 2> Integration::t2 = { 0.577'350'2692, -0.577'350'2692 };
4  const std::array<double, 2> Integration::tau2 = { 1.0, 1.0 };
5  const std::array<double, 3> Integration::t3{ 0.0, 0.774'596'669'24, -0.774'596'669'24 };
6  const std::array<double, 3> Integration::tau3{ 8.0 / 9.0, 5.0 / 9.0, 5.0 / 9.0 };
7  const std::array<double, 4> Integration::t4{ -0.861'136'311'6, -0.339'981'043'6,
8         0.339'981'043'6, 0.861'136'311'6 };
9  const std::array<double, 4> Integration::tau4{ 0.347'854'845'1, 0.652'145'154'9,
10         0.652'145'154'9, 0.347'854'845'1 };
11 const std::array<double, 5> Integration::t5{ -0.906'179'845'9, -0.538'469'310'1,
12        0.0,
13        0.538'469'310'1, 0.906'179'845'9 };
14 const std::array<double, 5> Integration::tau5{ 0.236'926'885'1, 0.478'628'670'5,
15        0.568'888'888'9, 0.478'628'670'5,
16        0.236'926'885'1 };
17
18 double Integration::Gauss2(function f) {
19     double ans(0.0);
20     for (size_t k(0); k < 2; ++k)
21         for (size_t j(0); j < 2; ++j)
22             for (size_t i(0); i < 2; ++i)
23                 ans += tau2[k] * tau2[j] * tau2[i] * f(t2[k], t2[j], t2[i]);
24     return ans; }
25
26 double Integration::Gauss3(function f) {
27     double ans(0.0);
28     for (size_t k(0); k < 3; ++k)
29         for (size_t j(0); j < 3; ++j)
30             for (size_t i(0); i < 3; ++i)
31                 ans += tau3[k] * tau3[j] * tau3[i] * f(t3[k], t3[j], t3[i]);
32     return ans; }
33
34 double Integration::Gauss4(function f) {
35     double ans(0.0);
36     for (size_t k(0); k < 4; ++k)
37         for (size_t j(0); j < 4; ++j)
38             for (size_t i(0); i < 4; ++i)
39                 ans += tau4[k] * tau4[j] * tau4[i] * f(t4[k], t4[j], t4[i]);
40     return ans; }
41

```

```

42 double Integration::Gauss5(function f) {
43     double ans(0.0);
44     for (size_t k(0); k < 5; ++k)
45         for (size_t j(0); j < 5; ++j)
46             for (size_t i(0); i < 5; ++i)
47                 ans += tau5[k] * tau5[j] * tau5[i] * f(t5[k], t5[j], t5[i]);
48     return ans; }

```

## GlobalMatrix.h

```

1  #pragma once
2
3  #include "../Logger/Logger.h"
4  #include "Matrix.h"
5  #include "LocalMatrix.h"
6  #include "GlobalVector.h"
7
8  #include <vector>
9  #include <array>
10 #include <algorithm>
11
12 class GlobalMatrix : public Matrix {
13 private:
14     size_t _size = 0;
15     bool _isPortraitGenerated = false;
16     inline bool isPortraitGenerated() const { return _isPortraitGenerated; }
17     void addLocalMatrixValues(const std::array<size_t, 12> localRibs, const LocalMatrix&
18         ↪ G, const LocalMatrix& M);
19     double getAlValue(size_t i, size_t j) const;
20     double getAuValue(size_t i, size_t j) const;
21 public:
22     inline size_t getSize() const override { return _di.size(); }
23     void GeneratePortrait(std::vector<std::array<size_t, 13>> areas, size_t ribsAmount);
24     void Fill(std::vector<std::array<size_t, 13>> areas, std::vector<std::array<double,
25         ↪ 3>> points,
26         std::vector<std::pair<size_t, size_t>> generatedRibs,
27         ↪ std::vector<std::pair<size_t, std::pair<double, double>>> areasInfo);
28     void CommitBoundaryConditions(std::vector<std::array<size_t, 6>> borderRibs);
29     std::vector<double> _al{};
30     std::vector<double> _au{};
31     std::vector<double> _di{};
32     std::vector<size_t> _ig{};
33     std::vector<size_t> _jg{};
34     inline double AL(size_t i) const { return i < _al.size() ? _al[i] : throw "_al
35         ↪ argument out of range."; }
36     inline double AU(size_t i) const { return i < _au.size() ? _au[i] : throw "_au
37         ↪ argument out of range."; }
38     inline double DI(size_t i) const { return i < _di.size() ? _di[i] : throw "_di
39         ↪ argument out of range."; }
40     inline size_t IG(size_t i) const { return i < _ig.size() ? _ig[i] : throw "_ig
41         ↪ argument out of range."; }
42     inline size_t JG(size_t i) const { return i < _jg.size() ? _jg[i] : throw "_jg
43         ↪ argument out of range."; }
44     __declspec(property(get = isPortraitGenerated)) bool IsPortraitGenerated;
45     __declspec(property(get = getSize)) size_t Size;
46     double getValue(size_t i, size_t j);
47     double operator() (size_t i, size_t j) const override;    // getter.
48     double& operator() (size_t i, size_t j) override;         // setter.
49     GlobalMatrix() {};
50     ~GlobalMatrix() {};
51     friend GlobalVector operator*(const GlobalMatrix A, const GlobalVector b);
52 };

```

## GlobalMatrix.cpp

```

1  #include "GlobalMatrix.h"
2
3  void GlobalMatrix::addLocalMatrixValues(const std::array<size_t, 12> localRibs, const
4  ↪ LocalMatrix& G, const LocalMatrix& M) {

```

```

4     const std::array<size_t, 12> switchV{
5         0, 3, 8, 11,
6         1, 2, 9, 10,
7         4, 5, 6, 7 };
8     int ii(0);
9     for (const auto& i : localRibs) {
10         int jj(0);
11         for (const auto& j : localRibs) {
12             int ind(0);
13             if (i - j == 0)
14                 _di[i] += G(switchV[ii], switchV[jj]) + M(switchV[ii], switchV[jj]);
15             else if (i - j < 0) {
16                 ind = _ig[j];
17                 for (; ind <= _ig[j + 1] - 1; ind++) if (_jg[ind] == i) break;
18                 _au[ind] += G(switchV[ii], switchV[jj]) + M(switchV[ii], switchV[jj]);
19             }
20             else if (i - j > 0) {
21                 ind = _ig[i];
22                 for (; ind <= _ig[i + 1] - 1; ind++) if (_jg[ind] == j) break;
23                 _al[ind] += G(switchV[ii], switchV[jj]) + M(switchV[ii], switchV[jj]);
24             } ++jj; } ++ii; }
25 }
26
27 double GlobalMatrix::getAlValue(size_t i, size_t j) const {
28     for (size_t ii = 0; ii < _ig[i + 1] - _ig[i]; ii++)
29         if (_jg[_ig[i] + ii] == j) return _al[_ig[i] + ii];
30     return 0.0;
31 }
32
33 double GlobalMatrix::getAuValue(size_t i, size_t j) const {
34     for (int ii = 0; ii < _ig[i + 1] - _ig[i]; ii++)
35         if (_jg[_ig[i] + ii] == j) return _au[_ig[i] + ii];
36     return 0.0;
37 }
38
39 void GlobalMatrix::GeneratePortrait(std::vector<std::array<size_t, 13>> areas, size_t
↪ ribsAmount) {
40     _ig.resize(ribsAmount + 1);
41     std::vector<std::vector<size_t>> additionalVector(ribsAmount);
42     for (const auto& area : areas)
43         for (size_t i(1); i < 13; ++i)
44             for (size_t j(1); j < 13; ++j)
45                 if (area[j] < area[i] and std::find(additionalVector[area[i]].begin(),
↪ additionalVector[area[i]].end(), area[j]) ==
↪ additionalVector[area[i]].end()) {
46                     additionalVector[area[i]].push_back(area[j]);
47                     std::sort(additionalVector[area[i]].begin(),
↪ additionalVector[area[i]].end()); }
48     _ig[0] = 0;
49     for (size_t i(0); i < ribsAmount; i++) {
50         _ig[i + 1] = _ig[i] + additionalVector[i].size();
51         _jg.insert(_jg.end(), additionalVector[i].begin(), additionalVector[i].end()); }
52     _di.resize(ribsAmount); _al.resize(_jg.size()); _au.resize(_jg.size());
53 }
54
55 void GlobalMatrix::Fill(std::vector<std::array<size_t, 13>> areas,
↪ std::vector<std::array<double, 3>> points,
56 std::vector<std::pair<size_t, size_t>> generatedRibs, std::vector<std::pair<size_t,
↪ std::pair<double, double>>> areasInfo) {
57     for (const auto& area : areas) {
58         auto mu = [area, areasInfo]() {
59             for (const auto& info : areasInfo)
60                 if (area[0] == info.first) return info.second.first; };
61         auto sigma = [area, areasInfo]() {
62             for (const auto& info : areasInfo)
63                 if (area[0] == info.first) return info.second.second; };
64         std::array<size_t, 12> localArea{ area[1], area[4], area[9], area[12],
65             area[2], area[3], area[10], area[11],
66             area[5], area[6], area[7], area[8] };
67         std::array<double, 8> xPoints = { points[generatedRibs[localArea[0]].first][0],
↪ points[generatedRibs[localArea[0]].second][0],
68             points[generatedRibs[localArea[1]].first][0],
↪ points[generatedRibs[localArea[1]].second][0],
69             points[generatedRibs[localArea[2]].first][0],
↪ points[generatedRibs[localArea[2]].second][0],

```

