СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Разработка программного обеспечения для моделирования трехмерного электромагнитного поля представляет собой сложную и актуальную задачу, особенно в контексте современных требований к точности и эффективности расчетов. Одним из перспективных подходов к решению таких задач является использование векторного метода конечных элементов (МКЭ), который обеспечивает высокую точность моделирования.

В данной работе рассматривается разработка программы для моделирования электромагнитного поля на шестигранных элементах, что особенно важно для задач, связанных с анализом поведения электромагнитного поля в электроразведке. Применение шестигранников в качестве базовых элементов сетки позволяет более гибко аппроксимировать сложные трехмерные структуры, что делает данный подход особенно востребованным для решения практических задач.

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:

$$\mathrm{rot}\left(\frac{1}{\mu}\mathrm{rot}\overrightarrow{\mathbf{A}}\right)+\sigma\frac{\partial\overrightarrow{\mathbf{A}}}{\partial t}+\epsilon\frac{\partial^2\overrightarrow{\mathbf{A}}}{\partial t^2}=\overrightarrow{\mathbf{J}}^{\mathbf{ст}}. \tag{1.1}$$

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:

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

В свою очередь, учёт от объектов, имеющих неоднородные значения удельной электропроводности, осуществляется за счёт математической модели, описываемой уравнением (1.2)

$$\mathrm{rot}\left(\frac{1}{\mu_0}\mathrm{rot}\overrightarrow{\mathbf{A}}^{+}\right)+\sigma\frac{\partial\overrightarrow{\mathbf{A}}^{+}}{\partial t}=(\sigma-\sigma_{\mathrm{n}})\,\overrightarrow{\mathbf{E}}_{\mathrm{n}}. \tag{1.2}$$

Для тестирования на правильность решения дифференциального уравнения (1.2) будем использовать уравнение, правая часть которого представляется в виде вектор-функции $\overrightarrow{\mathbf{F}}$, а также будет иметь место быть слагаемое $\gamma\overrightarrow{\mathbf{A}}$ в левой части уравнения:

$$\mathrm{rot}\left(\frac{1}{\mu_0}\mathrm{rot}\overrightarrow{\mathbf{A}}\right)+\gamma\overrightarrow{\mathbf{A}}+\sigma\frac{\partial\overrightarrow{\mathbf{A}}}{\partial t}=\overrightarrow{\mathbf{F}}. \tag{1.3}$$

1.2. ВАРИАЦИОННАЯ ПОСТАНОВКА

Будем считать, что на границе $S=S_1\cup S_2$ расчётной области Ω, в которой определено уравнение (1.3), заданы краевые условия двух типов:

$$\left(\overrightarrow{\mathbf{A}}\times\overrightarrow{\mathbf{n}}\right)\Big|_{S_1}=\overrightarrow{\mathbf{A}}^{g}\times\overrightarrow{\mathbf{n}}, \tag{1.4}$$

$$\left(\frac{1}{\mu}\mathrm{rot}\overrightarrow{\mathbf{A}}\times\overrightarrow{\mathbf{n}}\right)\Big|_{S_1}=\overrightarrow{\mathbf{H}}^{\Theta}\times\overrightarrow{\mathbf{n}}. \tag{1.5}$$

Тогда эквивалентная вариационная формулировка в форме Галёркина для уравнения (1.3) без производной по времени, и с учётом краевых условий (1.4) - (1.5) имеет вид:

$$\begin{aligned}\int\limits_{\Omega}\frac{1}{\mu_0}\mathrm{rot}\overrightarrow{\mathbf{A}}\cdot\mathrm{rot}\overrightarrow{\Psi}\,d\Omega+\int\limits_{\Omega}\gamma\overrightarrow{\mathbf{A}}\cdot\overrightarrow{\Psi}\,d\Omega=\int\limits_{\Omega}\overrightarrow{\mathbf{F}}\cdot\overrightarrow{\Psi}\,d\Omega+\\ +\int\limits_{S_2}\left(\overrightarrow{\mathbf{H}}^{\Theta}\times\overrightarrow{\mathbf{n}}\right)\cdot\overrightarrow{\Psi}\,dS\qquad\forall\overrightarrow{\Psi}\in H_0^{rot}.\end{aligned} \tag{1.6}$$

1.3. КОНЕЧНОЭЛЕМЕНТНАЯ ДИСКРЕТИЗАЦИЯ

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

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

$$x = \sum_{i=1}^{8} \hat{\varphi}_i(\xi,\eta,\zeta)\hat{x}_i, \qquad y = \sum_{i=1}^{8} \hat{\varphi}_i(\xi,\eta,\zeta)\hat{y}_i, \qquad z = \sum_{i=1}^{8} \hat{\varphi}_i(\xi,\eta,\zeta)\hat{z}_i, \quad (1.7)$$

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

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

$$\hat{\psi}_i(x,y,z) = \mathbf{J}^{-1}\hat{\varphi}_i(\xi(x,y,z),\eta(x,y,z),\zeta(x,y,z)), \quad (1.8)$$

где

$$\mathbf{J} = \begin{pmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} & \frac{\partial z}{\partial \xi} \\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta} & \frac{\partial z}{\partial \eta} \\ \frac{\partial x}{\partial \zeta} & \frac{\partial y}{\partial \zeta} & \frac{\partial z}{\partial \zeta} \end{pmatrix} \quad (1.9)$$

– функциональная матрица преобразования координат, переводящего кубик Ω^E в шестигранник Ω_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 Ω_m в мастер-элемент Ω_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:

$$\hat{\mathrm{G}}_{ij} = \int\limits_{\Omega_e} \frac{1}{\mu_0} \mathrm{rot}\hat{\psi}_i \cdot \mathrm{rot}\hat{\psi}_j d\Omega = \\ = \int\limits_{-1}^{1}\int\limits_{-1}^{1}\int\limits_{-1}^{1} \frac{1}{\mu_0}\frac{1}{|J|} \left(\mathbf{J}^{\mathrm{T}}\mathrm{rot}\hat{\varphi}_i\right) \cdot \left(\mathbf{J}^{\mathrm{T}}\mathrm{rot}\hat{\varphi}_j\right) \, d\xi d\eta d\zeta \quad , \tag{1.10}$$

а матрица масс, в свою очередь, по формуле:

$$\hat{\mathrm{M}}_{ij} = \int\limits_{\Omega_e} \gamma\hat{\psi}_i \cdot \hat{\psi}_j d\Omega = \int\limits_{-1}^{1}\int\limits_{-1}^{1}\int\limits_{-1}^{1} \gamma \left(\mathbf{J}^{\text{-}1}\hat{\varphi}_i\right) \cdot \left(\mathbf{J}^{\text{-}1}\hat{\varphi}_j\right) |J| \, d\xi d\eta d\zeta. \tag{1.11}$$

При расчёте локального вектора правой части будем использовать формулу:

$$\hat{\mathbf{b}}^{\mathbf{J^{CT}}} = \hat{\mathbf{M}}\,\hat{\mathbf{f}}. \tag{1.12}$$

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

Для произвольных шестигранников интегралы в соотношениях (1.10) - (1.11) берутся численно. Соответствующая схема вычисления значения инте-

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

$$\int\limits_{-1}^{1}\int\limits_{-1}^{1}\int\limits_{-1}^{1} f\left(\xi,\eta,\zeta\right)d\xi d\eta d\zeta \approx \sum_{k=1}^{n}\sum_{l=1}^{n}\sum_{r=1}^{n} f\left(t_k,t_l,t_r\right). \tag{1.13}$$

1.5. УЧЁТ КРАЕВЫХ УСЛОВИЙ

При решении уравнения (1.2) с использованием векторного МКЭ базисные вектор-функции $\overrightarrow{\psi}_i$ строятся так, что все базисные функции конечномерного пространства $\mathbf{V}^{\mathrm{rot}}$ с индексами $i \in N_0$ имеют нулевые касательные к S_1 составляющие (они образуют базис подпространства $\mathbf{V}_0^{\mathrm{rot}}$). Поэтому для выполнения однородных главных краевых условий достаточно к n_0 уравнениям системы (1.6) добавить $n-n_0$ уравнений вида:

$$q_i = 0, \qquad i \in N\backslash N_0.$$

Для учёта неоднородного краевого условия (1.4), 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 $\overrightarrow{\mathbf{A}}^h = \sum\limits_{j\in N} q_j^n \overrightarrow{\psi}_j$ к касательным составляющим вектора $\overrightarrow{\mathbf{A}}^g$.

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 $\overrightarrow{\mathbf{A}}^g$ из правой части краевых условий (1.4) заменяется некоторым интерполянтом $\overrightarrow{\mathbf{A}}^{g,h}$, представленными в виде линейной комбинации базисных вектор-функций:

$$\overrightarrow{\mathbf{A}}^{g,h} = \sum_j q_j^g \overrightarrow{\psi}_j. \tag{1.14}$$

Веса q_j^g в разложении (1.14) можно найти следующим образом. Поскольку в векторном МКЭ базисные функции $\overrightarrow{\psi}_i$ строятся так, что на поверхности S_1 ненулевые касательные имеют только $n - n_0$ базисных вектор-функций с номерами $i \in N \backslash N_0$, 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 $\overrightarrow{\tau}_i$ такой, что

$$\overrightarrow{\psi}_i(x_i, y_i, z_i) \cdot \overrightarrow{\tau}_i \neq 0, \qquad \overrightarrow{\psi}_j(x_i, y_i, z_i) \cdot \overrightarrow{\tau}_i = 0, \qquad \forall j \neq i. \tag{1.15}$$

Домножим левую и правую части уравнения (1.14) скалярно на вектор $\overrightarrow{\tau}_i$ в точке (x_i, y_i, z_i) с учётном (1.15) получим

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

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

$$\overrightarrow{\mathbf{A}}^{g,h} = \sum_{i \in N \backslash N_0} q_j^g \overrightarrow{\psi_j},$$

получим следующее выражение для учёта неоднородного главного краевого условия:

$$q_i = \frac{\overrightarrow{\mathbf{A}}^g(x_i, y_i, z_i) \cdot \overrightarrow{\tau}_i}{\overrightarrow{\psi}_i(x_i, y_i, z_i) \cdot \overrightarrow{\tau}_i}, \qquad i \in N \backslash N_0.$$

1.6. РЕШАТЕЛЬ СЛАУ PARDISO

PARDISO (Parallel Direct Sparse Solver) — это высокопроизводительная библиотека для решения систем линейных алгебраических уравнений (СЛАУ) с разреженными матрицами. Хотя PARDISO изначально разработан как прямой решатель (direct solver), он также поддерживает итерационные методы (iterative methods) для решения задач, где прямое решение может быть неэффективным или требовать слишком много ресурсов.

К основным особенностям PARDISO можно отнести несколько пунктов. Во-первых решатель оптимизирован для работы с разреженными матрицами, которые часто возникают в задачах математического моделирования, физики, инженерии и других областях. Разреженные матрицы содержат большое количество нулевых элементов, и PARDISO эффективно использует эту структуру для уменьшения вычислительных затрат.

Во-вторых это гибкость в выборе методов. PARDISO поддерживает как прямые методы (например, LU- или Cholesky-разложение), так и итерационные методы (например, метод сопряжённых градиентов, GMRES). Это позволяет выбирать наиболее подходящий метод в зависимости от задачи.

В-третьих имеется поддержка параллельных вычислений. PARDISO разработан для эффективного использования многопроцессорных систем и графических ускорителей (GPU). Это делает его одним из самых быстрых решателей для задач с большими разреженными матрицами.

Также при использовании решателя имеется поддержка различных типов матриц. PARDISO работает с симметричными, несимметричными, положительно определёнными и неопределёнными матрицами. Это делает его универсальным инструментом для широкого круга задач.

И наконец имеется возможность интеграции с популярными платформами. PARDISO легко интегрируется с такими языками программирования, как C, C++, Fortran, а также с математическими пакетами, например, MATLAB.

Хотя PARDISO изначально ориентирован на прямые методы, он также поддерживает итерационные методы для решения СЛАУ. Это особенно полезно для задач с очень большими разреженными матрицами, где прямое решение может быть слишком затратным по памяти или времени. Например, метод сопряжённых градиентов, который эффективен для симметричных положительно определённых матриц и если матрица хорошо обусловлена, то сходится за небольшое количество итераций. Также можно использовать метод обобщённый минимальных невязок (GMRES). Он подходит для несимметричных матриц. Использует процесс Арнольди для построения ортогонального базиса. Метод бисопряжённых градиентов подходит для несимметричных матриц, более устойчив, чем классический метод BiCG. Также имеется метод минимальных невязок (MINRES) в котором используется для симметричных неопределённых матриц.

