МИНИСТЕРСТВО НАУКИ
И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Новосибирский государственный технический университет»

НГТУ НЭТИ | Факультет прикладной математики и информатики

Кафедра прикладной математики

Лабораторная работа №3
по дисциплине «Уравнения математической физики»

Решение гармонических задач

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

Новосибирск, 2022

Цель работы

Разработать программу решения гармонической задачи методом конечных элементов. Провести сравнение прямого и итерационного методов решения получаемой в результате конечноэлементной аппроксимации СЛАУ.

Вариант: Решить трехмерную гармоническую задачу в декартовых координатах, базисные функции-трилинейные.

Теоретическая часть

Постановка задачи

Рассмотрим задачу для уравнения

$$\chi \frac{\partial u}{\partial t^2} + \sigma \frac{\partial u}{\partial t} - \operatorname{div}(\lambda \operatorname{grad} u) = f,$$

в котором правая часть f представима в виде:

$$f(x,y,z,t) = f^s(x,y,z)\sin\omega + f^c(x,y,z)\cos\omega.$$

Если остальные параметры рассматриваемого уравнения не зависят от времени, то тогда и его решения u может быть представлено в виде:

$$u(x,y,z,t) = u^s(x,y,z)\sin\omega t + u^c(x,y,z)\cos\omega t,$$

где u^s и u^c – две зависящие только от пространственнных координат функции, удовлетворяющие системе уравнений:

$$\begin{cases} -\operatorname{div}(\lambda \operatorname{grad} u^s) - \omega\sigma u^c - \omega^2\chi u^s = f^s, \\ -\operatorname{div}(\lambda \operatorname{grad} u^c) + \omega\sigma u^s - \omega^2\chi u^c = f^c. \end{cases}$$

То же самое можно сказать и о решении краевой задачи, если не только f, но и параметры краевхы условий являются гармонически изменяющимися по времени с одной и той же частотой w:

$$u_g(x,y,z,t) = u_g^s(x,y,z)\sin\omega t + u_g^c(x,y,z)\cos\omega t,$$

$$\theta(x,y,z,t) = \theta^s(x,y,z)\sin\omega t + \theta^c(x,y,z)\cos\omega t,$$

$$u_\beta(x,y,z,t) = u_\beta^s(x,y,z)\sin\omega t + u_\beta^c(x,y,z)\cos\omega t.$$

В этом случае функциии u^s и u^c должны удовлетворять краевым условиям:

$$u^s\big|_{S_1} = u_g^s, \quad u^c\big|_{S_1} = u_g^c,$$

$$\lambda \frac{\partial u^s}{\partial n}\bigg|_{S_2} = \theta^s, \quad \lambda \frac{\partial u^c}{\partial n}\bigg|_{S_2} = \theta^c$$

$$\lambda \frac{\partial u^s}{\partial n}\bigg|_{S_2} + \beta\left(u^s\big|_{S_3} - u_\beta^s\right) = 0, \quad \lambda \frac{\partial u^c}{\partial n}\bigg|_{S_2} + \beta\left(u^c\big|_{S_3} - u_\beta^c\right) = 0$$

Конечноэлементная аппроксимация

Выполним конечноэлементную аппроксимацию краевой задачи.

Сначала получим эквивалентную вариационную формулировку. Для этого умножим каждое из уравнений системы на пробную функцию υ и применим формулу Грина (интегрирования по частям). В результате получим систему двух вариацоинных уравнений:

$$\begin{cases} \int_{\Omega} \left(\lambda \operatorname{grad} u^s \operatorname{grad} \upsilon - \omega\sigma u^c \upsilon - \omega^2 \chi u^s \upsilon\right) d\Omega + \int_{S_3} \beta u^s \upsilon dS = \\ = \int_{\Omega} f^s \upsilon d\Omega + \int_{S_2} \theta^s \upsilon dS + \int_{S_3} \beta u^s_\beta \upsilon dS \\ \int_{\Omega} \left(\lambda \operatorname{grad} u^c \operatorname{grad} \upsilon - \omega\sigma u^s \upsilon - \omega^2 \chi u^c \upsilon\right) d\Omega + \int_{S_3} \beta u^c \upsilon = \\ = \int_{\Omega} f^c \upsilon d\Omega + \int_{S_2} \theta^c \upsilon dS + \int_{S_3} \beta u^c_\beta \upsilon dS. \end{cases}$$

Уравнения должны выполняться для любой $\upsilon \in H^1$, удовлетворяющей однородному первому краевому условию на границе S_1.