```

70         points[generatedRibs[localArea[3]].first][0],
71         ↪ points[generatedRibs[localArea[3]].second][0] };
72     std::array<double, 8> yPoints = { points[generatedRibs[localArea[0]].first][1],
73     ↪ points[generatedRibs[localArea[0]].second][1],
74     points[generatedRibs[localArea[1]].first][1],
75     ↪ points[generatedRibs[localArea[1]].second][1],
76     points[generatedRibs[localArea[2]].first][1],
77     ↪ points[generatedRibs[localArea[2]].second][1],
78     points[generatedRibs[localArea[3]].first][1],
79     ↪ points[generatedRibs[localArea[3]].second][1] };
80     std::array<double, 8> zPoints = { points[generatedRibs[localArea[0]].first][2],
81     ↪ points[generatedRibs[localArea[0]].second][2],
82     points[generatedRibs[localArea[1]].first][2],
83     ↪ points[generatedRibs[localArea[1]].second][2],
84     points[generatedRibs[localArea[2]].first][2],
85     ↪ points[generatedRibs[localArea[2]].second][2],
86     points[generatedRibs[localArea[3]].first][2],
87     ↪ points[generatedRibs[localArea[3]].second][2] };
88     LocalMatrix localG(mu(), xPoints, yPoints, zPoints, LMType::Stiffness);
89     LocalMatrix localM(sigma(), xPoints, yPoints, zPoints, LMType::Mass);
90     addLocalMatrixValues(localArea, localG, localM); }
91 }
92
93 void GlobalMatrix::CommitBoundaryConditions(std::vector<std::array<size_t, 6>>
94 ↪ borderRibs) {
95     for (const auto& rib : borderRibs) {
96         switch (rib[0]) {
97             case 2: case 3:
98                 Logger::ConsoleOutput("Can't commit boundary conditions of 2nd or 3rd type.",
99                 ↪ NotificationColor::Alert);
100                 exit(-1); break;
101             case 1:
102                 for (size_t i(2); i < 6; ++i) {
103                     for (size_t j(_ig[rib[i]]); j < _ig[rib[i] + 1]; ++j) _al[j] = 0;
104                     _di[rib[i]] = 1.0;
105                     for (size_t j = 0; j < _jg.size(); ++j) if (_jg[j] == rib[i]) _au[j] = 0;
106                     } break;
107             default: break; } }
108 }
109
110 double GlobalMatrix::getValue(size_t i, size_t j) {
111     if (i == j) return _di[i];
112     else if (i - j < 0) getAuValue(i, j);
113     else if (i - j > 0) getAlValue(j, i);
114 }
115
116 double GlobalMatrix::operator()(size_t i, size_t j) const {
117     if (i == j) return _di[i];
118     else if (i - j < 0) getAuValue(i, j);
119     else if (i - j > 0) getAlValue(i, j);
120 }
121
122 double& GlobalMatrix::operator()(size_t i, size_t j) {
123     Logger::ConsoleOutput("Can't set value for global matrix.", NotificationColor::Alert);
124     exit(-1); }
125
126 GlobalVector operator*(const GlobalMatrix A, const GlobalVector b) {
127     if (A.Size != b.Size) Logger::ConsoleOutput("Matrix and vector have different sizes
128     ↪ during multiplication", NotificationColor::Alert);
129     GlobalVector ans(b.Size);
130     for (size_t i(0); i < b.Size; ++i) {
131         for (size_t j(0); j < A._ig[i + 1] - A._ig[i]; ++j) {
132             ans(i) += A._al[A._ig[i] + j] * b(A._jg[A._ig[i] + j]);
133             ans(A._jg[A._ig[i] + j]) += A._au[A._ig[i] + j] * b(i); }
134         ans(i) += A._di[i] * b(i); }
135     return ans;
136 }

```



## LocalMatrix.h

```
1  #pragma once
2
3  #include "Matrix.h"
4  #include "JacobiMatrix.h"
5  #include "../Integration/Integration.h"
6  #include "../Functions/BasisFunction.h"
7
8  typedef JacobiMatrix J;
9
10 enum class LMType {
11     Stiffness,
12     Mass,
13     NotStated
14 };
15
16 class LocalMatrix : public Matrix {
17 private:
18     double _koef;
19     const size_t _localMatrixSize = 12;
20     LMType _matrixType = LMType::NotStated;
21     std::array<double, 8> _x{};
22     std::array<double, 8> _y{};
23     std::array<double, 8> _z{};
24     std::array<std::array<double, 12>, 12> _values{};
25     void generate();
26     void generateG();
27     void generateM();
28 public:
29     LocalMatrix() { _koef = 0.0; }
30     LocalMatrix(double koef, std::array<double, 8> x, std::array<double, 8> y,
31         ↪ std::array<double, 8> z, LMType matrixType) :
32         _koef(koef), _matrixType(matrixType), _x(x), _y(y), _z(z) {
33         generate();
34     }
35     ~LocalMatrix() {}
36     double operator() (size_t i, size_t j) const override { return _values[i][j]; };
37     double& operator() (size_t i, size_t j) override { return _values[i][j]; };
38     inline LMType GetMatrixType() const { return _matrixType; }
39     size_t getSize() const override { return _localMatrixSize; };
40 };
41
42 std::function<double(double, double, double)> operator* (std::function<double(double,
43     ↪ double, double)> f1,
44     std::function<double(double, double, double)> f2);
45
46 std::function<double(double, double, double)> operator+ (std::function<double(double,
47     ↪ double, double)> f1,
48     std::function<double(double, double, double)> f2);
49
50 std::function<double(double, double, double)> operator/ (std::function<double(double,
51     ↪ double, double)> f1,
52     std::function<double(double, double, double)> f2);
```

## LocalMatrix.cpp

```
1  #include "LocalMatrix.h"
2
3  std::function<double(double, double, double)> operator* (std::function<double(double,
4     ↪ double, double)> f1,
5     std::function<double(double, double, double)> f2) {
6     return [f1, f2](double t0, double t1, double t2) { return f1(t0, t1, t2) * f2(t0, t1,
7     ↪ t2); }; }
8
9  std::function<double(double, double, double)> operator+(std::function<double(double,
10     ↪ double, double)> f1,
11     std::function<double(double, double, double)> f2) {
12     return [f1, f2](double t0, double t1, double t2) { return f1(t0, t1, t2) + f2(t0, t1,
13     ↪ t2); }; }
```

```

11 std::function<double(double, double, double)> operator/(std::function<double(double,
    ↪ double, double)> f1,
12               std::function<double(double, double, double)> f2) {
13     return [f1, f2](double t0, double t1, double t2) { return f1(t0, t1, t2) / f2(t0, t1,
    ↪ t2); }; }
14
15 void LocalMatrix::generate() {
16     switch (_matrixType) {
17     case LMType::Stiffness: generateG(); break;
18     case LMType::Mass: generateM(); break;
19     case LMType::NotStated: default: break; }
20 }
21
22 void LocalMatrix::generateG() {
23     J::SetValues(_x, _y, _z);
24     for (size_t i(0); i < _localMatrixSize; ++i) {
25         for (size_t j(0); j < _localMatrixSize; ++j) {
26             auto rotPhi_i = BasisFunction::getRotAt(i);
27             auto rotPhi_j = BasisFunction::getRotAt(j);
28             std::array<std::function<double(double, double, double)>, 3> v1{
29                 J::GetValueAtTransposed(0, 0) * rotPhi_i[0] + J::GetValueAtTransposed(0, 1) *
    ↪ rotPhi_i[1] + J::GetValueAtTransposed(0, 2) * rotPhi_i[2],
30                 J::GetValueAtTransposed(1, 0) * rotPhi_i[0] + J::GetValueAtTransposed(1, 1) *
    ↪ rotPhi_i[1] + J::GetValueAtTransposed(1, 2) * rotPhi_i[2],
31                 J::GetValueAtTransposed(2, 0) * rotPhi_i[0] + J::GetValueAtTransposed(2, 1) *
    ↪ rotPhi_i[1] + J::GetValueAtTransposed(2, 2) * rotPhi_i[2], };
32             std::array<std::function<double(double, double, double)>, 3> v2{
33                 J::GetValueAtTransposed(0, 0) * rotPhi_j[0] + J::GetValueAtTransposed(0, 1) *
    ↪ rotPhi_j[1] + J::GetValueAtTransposed(0, 2) * rotPhi_j[2],
34                 J::GetValueAtTransposed(1, 0) * rotPhi_j[0] + J::GetValueAtTransposed(1, 1) *
    ↪ rotPhi_j[1] + J::GetValueAtTransposed(1, 2) * rotPhi_j[2],
35                 J::GetValueAtTransposed(2, 0) * rotPhi_j[0] + J::GetValueAtTransposed(2, 1) *
    ↪ rotPhi_j[1] + J::GetValueAtTransposed(2, 2) * rotPhi_j[2], };
36             for (size_t k(0); k < 3; ++k) _values[i][j] += Integration::Gauss5((v1[k] * v2[k])
    ↪ / J::GetDeterminant());
37             _values[i][j] /= _koef; } }
38 }
39
40 void LocalMatrix::generateM() {
41     J::SetValues(_x, _y, _z);
42     for (size_t i(0); i < _localMatrixSize; ++i) {
43         for (size_t j(0); j < _localMatrixSize; ++j) {
44             for (size_t k(0); k < 3; ++k) {
45                 _values[i][j] += Integration::Gauss5(J::GetValueAtInverse(k, i / 4) *
    ↪ BasisFunction::getAt(i) *
46                 J::GetValueAtInverse(k, j / 4) * BasisFunction::getAt(j) *
47                 J::GetDeterminant());
48             }
49             _values[i][j] *= _koef; } }
50 }

```

## GlobalVector.h