К преимуществам использования PARDISO можно отнести высокую производительность (PARDISO использует современные алгоритмы и оптимизации для достижения максимальной скорости решения), экономию памяти (благодаря работе с разреженными матрицами PARDISO минимизирует использование памяти), гибкость (поддержка как прямых, так и итерационных методов позволяет адаптировать решение под конкретную задачу) и масштабируемость (PARDISO эффективно использует многопроцессорные системы, что делает его пригодным для решения задач высокой размерности).

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 перечисленны 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 μ и σ. 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 μ и σ 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).

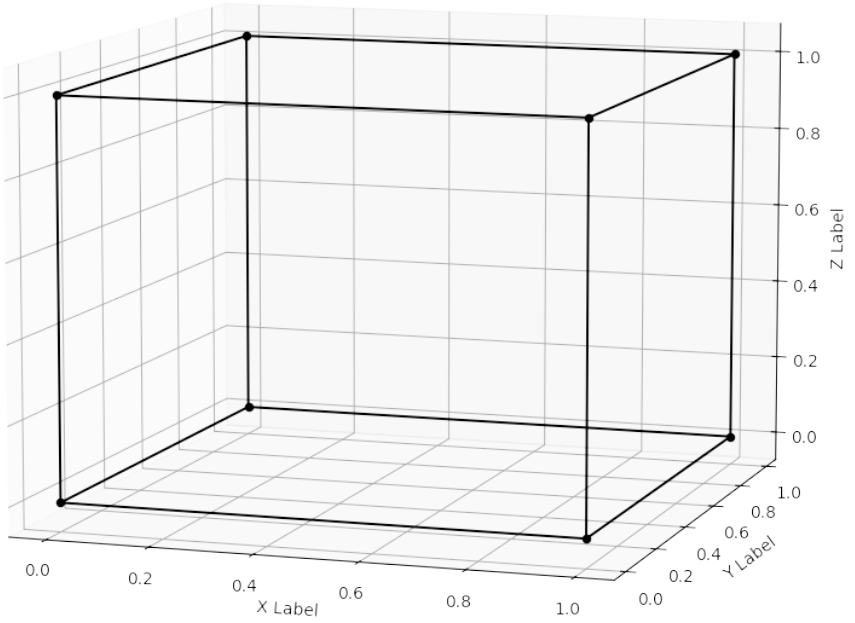


Рисунок 2.1 – Расчетная область для кубика

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

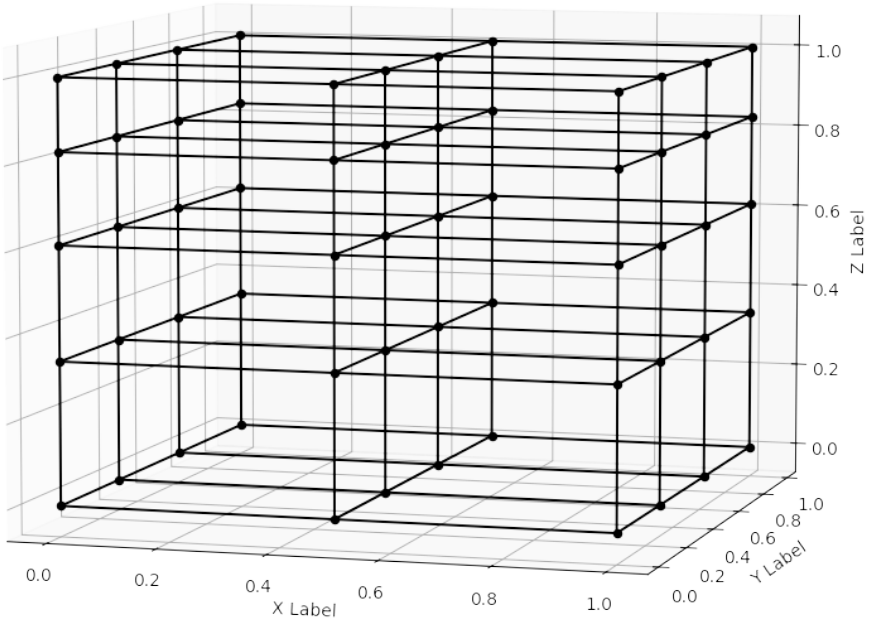
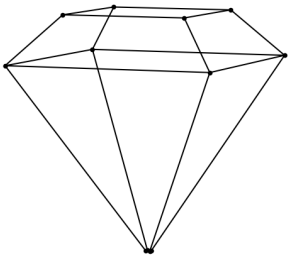
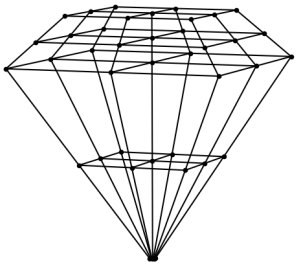


Рисунок 2.2 – Секта для кубика

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

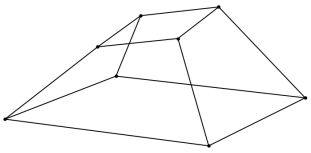


а)

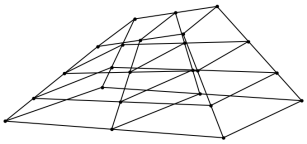


б)

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

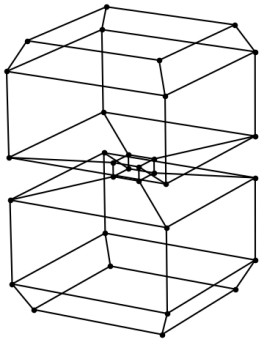


а)

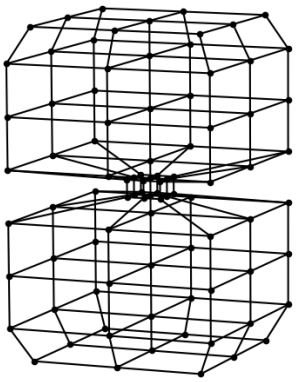


б)

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

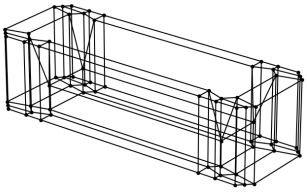


а)

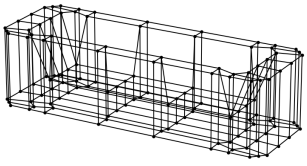


б)

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

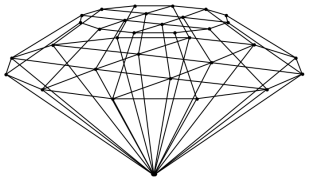


а)

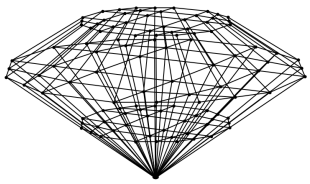


б)

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 сложные геометрически фигуры.

2.2. ЧИСЛЕННОЕ ИНТЕГРИРОВАНИЕ

При расчёте элементов локальных матриц жёсткости (1.10) и масс (1.11) 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

Таким образом, программа численного интегрирования методами Гаусса находит верное решение на соответствующем порядке.

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

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

где $\mathbf{0}$ - матрица размером 4×4, 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 (1.11). Элементы сгенерированной матрицы в виде консольного вывода программы изображены на рисунке 2.8.

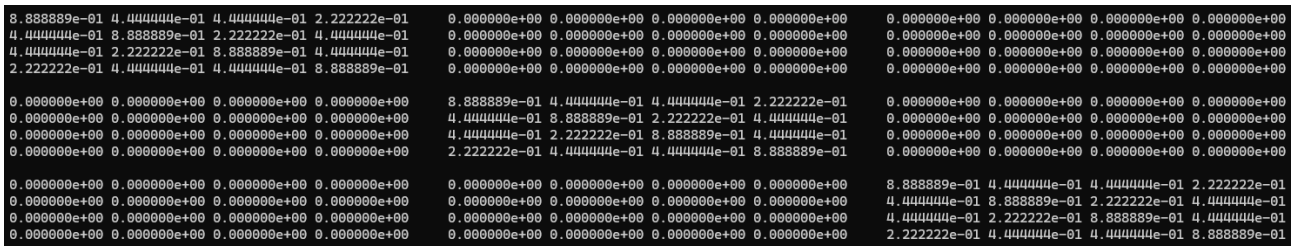


Рисунок 2.8 – Консольный вывод сгенерированной матрицы масс на Ω_E

Локальная матрица жёсткости $\hat{\mathbf{G}}$ на параллелепипеде $\Omega_E \in [-1,1]_x \times [-1,1]_y \times [-1,1]_z$ при $\overline{\mu} = 1$ принимает вид:

$$\hat{\mathbf{G}} = \begin{pmatrix} \frac{1}{3}\mathbf{G}_1 + \frac{1}{3}\mathbf{G}_2 & -\frac{1}{3}\mathbf{G}_2 & \frac{1}{3}\mathbf{G}_3 \\ -\frac{1}{3}\mathbf{G}_2 & \frac{1}{3}\mathbf{G}_1 + \frac{1}{3}\mathbf{G}_2 & -\frac{1}{3}\mathbf{G}_1 \\ \frac{1}{3}\mathbf{G}_3^{\mathrm{T}} & -\frac{1}{3}\mathbf{G}_1 & \frac{1}{3}\mathbf{G}_1 + \frac{1}{3}\mathbf{G}_2 \end{pmatrix},$$

где

$$\mathbf{G}_1 = \begin{pmatrix} 2 & 1 & -2 & -1 \\ 1 & 2 & -1 & -2 \\ -2 & -1 & 2 & 1 \\ -1 & -2 & 1 & 2 \end{pmatrix}, \qquad \mathbf{G}_2 = \begin{pmatrix} 2 & -2 & 1 & -1 \\ -2 & 2 & -1 & 2 \\ 1 & -1 & 2 & -2 \\ -1 & 1 & -2 & 2 \end{pmatrix},$$

$$\mathbf{G}_3 = \begin{pmatrix} -2 & 2 & -1 & 1 \\ -1 & 1 & -2 & 2 \\ 2 & -2 & 1 & -1 \\ 1 & -1 & 2 & -2 \end{pmatrix}.$$

Теперь, проверим генерацию матрицы жёсткости через формулу (1.10). Консольный вывод содержимого матрицы указан на рисунке 2.9

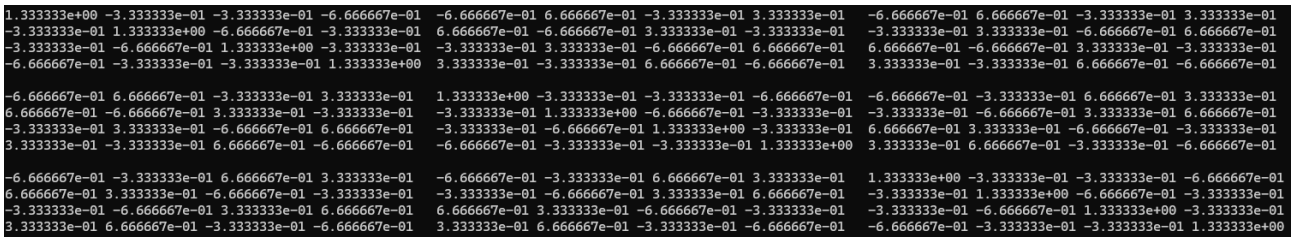


Рисунок 2.9 – Консольный вывод сгенерированной матрицы масс на Ω_E

2.4. ИСПОЛЬЗОВАНИЕ PARDISO

Использование PARDISO в C++ требует подключения библиотеки PARDISO и настройки параметров для решения системы линейных уравнений. PARDISO поставляется в составе Intel MKL (Math Kernel Library) или как отдельная библиотека.

PARDISO требует настройки множества параметров, таких как указатель на внутренние данные PARDISO (указатель на массив из 64 элементов, который хранит внутренние данные PARDISO, массив должен быть инициализирован нулями перед первым вызовом PARDISO и используется для хранения информации о факторизации и других внутренних данных), максимальное количество факторов (обычно 1, но если решать несколько систем с одинаковой матрицей, но разными правыми частями, это значение может быть больше 1), номер фактора, тип матрицы (1-вещественная симметричная положительно определённая, 2-вещественная симметричная неопределённая, -2 - вещественная симметричная неопределённая (альтернативный вариант), 11 - вещественная несимметричная, 6 - комплексная симметричная, 13 - комплексная несимметричная.), фаза решения (11 - анализ, 12 - анализ, факторизация и решение, 13 - анализ, факторизация и решение с итерационным уточнением, 22 - факторизация, 33 - решение, -1 - освобождение памяти), размер матрицы (число строк/столбцов), указатель на массив ненулевых элементов матрицы в разреженном формате, указатель на массив индексов начала строк в разреженном формате, указатель на массив индексов столбцов для каждого

ненулевого элемента в разреженном формате, массив перестановок (обычно NULL), количество правых частей (обычно 1), массив из 64 элементов, содержащий параметры решения (iparm[0] - использовать ли пользовательские настройки, iparm[1] - метод упорядочивания, iparm[2] - количество потоков, iparm[9] - порог для точности решения, iparm[10] - включить масштабирование), уровень вывода сообщений (0 - без сообщений, 1 - вывод информации о ходе решения), указатель на массив правой части системы, указатель на массив, в который будет записано решение, код ошибки (0 - успешное выполнение, отрицательные значения - фатальные ошибки, положительные значения - предупреждения).