Построим конечноэлементную аппроксимацию на основе вариационнной формулировки. Пусть $\{\psi_i\}$ – набор базисных функций. Чтобы получить конечноэлементную СЛАУ, заменим в каждую из искомых функций u^s и u^c на функции $u^{s,h} = \sum_{i=1}^n q_i^s \psi_i$ и $u^{c,h} = \sum_{i=1}^n q_i^c \psi_i$, а вместо пробной функции подставим поочередно базисные функции ψ_i :

$$\begin{cases} \sum\limits_{j=1}^{n} \left(\int_{\Omega} \left(\lambda \operatorname{grad} \psi_i \operatorname{grad} \psi_j - \omega^2 \chi \psi_i \psi_j\right) d\Omega + \int_{S_3} \beta \psi_i \psi_j \right) q_j^s - \omega \sum\limits_{j=1}^{n} \left(\int_{\Omega} \sigma \psi_i \psi_j d\Omega \right) q_j^c = \\ \qquad = \int_{\Omega} f^s \psi_i d\Omega + \int_{S_2} \theta^s \psi_i dS + \int_{S_3} \beta u^s_\beta \psi_i dS, \quad i = 1 \dots n, \\ \sum\limits_{j=1}^{n} \left(\int_{\Omega} \left(\lambda \operatorname{grad} \psi_i \operatorname{grad} \psi_j - \omega^2 \chi \psi_i \psi_j\right) d\Omega + \int_{S_3} \beta \psi_i \psi_j \right) q_j - \omega \sum\limits_{j=1}^{n} \left(\int_{\Omega} \sigma \psi_i \psi_j d\Omega \right) q_j^s = \\ \qquad = \int_{\Omega} f^c \psi_i d\Omega + \int_{S_2} \theta^c \psi_i dS + \int_{S_3} \beta u^c_\beta \psi_i dS, \quad i = 1 \dots n. \end{cases}$$

В результате мы получили ситсему из $2n$ уравнений с $2n$ неизвестными q_j^s и q_j^c. Чтобы определить матрицу и вектор правой части полученной конечноэлементной СЛАУ, пронумеруем ее уравнения как $2i - 1$ и $2i$. Соответственно пронумеруем и неизвестные этой системы, которые обзначим $q_j, \ j = 1 \dots 2n : q_{2j-1} = q_j^s, \ q_{2j} = q_j^c, \ j = 1 \dots n$.

Обозначим

$$p_{ij} = \int_{\Omega} \left(\lambda \operatorname{grad} \psi_i \operatorname{grad} \psi_j - \omega^2 \chi \psi_i \psi_j\right) d\Omega + \int_{S_3} \beta \psi_i \psi_j,$$

$$c_{ij} = \omega \int_{\Omega} \sigma \psi_i \psi_j d\Omega.$$

Тогда матрица конечноэлементной СЛАУ будет иметь следующую струкутуру:

$$
A = \begin{pmatrix}
p_{11} & -c_{11} & p_{12} & -c_{12} & \dots & \dots & p_{1n} & -c_{1n} \\
c_{11} & p_{11} & c_{12} & p_{12} & \dots & \dots & c_{1n} & p_{1n} \\
p_{21} & -c_{21} & p_{22} & -c_{22} & \dots & \dots & p_{2n} & -c_{2n} \\
c_{21} & p_{21} & c_{22} & p_{22} & \dots & \dots & c_{2n} & p_{2n} \\
 & & & & \ddots & & & \\
 & & & & & \ddots & & \\
p_{n1} & -c_{n1} & p_{n2} & -c_{n2} & \dots & \dots & p_{nn} & -c_{nn} \\
c_{n1} & p_{n1} & c_{n2} & p_{n2} & \dots & \dots & c_{nn} & p_{nn}
\end{pmatrix}.
$$