```

1  #pragma once
2
3  #include "../Logger/Logger.h"
4  #include "Vector.h"
5  #include "LocalVector.h"
6
7  #include <vector>
8  #include <array>
9
10 class GlobalVector : public Vector
11 {
12 private:
13     std::vector<double> _values{};
14     void addLocalVectorValues(const std::array<size_t, 12> localRibs, const LocalVector&
    ↪ b);
15 public:
16     size_t getSize() const override { return _values.size(); }
17     double operator() (size_t i) const override;

```

```

18     double& operator() (size_t i) override;
19     __declspec(property(get = getSize)) size_t Size;
20     GlobalVector();
21     GlobalVector(size_t size) { _values.resize(size); }
22     ~GlobalVector() {}
23     void Fill(std::vector<std::array<size_t, 13>> areas, std::vector<std::array<double,
    ↪ 3>> points,
24             std::vector<std::pair<size_t, size_t>> generatedRibs);
25     void CommitBoundaryConditions(std::vector<std::array<size_t, 6>> borderRibs,
    ↪ std::vector<std::array<double, 3>> points, std::vector<std::pair<size_t, size_t>>
    ↪ generatedRibs);
26     double Norma() const;
27     friend double operator* (const GlobalVector v1, const GlobalVector v2);
28     friend GlobalVector operator* (const double a, const GlobalVector v);
29     friend GlobalVector operator+ (const GlobalVector v1, const GlobalVector v2);
30     friend GlobalVector operator- (const GlobalVector v1, const GlobalVector v2);
31 };
32

```

## GlobalVector.cpp

```

1  #include "GlobalVector.h"
2
3  void GlobalVector::addLocalVectorValues(const std::array<size_t, 12> localRibs, const
    ↪ LocalVector& b) {
4      const std::array<size_t, 12> switchV{
5          0, 3, 8, 11,
6          1, 2, 9, 10,
7          4, 5, 6, 7 };
8      for (size_t i(0); i < b.getSize(); ++i) _values[localRibs[i]] += b(switchV[i]);
9  }
10
11  double GlobalVector::operator()(size_t i) const {
12      if (i >= Size) {
13          Logger::ConsoleOutput("Index run out of Vector range.", NotificationColor::Alert);
14          exit(-1); }
15      return i < Size ? _values[i] : throw "Index run out of Vector range.";
16  }
17
18  double& GlobalVector::operator()(size_t i) {
19      if (i >= Size) {
20          Logger::ConsoleOutput("Index run out of Vector range.", NotificationColor::Alert);
21          exit(-1); }
22      return _values[i];
23  }
24
25  GlobalVector::GlobalVector() {
26      Logger::ConsoleOutput("Global vector initialized, but it's empty",
    ↪ NotificationColor::Warning);
27  }
28
29  void GlobalVector::Fill(std::vector<std::array<size_t, 13>> areas,
    ↪ std::vector<std::array<double, 3>> points,
30                      std::vector<std::pair<size_t, size_t>> generatedRibs) {
31      for (const auto& area : areas) {
32          std::array<size_t, 12> localArea{ area[1], area[4], area[9], area[12],
33              area[2], area[3], area[10], area[11],
34              area[5], area[6], area[7], area[8] };
35          std::array<double, 8> xPoints = { points[generatedRibs[localArea[0]].first][0],
    ↪ points[generatedRibs[localArea[0]].second][0],
36              points[generatedRibs[localArea[1]].first][0],
    ↪ points[generatedRibs[localArea[1]].second][0],
37              points[generatedRibs[localArea[2]].first][0],
    ↪ points[generatedRibs[localArea[2]].second][0],
38              points[generatedRibs[localArea[3]].first][0],
    ↪ points[generatedRibs[localArea[3]].second][0] };
39          std::array<double, 8> yPoints = { points[generatedRibs[localArea[0]].first][1],
    ↪ points[generatedRibs[localArea[0]].second][1],
40              points[generatedRibs[localArea[1]].first][1],
    ↪ points[generatedRibs[localArea[1]].second][1],
41              points[generatedRibs[localArea[2]].first][1],
    ↪ points[generatedRibs[localArea[2]].second][1],

```

```

42         points[generatedRibs[localArea[3]].first][1],
43         ↪ points[generatedRibs[localArea[3]].second][1] };
44     std::array<double, 8> zPoints = { points[generatedRibs[localArea[0]].first][2],
45     ↪ points[generatedRibs[localArea[0]].second][2],
46     ↪ points[generatedRibs[localArea[1]].first][2],
47     ↪ points[generatedRibs[localArea[1]].second][2],
48     ↪ points[generatedRibs[localArea[2]].first][2],
49     ↪ points[generatedRibs[localArea[2]].second][2],
50     ↪ points[generatedRibs[localArea[3]].first][2],
51     ↪ points[generatedRibs[localArea[3]].second][2] };
52     LocalVector b(xPoints, yPoints, zPoints); }
53 }
54
55 void GlobalVector::CommitBoundaryConditions(std::vector<std::array<size_t, 6>>
56 ↪ borderRibs, std::vector<std::array<double, 3>> points, std::vector<std::pair<size_t,
57 ↪ size_t>> generatedRibs) {
58     for (const auto& square : borderRibs) {
59         size_t r0 = square[2]; size_t r1 = square[3];
60         std::array<double, 3> _x = { points[generatedRibs[r0].first][0],
61         ↪ points[generatedRibs[r0].second][0], points[generatedRibs[r1].second][0] };
62         std::array<double, 3> _y = { points[generatedRibs[r0].first][1],
63         ↪ points[generatedRibs[r0].second][1], points[generatedRibs[r1].second][1] };
64         std::array<double, 3> _z = { points[generatedRibs[r0].first][2],
65         ↪ points[generatedRibs[r0].second][2], points[generatedRibs[r1].second][2] };
66         auto getNormal = [_x, _y, _z]() -> vector {
67             auto v = vector{ (_y[1] - _y[0]) * (_z[2] - _z[0]) - (_z[1] - _z[0]) * (_y[2]
68             ↪ - _y[0]),
69             ↪ -1.0 * ((_x[1] - _x[0]) * (_z[2] - _z[0]) - (_z[1] - _z[0]) * (_x[2] -
70             ↪ _x[0])),
71             ↪ (_x[1] - _x[0]) * (_y[2] - _y[0]) - (_y[1] - _y[0]) * (_x[2] - _x[0]) };
72             double len = sqrt(v[0] * v[0] + v[1] * v[1] + v[2] * v[2]);
73             return vector{ v[0] / len, v[1] / len, v[2] / len }; };
74         auto normal = getNormal();
75         switch (square[0]) {
76             case 2: case 3:
77                 Logger::ConsoleOutput("Can't commit boundary conditions of 2nd or 3rd type.",
78                 ↪ NotificationColor::Alert);
79                 exit(-1); break;
80             case 1:
81                 for (size_t ii(2); ii < 6; ++ii) {
82                     std::array<double, 3> middlePoint{ 0.5 *
83                     ↪ (points[generatedRibs[square[ii]].first][0] +
84                     ↪ points[generatedRibs[square[ii]].second][0]),
85                     ↪ 0.5 * (points[generatedRibs[square[ii]].first][1] +
86                     ↪ points[generatedRibs[square[ii]].second][1]),
87                     ↪ 0.5 * (points[generatedRibs[square[ii]].first][2] +
88                     ↪ points[generatedRibs[square[ii]].second][2]), };
89                     auto fVector = Function::TestA(middlePoint[0], middlePoint[1],
90                     ↪ middlePoint[2], 0.0);
91                     auto fValue = fVector[0] * normal[0] + fVector[1] * normal[1] +
92                     ↪ fVector[2] * normal[2];
93                     _values[square[ii]] = fValue;
94                 } break;
95             default: break; } }
96 }
97
98 double GlobalVector::Norma() const {
99     double sum(0.0);
100     for (const auto& value : _values) sum += value * value;
101     return sqrt(sum);
102 }
103
104 double operator*(const GlobalVector v1, const GlobalVector v2) {
105     if (v1.Size != v2.Size) Logger::ConsoleOutput("During vector multiplication vectors
106     ↪ have different size", NotificationColor::Alert);
107     double sum(0.0);
108     for (size_t i(0); i < v1.Size; ++i) sum += v1(i) * v2(i);
109     return sum;
110 }
111
112 GlobalVector operator*(const double a, const GlobalVector v) {
113     GlobalVector result(v.Size);
114     for (size_t i(0); i < v.Size; ++i) result(i) = a * v(i);
115     return result;
116 }

```