Использование PARDISO всегда подразумевает процедуру очистки буфера памяти для решателя. Для этого необходимо вызвать функцию PARDISO ещё раз, указав значение переменной фазы решения, как -1. Если все особенности решателя были настроены правильно, то получится консольный вывод, представленный на рисунке 2.10.

На консоли будет выводиться процесс факторизации матрицы, состояние некоторых параметров при вызове функции, временные интервалы, затраченные на каждый этап при решении СЛАУ, некоторые статистические данные, например размерность матрицы, вектора, процент ненулевых значений матрицы относительно её общего размера, а также информацию о процессе факторизации.

Рисунок 2.10 – Консольный вывод после решения СЛАУ через PARDISO

Помимо консольного вывода вызов программы PARDISO создаст текстовый файл pardiso64.log, в котором будет написан статус вызова функции (0 если нет ни ошибок, ни предупреждений; значение больше нуля если воз-

никли предупреждения; значение меньше нуля если при вызове PARDISO произошла фатальная ошибка).

ЗАКЛЮЧЕНИЕ

При выполнений учебной практики было разработано программное обеспечение, позволяющее строить сетку достаточно сложных геометрических фигур, используя трёхмерные "шестигранники"в качестве конечных элементов. Также разработано программное обеспечение, позволяющее вычислять локальные матрицы масс и жёсткости, а также значения локального вектора правой части на шестигранниках.

Однако, не был реализован учёт неоднородных главных краевых условий, из-за чего полноценную проверку решения на порядок сходимости и аппроксимации провести не удалось.

Помимо этого, был теоретически изучен и опробован на практике при решении СЛАУ метод PARDISO. Во многом PARDISO на тестовых данных выдавал почти мгновенный результат.

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 1. ТЕКСТ ПРОГРАММЫ

FEM.h

```
#pragma once

#include <vector>
#include <array>
#include <fstream>

#include "../Mathematical_objects/MathematicalHeader.h"
#include "../Logger/Logger.h"
#include "../Mesh/Mesh.h"
#include "../Solvers/MainSolverHeader.h"

typedef std::array<double, 3> vector;

enum class EquationType {
    Hyperbolical, Parabolical,
    Elliptical, NotStated };

enum class InputExtension { Txt, Bin };

class FEM {
    bool _isDataCommited = false;
    bool isDataCommited() const { return _isDataCommited; }
    bool _isSolved = false;
    bool isSolved() const { return _isSolved; }
    // Based.
    std::vector<std::array<size_t, 13>> _areas{};
    std::vector<std::array<size_t, 3>> _borderRibs{};
    std::vector<std::array<size_t, 6>> _newBorderRibs{};
    std::vector<std::array<double, 3>> _points{};
    std::vector<std::pair<size_t, size_t>> _generatedRibs{};
    std::vector<std::pair<size_t, std::pair<double, double>>> _areasInfo{};
    std::vector<double> _time{};
    void SolveElliptical();
    void SolveParabolical();
    void SolveHyperbolical();
    GlobalMatrix* A;
    GlobalVector* b;
    GlobalVector x;
    Solver* _s;
public:
    void ReadMeshData(InputExtension ie);
    void GetMeshData(const Mesh* mesh);
    void BuildMatrixAndVector();
    vector GetSolutionAtPoint(double x, double y, double z);
    void SetSolver(Solver* s);
    void Solve();
    void WriteAnswer();
    EquationType Type = EquationType::NotStated;
    __declspec(property(get = isDataCommited)) bool IsDataCommited;
    __declspec(property(get = isSolved)) bool IsSolved;
    FEM();
    FEM(std::vector<std::array<size_t, 13>> areas,
        std::vector<std::array<size_t, 3>> borderRibs,
        std::vector<std::array<double, 3>> points,
        std::vector<std::pair<size_t, size_t>> generatedRibs) :
        _areas(areas), _borderRibs(borderRibs),
        _points(points), _generatedRibs(generatedRibs) {}
    ~FEM();
};
```

FEM.cpp

```cpp
#include "FEM.h"

void FEM::SolveElliptical() {
    A = new GlobalMatrix(); b = new GlobalVector(_generatedRibs.size());
    x = GlobalVector(_generatedRibs.size());
    A->GeneratePortrait(_areas, _generatedRibs.size());
    A->Fill(_areas, _points, _generatedRibs, _areasInfo);
    b->Fill(_areas, _points, _generatedRibs);
    A->CommitBoundaryConditions(_newBorderRibs);
    b->CommitBoundaryConditions(_newBorderRibs, _points, _generatedRibs); }

void FEM::SolveParabolical() {
    Logger::ConsoleOutput("Couldn't read Parabolic problem", NotificationColor::Alert);
    exit(-1); }

void FEM::SolveHyperbolical() {
    Logger::ConsoleOutput("Couldn't read Hyperbolic problem", NotificationColor::Alert);
    exit(-1); }

void FEM::ReadMeshData(InputExtension ie) {
    Logger::ConsoleOutput("Couldn't read from file", NotificationColor::Alert);
    exit(-1); }

void FEM::GetMeshData(const Mesh* mesh) {
    for (const auto& point : mesh->Points) {
        std::array<double, 3> _point = { point.x, point.y, point.z };
        _points.push_back(_point); }
    for (const auto& area : mesh->getAreasAsRibs()) {
        std::array<size_t, 13> _area = { area.subdomainNum_,
            area.refs_[0], area.refs_[1], area.refs_[2], area.refs_[3],
            area.refs_[4], area.refs_[5], area.refs_[6], area.refs_[7],
            area.refs_[8], area.refs_[9], area.refs_[10], area.refs_[11] };
        _areas.push_back(_area); }
    for (const auto& rib : mesh->getRibsRefs()) {
        std::pair<size_t, size_t> _rib{ rib.p1, rib.p2 };
        _generatedRibs.push_back(_rib); }
    for (const auto& border : mesh->getNewBorderRibs()) {
        std::array<size_t, 6> rwe{border[0], border[1], border[2], border[3], border[4],
        ↪  border[5]};
        _newBorderRibs.push_back(rwe); }
    for (const auto& areasInfos : mesh->AreasInfo) {
        std::pair<size_t, std::pair<double, double>> _area{ areasInfos.subdomainNum_,
        ↪  {areasInfos.mu_, areasInfos.sigma_} };
        _areasInfo.push_back(_area); }
    _isDataCommited = true; }

void FEM::BuildMatrixAndVector() {
    switch (Type) {
    case EquationType::Hyperbolical: SolveHyperbolical(); break;
    case EquationType::Parabolical: SolveParabolical(); break;
    case EquationType::Elliptical: SolveElliptical(); break;
    case EquationType::NotStated: default:
        Logger::ConsoleOutput("Equation type didn't stated. Exit program.",
        ↪  NotificationColor::Alert);
        exit(-1); break; } }

vector FEM::GetSolutionAtPoint(double x, double y, double z) {
    Logger::ConsoleOutput("Can't get solution at point.", NotificationColor::Alert);
    exit(-1);
    return vector(); }

void FEM::SetSolver(Solver* s) { _s = s;}

void FEM::Solve() { x = _s->Solve(*A, *b);}

void FEM::WriteAnswer() {
    std::ofstream fout("Data/Output/solution.txt");
    for (size_t i(0); i < x.getSize(); ++i)
        fout << i << ". " << std::setprecision(15) << std::scientific << x(i) <<
        ↪  std::endl;
    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
```

```cpp
double Integration::Gauss5(function f) {
    double ans(0.0);
    for (size_t k(0); k < 5; ++k)
        for (size_t j(0); j < 5; ++j)
            for (size_t i(0); i < 5; ++i)
                ans += tau5[k] * tau5[j] * tau5[i] * f(t5[k], t5[j], t5[i]);
    return ans; }
```

GlobalMatrix.h

```cpp
#pragma once

#include "..\Logger\Logger.h"
#include "Matrix.h"
#include "LocalMatrix.h"
#include "GlobalVector.h"

#include <vector>
#include <array>
#include <algorithm>

class GlobalMatrix : public Matrix {
private:
    size_t _size = 0;
    bool _isPortraitGenerated = false;
    inline bool isPortraitGenerated() const { return _isPortraitGenerated; }
    void addLocalMatrixValues(const std::array<size_t, 12> localRibs, const LocalMatrix&
    ↪  G, const LocalMatrix& M);
    double getAlValue(size_t i, size_t j) const;
    double getAuValue(size_t i, size_t j) const;
public:
    inline size_t getSize() const override { return _di.size(); }
    void GeneratePortrait(std::vector<std::array<size_t, 13>> areas, size_t ribsAmount);
    void Fill(std::vector<std::array<size_t, 13>> areas, std::vector<std::array<double,
    ↪  3>> points,
        std::vector<std::pair<size_t, size_t>> generatedRibs,
        ↪  std::vector<std::pair<size_t, std::pair<double, double>>> areasInfo);
    void CommitBoundaryConditions(std::vector<std::array<size_t, 6>> borderRibs);
    std::vector<double> _al{};
    std::vector<double> _au{};
    std::vector<double> _di{};
    std::vector<size_t> _ig{};
    std::vector<size_t> _jg{};
    inline double AL(size_t i) const { return i < _al.size() ? _al[i] : throw "_al
    ↪  argument out of range."; }
    inline double AU(size_t i) const { return i < _au.size() ? _au[i] : throw "_au
    ↪  argument out of range."; }
    inline double DI(size_t i) const { return i < _di.size() ? _di[i] : throw "_di
    ↪  argument out of range."; }
    inline size_t IG(size_t i) const { return i < _ig.size() ? _ig[i] : throw "_ig
    ↪  argument out of range."; }
    inline size_t JG(size_t i) const { return i < _jg.size() ? _jg[i] : throw "_jg
    ↪  argument out of range."; }
    __declspec(property(get = isPortraitGenerated)) bool IsPortraitGenerated;
    __declspec(property(get = getSize)) size_t Size;
    double getValue(size_t i, size_t j);
    double operator() (size_t i, size_t j) const override;   // getter.
    double& operator() (size_t i, size_t j) override;        // setter.
    GlobalMatrix() {};
    ~GlobalMatrix() {};
    friend GlobalVector operator*(const GlobalMatrix A, const GlobalVector b);
};
```

GlobalMatrix.cpp

```cpp
#include "GlobalMatrix.h"

void GlobalMatrix::addLocalMatrixValues(const std::array<size_t, 12> localRibs, const
↪  LocalMatrix& G, const LocalMatrix& M) {
```

```
    const std::array<size_t, 12> switchV{
        0, 3, 8, 11,
        1, 2, 9, 10,
        4, 5, 6, 7 };
    int ii(0);
    for (const auto& i : localRibs) {
        int jj(0);
        for (const auto& j : localRibs) {
            int ind(0);
            if (i - j == 0)
                _di[i] += G(switchV[ii], switchV[jj]) + M(switchV[ii], switchV[jj]);
            else if (i - j < 0) {
                ind = _ig[j];
                for (; ind <= _ig[j + 1] - 1; ind++) if (_jg[ind] == i) break;
                _au[ind] += G(switchV[ii], switchV[jj]) + M(switchV[ii], switchV[jj]);
            }
            else if (i - j > 0) {
                ind = _ig[i];
                for (; ind <= _ig[i + 1] - 1; ind++) if (_jg[ind] == j) break;
                _al[ind] += G(switchV[ii], switchV[jj]) + M(switchV[ii], switchV[jj]);
            } ++jj; } ++ii; }
}

double GlobalMatrix::getAlValue(size_t i, size_t j) const {
    for (size_t ii = 0; ii < _ig[i + 1] - _ig[i]; ii++)
        if (_jg[_ig[i] + ii] == j) return _al[_ig[i] + ii];
    return 0.0;
}

double GlobalMatrix::getAuValue(size_t i, size_t j) const {
    for (int ii = 0; ii < _ig[i + 1] - _ig[i]; ii++)
        if (_jg[_ig[i] + ii] == j) return _au[_ig[i] + ii];
    return 0.0;
}

void GlobalMatrix::GeneratePortrait(std::vector<std::array<size_t, 13>> areas, size_t
↪  ribsAmount) {
    _ig.resize(ribsAmount + 1);
    std::vector<std::vector<size_t>> additionalVector(ribsAmount);
    for (const auto& area : areas)
        for (size_t i(1); i < 13; ++i)
            for (size_t j(1); j < 13; ++j)
                if (area[j] < area[i] and std::find(additionalVector[area[i]].begin(),
                ↪  additionalVector[area[i]].end(), area[j]) ==
                ↪  additionalVector[area[i]].end()) {
                    additionalVector[area[i]].push_back(area[j]);
                    std::sort(additionalVector[area[i]].begin(),
                    ↪  additionalVector[area[i]].end()); }
    _ig[0] = 0;
    for (size_t i(0); i < ribsAmount; i++) {
        _ig[i + 1] = _ig[i] + additionalVector[i].size();
        _jg.insert(_jg.end(), additionalVector[i].begin(), additionalVector[i].end()); }
    _di.resize(ribsAmount); _al.resize(_jg.size()); _au.resize(_jg.size());
}

void GlobalMatrix::Fill(std::vector<std::array<size_t, 13>> areas,
↪  std::vector<std::array<double, 3>> points,
    std::vector<std::pair<size_t, size_t>> generatedRibs, std::vector<std::pair<size_t,
    ↪  std::pair<double, double>>> areasInfo) {
    for (const auto& area : areas) {
        auto mu = [area, areasInfo](){
            for (const auto& info : areasInfo)
                if (area[0] == info.first) return info.second.first; };
        auto sigma = [area, areasInfo]() {
            for (const auto& info : areasInfo)
                if (area[0] == info.first) return info.second.second; };
        std::array<size_t, 12> localArea{ area[1], area[4], area[9], area[12],
            area[2], area[3], area[10], area[11],
            area[5], area[6], area[7], area[8] };
        std::array<double, 8> xPoints = { points[generatedRibs[localArea[0]].first][0],
        ↪  points[generatedRibs[localArea[0]].second][0],
            points[generatedRibs[localArea[1]].first][0],
            ↪  points[generatedRibs[localArea[1]].second][0],
            points[generatedRibs[localArea[2]].first][0],
            ↪  points[generatedRibs[localArea[2]].second][0],
```

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

LocalMatrix.h

```cpp
#pragma once

#include "Matrix.h"
#include "JacobiMatrix.h"
#include "..\Integration\Integration.h"
#include "..\Functions\BasisFunction.h"

typedef JacobiMatrix J;

enum class LMType {
    Stiffness,
    Mass,
    NotStated
};

class LocalMatrix : public Matrix {
private:
    double _koef;
    const size_t _localMatrixSize = 12;
    LMType _matrixType = LMType::NotStated;
    std::array<double, 8> _x{};
    std::array<double, 8> _y{};
    std::array<double, 8> _z{};
    std::array<std::array<double, 12>, 12> _values{};
    void generate();
    void generateG();
    void generateM();
public:
    LocalMatrix() { _koef = 0.0; }
    LocalMatrix(double koef, std::array<double, 8> x, std::array<double, 8> y,
    ↪  std::array<double, 8> z, LMType matrixType) :
        _koef(koef), _matrixType(matrixType), _x(x), _y(y), _z(z) {
        generate();
    }
    ~LocalMatrix() {}
    double operator() (size_t i, size_t j) const override { return _values[i][j]; };
    double& operator() (size_t i, size_t j) override { return _values[i][j]; };
    inline LMType GetMatrixType() const { return _matrixType; }
    size_t getSize() const override { return _localMatrixSize; };
};

std::function<double(double, double, double)> operator* (std::function<double(double,
↪  double, double)> f1,
    std::function<double(double, double, double)> f2);

std::function<double(double, double, double)> operator+ (std::function<double(double,
↪  double, double)> f1,
    std::function<double(double, double, double)> f2);

std::function<double(double, double, double)> operator/ (std::function<double(double,
↪  double, double)> f1,
    std::function<double(double, double, double)> f2);
```

LocalMatrix.cpp

```cpp
#include "LocalMatrix.h"

std::function<double(double, double, double)> operator* (std::function<double(double,
↪  double, double)> f1,
                                                         std::function<double(double, double, double)> f2) {
  return [f1, f2](double t0, double t1, double t2) { return f1(t0, t1, t2) * f2(t0, t1,
  ↪  t2); }; }

std::function<double(double, double, double)> operator+(std::function<double(double,
↪  double, double)> f1,
                                                        std::function<double(double, double, double)> f2) {
  return [f1, f2](double t0, double t1, double t2) { return f1(t0, t1, t2) + f2(t0, t1,
  ↪  t2); }; }

```

```cpp
std::function<double(double, double, double)> operator/(std::function<double(double,
    double, double)> f1,
                            std::function<double(double, double, double)> f2) {
  return [f1, f2](double t0, double t1, double t2) { return f1(t0, t1, t2) / f2(t0, t1,
      t2); }; }

void LocalMatrix::generate() {
  switch (_matrixType) {
  case LMType::Stiffness: generateG(); break;
  case LMType::Mass: generateM(); break;
  case LMType::NotStated:  default: break; }
}

void LocalMatrix::generateG() {
  J::SetValues(_x, _y, _z);
  for (size_t i(0); i < _localMatrixSize; ++i) {
    for (size_t j(0); j < _localMatrixSize; ++j) {
      auto rotPhi_i = BasisFunction::getRotAt(i);
      auto rotPhi_j = BasisFunction::getRotAt(j);
      std::array < std::function<double(double, double, double)>, 3> v1{
        J::GetValueAtTransposed(0, 0) * rotPhi_i[0] + J::GetValueAtTransposed(0, 1) *
          rotPhi_i[1] + J::GetValueAtTransposed(0, 2) * rotPhi_i[2],
        J::GetValueAtTransposed(1, 0) * rotPhi_i[0] + J::GetValueAtTransposed(1, 1) *
          rotPhi_i[1] + J::GetValueAtTransposed(1, 2) * rotPhi_i[2],
        J::GetValueAtTransposed(2, 0) * rotPhi_i[0] + J::GetValueAtTransposed(2, 1) *
          rotPhi_i[1] + J::GetValueAtTransposed(2, 2) * rotPhi_i[2], };
      std::array < std::function<double(double, double, double)>, 3> v2{
        J::GetValueAtTransposed(0, 0) * rotPhi_j[0] + J::GetValueAtTransposed(0, 1) *
          rotPhi_j[1] + J::GetValueAtTransposed(0, 2) * rotPhi_j[2],
        J::GetValueAtTransposed(1, 0) * rotPhi_j[0] + J::GetValueAtTransposed(1, 1) *
          rotPhi_j[1] + J::GetValueAtTransposed(1, 2) * rotPhi_j[2],
        J::GetValueAtTransposed(2, 0) * rotPhi_j[0] + J::GetValueAtTransposed(2, 1) *
          rotPhi_j[1] + J::GetValueAtTransposed(2, 2) * rotPhi_j[2], };
      for (size_t k(0); k < 3; ++k) _values[i][j] += Integration::Gauss5((v1[k] * v2[k])
        / J::GetDeterminant());
      _values[i][j] /= _koef; } }
}

void LocalMatrix::generateM() {
  J::SetValues(_x, _y, _z);
  for (size_t i(0); i < _localMatrixSize; ++i) {
    for (size_t j(0); j < _localMatrixSize; ++j) {
      for (size_t k(0); k < 3; ++k) {
        _values[i][j] += Integration::Gauss5(J::GetValueAtInverse(k, i / 4) *
          BasisFunction::getAt(i) *
          J::GetValueAtInverse(k, j / 4) * BasisFunction::getAt(j) *
          J::GetDeterminant());
      }
      _values[i][j] *= _koef;  } }
}
```

GlobalVector.h

```cpp
#pragma once

#include "..\Logger\Logger.h"
#include "Vector.h"
#include "LocalVector.h"

#include <vector>
#include <array>

class GlobalVector : public Vector
{
private:
    std::vector<double> _values{};
    void addLocalVectorValues(const std::array<size_t, 12> localRibs, const LocalVector&
      b);
public:
    size_t getSize() const override { return _values.size(); }
    double operator() (size_t i) const override;
```

```
    double& operator() (size_t i) override;
    __declspec(property(get = getSize)) size_t Size;
    GlobalVector();
    GlobalVector(size_t size) { _values.resize(size); }
    ~GlobalVector() {}
    void Fill(std::vector<std::array<size_t, 13>> areas, std::vector<std::array<double, 3>> points,
        std::vector<std::pair<size_t, size_t>> generatedRibs);
    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);
    double Norma() const;
    friend double operator* (const GlobalVector v1, const GlobalVector v2);
    friend GlobalVector operator* (const double a, const GlobalVector v);
    friend GlobalVector operator+ (const GlobalVector v1, const GlobalVector v2);
    friend GlobalVector operator- (const GlobalVector v1, const GlobalVector v2);
};

```

GlobalVector.cpp

```
#include "GlobalVector.h"

void GlobalVector::addLocalVectorValues(const std::array<size_t, 12> localRibs, const LocalVector& b) {
    const std::array<size_t, 12> switchV{
        0, 3, 8, 11,
        1, 2, 9, 10,
        4, 5, 6, 7 };
    for (size_t i(0); i < b.getSize(); ++i) _values[localRibs[i]] += b(switchV[i]);
}

double GlobalVector::operator()(size_t i) const {
    if (i >= Size) {
        Logger::ConsoleOutput("Index run out of Vector range.", NotificationColor::Alert);
        exit(-1); }
    return i < Size ? _values[i] : throw "Index run out of Vector range.";
}

double& GlobalVector::operator()(size_t i) {
    if (i >= Size) {
        Logger::ConsoleOutput("Index run out of Vector range.", NotificationColor::Alert);
        exit(-1); }
    return _values[i];
}

GlobalVector::GlobalVector() {
    Logger::ConsoleOutput("Global vector initialized, but it's empty", NotificationColor::Warning);
}

void GlobalVector::Fill(std::vector<std::array<size_t, 13>> areas, std::vector<std::array<double, 3>> points,
                        std::vector<std::pair<size_t, size_t>> generatedRibs) {
    for (const auto& area : areas) {
        std::array<size_t, 12> localArea{ area[1], area[4], area[9], area[12],
            area[2], area[3], area[10], area[11],
            area[5], area[6], area[7], area[8] };
        std::array<double, 8> xPoints = { points[generatedRibs[localArea[0]].first][0], points[generatedRibs[localArea[0]].second][0],
            points[generatedRibs[localArea[1]].first][0], points[generatedRibs[localArea[1]].second][0],
            points[generatedRibs[localArea[2]].first][0], points[generatedRibs[localArea[2]].second][0],
            points[generatedRibs[localArea[3]].first][0], points[generatedRibs[localArea[3]].second][0] };
        std::array<double, 8> yPoints = { points[generatedRibs[localArea[0]].first][1], points[generatedRibs[localArea[0]].second][1],
            points[generatedRibs[localArea[1]].first][1], points[generatedRibs[localArea[1]].second][1],
            points[generatedRibs[localArea[2]].first][1], points[generatedRibs[localArea[2]].second][1],
```

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

```
}

GlobalVector operator+(const GlobalVector v1, const GlobalVector v2) {
    if (v1.Size != v2.Size) Logger::ConsoleOutput("During vector multiplication vectors have different size", NotificationColor::Alert);
    GlobalVector result(v1.Size);
    for (size_t i(0); i < v1.Size; ++i) result(i) = v1(i) + v2(i);
    return result;
}

GlobalVector operator-(const GlobalVector v1, const GlobalVector v2) {
    if (v1.Size != v2.Size) Logger::ConsoleOutput("During vector multiplication vectors have different size", NotificationColor::Alert);
    GlobalVector result(v1.Size);
    for (size_t i(0); i < v1.Size; ++i) result(i) = v1(i) - v2(i);
    return result;
}
```

LocalVector.h

```
#pragma once

#include "Vector.h"
#include "LocalMatrix.h"
#include "..\Functions\Function.h"

class LocalVector : public Vector {
public:
    LocalVector() {};
    LocalVector(std::array<double, 8> x, std::array<double, 8> y, std::array<double, 8> z) : _x(x), _y(y), _z(z) {
        M = new LocalMatrix(1.0, _x, _y, _z, LMType::Mass);
        generate();
    }
    ~LocalVector() {};
    double operator() (size_t i) const override { return _values[i]; }
    double& operator() (size_t i) override { return _values[i]; }
    size_t getSize() const override { return 12; }
private:
    std::array<double, 8> _x{};
    std::array<double, 8> _y{};
    std::array<double, 8> _z{};
    std::array<double, 12> _values{};
    void generate();
    LocalMatrix* M = nullptr;
};
```