```

96 }
97
98 GlobalVector operator+(const GlobalVector v1, const GlobalVector v2) {
99     if (v1.Size != v2.Size) Logger::ConsoleOutput("During vector multiplication vectors
    ↳ have different size", NotificationColor::Alert);
100     GlobalVector result(v1.Size);
101     for (size_t i(0); i < v1.Size; ++i) result(i) = v1(i) + v2(i);
102     return result;
103 }
104
105 GlobalVector operator-(const GlobalVector v1, const GlobalVector v2) {
106     if (v1.Size != v2.Size) Logger::ConsoleOutput("During vector multiplication vectors
    ↳ have different size", NotificationColor::Alert);
107     GlobalVector result(v1.Size);
108     for (size_t i(0); i < v1.Size; ++i) result(i) = v1(i) - v2(i);
109     return result;
110 }

```

## LocalVector.h

```

1  #pragma once
2
3  #include "Vector.h"
4  #include "LocalMatrix.h"
5  #include "..\Functions\Function.h"
6
7  class LocalVector : public Vector {
8  public:
9      LocalVector() {};
10     LocalVector(std::array<double, 8> x, std::array<double, 8> y, std::array<double, 8>
    ↳ z) : _x(x), _y(y), _z(z) {
11         M = new LocalMatrix(1.0, _x, _y, _z, LMType::Mass);
12         generate();
13     }
14     ~LocalVector() {};
15     double operator() (size_t i) const override { return _values[i]; }
16     double& operator() (size_t i) override { return _values[i]; }
17     size_t getSize() const override { return 12; }
18 private:
19     std::array<double, 8> _x{};
20     std::array<double, 8> _y{};
21     std::array<double, 8> _z{};
22     std::array<double, 12> _values{};
23     void generate();
24     LocalMatrix* M = nullptr;
25 };

```

## LocalVector.cpp

```

1  #include "LocalVector.h"
2
3  #include <iostream>
4
5  void LocalVector::generate() {
6      auto findLen = [](vector v) {return sqrt(v[0] * v[0] + v[1] * v[1] + v[2] * v[2]); };
7      auto scMult = [](vector v1, vector v2) { return v1[0] * v2[0] + v1[1] * v2[1] + v1[2]
    ↳ * v2[2]; };
8      std::array<vector, 12> diffVec = {
9          vector{_x[1] - _x[0], _y[1] - _y[0], _z[1] - _z[0]},
10         vector{_x[3] - _x[2], _y[3] - _y[2], _z[3] - _z[2]},
11         vector{_x[5] - _x[4], _y[5] - _y[4], _z[5] - _z[4]},
12         vector{_x[7] - _x[6], _y[7] - _y[6], _z[7] - _z[6]},
13         vector{_x[2] - _x[0], _y[2] - _y[0], _z[2] - _z[0]},
14         vector{_x[3] - _x[1], _y[3] - _y[1], _z[3] - _z[1]},
15         vector{_x[6] - _x[4], _y[6] - _y[4], _z[6] - _z[4]},
16         vector{_x[7] - _x[5], _y[7] - _y[5], _z[7] - _z[5]},
17         vector{_x[4] - _x[0], _y[4] - _y[0], _z[4] - _z[0]},
18         vector{_x[5] - _x[1], _y[5] - _y[1], _z[5] - _z[1]},
19         vector{_x[6] - _x[2], _y[6] - _y[2], _z[6] - _z[2]},

```

```

20     vector{_x[7] - _x[3], _y[7] - _y[3], _z[7] - _z[3]}};
21     std::array<double, 12> vf = {
22         scMult(Function::TestF1(_x[0] + 0.5 * diffVec[0][0], _y[0] + 0.5 * diffVec[0][1],
23             ↪ _z[0] + 0.5 * diffVec[0][2], 0.0), diffVec[0]) / findLen(diffVec[0]),
24         scMult(Function::TestF1(_x[2] + 0.5 * diffVec[2][0], _y[2] + 0.5 * diffVec[2][1],
25             ↪ _z[2] + 0.5 * diffVec[2][2], 0.0), diffVec[2]) / findLen(diffVec[2]),
26         scMult(Function::TestF1(_x[4] + 0.5 * diffVec[4][0], _y[4] + 0.5 * diffVec[4][1],
27             ↪ _z[4] + 0.5 * diffVec[4][2], 0.0), diffVec[4]) / findLen(diffVec[4]),
28         scMult(Function::TestF1(_x[6] + 0.5 * diffVec[6][0], _y[6] + 0.5 * diffVec[6][1],
29             ↪ _z[6] + 0.5 * diffVec[6][2], 0.0), diffVec[6]) / findLen(diffVec[6]),
30         scMult(Function::TestF1(_x[0] + 0.5 * diffVec[0][0], _y[0] + 0.5 * diffVec[0][1],
31             ↪ _z[0] + 0.5 * diffVec[0][2], 0.0), diffVec[0]) / findLen(diffVec[0]),
32         scMult(Function::TestF1(_x[1] + 0.5 * diffVec[1][0], _y[1] + 0.5 * diffVec[1][1],
33             ↪ _z[1] + 0.5 * diffVec[1][2], 0.0), diffVec[1]) / findLen(diffVec[1]),
34         scMult(Function::TestF1(_x[4] + 0.5 * diffVec[4][0], _y[4] + 0.5 * diffVec[4][1],
35             ↪ _z[4] + 0.5 * diffVec[4][2], 0.0), diffVec[4]) / findLen(diffVec[4]),
36         scMult(Function::TestF1(_x[5] + 0.5 * diffVec[5][0], _y[5] + 0.5 * diffVec[5][1],
37             ↪ _z[5] + 0.5 * diffVec[5][2], 0.0), diffVec[5]) / findLen(diffVec[5]),
38         scMult(Function::TestF1(_x[0] + 0.5 * diffVec[0][0], _y[0] + 0.5 * diffVec[0][1],
39             ↪ _z[0] + 0.5 * diffVec[0][2], 0.0), diffVec[0]) / findLen(diffVec[0]),
40         scMult(Function::TestF1(_x[1] + 0.5 * diffVec[1][0], _y[1] + 0.5 * diffVec[1][1],
41             ↪ _z[2] + 0.5 * diffVec[1][2], 0.0), diffVec[1]) / findLen(diffVec[1]),
42         scMult(Function::TestF1(_x[2] + 0.5 * diffVec[2][0], _y[2] + 0.5 * diffVec[2][1],
43             ↪ _z[4] + 0.5 * diffVec[2][2], 0.0), diffVec[2]) / findLen(diffVec[2]),
44         scMult(Function::TestF1(_x[3] + 0.5 * diffVec[3][0], _y[3] + 0.5 * diffVec[3][1],
45             ↪ _z[6] + 0.5 * diffVec[3][2], 0.0), diffVec[3]) / findLen(diffVec[3]), };
46     for (size_t i(0); i < 12; ++i)
47         for (size_t j(0); j < 12; ++j)
48             _values[i] += M->operator()(i, j) * vf[j];
49 }

```

## JacobiMatrix.h

```

1  #pragma once
2
3  #include <functional>
4  #include <array>
5
6  class JacobiMatrix {
7  private:
8      // Points arrays.
9      static std::array<double, 8> _x;
10     static std::array<double, 8> _y;
11     static std::array<double, 8> _z;
12     // Template functions.
13     static inline double const W(double t) { return 0.5 * (1.0 - t); }
14     static inline double const W(double t) { return 0.5 * (1.0 + t); }
15 public:
16     JacobiMatrix() = delete;
17     static void SetValues(std::array<double, 8> x, std::array<double, 8> y,
18         ↪ std::array<double, 8> z);
19     static inline double dxde(double eps, double eta, double zeta);
20     static inline double dyde(double eps, double eta, double zeta);
21     static inline double dzde(double eps, double eta, double zeta);
22     static inline double dxdn(double eps, double eta, double zeta);
23     static inline double dydn(double eps, double eta, double zeta);
24     static inline double dzdn(double eps, double eta, double zeta);
25     static inline double dxdc(double eps, double eta, double zeta);
26     static inline double dydc(double eps, double eta, double zeta);
27     static inline double dzdc(double eps, double eta, double zeta);
28     static std::function<double(double, double, double)> const GetValueAt(size_t i,
29         ↪ size_t j);
30     static std::function<double(double, double, double)> const GetValueAtInverse(size_t
31         ↪ i, size_t j);
32     static std::function<double(double, double, double)> const
33         ↪ GetValueAtTransposed(size_t i, size_t j);
34     static std::function<double(double, double, double)> const GetDeterminant();
35 }

```

## JacobiMatrix.cpp