LocalVector.cpp

```
#include "LocalVector.h"

#include <iostream>

void LocalVector::generate() {
    auto findLen = [](vector v) {return sqrt(v[0] * v[0] + v[1] * v[1] + v[2] * v[2]); };
    auto scMult = [](vector v1, vector v2) { return v1[0] * v2[0] + v1[1] * v2[1] + v1[2] * v2[2]; };
    std::array<vector, 12> diffVec = {
        vector{_x[1] - _x[0], _y[1] - _y[0], _z[1] - _z[0]},
        vector{_x[3] - _x[2], _y[3] - _y[2], _z[3] - _z[2]},
        vector{_x[5] - _x[4], _y[5] - _y[4], _z[5] - _z[4]},
        vector{_x[7] - _x[6], _y[7] - _y[6], _z[7] - _z[6]},
        vector{_x[2] - _x[0], _y[2] - _y[0], _z[2] - _z[0]},
        vector{_x[3] - _x[1], _y[3] - _y[1], _z[3] - _z[1]},
        vector{_x[6] - _x[4], _y[6] - _y[4], _z[6] - _z[4]},
        vector{_x[7] - _x[5], _y[7] - _y[5], _z[7] - _z[5]},
        vector{_x[4] - _x[0], _y[4] - _y[0], _z[4] - _z[0]},
        vector{_x[5] - _x[1], _y[5] - _y[1], _z[5] - _z[1]},
        vector{_x[6] - _x[2], _y[6] - _y[2], _z[6] - _z[2]},
```

```
        vector{_x[7] - _x[3], _y[7] - _y[3], _z[7] - _z[3]}};
    std::array<double, 12> vf = {
        scMult(Function::TestF1(_x[0] + 0.5 * diffVec[0][0], _y[0] + 0.5 * diffVec[0][1],
        ↪  _z[0] + 0.5 * diffVec[0][2], 0.0), diffVec[0]) / findLen(diffVec[0]),
        scMult(Function::TestF1(_x[2] + 0.5 * diffVec[2][0], _y[2] + 0.5 * diffVec[2][1],
        ↪  _z[2] + 0.5 * diffVec[2][2], 0.0), diffVec[2]) / findLen(diffVec[2]),
        scMult(Function::TestF1(_x[4] + 0.5 * diffVec[4][0], _y[4] + 0.5 * diffVec[4][1],
        ↪  _z[4] + 0.5 * diffVec[4][2], 0.0), diffVec[4]) / findLen(diffVec[4]),
        scMult(Function::TestF1(_x[6] + 0.5 * diffVec[6][0], _y[6] + 0.5 * diffVec[6][1],
        ↪  _z[6] + 0.5 * diffVec[6][2], 0.0), diffVec[6]) / findLen(diffVec[6]),
        scMult(Function::TestF1(_x[0] + 0.5 * diffVec[0][0], _y[0] + 0.5 * diffVec[0][1],
        ↪  _z[0] + 0.5 * diffVec[0][2], 0.0), diffVec[0]) / findLen(diffVec[0]),
        scMult(Function::TestF1(_x[1] + 0.5 * diffVec[1][0], _y[1] + 0.5 * diffVec[1][1],
        ↪  _z[1] + 0.5 * diffVec[1][2], 0.0), diffVec[1]) / findLen(diffVec[1]),
        scMult(Function::TestF1(_x[4] + 0.5 * diffVec[4][0], _y[4] + 0.5 * diffVec[4][1],
        ↪  _z[4] + 0.5 * diffVec[4][2], 0.0), diffVec[4]) / findLen(diffVec[4]),
        scMult(Function::TestF1(_x[5] + 0.5 * diffVec[5][0], _y[5] + 0.5 * diffVec[5][1],
        ↪  _z[5] + 0.5 * diffVec[5][2], 0.0), diffVec[5]) / findLen(diffVec[5]),
        scMult(Function::TestF1(_x[0] + 0.5 * diffVec[0][0], _y[0] + 0.5 * diffVec[0][1],
        ↪  _z[0] + 0.5 * diffVec[0][2], 0.0), diffVec[0]) / findLen(diffVec[0]),
        scMult(Function::TestF1(_x[1] + 0.5 * diffVec[1][0], _y[1] + 0.5 * diffVec[1][1],
        ↪  _z[2] + 0.5 * diffVec[1][2], 0.0), diffVec[1]) / findLen(diffVec[1]),
        scMult(Function::TestF1(_x[2] + 0.5 * diffVec[2][0], _y[2] + 0.5 * diffVec[2][1],
        ↪  _z[4] + 0.5 * diffVec[2][2], 0.0), diffVec[2]) / findLen(diffVec[2]),
        scMult(Function::TestF1(_x[3] + 0.5 * diffVec[3][0], _y[3] + 0.5 * diffVec[3][1],
        ↪  _z[6] + 0.5 * diffVec[3][2], 0.0), diffVec[3]) / findLen(diffVec[3]), };
    for (size_t i(0); i < 12; ++i)
        for (size_t j(0); j < 12; ++j)
            _values[i] += M->operator()(i, j) * vf[j];
}
```

JacobiMatrix.h

```
#pragma once

#include <functional>
#include <array>

class JacobiMatrix {
private:
    // Points arrays.
    static std::array<double, 8> _x;
    static std::array<double, 8> _y;
    static std::array<double, 8> _z;
    // Template functions.
    static inline double const W_(double t) { return 0.5 * (1.0 - t); }
    static inline double const W(double t)  { return 0.5 * (1.0 + t); }
public:
    JacobiMatrix() = delete;
    static void SetValues(std::array<double, 8> x, std::array<double, 8> y,
    ↪  std::array<double, 8> z);
    static inline double dxde(double eps, double eta, double zeta);
    static inline double dyde(double eps, double eta, double zeta);
    static inline double dzde(double eps, double eta, double zeta);
    static inline double dxdn(double eps, double eta, double zeta);
    static inline double dydn(double eps, double eta, double zeta);
    static inline double dzdn(double eps, double eta, double zeta);
    static inline double dxdc(double eps, double eta, double zeta);
    static inline double dydc(double eps, double eta, double zeta);
    static inline double dzdc(double eps, double eta, double zeta);
    static std::function<double(double, double, double)> const GetValueAt(size_t i,
    ↪  size_t j);
    static std::function<double(double, double, double)> const GetValueAtInverse(size_t
    ↪  i, size_t j);
    static std::function<double(double, double, double)> const
    ↪  GetValueAtTransposed(size_t i, size_t j);
    static std::function<double(double, double, double)> const GetDeterminant();
};

```

JacobiMatrix.cpp

```cpp
#include "JacobiMatrix.h"

std::array<double, 8> JacobiMatrix::_x = {};
std::array<double, 8> JacobiMatrix::_y = {};
std::array<double, 8> JacobiMatrix::_z = {};

inline double JacobiMatrix::dxde(double eps, double eta, double zeta) {
    return 0.5 * (W_(eta) * W_(zeta) * (_x[1] - _x[0]) +
                  W(eta) * W_(zeta) * (_x[3] - _x[2]) +
                 W_(eta) *  W(zeta) * (_x[5] - _x[4]) +
                  W(eta) *  W(zeta) * (_x[7] - _x[6])); }

inline double JacobiMatrix::dyde(double eps, double eta, double zeta) {
    return 0.5 * (W_(eta) * W_(zeta) * (_y[1] - _y[0]) +
                  W(eta) * W_(zeta) * (_y[3] - _y[2]) +
                 W_(eta) *  W(zeta) * (_y[5] - _y[4]) +
                  W(eta) *  W(zeta) * (_y[7] - _y[6])); }

inline double JacobiMatrix::dzde(double eps, double eta, double zeta) {
    return 0.5 * (W_(eta) * W_(zeta) * (_z[1] - _z[0]) +
                  W(eta) * W_(zeta) * (_z[3] - _z[2]) +
                 W_(eta) *  W(zeta) * (_z[5] - _z[4]) +
                  W(eta) *  W(zeta) * (_z[7] - _z[6]));}

inline double JacobiMatrix::dxdn(double eps, double eta, double zeta) {
    return 0.5 * (W_(eps) * W_(zeta) * (_x[2] - _x[0]) +
                  W(eps) * W_(zeta) * (_x[3] - _x[1]) +
                 W_(eps) *  W(zeta) * (_x[6] - _x[4]) +
                  W(eps) *  W(zeta) * (_x[7] - _x[5])); }

inline double JacobiMatrix::dydn(double eps, double eta, double zeta) {
    return 0.5 * (W_(eps) * W_(zeta) * (_y[2] - _y[0]) +
                  W(eps) * W_(zeta) * (_y[3] - _y[1]) +
                 W_(eps) *  W(zeta) * (_y[6] - _y[4]) +
                  W(eps) *  W(zeta) * (_y[7] - _y[5])); }

inline double JacobiMatrix::dzdn(double eps, double eta, double zeta) {
    return 0.5 * (W_(eps) * W_(zeta) * (_z[2] - _z[0]) +
                  W(eps) * W_(zeta) * (_z[3] - _z[1]) +
                 W_(eps) *  W(zeta) * (_z[6] - _z[4]) +
                  W(eps) *  W(zeta) * (_z[7] - _z[5])); }


inline double JacobiMatrix::dxdc(double eps, double eta, double zeta) {
    return 0.5 * (W_(eps) * W_(eta) * (_x[4] - _x[0]) +
                  W(eps) * W_(eta) * (_x[5] - _x[1]) +
                 W_(eps) *  W(eta) * (_x[6] - _x[2]) +
                  W(eps) *  W(eta) * (_x[7] - _x[3])); }

inline double JacobiMatrix::dydc(double eps, double eta, double zeta) {
    return 0.5 * (W_(eps) * W_(eta) * (_y[4] - _y[0]) +
                  W(eps) * W_(eta) * (_y[5] - _y[1]) +
                 W_(eps) *  W(eta) * (_y[6] - _y[2]) +
                  W(eps) *  W(eta) * (_y[7] - _y[3])); }

inline double JacobiMatrix::dzdc(double eps, double eta, double zeta) {
    return 0.5 * (W_(eps) * W_(eta) * (_z[4] - _z[0]) +
                  W(eps) * W_(eta) * (_z[5] - _z[1]) +
                 W_(eps) *  W(eta) * (_z[6] - _z[2]) +
                  W(eps) *  W(eta) * (_z[7] - _z[3])); }


void JacobiMatrix::SetValues(std::array<double, 8> x, std::array<double, 8> y,
↪  std::array<double, 8> z) {
    _x = x; _y = y; _z = z; }

std::function<double(double, double, double)> const JacobiMatrix::GetValueAt(size_t i,
↪  size_t j) {
    if (i == 0 and j == 0) return dxde;
    if (i == 0 and j == 1) return dyde;
    if (i == 0 and j == 2) return dzde;
    if (i == 1 and j == 0) return dxdn;
    if (i == 1 and j == 1) return dydn;
```

```
    if (i == 1 and j == 2) return dzdn;
    if (i == 2 and j == 0) return dxdc;
    if (i == 2 and j == 1) return dydc;
    if (i == 2 and j == 2) return dzdc;
}

std::function<double(double, double, double)> const
↪  JacobiMatrix::GetValueAtInverse(size_t i, size_t j) {
    if (i == 0 and j == 0)
        return [](double t0, double t1, double t2) { return (dydn(t0, t1, t2) * dzdc(t0,
        ↪  t1, t2) -
            dydc(t0, t1, t2) * dzdn(t0, t1, t2)) / GetDeterminant()(t0, t1, t2); };
    if (i == 0 and j == 1)
        return [](double t0, double t1, double t2) { return -1.0 * (dyde(t0, t1, t2) *
        ↪  dzdc(t0, t1, t2) -
            dydc(t0, t1, t2) * dzde(t0, t1, t2)) / GetDeterminant()(t0, t1, t2); };
    if (i == 0 and j == 2)
        return [](double t0, double t1, double t2) { return (dyde(t0, t1, t2) * dzdn(t0,
        ↪  t1, t2) -
            dydn(t0, t1, t2) * dzde(t0, t1, t2)) / GetDeterminant()(t0, t1, t2); };
    if (i == 1 and j == 0)
        return [](double t0, double t1, double t2) { return -1.0 * (dxdn(t0, t1, t2) *
        ↪  dzdc(t0, t1, t2) -
            dxdc(t0, t1, t2) * dzdn(t0, t1, t2)) / GetDeterminant()(t0, t1, t2); };
    if (i == 1 and j == 1)
        return [](double t0, double t1, double t2) { return (dxde(t0, t1, t2) * dzdc(t0,
        ↪  t1, t2) -
            dxdc(t0, t1, t2) * dzde(t0, t1, t2)) / GetDeterminant()(t0, t1, t2); };
    if (i == 1 and j == 2)
        return [](double t0, double t1, double t2) { return -1.0 * (dxde(t0, t1, t2) *
        ↪  dzdn(t0, t1, t2) -
            dxdn(t0, t1, t2) * dzde(t0, t1, t2)) / GetDeterminant()(t0, t1, t2); };
    if (i == 2 and j == 0)
        return [](double t0, double t1, double t2) { return (dxdn(t0, t1, t2) * dydc(t0,
        ↪  t1, t2) -
            dxdc(t0, t1, t2) * dydn(t0, t1, t2)) / GetDeterminant()(t0, t1, t2); };
    if (i == 2 and j == 1)
        return [](double t0, double t1, double t2) { return -1.0 * (dxde(t0, t1, t2) *
        ↪  dydc(t0, t1, t2) -
            dxdc(t0, t1, t2) * dyde(t0, t1, t2)) / GetDeterminant()(t0, t1, t2); };
    if (i == 2 and j == 2)
        return [](double t0, double t1, double t2) { return (dxde(t0, t1, t2) * dydn(t0,
        ↪  t1, t2) -
            dxdn(t0, t1, t2) * dyde(t0, t1, t2)) / GetDeterminant()(t0, t1, t2); };
}

std::function<double(double, double, double)> const
↪  JacobiMatrix::GetValueAtTransposed(size_t i, size_t j) { return GetValueAt(j, i); }

std::function<double(double, double, double)> const JacobiMatrix::GetDeterminant() {
    return [](double t0, double t1, double t2) { return
        dxde(t0, t1, t2) * dydn(t0, t1, t2) * dzdc(t0, t1, t2) +
        dyde(t0, t1, t2) * dzdn(t0, t1, t2) * dxdc(t0, t1, t2) +
        dzde(t0, t1, t2) * dxdn(t0, t1, t2) * dydc(t0, t1, t2) -
        dzde(t0, t1, t2) * dydn(t0, t1, t2) * dxdc(t0, t1, t2) -
        dyde(t0, t1, t2) * dxdn(t0, t1, t2) * dzdc(t0, t1, t2) -
        dxde(t0, t1, t2) * dzdn(t0, t1, t2) * dydc(t0, t1, t2); };
}
```

Mesh.h

```
#pragma once

#include <cassert>
#include <cmath>
#include <string>
#include <algorithm>

#include "../DataTypes.h"
#include "../Logger/Logger.h"

```

```cpp
class Mesh {
private:
    // Based.
    bool isGenerated_ = false;
    bool isDeclarated_ = false;
    size_t linesAmountX_ = 0;
    size_t linesAmountY_ = 0;
    size_t linesAmountZ_ = 0;
    std::vector<Point> points_{};
    size_t subdomainsAmount_ = 0;
    std::vector<std::array<size_t, 7>> subdomains_{};
    std::vector<AreaInfo> areasInfo_{};
    std::vector<AreaRibs> areasRibs_{};
    std::vector<std::pair<size_t, double_t>> delimitersX_{};
    std::vector<std::pair<size_t, double_t>> delimitersY_{};
    std::vector<std::pair<size_t, double_t>> delimitersZ_{};
    size_t bordersAmount_ = 0;
    std::vector<Border> borders_{};
    std::vector<BorderLine> borderRibs_{};
    std::vector<std::array<size_t, 6>> newBorders_{};
    std::vector<AreaPoints> areasPoints_{};
    std::vector<RibRef> referableRibs_{};
    // Additional.
    std::vector<Point> immutablePoints_{};
    std::vector<std::array<size_t, 7>> immutableSubdomains_{};
    std::vector<Border> immutableBorders_{};
    std::vector<size_t> numRefsOfLinesAboveX{};
    std::vector<size_t> numRefsOfLinesAboveY{};
    std::vector<size_t> numRefsOfLinesAboveZ{};
    void organizeBorders();
public:
    Mesh() { Logger::ConsoleOutput("Mesh declared, but it's empty.",
    ↪  NotificationColor::Warning); };
    ~Mesh() {};
    bool CheckData();
    void CommitData(std::vector<std::string>* data);
    inline bool isGenerated() const { return isGenerated_; }
    inline bool isDeclarated() const { return isDeclarated_; }
    inline size_t getLinesAmountX() const { return linesAmountX_; }
    inline size_t getLinesAmountY() const { return linesAmountY_; }
    inline size_t getLinesAmountZ() const { return linesAmountZ_; }
    inline std::vector<Point> getPoints() const { return points_; }
    inline std::vector<AreaRibs> getAreasAsRibs() const { return areasRibs_; }
    inline std::vector<RibRef> getRibsRefs() const { return referableRibs_; }
    inline std::vector<Border> getBorders() const { return borders_; }
    inline std::vector<BorderLine> getBorderRibs() const { return borderRibs_; }
    inline std::vector<std::array<size_t, 6>> getNewBorderRibs() const { return
    ↪  newBorders_; }
    inline std::vector<AreaInfo> getAreaInfo() const { return areasInfo_; }
    __declspec(property(get = getLinesAmountX)) size_t LinesAmountX;
    __declspec(property(get = getLinesAmountY)) size_t LinesAmountY;
    __declspec(property(get = getLinesAmountZ)) size_t LinesAmountZ;
    __declspec(property(get = isGeneraed)) bool IsGenerated;
    __declspec(property(get = isDeclarated)) bool IsDeclarated;
    __declspec(property(get = getPoints)) std::vector<Point> Points;
    __declspec(property(get = getAreaInfo)) std::vector<AreaInfo> AreasInfo;
    friend class MeshGenerator;
};
```

Mesh.cpp

```cpp
#include "Mesh.h"

void Mesh::organizeBorders() {
    size_t bordersIISwifted(0); size_t bordersIIISwifted(0);
    for (auto& border : borders_) {
        if (border.type_ == 2) {
            std::iter_swap(borders_.begin() + bordersIISwifted, &border);
            bordersIISwifted++; }
        if (border.type_ == 3) {
            std::iter_swap(borders_.begin() + bordersIISwifted + bordersIIISwifted,
            ↪  &border);
```

```
            bordersIIISwifted++; } }
}

bool Mesh::CheckData() {
    if (linesAmountX_ * linesAmountY_ * linesAmountZ_ != points_.size()) return false;
    if (linesAmountX_ - 1 != delimitersX_.size()) return false;
    if (linesAmountY_ - 1 != delimitersY_.size()) return false;
    if (linesAmountZ_ - 1 != delimitersZ_.size()) return false;
    size_t maxLineX = 0; size_t maxLineY = 0; size_t maxLineZ = 0;
    for (const auto& subdomain : subdomains_) {
        maxLineX = maxLineX < subdomain[2] ? subdomain[2] : maxLineX;
        maxLineY = maxLineY < subdomain[4] ? subdomain[4] : maxLineY;
        maxLineZ = maxLineZ < subdomain[6] ? subdomain[6] : maxLineZ; }
    if (linesAmountX_ - 1 != maxLineX) return false;
    if (linesAmountY_ - 1 != maxLineY) return false;
    if (linesAmountZ_ - 1 != maxLineZ) return false;
    Logger::ConsoleOutput("Mesh checked and declared.", NotificationColor::Passed);
    isDeclarated_ = true; return true;
}

void Mesh::CommitData(std::vector<std::string>* data) {
    auto currentItem = data->begin(); // Select first item of vector.
    // Commit lines amount above X,Y,Z axis.
    linesAmountX_ = std::stoul(*currentItem); currentItem++;
    linesAmountY_ = std::stoul(*currentItem); currentItem++;
    linesAmountZ_ = std::stoul(*currentItem); currentItem++;
    // Commit area description.
    for (size_t i(0); i < linesAmountX_ * linesAmountY_ * linesAmountZ_; ++i) {
        points_.emplace_back(std::stod(*currentItem),           // X
                             std::stod(*(currentItem + 1)),     // Y
                             std::stod(*(currentItem + 2)));    // Z
        currentItem += 3; }
    immutablePoints_ = points_;
    // Commit unique areas description.
    subdomainsAmount_ = std::stoul(*currentItem); currentItem++;
    for (size_t i(0); i < subdomainsAmount_; ++i) {
        std::array<size_t, 7> currentArray = { std::stoul(*currentItem), // Formula num.
            std::stoul(*(currentItem + 1)),      // X0.
            std::stoul(*(currentItem + 2)),      // X1.
            std::stoul(*(currentItem + 3)),      // Y0.
            std::stoul(*(currentItem + 4)),      // Y1.
            std::stoul(*(currentItem + 5)),      // Z0.
            std::stoul(*(currentItem + 6)) };    // Z1.
        subdomains_.push_back(currentArray);
        currentItem += 7; }
    immutableSubdomains_ = subdomains_;
    // Commit unique areas coefficients description.
    for (size_t i(0); i < subdomainsAmount_; ++i) {
        areasInfo_.emplace_back(std::stoul(*currentItem),           // Area num.
                                std::stod(*(currentItem + 1)),      // Mu_i.
                                std::stod(*(currentItem + 2)));     // Sigma_i.
        currentItem += 3; }
    // Commit delimiters above X description.
    for (size_t i(0); i < linesAmountX_ - 1; ++i) {
        delimitersX_.emplace_back(std::stoul(*currentItem),
            std::stod(*(currentItem + 1)));
        currentItem += 2; }
    // Commit delimiters above Y description.
    for (size_t i(0); i < linesAmountY_ - 1; ++i) {
        delimitersY_.emplace_back(std::stoul(*currentItem),
            std::stod(*(currentItem + 1)));
        currentItem += 2; }
    // Commit delimiters above Z description.
    for (size_t i(0); i < linesAmountZ_ - 1; ++i) {
        delimitersZ_.emplace_back(std::stoul(*currentItem),
            std::stod(*(currentItem + 1)));
        currentItem += 2; }
    // Commit information about borders.
    bordersAmount_ = std::stoul(*currentItem); currentItem++;
    for (size_t i(0); i < bordersAmount_; ++i) {
        borders_.emplace_back(std::stoul(*currentItem), // Border type.
            std::stoul(*(currentItem + 1)),       // Border formula num.
            std::stoul(*(currentItem + 2)),       // X0.
            std::stoul(*(currentItem + 3)),       // X1.
            std::stoul(*(currentItem + 4)),       // Y0.
```

```
            std::stoul(*(currentItem + 5)),       // Y1.
            std::stoul(*(currentItem + 6)),       // Z0.
            std::stoul(*(currentItem + 7)));      // Z1.
        currentItem += 8; }
    organizeBorders(); immutableBorders_ = borders_;
}
```

MeshGenerator.h

```
#pragma once

#include "Mesh.h"

#include <cassert>
#include <vector>

typedef std::vector<std::vector<std::vector<Point>>> area3D;
typedef std::vector<std::vector<Point>> square2D;
typedef std::vector<Point> line1D;

class MeshGenerator {
private:
    static int SelectAreaNum(Mesh& mesh, std::array<size_t, 12> arr);
    static void GenerateListOfPoints(Mesh& mesh);
    static void GenerateListOfAreas(Mesh& mesh);
    static void GenerateListOfRibs(Mesh& mesh);
    static void GenerateListOfBorders(Mesh& mesh);
public:
    MeshGenerator() = delete;
    static void Generate3DMesh(Mesh& mesh);
};
```