```
1  #include "JacobiMatrix.h"
2
3  std::array<double, 8> JacobiMatrix::_x = {};
4  std::array<double, 8> JacobiMatrix::_y = {};
5  std::array<double, 8> JacobiMatrix::_z = {};
6
7  inline double JacobiMatrix::dxde(double eps, double eta, double zeta) {
8      return 0.5 * (W_(eta) * W_(zeta) * (_x[1] - _x[0]) +
9                  W_(eta) * W_(zeta) * (_x[3] - _x[2]) +
10                 W_(eta) * W_(zeta) * (_x[5] - _x[4]) +
11                 W_(eta) * W_(zeta) * (_x[7] - _x[6])); }
12
13  inline double JacobiMatrix::dyde(double eps, double eta, double zeta) {
14      return 0.5 * (W_(eta) * W_(zeta) * (_y[1] - _y[0]) +
15                  W_(eta) * W_(zeta) * (_y[3] - _y[2]) +
16                  W_(eta) * W_(zeta) * (_y[5] - _y[4]) +
17                  W_(eta) * W_(zeta) * (_y[7] - _y[6])); }
18
19  inline double JacobiMatrix::dzde(double eps, double eta, double zeta) {
20      return 0.5 * (W_(eta) * W_(zeta) * (_z[1] - _z[0]) +
21                  W_(eta) * W_(zeta) * (_z[3] - _z[2]) +
22                  W_(eta) * W_(zeta) * (_z[5] - _z[4]) +
23                  W_(eta) * W_(zeta) * (_z[7] - _z[6])); }
24
25  inline double JacobiMatrix::dxdn(double eps, double eta, double zeta) {
26      return 0.5 * (W_(eps) * W_(zeta) * (_x[2] - _x[0]) +
27                  W_(eps) * W_(zeta) * (_x[3] - _x[1]) +
28                  W_(eps) * W_(zeta) * (_x[6] - _x[4]) +
29                  W_(eps) * W_(zeta) * (_x[7] - _x[5])); }
30
31  inline double JacobiMatrix::dydn(double eps, double eta, double zeta) {
32      return 0.5 * (W_(eps) * W_(zeta) * (_y[2] - _y[0]) +
33                  W_(eps) * W_(zeta) * (_y[3] - _y[1]) +
34                  W_(eps) * W_(zeta) * (_y[6] - _y[4]) +
35                  W_(eps) * W_(zeta) * (_y[7] - _y[5])); }
36
37  inline double JacobiMatrix::dzdn(double eps, double eta, double zeta) {
38      return 0.5 * (W_(eps) * W_(zeta) * (_z[2] - _z[0]) +
39                  W_(eps) * W_(zeta) * (_z[3] - _z[1]) +
40                  W_(eps) * W_(zeta) * (_z[6] - _z[4]) +
41                  W_(eps) * W_(zeta) * (_z[7] - _z[5])); }
42
43
44  inline double JacobiMatrix::dxdc(double eps, double eta, double zeta) {
45      return 0.5 * (W_(eps) * W_(eta) * (_x[4] - _x[0]) +
46                  W_(eps) * W_(eta) * (_x[5] - _x[1]) +
47                  W_(eps) * W_(eta) * (_x[6] - _x[2]) +
48                  W_(eps) * W_(eta) * (_x[7] - _x[3])); }
49
50  inline double JacobiMatrix::dydc(double eps, double eta, double zeta) {
51      return 0.5 * (W_(eps) * W_(eta) * (_y[4] - _y[0]) +
52                  W_(eps) * W_(eta) * (_y[5] - _y[1]) +
53                  W_(eps) * W_(eta) * (_y[6] - _y[2]) +
54                  W_(eps) * W_(eta) * (_y[7] - _y[3])); }
55
56  inline double JacobiMatrix::dzdc(double eps, double eta, double zeta) {
57      return 0.5 * (W_(eps) * W_(eta) * (_z[4] - _z[0]) +
58                  W_(eps) * W_(eta) * (_z[5] - _z[1]) +
59                  W_(eps) * W_(eta) * (_z[6] - _z[2]) +
60                  W_(eps) * W_(eta) * (_z[7] - _z[3])); }
61
62
63  void JacobiMatrix::SetValues(std::array<double, 8> x, std::array<double, 8> y,
64      ↪ std::array<double, 8> z) {
65      _x = x; _y = y; _z = z; }
66
67  std::function<double(double, double, double)> const JacobiMatrix::GetValueAt(size_t i,
68      ↪ size_t j) {
69      if (i == 0 and j == 0) return dxde;
70      if (i == 0 and j == 1) return dyde;
71      if (i == 0 and j == 2) return dzde;
72      if (i == 1 and j == 0) return dxdn;
73      if (i == 1 and j == 1) return dydn;
```

```

72     if (i == 1 and j == 2) return dzdn;
73     if (i == 2 and j == 0) return dxdc;
74     if (i == 2 and j == 1) return dydc;
75     if (i == 2 and j == 2) return dzdc;
76 }
77
78 std::function<double(double, double, double)> const
79   ↳ JacobiMatrix::GetValueAtInverse(size_t i, size_t j) {
80     if (i == 0 and j == 0)
81       return [](double t0, double t1, double t2) { return (dydn(t0, t1, t2) * dzdc(t0,
82         ↳ t1, t2) -
83           dydc(t0, t1, t2) * dzdn(t0, t1, t2)) / GetDeterminant()(t0, t1, t2); };
84     if (i == 0 and j == 1)
85       return [](double t0, double t1, double t2) { return -1.0 * (dyde(t0, t1, t2) *
86         ↳ dzdc(t0, t1, t2) -
87           dydc(t0, t1, t2) * dzde(t0, t1, t2)) / GetDeterminant()(t0, t1, t2); };
88     if (i == 0 and j == 2)
89       return [](double t0, double t1, double t2) { return (dyde(t0, t1, t2) * dzdn(t0,
90         ↳ t1, t2) -
91           dydn(t0, t1, t2) * dzde(t0, t1, t2)) / GetDeterminant()(t0, t1, t2); };
92     if (i == 1 and j == 0)
93       return [](double t0, double t1, double t2) { return -1.0 * (dxdn(t0, t1, t2) *
94         ↳ dzdc(t0, t1, t2) -
95           dxdc(t0, t1, t2) * dzdn(t0, t1, t2)) / GetDeterminant()(t0, t1, t2); };
96     if (i == 1 and j == 1)
97       return [](double t0, double t1, double t2) { return (dxde(t0, t1, t2) * dzdc(t0,
98         ↳ t1, t2) -
99           dxdc(t0, t1, t2) * dzde(t0, t1, t2)) / GetDeterminant()(t0, t1, t2); };
100    if (i == 1 and j == 2)
101      return [](double t0, double t1, double t2) { return -1.0 * (dxde(t0, t1, t2) *
102        ↳ dzdn(t0, t1, t2) -
103          dxdn(t0, t1, t2) * dzde(t0, t1, t2)) / GetDeterminant()(t0, t1, t2); };
104    if (i == 2 and j == 0)
105      return [](double t0, double t1, double t2) { return (dxdn(t0, t1, t2) * dydc(t0,
106        ↳ t1, t2) -
107          dxdc(t0, t1, t2) * dydn(t0, t1, t2)) / GetDeterminant()(t0, t1, t2); };
108    if (i == 2 and j == 1)
109      return [](double t0, double t1, double t2) { return -1.0 * (dxde(t0, t1, t2) *
110        ↳ dydc(t0, t1, t2) -
111          dxdc(t0, t1, t2) * dyde(t0, t1, t2)) / GetDeterminant()(t0, t1, t2); };
112    if (i == 2 and j == 2)
113      return [](double t0, double t1, double t2) { return (dxde(t0, t1, t2) * dydn(t0,
114        ↳ t1, t2) -
115          dxdn(t0, t1, t2) * dyde(t0, t1, t2)) / GetDeterminant()(t0, t1, t2); };
116  }
117
118 std::function<double(double, double, double)> const
119   ↳ JacobiMatrix::GetValueAtTransposed(size_t i, size_t j) { return GetValueAt(j, i); }
120
121 std::function<double(double, double, double)> const JacobiMatrix::GetDeterminant() {
122   return [](double t0, double t1, double t2) { return
123     dxde(t0, t1, t2) * dydn(t0, t1, t2) * dzdc(t0, t1, t2) +
124     dyde(t0, t1, t2) * dzdn(t0, t1, t2) * dxdc(t0, t1, t2) +
125     dzde(t0, t1, t2) * dxdn(t0, t1, t2) * dydc(t0, t1, t2) -
126     dzdn(t0, t1, t2) * dydn(t0, t1, t2) * dxdc(t0, t1, t2) -
127     dyde(t0, t1, t2) * dxdn(t0, t1, t2) * dzdc(t0, t1, t2) -
128     dxde(t0, t1, t2) * dzdn(t0, t1, t2) * dydc(t0, t1, t2); };
129 }

```

## Mesh.h

```

1  #pragma once
2
3  #include <cassert>
4  #include <cmath>
5  #include <string>
6  #include <algorithm>
7
8  #include "../DataTypes.h"
9  #include "../Logger/Logger.h"
10

```



```

11 class Mesh {
12 private:
13     // Based.
14     bool isGenerated_ = false;
15     bool isDeclarated_ = false;
16     size_t linesAmountX_ = 0;
17     size_t linesAmountY_ = 0;
18     size_t linesAmountZ_ = 0;
19     std::vector<Point> points_{};
20     size_t subdomainsAmount_ = 0;
21     std::vector<std::array<size_t, 7>> subdomains_{};
22     std::vector<AreaInfo> areasInfo_{};
23     std::vector<AreaRibs> areasRibs_{};
24     std::vector<std::pair<size_t, double_t>> delimitersX_{};
25     std::vector<std::pair<size_t, double_t>> delimitersY_{};
26     std::vector<std::pair<size_t, double_t>> delimitersZ_{};
27     size_t bordersAmount_ = 0;
28     std::vector<Border> borders_{};
29     std::vector<BorderLine> borderRibs_{};
30     std::vector<std::array<size_t, 6>> newBorders_{};
31     std::vector<AreaPoints> areasPoints_{};
32     std::vector<RibRef> referableRibs_{};
33     // Additional.
34     std::vector<Point> immutablePoints_{};
35     std::vector<std::array<size_t, 7>> immutableSubdomains_{};
36     std::vector<Border> immutableBorders_{};
37     std::vector<size_t> numRefsOfLinesAboveX{};
38     std::vector<size_t> numRefsOfLinesAboveY{};
39     std::vector<size_t> numRefsOfLinesAboveZ{};
40     void organizeBorders();
41 public:
42     Mesh() { Logger::ConsoleOutput("Mesh declared, but it's empty.",
43     ↪ NotificationColor::Warning); };
44     ~Mesh() {};
45     bool CheckData();
46     void CommitData(std::vector<std::string>* data);
47     inline bool isGenerated() const { return isGenerated_; }
48     inline bool isDeclarated() const { return isDeclarated_; }
49     inline size_t getLinesAmountX() const { return linesAmountX_; }
50     inline size_t getLinesAmountY() const { return linesAmountY_; }
51     inline size_t getLinesAmountZ() const { return linesAmountZ_; }
52     inline std::vector<Point> getPoints() const { return points_; }
53     inline std::vector<AreaRibs> getAreasAsRibs() const { return areasRibs_; }
54     inline std::vector<RibRef> getRibsRefs() const { return referableRibs_; }
55     inline std::vector<Border> getBorders() const { return borders_; }
56     inline std::vector<BorderLine> getBorderRibs() const { return borderRibs_; }
57     inline std::vector<std::array<size_t, 6>> getNewBorderRibs() const { return
58     ↪ newBorders_; }
59     inline std::vector<AreaInfo> getAreaInfo() const { return areasInfo_; }
60     __declspec(property(get = getLinesAmountX)) size_t LinesAmountX;
61     __declspec(property(get = getLinesAmountY)) size_t LinesAmountY;
62     __declspec(property(get = getLinesAmountZ)) size_t LinesAmountZ;
63     __declspec(property(get = isGenerated)) bool IsGenerated;
64     __declspec(property(get = isDeclarated)) bool IsDeclarated;
65     __declspec(property(get = getPoints)) std::vector<Point> Points;
66     __declspec(property(get = getAreaInfo)) std::vector<AreaInfo> AreasInfo;
67     friend class MeshGenerator;
68 };

```

## Mesh.cpp

```

1  #include "Mesh.h"
2
3  void Mesh::organizeBorders() {
4      size_t bordersIISwifted(0); size_t bordersIIISwifted(0);
5      for (auto& border : borders_) {
6          if (border.type_ == 2) {
7              std::iter_swap(borders_.begin() + bordersIISwifted, &border);
8              bordersIISwifted++; }
9          if (border.type_ == 3) {
10             std::iter_swap(borders_.begin() + bordersIISwifted + bordersIIISwifted,
11             ↪ &border);

```

```

11         bordersIIISwifted++; } }
12     }
13
14     bool Mesh::CheckData() {
15         if (linesAmountX_ * linesAmountY_ * linesAmountZ_ != points_.size()) return false;
16         if (linesAmountX_ - 1 != delimitersX_.size()) return false;
17         if (linesAmountY_ - 1 != delimitersY_.size()) return false;
18         if (linesAmountZ_ - 1 != delimitersZ_.size()) return false;
19         size_t maxLineX = 0; size_t maxLineY = 0; size_t maxLineZ = 0;
20         for (const auto& subdomain : subdomains_) {
21             maxLineX = maxLineX < subdomain[2] ? subdomain[2] : maxLineX;
22             maxLineY = maxLineY < subdomain[4] ? subdomain[4] : maxLineY;
23             maxLineZ = maxLineZ < subdomain[6] ? subdomain[6] : maxLineZ; }
24         if (linesAmountX_ - 1 != maxLineX) return false;
25         if (linesAmountY_ - 1 != maxLineY) return false;
26         if (linesAmountZ_ - 1 != maxLineZ) return false;
27         Logger::ConsoleOutput("Mesh checked and declared.", NotificationColor::Passed);
28         isDeclarated_ = true; return true;
29     }
30
31     void Mesh::CommitData(std::vector<std::string>* data) {
32         auto currentItem = data->begin(); // Select first item of vector.
33         // Commit lines amount above X,Y,Z axis.
34         linesAmountX_ = std::stoul(*currentItem); currentItem++;
35         linesAmountY_ = std::stoul(*currentItem); currentItem++;
36         linesAmountZ_ = std::stoul(*currentItem); currentItem++;
37         // Commit area description.
38         for (size_t i(0); i < linesAmountX_ * linesAmountY_ * linesAmountZ_; ++i) {
39             points_.emplace_back(std::stod(*currentItem), // X
40                                 std::stod(*(currentItem + 1)), // Y
41                                 std::stod(*(currentItem + 2))); // Z
42             currentItem += 3; }
43         immutablePoints_ = points_;
44         // Commit unique areas description.
45         subdomainsAmount_ = std::stoul(*currentItem); currentItem++;
46         for (size_t i(0); i < subdomainsAmount_; ++i) {
47             std::array<size_t, 7> currentArray = { std::stoul(*currentItem), // Formula num.
48             std::stoul(*(currentItem + 1)), // X0.
49             std::stoul(*(currentItem + 2)), // X1.
50             std::stoul(*(currentItem + 3)), // Y0.
51             std::stoul(*(currentItem + 4)), // Y1.
52             std::stoul(*(currentItem + 5)), // Z0.
53             std::stoul(*(currentItem + 6)) }; // Z1.
54             subdomains_.push_back(currentArray);
55             currentItem += 7; }
56         immutableSubdomains_ = subdomains_;
57         // Commit unique areas coefficients description.
58         for (size_t i(0); i < subdomainsAmount_; ++i) {
59             areasInfo_.emplace_back(std::stoul(*currentItem), // Area num.
60                                     std::stod(*(currentItem + 1)), // Mu_i.
61                                     std::stod(*(currentItem + 2))); // Sigma_i.
62             currentItem += 3; }
63         // Commit delimiters above X description.
64         for (size_t i(0); i < linesAmountX_ - 1; ++i) {
65             delimitersX_.emplace_back(std::stoul(*currentItem),
66                                     std::stod(*(currentItem + 1)));
67             currentItem += 2; }
68         // Commit delimiters above Y description.
69         for (size_t i(0); i < linesAmountY_ - 1; ++i) {
70             delimitersY_.emplace_back(std::stoul(*currentItem),
71                                     std::stod(*(currentItem + 1)));
72             currentItem += 2; }
73         // Commit delimiters above Z description.
74         for (size_t i(0); i < linesAmountZ_ - 1; ++i) {
75             delimitersZ_.emplace_back(std::stoul(*currentItem),
76                                     std::stod(*(currentItem + 1)));
77             currentItem += 2; }
78         // Commit information about borders.
79         bordersAmount_ = std::stoul(*currentItem); currentItem++;
80         for (size_t i(0); i < bordersAmount_; ++i) {
81             borders_.emplace_back(std::stoul(*currentItem), // Border type.
82                                  std::stoul(*(currentItem + 1)), // Border formula num.
83                                  std::stoul(*(currentItem + 2)), // X0.
84                                  std::stoul(*(currentItem + 3)), // X1.
85                                  std::stoul(*(currentItem + 4)), // Y0.

```

```

86         std::stoul(*(currentItem + 5)),          // Y1.
87         std::stoul(*(currentItem + 6)),          // Z0.
88         std::stoul(*(currentItem + 7)));         // Z1.
89         currentItem += 8; }
90     organizeBorders(); immutableBorders_ = borders_;
91 }

```

## MeshGenerator.h

```

1  #pragma once
2
3  #include "Mesh.h"
4
5  #include <cassert>
6  #include <vector>
7
8  typedef std::vector<std::vector<std::vector<Point>>> area3D;
9  typedef std::vector<std::vector<Point>> square2D;
10 typedef std::vector<Point> line1D;
11
12 class MeshGenerator {
13 private:
14     static int SelectAreaNum(Mesh& mesh, std::array<size_t, 12> arr);
15     static void GenerateListOfPoints(Mesh& mesh);
16     static void GenerateListOfAreas(Mesh& mesh);
17     static void GenerateListOfRibs(Mesh& mesh);
18     static void GenerateListOfBorders(Mesh& mesh);
19 public:
20     MeshGenerator() = delete;
21     static void Generate3DMesh(Mesh& mesh);
22 };

```

## MeshGenerator.cpp