MeshGenerator.cpp

```
#include "MeshGenerator.h"

void MeshGenerator::Generate3DMesh(Mesh& mesh) {
    assert(mesh.IsDeclarated);
    for (size_t i(0); i < mesh.LinesAmountX; ++i) mesh.numRefsOfLinesAboveX.push_back(i);
    for (size_t i(0); i < mesh.LinesAmountY; ++i) mesh.numRefsOfLinesAboveY.push_back(i);
    for (size_t i(0); i < mesh.LinesAmountZ; ++i) mesh.numRefsOfLinesAboveZ.push_back(i);
    GenerateListOfPoints(mesh); GenerateListOfRibs(mesh);
    GenerateListOfAreas(mesh); GenerateListOfBorders(mesh);
}

int MeshGenerator::SelectAreaNum(Mesh& mesh, std::array<size_t, 12> arr) {
    auto sxy = mesh.LinesAmountX * mesh.LinesAmountY;
    auto p1 = mesh.referableRibs_[arr[0]].p1;
    auto p2 = mesh.referableRibs_[arr[11]].p2;
    auto lx0 = p1 % mesh.LinesAmountX;
    auto lx1 = p2 % mesh.LinesAmountX;
    auto ly0 = (p1 % sxy) / mesh.LinesAmountX;
    auto ly1 = (p2 % sxy) / mesh.LinesAmountX;
    auto lz0 = p1 / sxy; auto lz1 = p2 / sxy;
    for (const auto& area : mesh.subdomains_)
        if (mesh.numRefsOfLinesAboveX[area[1]] <= lx0 and
            lx1 <= mesh.numRefsOfLinesAboveX[area[2]] and
            mesh.numRefsOfLinesAboveY[area[3]] <= ly0 and
            ly1 <= mesh.numRefsOfLinesAboveY[area[4]] and
            mesh.numRefsOfLinesAboveZ[area[5]] <= lz0 and
            lz1 <= mesh.numRefsOfLinesAboveZ[area[6]])
            return area[0];
    Logger::ConsoleOutput("Error during selection of area num", NotificationColor::Alert);
    return NAN;
}

// Try to optimize memory.
void MeshGenerator::GenerateListOfPoints(Mesh& mesh) {
    // Construct 3D area.
    area3D figure{};
```

```
    figure.resize(mesh.LinesAmountZ);
    for (auto& square : figure) {
        square.resize(mesh.LinesAmountY);
        for (auto& line : square) line.resize(mesh.LinesAmountX); }
    // Fill 3D area.
    auto sxy = mesh.LinesAmountX * mesh.LinesAmountY;
    auto lx = mesh.LinesAmountX;
    for (size_t k(0); k < mesh.LinesAmountZ; ++k)
        for (size_t j(0); j < mesh.LinesAmountY; ++j)
            for (size_t i(0); i < mesh.LinesAmountX; ++i)
                figure[k][j][i] = mesh.immutablePoints_[k * sxy + j * lx + i];
    // Generation above X-axis.
    for (size_t k(0); k < mesh.LinesAmountZ; ++k) {
        for (size_t j(0); j < mesh.LinesAmountY; ++j) {
            line1D lineToBuild{};
            lineToBuild = figure[k][j];
            size_t shift = mesh.LinesAmountX - 1;
            auto iterOnXRefs = mesh.numRefsOfLinesAboveX.begin() + 1;
            for (const auto& info : mesh.delimitersX_) {
                auto rightBorderIter = lineToBuild.end() - shift;
                auto amountOfDelimiters = info.first;
                auto coefficientOfDelimiter = info.second;
                *iterOnXRefs = *(iterOnXRefs - 1) + amountOfDelimiters;
                iterOnXRefs++;
                double denum = 0.0;
                for (size_t ii(0); ii < amountOfDelimiters; ++ii)
                    denum += pow(coefficientOfDelimiter, ii);
                double x0 = (*(rightBorderIter - 1)).x; double x1 = (*rightBorderIter).x;
                double y0 = (*(rightBorderIter - 1)).y; double y1 = (*rightBorderIter).y;
                double z0 = (*(rightBorderIter - 1)).z; double z1 = (*rightBorderIter).z;
                double deltX = x1 - x0; double deltY = y1 - y0; double deltZ = z1 - z0;
                double xh = deltX / denum; double yh = deltY / denum;
                double zh = deltZ / denum; double multiplier = 0.0;
                for (size_t ii(0); ii < amountOfDelimiters - 1; ++ii) {
                    multiplier += pow(coefficientOfDelimiter, ii);
                    auto pointToInsert = Point(x0 + xh * multiplier,
                        y0 + yh * multiplier, z0 + zh * multiplier);
                    lineToBuild.insert(lineToBuild.end() - shift, pointToInsert);
                } shift--;
            } figure[k][j] = lineToBuild;
        }
    } mesh.linesAmountX_ = figure[0][0].size();
    // Generation above Y-axis.
    for (size_t k(0); k < mesh.LinesAmountZ; ++k) {
        square2D squareToBuild{};
        squareToBuild = figure[k];
        size_t shift = mesh.LinesAmountY - 1;
        auto iterOnYRefs = mesh.numRefsOfLinesAboveY.begin() + 1;
        for (const auto& info : mesh.delimitersY_) {
            auto rightBorderIter = squareToBuild.end() - shift;
            auto amountOfDelimiters = info.first;
            auto coefficientOfDelimiter = info.second;
            *iterOnYRefs = *(iterOnYRefs - 1) + amountOfDelimiters;
            iterOnYRefs++;
            line1D v0(mesh.LinesAmountX); line1D v1(mesh.LinesAmountX);
            std::copy((*(rightBorderIter - 1)).begin(), (*(rightBorderIter - 1)).end(),
            ↪  v0.begin());
            std::copy((*(rightBorderIter)).begin(), (*(rightBorderIter)).end(),
            ↪  v1.begin());
            square2D subSquareToBuild(amountOfDelimiters - 1);
            for (auto& line : subSquareToBuild) line.resize(mesh.LinesAmountX);
            double denum = 0.0;
            for (size_t ii(0); ii < amountOfDelimiters; ++ii)
                denum += pow(coefficientOfDelimiter, ii);
            for (size_t i(0); i < mesh.LinesAmountX; ++i) {
                double x0 = v0[i].x; double x1 = v1[i].x;
                double y0 = v0[i].y; double y1 = v1[i].y;
                double z0 = v0[i].z; double z1 = v1[i].z;
                double deltX = x1 - x0; double deltY = y1 - y0; double deltZ = z1 - z0;
                double xh = deltX / denum; double yh = deltY / denum;
                double zh = deltZ / denum; double multiplier = 0.0;
                for (size_t ii(0); ii < amountOfDelimiters - 1; ++ii) {
                    multiplier += pow(coefficientOfDelimiter, ii);
                    auto pointToInsert = Point(x0 + xh * multiplier,
                        y0 + yh * multiplier, z0 + zh * multiplier);
                    subSquareToBuild[ii][i] = pointToInsert;
```

```
                }}
            for (auto line : subSquareToBuild)
                squareToBuild.insert(squareToBuild.end() - shift, line);
            shift--;
        } figure[k] = squareToBuild;
    } mesh.linesAmountY_ = figure[0].size();
    // Generate above Z-axis.
    area3D areaToBuild{};
    areaToBuild = figure;
    size_t shift = mesh.LinesAmountZ - 1;
    auto iterOnZRefs = mesh.numRefsOfLinesAboveZ.begin() + 1;
    for (const auto& info : mesh.delimitersZ_) {
        auto rightBorderIter = areaToBuild.end() - shift;
        auto amountOfDelimiters = info.first;
        auto coefficientOfDelimiter = info.second;
        *iterOnZRefs = *(iterOnZRefs - 1) + amountOfDelimiters;
        iterOnZRefs++;
        square2D s0(mesh.LinesAmountY); for (auto& line : s0)
        ↪  line.resize(mesh.LinesAmountX);
        square2D s1(mesh.LinesAmountY); for (auto& line : s1)
        ↪  line.resize(mesh.LinesAmountX);
        std::copy((*(rightBorderIter - 1)).begin(), (*(rightBorderIter - 1)).end(),
        ↪  s0.begin());
        std::copy((*(rightBorderIter)).begin(), (*(rightBorderIter)).end(), s1.begin());
        area3D subAreaToBuild(amountOfDelimiters - 1);
        for (auto& square : subAreaToBuild) {
            square.resize(mesh.LinesAmountY);
            for (auto& line : square)
                line.resize(mesh.LinesAmountX); }
        double denum = 0.0;
        for (size_t ii(0); ii < amountOfDelimiters; ++ii)
            denum += pow(coefficientOfDelimiter, ii);
        for (size_t j(0); j < mesh.LinesAmountY; ++j) {
            for (size_t i(0); i < mesh.LinesAmountX; ++i) {
                double x0 = s0[j][i].x; double x1 = s1[j][i].x;
                double y0 = s0[j][i].y; double y1 = s1[j][i].y;
                double z0 = s0[j][i].z; double z1 = s1[j][i].z;
                double deltX = x1 - x0; double deltY = y1 - y0;
                double deltZ = z1 - z0;
                double xh = deltX / denum;
                double yh = deltY / denum;
                double zh = deltZ / denum;
                double multiplier = 0.0;
                for (size_t ii(0); ii < amountOfDelimiters - 1; ++ii) {
                    multiplier += pow(coefficientOfDelimiter, ii);
                    auto pointToInsert = Point(x0 + xh * multiplier,
                        y0 + yh * multiplier,
                        z0 + zh * multiplier);
                    subAreaToBuild[ii][j][i] = pointToInsert; } } }
        for (auto square : subAreaToBuild)
            areaToBuild.insert(areaToBuild.end() - shift, square);
        shift--;
    } figure = areaToBuild;
    mesh.linesAmountZ_ = figure.size();
    // Convert borders array.
    for (auto& border : mesh.borders_) {
        border.refs_[0] = mesh.numRefsOfLinesAboveX[border.refs_[0]];
        border.refs_[1] = mesh.numRefsOfLinesAboveX[border.refs_[1]];
        border.refs_[2] = mesh.numRefsOfLinesAboveY[border.refs_[2]];
        border.refs_[3] = mesh.numRefsOfLinesAboveY[border.refs_[3]];
        border.refs_[4] = mesh.numRefsOfLinesAboveZ[border.refs_[4]];
        border.refs_[5] = mesh.numRefsOfLinesAboveZ[border.refs_[5]]; }
    // Convert to line-format.
    mesh.linesAmountZ_ = figure.size();
    mesh.points_.resize(mesh.linesAmountX_ * mesh.linesAmountY_ * mesh.linesAmountZ_);
    sxy = mesh.LinesAmountX * mesh.LinesAmountY;
    lx = mesh.LinesAmountX;
    for (size_t k(0); k < mesh.LinesAmountZ; ++k)
        for (size_t j(0); j < mesh.LinesAmountY; ++j)
            for (size_t i(0); i < mesh.LinesAmountX; ++i)
                mesh.points_[k * sxy + j * lx + i] = figure[k][j][i];
}

void MeshGenerator::GenerateListOfAreas(Mesh& mesh) {
    size_t rx = mesh.LinesAmountX - 1; size_t ry = mesh.LinesAmountY - 1;
    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 *
                        ↪  rx, curr + rxy + nx + 1 - j * rx,
193                     curr + rxy + nxy, curr + rxy + nxy + rx, curr + rxy + nxy + rx + 1,
                        ↪  curr + rxy + nxy + rx + nx };
194                 int areaNumber = SelectAreaNum(mesh, arrI);
195                 mesh.areasRibs_.emplace_back(areaNumber, arrI); }
196 }
197
198 void MeshGenerator::GenerateListOfRibs(Mesh& mesh) {
199     auto nx = mesh.LinesAmountX; auto ny = mesh.LinesAmountY;
200     auto nz = mesh.LinesAmountZ;
201     auto nxny = mesh.LinesAmountX * mesh.LinesAmountY;
202     for (int k = 0; k < mesh.LinesAmountZ; k++) {
203         for (int j = 0; j < mesh.LinesAmountY; j++) {
204             for (int i = 0; i < mesh.LinesAmountX - 1; i++)
205                 mesh.referableRibs_.emplace_back(k * nxny + nx * j + i, k * nxny + nx * j
                    ↪  + i + 1);
206             if (j != mesh.LinesAmountY - 1)
207                 for (int i = 0; i < nx; i++)
208                     mesh.referableRibs_.emplace_back(k * nxny + nx * j + i, k * nxny + nx
                        ↪  * (j + 1) + i); }
209         if (k != mesh.LinesAmountZ - 1)
210             for (int j = 0; j < mesh.LinesAmountY; j++)
211                 for (int i = 0; i < mesh.LinesAmountX; i++)
212                     mesh.referableRibs_.emplace_back(k * nxny + nx * j + i, (k + 1) *
                        ↪  nxny + nx * j + i); }
213     Logger::ConsoleOutput("Ribs array generated.", NotificationColor::Passed);
214 }
215
216 void MeshGenerator::GenerateListOfBorders(Mesh& mesh) {
217     Logger::ConsoleOutput("Borders generates just for 1st type and formula num 1!",
        ↪  NotificationColor::Warning);
218     auto nx = mesh.getLinesAmountX(); auto ny = mesh.getLinesAmountY();
219     auto nz = mesh.getLinesAmountZ(); auto nxny = nx * ny;
220     auto rxy = (nx - 1) * ny + (ny - 1) * nx;
221     // XY0
222     for (size_t i = 0; i < ny - 1; i++)
223         for (size_t j = 0; j < nx - 1; j++)
224             mesh.newBorders_.push_back(std::array<size_t, 6> {1, 1,
225                 i * (2 * nx - 1) + j, i * (2 * nx - 1) + j + nx - 1,
226                 i * (2 * nx - 1) + j + nx, i * (2 * nx - 1) + j + nx + nx - 1});
227     // X0Z
228     for (size_t i = 0; i < nz - 1; i++)
229         for (size_t j = 0; j < nx - 1; j++)
230             mesh.newBorders_.push_back(std::array<size_t, 6> {1, 1,
231                 i * (rxy + nxny) + j, i * (rxy + nxny) + j + rxy,
232                 i * (rxy + nxny) + j + rxy + 1, i * (rxy + nxny) + j + rxy + nxny});
233     // 0YZ
234     for (size_t i = 0; i < nz - 1; i++)
235         for (size_t j = 0; j < ny - 1; j++)
236             mesh.newBorders_.push_back(std::array<size_t, 6> {1, 1,
237                 nx - 1 + i * (rxy + nxny) + j * (2 * nx - 1), rxy + i * (rxy + nxny) + j
                    ↪  * nx,
238                 rxy + nx + i * (rxy + nxny) + j * nx, rxy + nxny + nx - 1 + i * (rxy +
                    ↪  nxny) + j * (2 * nx - 1)});
239     // XY1
240     for (size_t i = 0; i < ny - 1; i++)
241         for (size_t j = 0; j < nx - 1; j++)
242             mesh.newBorders_.push_back(std::array<size_t, 6> {1, 1,
243                 (nz - 1)* (rxy + nxny) + j + i * (2 * nx - 1),
244                 (nz - 1)* (rxy + nxny) + nx - 1 + j + i * (2 * nx - 1),
245                 (nz - 1)* (rxy + nxny) + nx + j + i * (2 * nx - 1),
246                 (nz - 1)* (rxy + nxny) + nx + nx - 1 + j + i * (2 * nx - 1)});
247     // X1Z
248     for (size_t i = 0; i < nz - 1; i++)
249         for (size_t j = 0; j < nx - 1; j++)
250             mesh.newBorders_.push_back(std::array<size_t, 6> {1, 1,
```

```
                (ny - 1)* nx + (ny - 1) * (nx - 1) + j + i * (rxy + nxny),
                (ny - 1)* nx + (ny - 1) * (nx - 1) + nxny - 1 + j + i * (rxy + nxny),
                (ny - 1)* nx + (ny - 1) * (nx - 1) + nxny + j + i * (rxy + nxny),
                (ny - 1)* nx + (ny - 1) * (nx - 1) + nxny + j + rxy + i * (rxy + nxny)});
    // 1YZ
    for (size_t i = 0; i < nz - 1; i++)
        for (size_t j = 0; j < ny - 1; j++)
            mesh.newBorders_.push_back(std::array<size_t, 6> {1, 1,
                nx - 1 + i * (rxy + nxny) + j * (2 * nx - 1) + nx - 1,
                rxy + i * (rxy + nxny) + j * nx + nx - 1,
                rxy + nx + i * (rxy + nxny) + j * nx + nx - 1,
                rxy + nxny + nx - 1 + i * (rxy + nxny) + j * (2 * nx - 1) + nx - 1});
}
```

LOS.h

```
#pragma once

#include "Solver.h"

class LOS : public Solver {
public:
    LOS() : Solver() {};
    LOS(double eps, size_t maxIters) : Solver(eps, maxIters) {};
    ~LOS() {};
    GlobalVector Solve(const GlobalMatrix& A, const GlobalVector& b) const override;
};

```

LOS.cpp

```
#include "LOS.h"

GlobalVector LOS::Solve(const GlobalMatrix& A, const GlobalVector& b) const {
    GlobalVector x(b.Size); GlobalVector _x(b.Size);
    GlobalVector r(b.Size); GlobalVector _r(b.Size);
    GlobalVector z(b.Size); GlobalVector _z(b.Size);
    GlobalVector p(b.Size); GlobalVector _p(b.Size);
    double alph(0.0); double beta(0.0);
    r = b - A * x; z = r; p = A * r; size_t iter(0);
    do {
        _x = x; _z = z; _r = r; _p = p;
        alph = (_p * _r) / (_p * _p);
        x = _x + alph * _z; r = _r - alph * _p;
        beta = -1.0 * (_p * (A * r)) / (_p * _p);
        z = r + beta * _z; p = A * r + beta * _p;
        if (iter % 4 == 0) std::cout << iter << ". " << std::scientific << r.Norma() /
        ↪  b.Norma() << std::endl;
        ++iter;
    } while (iter < _maxIters and r.Norma() / b.Norma() >= _eps);
    return x;
}
```

Pardiso.cpp

```
#include "stdafx.h"

ofstream logfile;

int Write_Txt_File_Of_Double(const char *fname, double *massiv, int n_of_records, int
↪  len_of_record) {
  FILE *fp;
  int i, j;
  if ((fp = fopen(fname, "w")) == 0) {
    printf("Error: Cannot open file \"%s\" for writing.\n", fname);
    return 1; }
```

```
  printf("writing %s... ", fname);
  for (i = 0; i < n_of_records; i++) {
    for (j = 0; j < len_of_record; j++)
      fprintf(fp, "%25.13e\t", massiv[i*len_of_record + j]);
    fprintf(fp, "\n");}
  printf("done\n");
  fclose(fp);
  return 0;
}

int Read_Long_From_Txt_File(const char *fname, int *number) {
  FILE *fp;
  int temp; int retcode;
  if ((fp = fopen(fname, "r")) == 0) {
    char str[100];
    sprintf(str, "Error: Cannot open file \"%s\" for reading.\n", fname);
    return 1;} retcode = fscanf(fp, "%ld", &temp);
  if (retcode != 1){
    char str[100];
    sprintf(str, "Error reading file \"%s\".\n", fname);
    fclose(fp);  } *number = temp;
  fclose(fp); return 0;
}

int Read_Bin_File_Of_Double(const char *fname, double *massiv, int n_of_records, int
↪  len_of_record) {
  int temp; FILE *fp;
  if ((fp = fopen(fname, "r+b")) == 0) {
    char str[100];
    sprintf(str, "Error: Cannot open file \"%s\" for reading.\n", fname);
    return 1; }
  temp = fread(massiv, sizeof(double)*len_of_record, n_of_records, fp);
  if (temp != n_of_records) {
    char str[100];
    sprintf(str, "Error reading file \"%s\". %ld of %ld records was read.\n", fname,
    ↪  temp, n_of_records);
    fclose(fp);
    return 1; }
  fclose(fp);  return 0;
}

int Read_Bin_File_Of_Long(const char *fname, int *massiv, int n_of_records, int
↪  len_of_record) {
  int temp; FILE *fp;
  if ((fp = fopen(fname, "r+b")) == 0) {
    char str[100];
    sprintf(str, "Cannot open file %s.\n", fname);
    return 1; }
  temp = fread(massiv, sizeof(int)*len_of_record, n_of_records, fp);
  if (temp != n_of_records){
    char str[100];
    sprintf(str, "Error reading file \"%s\". %ld of %ld records was read.\n", fname,
    ↪  temp, n_of_records);
    fclose(fp);
    return 1;}
  fclose(fp); return 0;
}

void FromRSFToCSR_Real_1_Sym(int nb, int *ig, int *sz_ia, int *sz_ja) {
  *sz_ia = nb + 1; *sz_ja = ig[nb] + nb;
}

void FromRSFToCSR_Real_2_Sym(int nb, int *ig, int *jg, double *di, double *gg,
  MKL_INT *ia, MKL_INT *ja, double *a) {
  int i, j, k;
  vector<MKL_INT> adr;
  adr.resize(nb, 0);
  for (i = 0; i<nb; i++) {
    adr[i] += 1;
    for (j = ig[i]; j <= ig[i + 1] - 1; j++) {
      k = jg[j];
      adr[k]++;}}
  // ia
  ia[0] = 0;
  for (i = 0; i<nb; i++) ia[i + 1] = ia[i] + adr[i];
```

```
82
83    // ja,  a
84    for (i = 0; i<ig[nb] + nb; i++)  a[i] = 0;
85
86    for (i = 0; i<nb; i++) adr[i] = ia[i];
87
88    for (i = 0; i<nb; i++) {
89      ja[adr[i]] = i;
90      a[adr[i]] = di[i];
91      adr[i]++; }
92    for (i = 0; i<nb; i++) {
93      for (j = ig[i]; j <= ig[i + 1] - 1; j++) {
94        k = jg[j]; ja[adr[k]] = i;
95        a[adr[k]] = gg[j]; adr[k]++; } }
96  }
97
98
99  int _tmain(int argc, _TCHAR* argv[]) {
100   logfile.open("pardiso64.log");
101   if (!logfile) {
102     cerr << "Cannot open pardiso64.log" << endl;
103     return 1;}
104   int i; int ig_n_1 = 0;
105   int sz_ia = 0; int sz_ja = 0;
106   int nb;  int *ig = NULL;
107   int *jg = NULL;  double *di = NULL;
108   double *ggl = NULL;
109   clock_t begin = clock();
110   MKL_INT pt[64]; for (int i(0); i < 64; ++i) pt[i] = 0;
111   MKL_INT maxfct = 1;  MKL_INT  mnum = 1;
112   MKL_INT mtype = 2; MKL_INT phase = 13;
113   MKL_INT n = 0;
114   double *a = NULL; MKL_INT *ia = NULL;
115   MKL_INT *ja = NULL;  MKL_INT *perm = NULL;
116   MKL_INT nrhs = 1; MKL_INT iparm[64];
117   iparm[0] = 1; for (i = 1; i < 64; i++) iparm[i] = 0;
118   iparm[0] = 1; MKL_INT msglvl = 1; double *pr = NULL;
119   double *x = NULL; MKL_INT error = 0;
120   Read_Long_From_Txt_File("kuslau2", &nb);
121   ig = new int[nb + 1];
122   Read_Bin_File_Of_Long("ig", ig, nb + 1, 1);
123   for (i = 0; i<nb + 1; i++) ig[i]--;
124   ig_n_1 = ig[nb];
125   jg = new int[ig_n_1];
126   Read_Bin_File_Of_Long("jg", jg, ig_n_1, 1);
127   for (i = 0; i<ig_n_1; i++) jg[i]--;
128   di = new double[nb];
129   Read_Bin_File_Of_Double("di", di, nb, 1);
130   ggl = new double[ig_n_1];
131   Read_Bin_File_Of_Double("gg", ggl, ig_n_1, 1);
132   pr = new double[nb];
133   Read_Bin_File_Of_Double("pr", pr, nb, 1);
134   FromRSFToCSR_Real_1_Sym(nb, ig, &sz_ia, &sz_ja);
135   ia = new MKL_INT[sz_ia]; ja = new MKL_INT[sz_ja];
136   a = new double[sz_ja];
137   FromRSFToCSR_Real_2_Sym(nb, ig, jg, di, ggl, ia, ja, a);
138   for (i = 0; i<sz_ia; i++) ia[i]++;
139   for (i = 0; i<sz_ja; i++) ja[i]++;
140   if (ig) { delete[] ig; ig = NULL; }
141   if (jg) { delete[] jg; jg = NULL; }
142   if (di) { delete[] di; di = NULL; }
143   if (ggl) { delete[] ggl; ggl = NULL; }
144   n = nb;  x = new double[nb];
145   perm = new MKL_INT[nb];
146   cout << "pardiso start.." << endl << flush;
147   PARDISO(pt,  &maxfct, &mnum,  &mtype,  &phase,
148     &n,  a, ia, ja, perm, &nrhs,  iparm,
149     &msglvl, pr, x,  &error); phase = -1;
150   PARDISO(pt,  &maxfct, &mnum,  &mtype,  &phase,
151     &n, a, ia, ja, perm, &nrhs, iparm,
152     &msglvl, pr, x,  &error);
153   clock_t time = (clock() - begin) / CLOCKS_PER_SEC;
154   ofstream fout; fout.open("kit", ios_base::app);
155   fout << "n=" << nb << " pardiso time=" << time << " s" << endl;
156   fout.close();
157   Write_Txt_File_Of_Double("x.txt", x, nb, 1);
```

```
158     cout << "info=" << error << endl;
159     logfile << "info=" << error << endl;
160     if (a) { delete[] a;  a = NULL; }
161     if (x) { delete[] x;  x = NULL; }
162     if (pr) { delete[] pr; pr = NULL; }
163     if (ia) { delete[] ia; ia = NULL; }
164     if (ja) { delete[] ja; ja = NULL; }
165     if (perm) { delete[] perm; perm = NULL; }
166     logfile.close(); logfile.clear();
167     return 0;
168 }
```