```

1  #include "MeshGenerator.h"
2
3  void MeshGenerator::Generate3DMesh(Mesh& mesh) {
4      assert(mesh.IsDeclared);
5      for (size_t i(0); i < mesh.LinesAmountX; ++i) mesh.numRefsOfLinesAboveX.push_back(i);
6      for (size_t i(0); i < mesh.LinesAmountY; ++i) mesh.numRefsOfLinesAboveY.push_back(i);
7      for (size_t i(0); i < mesh.LinesAmountZ; ++i) mesh.numRefsOfLinesAboveZ.push_back(i);
8      GenerateListOfPoints(mesh); GenerateListOfRibs(mesh);
9      GenerateListOfAreas(mesh); GenerateListOfBorders(mesh);
10 }
11
12 int MeshGenerator::SelectAreaNum(Mesh& mesh, std::array<size_t, 12> arr) {
13     auto sxy = mesh.LinesAmountX * mesh.LinesAmountY;
14     auto p1 = mesh.referableRibs_[arr[0]].p1;
15     auto p2 = mesh.referableRibs_[arr[11]].p2;
16     auto lx0 = p1 % mesh.LinesAmountX;
17     auto lx1 = p2 % mesh.LinesAmountX;
18     auto ly0 = (p1 % sxy) / mesh.LinesAmountX;
19     auto ly1 = (p2 % sxy) / mesh.LinesAmountX;
20     auto lz0 = p1 / sxy; auto lz1 = p2 / sxy;
21     for (const auto& area : mesh.subdomains_)
22         if (mesh.numRefsOfLinesAboveX[area[1]] <= lx0 and
23             lx1 <= mesh.numRefsOfLinesAboveX[area[2]] and
24             mesh.numRefsOfLinesAboveY[area[3]] <= ly0 and
25             ly1 <= mesh.numRefsOfLinesAboveY[area[4]] and
26             mesh.numRefsOfLinesAboveZ[area[5]] <= lz0 and
27             lz1 <= mesh.numRefsOfLinesAboveZ[area[6]])
28             return area[0];
29     Logger::ConsoleOutput("Error during selection of area num", NotificationColor::Alert);
30     return NAN;
31 }
32
33 // Try to optimize memory.
34 void MeshGenerator::GenerateListOfPoints(Mesh& mesh) {
35     // Construct 3D area.
36     area3D figure{};

```

```

37 figure.resize(mesh.LinesAmountZ);
38 for (auto& square : figure) {
39     square.resize(mesh.LinesAmountY);
40     for (auto& line : square) line.resize(mesh.LinesAmountX); }
41 // Fill 3D area.
42 auto sxy = mesh.LinesAmountX * mesh.LinesAmountY;
43 auto lx = mesh.LinesAmountX;
44 for (size_t k(0); k < mesh.LinesAmountZ; ++k)
45     for (size_t j(0); j < mesh.LinesAmountY; ++j)
46         for (size_t i(0); i < mesh.LinesAmountX; ++i)
47             figure[k][j][i] = mesh.immutablePoints_[k * sxy + j * lx + i];
48 // Generation above X-axis.
49 for (size_t k(0); k < mesh.LinesAmountZ; ++k) {
50     for (size_t j(0); j < mesh.LinesAmountY; ++j) {
51         line1D lineToBuild{};
52         lineToBuild = figure[k][j];
53         size_t shift = mesh.LinesAmountX - 1;
54         auto iterOnXRefs = mesh.numRefsOfLinesAboveX.begin() + 1;
55         for (const auto& info : mesh.delimitersX_) {
56             auto rightBorderIter = lineToBuild.end() - shift;
57             auto amountOfDelimiters = info.first;
58             auto coefficientOfDelimiter = info.second;
59             *iterOnXRefs = *(iterOnXRefs - 1) + amountOfDelimiters;
60             iterOnXRefs++;
61             double denum = 0.0;
62             for (size_t ii(0); ii < amountOfDelimiters; ++ii)
63                 denum += pow(coefficientOfDelimiter, ii);
64             double x0 = (*(rightBorderIter - 1)).x; double x1 = (*rightBorderIter).x;
65             double y0 = (*(rightBorderIter - 1)).y; double y1 = (*rightBorderIter).y;
66             double z0 = (*(rightBorderIter - 1)).z; double z1 = (*rightBorderIter).z;
67             double deltax = x1 - x0; double deltay = y1 - y0; double deltax = z1 - z0;
68             double xh = deltax / denum; double yh = deltay / denum;
69             double zh = deltax / denum; double multiplier = 0.0;
70             for (size_t ii(0); ii < amountOfDelimiters - 1; ++ii) {
71                 multiplier += pow(coefficientOfDelimiter, ii);
72                 auto pointToInsert = Point(x0 + xh * multiplier,
73                     y0 + yh * multiplier, z0 + zh * multiplier);
74                 lineToBuild.insert(lineToBuild.end() - shift, pointToInsert);
75             } shift--;
76         } figure[k][j] = lineToBuild;
77     }
78 } mesh.linesAmountX_ = figure[0][0].size();
79 // Generation above Y-axis.
80 for (size_t k(0); k < mesh.LinesAmountZ; ++k) {
81     square2D squareToBuild{};
82     squareToBuild = figure[k];
83     size_t shift = mesh.LinesAmountY - 1;
84     auto iterOnYRefs = mesh.numRefsOfLinesAboveY.begin() + 1;
85     for (const auto& info : mesh.delimitersY_) {
86         auto rightBorderIter = squareToBuild.end() - shift;
87         auto amountOfDelimiters = info.first;
88         auto coefficientOfDelimiter = info.second;
89         *iterOnYRefs = *(iterOnYRefs - 1) + amountOfDelimiters;
90         iterOnYRefs++;
91         line1D v0(mesh.LinesAmountX); line1D v1(mesh.LinesAmountX);
92         std::copy((*(rightBorderIter - 1)).begin(), (*(rightBorderIter - 1)).end(),
93             ↪ v0.begin());
93         std::copy((*(rightBorderIter)).begin(), (*(rightBorderIter)).end(),
94             ↪ v1.begin());
94         square2D subSquareToBuild(amountOfDelimiters - 1);
95         for (auto& line : subSquareToBuild) line.resize(mesh.LinesAmountX);
96         double denum = 0.0;
97         for (size_t ii(0); ii < amountOfDelimiters; ++ii)
98             denum += pow(coefficientOfDelimiter, ii);
99         for (size_t i(0); i < mesh.LinesAmountX; ++i) {
100             double x0 = v0[i].x; double x1 = v1[i].x;
101             double y0 = v0[i].y; double y1 = v1[i].y;
102             double z0 = v0[i].z; double z1 = v1[i].z;
103             double deltax = x1 - x0; double deltay = y1 - y0; double deltax = z1 - z0;
104             double xh = deltax / denum; double yh = deltay / denum;
105             double zh = deltax / denum; double multiplier = 0.0;
106             for (size_t ii(0); ii < amountOfDelimiters - 1; ++ii) {
107                 multiplier += pow(coefficientOfDelimiter, ii);
108                 auto pointToInsert = Point(x0 + xh * multiplier,
109                     y0 + yh * multiplier, z0 + zh * multiplier);
109                 subSquareToBuild[ii][i] = pointToInsert;
110

```

```

111     }}
112     for (auto line : subSquareToBuild)
113         squareToBuild.insert(squareToBuild.end() - shift, line);
114     shift--;
115     } figure[k] = squareToBuild;
116 } mesh.linesAmountY_ = figure[0].size();
117 // Generate above Z-axis.
118 area3D areaToBuild{};
119 areaToBuild = figure;
120 size_t shift = mesh.LinesAmountZ - 1;
121 auto iterOnZRefs = mesh.numRefsOfLinesAboveZ.begin() + 1;
122 for (const auto& info : mesh.delimitersZ_) {
123     auto rightBorderIter = areaToBuild.end() - shift;
124     auto amountOfDelimiters = info.first;
125     auto coefficientOfDelimiter = info.second;
126     *iterOnZRefs = *(iterOnZRefs - 1) + amountOfDelimiters;
127     iterOnZRefs++;
128     square2D s0(mesh.LinesAmountY); for (auto& line : s0)
129         line.resize(mesh.LinesAmountX);
130     square2D s1(mesh.LinesAmountY); for (auto& line : s1)
131         line.resize(mesh.LinesAmountX);
132     std::copy((*(rightBorderIter - 1)).begin(), (*(rightBorderIter - 1)).end(),
133             s0.begin());
134     std::copy((*(rightBorderIter)).begin(), (*(rightBorderIter)).end(), s1.begin());
135     area3D subAreaToBuild(amountOfDelimiters - 1);
136     for (auto& square : subAreaToBuild) {
137         square.resize(mesh.LinesAmountY);
138         for (auto& line : square)
139             line.resize(mesh.LinesAmountX); }
140     double denum = 0.0;
141     for (size_t ii(0); ii < amountOfDelimiters; ++ii)
142         denum += pow(coefficientOfDelimiter, ii);
143     for (size_t j(0); j < mesh.LinesAmountY; ++j) {
144         for (size_t i(0); i < mesh.LinesAmountX; ++i) {
145             double x0 = s0[j][i].x; double x1 = s1[j][i].x;
146             double y0 = s0[j][i].y; double y1 = s1[j][i].y;
147             double z0 = s0[j][i].z; double z1 = s1[j][i].z;
148             double delTX = x1 - x0; double delTY = y1 - y0;
149             double delTZ = z1 - z0;
150             double xh = delTX / denum;
151             double yh = delTY / denum;
152             double zh = delTZ / denum;
153             double multiplier = 0.0;
154             for (size_t ii(0); ii < amountOfDelimiters - 1; ++ii) {
155                 multiplier += pow(coefficientOfDelimiter, ii);
156                 auto pointToInsert = Point(x0 + xh * multiplier,
157                                             y0 + yh * multiplier,
158                                             z0 + zh * multiplier);
159                 subAreaToBuild[ii][j][i] = pointToInsert; } } }
160     for (auto square : subAreaToBuild)
161         areaToBuild.insert(areaToBuild.end() - shift, square);
162     shift--;
163 } figure = areaToBuild;
164 mesh.linesAmountZ_ = figure.size();
165 // Convert borders array.
166 for (auto& border : mesh.borders_) {
167     border.refs_[0] = mesh.numRefsOfLinesAboveX[border.refs_[0]];
168     border.refs_[1] = mesh.numRefsOfLinesAboveX[border.refs_[1]];
169     border.refs_[2] = mesh.numRefsOfLinesAboveY[border.refs_[2]];
170     border.refs_[3] = mesh.numRefsOfLinesAboveY[border.refs_[3]];
171     border.refs_[4] = mesh.numRefsOfLinesAboveZ[border.refs_[4]];
172     border.refs_[5] = mesh.numRefsOfLinesAboveZ[border.refs_[5]]; }
173 // Convert to line-format.
174 mesh.linesAmountZ_ = figure.size();
175 mesh.points_.resize(mesh.linesAmountX * mesh.linesAmountY * mesh.linesAmountZ_);
176 sxy = mesh.LinesAmountX * mesh.LinesAmountY;
177 lx = mesh.LinesAmountX;
178 for (size_t k(0); k < mesh.LinesAmountZ; ++k)
179     for (size_t j(0); j < mesh.LinesAmountY; ++j)
180         for (size_t i(0); i < mesh.LinesAmountX; ++i)
181             mesh.points_[k * sxy + j * lx + i] = figure[k][j][i];
182 }
183
184 void MeshGenerator::GenerateListOfAreas(Mesh& mesh) {
185     size_t rx = mesh.LinesAmountX - 1; size_t ry = mesh.LinesAmountY - 1;
186     size_t nx = mesh.LinesAmountX; size_t ny = mesh.LinesAmountY;

```

```

184     size_t rxy = rx * ny + ry * nx; size_t nxy = nx * ny;
185     size_t nz = mesh.LinesAmountZ;
186     for (size_t k(0); k < nz - 1; ++k)
187         for (size_t j(0); j < ny - 1; ++j)
188             for (size_t i(0); i < nx - 1; ++i) {
189                 size_t curr = i + j * (nx + rx) + k * (rxy + nxy);
190                 std::array<size_t, 12> arrI = {
191                     curr, curr + rx, curr + rx + 1, curr + rx + nx,
192                     curr + rxy - j * rx, curr + rxy + 1 - j * rx, curr + rxy + nx - j *
193                     ↪ rx, curr + rxy + nx + 1 - j * rx,
194                     curr + rxy + nxy, curr + rxy + nxy + rx, curr + rxy + nxy + rx + 1,
195                     ↪ curr + rxy + nxy + rx + nx };
196                 int areaNumber = SelectAreaNum(mesh, arrI);
197                 mesh.areasRibs_.emplace_back(areaNumber, arrI); }
198 }
199
200 void MeshGenerator::GenerateListOfRibs(Mesh& mesh) {
201     auto nx = mesh.LinesAmountX; auto ny = mesh.LinesAmountY;
202     auto nz = mesh.LinesAmountZ;
203     auto nxny = mesh.LinesAmountX * mesh.LinesAmountY;
204     for (int k = 0; k < mesh.LinesAmountZ; k++) {
205         for (int j = 0; j < mesh.LinesAmountY; j++) {
206             for (int i = 0; i < mesh.LinesAmountX - 1; i++)
207                 mesh.referableRibs_.emplace_back(k * nxny + nx * j + i, k * nxny + nx * j
208                 ↪ + i + 1);
209             if (j != mesh.LinesAmountY - 1)
210                 for (int i = 0; i < nx; i++)
211                     mesh.referableRibs_.emplace_back(k * nxny + nx * j + i, k * nxny + nx
212                     ↪ * (j + 1) + i); }
213             if (k != mesh.LinesAmountZ - 1)
214                 for (int j = 0; j < mesh.LinesAmountY; j++)
215                     for (int i = 0; i < mesh.LinesAmountX; i++)
216                         mesh.referableRibs_.emplace_back(k * nxny + nx * j + i, (k + 1) *
217                         ↪ nxny + nx * j + i); }
218         Logger::ConsoleOutput("Ribs array generated.", NotificationColor::Passed);
219     }
220 }
221
222 void MeshGenerator::GenerateListOfBorders(Mesh& mesh) {
223     Logger::ConsoleOutput("Borders generates just for 1st type and formula num 1!",
224     ↪ NotificationColor::Warning);
225     auto nx = mesh.getLinesAmountX(); auto ny = mesh.getLinesAmountY();
226     auto nz = mesh.getLinesAmountZ(); auto nxny = nx * ny;
227     auto rxy = (nx - 1) * ny + (ny - 1) * nx;
228     // XOY
229     for (size_t i = 0; i < ny - 1; i++)
230         for (size_t j = 0; j < nx - 1; j++)
231             mesh.newBorders_.push_back(std::array<size_t, 6> {1, 1,
232             ↪ i * (2 * nx - 1) + j, i * (2 * nx - 1) + j + nx - 1,
233             ↪ i * (2 * nx - 1) + j + nx, i * (2 * nx - 1) + j + nx + nx - 1});
234     // XOZ
235     for (size_t i = 0; i < nz - 1; i++)
236         for (size_t j = 0; j < nx - 1; j++)
237             mesh.newBorders_.push_back(std::array<size_t, 6> {1, 1,
238             ↪ i * (rxy + nxny) + j, i * (rxy + nxny) + j + rxy,
239             ↪ i * (rxy + nxny) + j + rxy + 1, i * (rxy + nxny) + j + rxy + nxny});
240     // OYZ
241     for (size_t i = 0; i < nz - 1; i++)
242         for (size_t j = 0; j < ny - 1; j++)
243             mesh.newBorders_.push_back(std::array<size_t, 6> {1, 1,
244             ↪ nx - 1 + i * (rxy + nxny) + j * (2 * nx - 1), rxy + i * (rxy + nxny) + j
245             ↪ * nx,
246             ↪ rxy + nx + i * (rxy + nxny) + j * nx, rxy + nxny + nx - 1 + i * (rxy +
247             ↪ nxny) + j * (2 * nx - 1)});
248     // XY1
249     for (size_t i = 0; i < ny - 1; i++)
250         for (size_t j = 0; j < nx - 1; j++)
251             mesh.newBorders_.push_back(std::array<size_t, 6> {1, 1,
252             ↪ (nz - 1) * (rxy + nxny) + j + i * (2 * nx - 1),
253             ↪ (nz - 1) * (rxy + nxny) + nx - 1 + j + i * (2 * nx - 1),
254             ↪ (nz - 1) * (rxy + nxny) + nx + j + i * (2 * nx - 1),
255             ↪ (nz - 1) * (rxy + nxny) + nx + nx - 1 + j + i * (2 * nx - 1)});
256     // X1Z
257     for (size_t i = 0; i < nz - 1; i++)
258         for (size_t j = 0; j < nx - 1; j++)
259             mesh.newBorders_.push_back(std::array<size_t, 6> {1, 1,

```

```

251         (ny - 1)* nx + (ny - 1) * (nx - 1) + j + i * (rxy + nxny),
252         (ny - 1)* nx + (ny - 1) * (nx - 1) + nxny - 1 + j + i * (rxy + nxny),
253         (ny - 1)* nx + (ny - 1) * (nx - 1) + nxny + j + i * (rxy + nxny),
254         (ny - 1)* nx + (ny - 1) * (nx - 1) + nxny + j + rxy + i * (rxy + nxny)});
255 // 1YZ
256 for (size_t i = 0; i < nz - 1; i++)
257     for (size_t j = 0; j < ny - 1; j++)
258         mesh.newBorders_.push_back(std::array<size_t, 6> {1, 1,
259             nx - 1 + i * (rxy + nxny) + j * (2 * nx - 1) + nx - 1,
260             rxy + i * (rxy + nxny) + j * nx + nx - 1,
261             rxy + nx + i * (rxy + nxny) + j * nx + nx - 1,
262             rxy + nxny + nx - 1 + i * (rxy + nxny) + j * (2 * nx - 1) + nx - 1});
263 }

```

## LOS.h

```

1  #pragma once
2
3  #include "Solver.h"
4
5  class LOS : public Solver {
6  public:
7      LOS() : Solver() {};
8      LOS(double eps, size_t maxIters) : Solver(eps, maxIters) {};
9      ~LOS() {};
10     GlobalVector Solve(const GlobalMatrix& A, const GlobalVector& b) const override;
11 };
12

```

## LOS.cpp

```

1  #include "LOS.h"
2
3  GlobalVector LOS::Solve(const GlobalMatrix& A, const GlobalVector& b) const {
4      GlobalVector x(b.Size()); GlobalVector _x(b.Size());
5      GlobalVector r(b.Size()); GlobalVector _r(b.Size());
6      GlobalVector z(b.Size()); GlobalVector _z(b.Size());
7      GlobalVector p(b.Size()); GlobalVector _p(b.Size());
8      double alph(0.0); double beta(0.0);
9      r = b - A * x; z = r; p = A * r; size_t iter(0);
10     do {
11         _x = x; _z = z; _r = r; _p = p;
12         alph = (_p * _r) / (_p * _p);
13         x = _x + alph * _z; r = _r - alph * _p;
14         beta = -1.0 * (_p * (A * r)) / (_p * _p);
15         z = r + beta * _z; p = A * r + beta * _p;
16         if (iter % 4 == 0) std::cout << iter << ". " << std::scientific << r.Norma() /
            << b.Norma() << std::endl;
17         ++iter;
18     } while (iter < _maxIters and r.Norma() / b.Norma() >= _eps);
19     return x;
20 }

```