Очевидно, что в этой матрице выделяются блоки размера 2 x 2 вида:

$$
A^{ij} = \begin{pmatrix} p_{ij} & -c_{ij} \\ c_{ij} & p_{ij} \end{pmatrix},
$$

для хранения которых достаточно только двух ячеек памяти. Таким образом, для хранения всей матрицы в блочном формате с учетом структуры блока A^{ij} требуется практически в два раза меньше памяти, чем при хранении ее покомпонентно. Поэтому при программной реализации в таких случаях, как правило, используются блочные форматы хранения данных, в том числе и с учетом разреженной структуры матрицы.

Практическая часть

1. Выполнить конечноэлементую аппроксимацию исходного уравнения в соответсвии с заданием. Получить формулы для вычисления компонент матрицы A и вектора правой части b для метода простой итерации.

2. Реализовать программу для решения гармонической задачи.

3. Протестировать разработанную программу на полиномах первой степени.

4. Провести исследования реализованных методов для сеток с небольшим количеством узлов 500 - 1000 и большим количеством узлов - порядка 20000 - 50000 для различных значений параметров $10^{-4} \le \omega \le 10^9$, $10^2 \le \lambda \le 8 \cdot 10^5$, $0 \le \sigma \le 10^8$, $8.81 \cdot 10^{-12} \le \chi \le 10^{-10}$. Для всех решенных задач сравнить вычислителные затраты, требуемые для решения СЛАУ итерационным и прямым методом.

Описание программы

Программа состоит из нескольких модулей:

- Класс FEM, в котором происходят основные вычислительные операции.
- Класс SparseMatrix для представления матрицы в разреженном формате.
- Класс Matrix для представления матрицы в плотном формате.
- Класс Vector для представления вектора.
- Абстрактный класс Solver и его класс-наследник LOSLU для решения СЛАУ итерационным методом.
- Абстрактный класс Decomposer и его класс-наследник DecomposerLDU для решения СЛАУ прямым методом.
- Структура Interval для представления интервала.
- Статические классы Integration и Quadratures для интегрирования.
- Статический класс LinearBasis, в котором записаны базисные функции - трилинейные.
- Статический класс ArrayHelper для методов расширения одномерного массива (копирование, заполнение).
- Класс GridParameters для различных параметров сетки.
- Класс GridFactory для генерации сеток через фабричный метод.
- Абстрактный класс Grid и классы-наследнки, которые его реализуют RegularGrid, IrregularGrid для представления сеток.
- Структура Point3D для представления трехмерной точки.
- Интерфейс ITest и классы, которые его реализуют для тестирования.

Тестирование

Исследования для сетки с небольшим количеством узлов

Функция:

- $u^s = x + y + z$
- $u^c = x - y - z.$

Правая часть:

- $f^s = -\omega\sigma\,(x + y + z) - \omega^2\chi(x - y - z)$
- $f^c = \omega\sigma\,(x - y - z) - \omega^2\chi(x + y + z).$

Заданы краевые условия первого рода.

Изменение параметра ω

Коэффициенты:

- $\sigma = 2$
- $\chi = 10^{-11}$
- $\lambda = 110.$

ω	LOSLU (ms)	LDU (ms)	Погрешность u^s (LOS)	Погрешность u^c (LOS)	Погрешность u^s (LDU)	Погрешность u^c (LDU)
$1 \cdot 10^{-4}$	16	46	$1.62 \cdot 10^{-11}$	$8.16 \cdot 10^{-12}$	$1.74 \cdot 10^{-16}$	$7.13 \cdot 10^{-17}$
$1 \cdot 10^{-3}$	5	43	$1.6 \cdot 10^{-11}$	$8.4 \cdot 10^{-12}$	$1.84 \cdot 10^{-16}$	$8.65 \cdot 10^{-17}$
$1 \cdot 10^{-2}$	5	43	$1.53 \cdot 10^{-11}$	$8.51 \cdot 10^{-12}$	$2.53 \cdot 10^{-16}$	$7.37 \cdot 10^{-17}$
0.1	5	42	$6.49 \cdot 10^{-11}$	$2.66 \cdot 10^{-11}$	$8.42 \cdot 10^{-16}$	$2.9 \cdot 10^{-16}$
1	7	51	$7.37 \cdot 10^{-11}$	$4.67 \cdot 10^{-11}$	$3.36 \cdot 10^{-16}$	$1.34 \cdot 10^{-16}$
10	6	51	$5.65 \cdot 10^{-11}$	$2.08 \cdot 10^{-10}$	$3.86 \cdot 10^{-16}$	$2.79 \cdot 10^{-16}$
100	4	42	$1.75 \cdot 10^{-11}$	$1.6 \cdot 10^{-11}$	$1.71 \cdot 10^{-16}$	$7.98 \cdot 10^{-17}$
$1{,}000$	4	49	$3.56 \cdot 10^{-11}$	$1.32 \cdot 10^{-11}$	$3.97 \cdot 10^{-16}$	$1.32 \cdot 10^{-16}$
$10{,}000$	7	50	$6.85 \cdot 10^{-12}$	$4.37 \cdot 10^{-12}$	$3.11 \cdot 10^{-15}$	$1.78 \cdot 10^{-16}$
$1 \cdot 10^{5}$	13	45	$5.22 \cdot 10^{-11}$	$3.33 \cdot 10^{-11}$	$3.46 \cdot 10^{-16}$	$1.5 \cdot 10^{-16}$
$1 \cdot 10^{6}$	4	41	$5.22 \cdot 10^{-11}$	$3.33 \cdot 10^{-11}$	$3.56 \cdot 10^{-16}$	$1.14 \cdot 10^{-16}$
$1 \cdot 10^{7}$	4	41	$5.22 \cdot 10^{-11}$	$3.33 \cdot 10^{-11}$	$3.63 \cdot 10^{-16}$	$1.04 \cdot 10^{-16}$
$1 \cdot 10^{8}$	4	41	$5.22 \cdot 10^{-11}$	$3.33 \cdot 10^{-11}$	$3.19 \cdot 10^{-16}$	$1.08 \cdot 10^{-16}$
$1 \cdot 10^{9}$	4	41	$5.22 \cdot 10^{-11}$	$3.33 \cdot 10^{-11}$	$3.41 \cdot 10^{-16}$	$1.21 \cdot 10^{-16}$

Изменение параметра λ

Коэффициенты:

- $\omega = 1$
- $\sigma = 2$
- $\chi = 10^{-11}$

λ	LOSLU (ms)	LDU (ms)	Погрешность u^s (LOS)	Погрешность u^c (LOS)	Погрешность u^s (LDU)	Погрешность u^c (LDU)
100	6	48	$7.89 \cdot 10^{-11}$	$6.77 \cdot 10^{-11}$	$6 \cdot 10^{-16}$	$2.05 \cdot 10^{-16}$
$1{,}000$	5	42	$7.59 \cdot 10^{-11}$	$3.26 \cdot 10^{-11}$	$3.9 \cdot 10^{-16}$	$1.29 \cdot 10^{-16}$
$10{,}000$	6	51	$1.53 \cdot 10^{-11}$	$8.46 \cdot 10^{-12}$	$1.79 \cdot 10^{-16}$	$7.49 \cdot 10^{-17}$
$1 \cdot 10^{5}$	5	42	$1.6 \cdot 10^{-11}$	$8.42 \cdot 10^{-12}$	$1.83 \cdot 10^{-16}$	$6.36 \cdot 10^{-17}$
$8 \cdot 10^{5}$	5	45	$1.62 \cdot 10^{-11}$	$8.17 \cdot 10^{-12}$	$1.21 \cdot 10^{-16}$	$5.19 \cdot 10^{-17}$

Изменение параметра σ

Коэффициенты:

- $\omega = 1$
- $\chi = 10^{-11}$
- $\lambda = 110.$

σ	LOSLU (ms)	LDU (ms)	Погрешность u^s (LOS)	Погрешность u^c (LOS)	Погрешность u^s (LDU)	Погрешность u^c (LDU)
0	2	42	0.5	0.23	NaN	NaN
10	6	47	$3.47 \cdot 10^{-10}$	$1.45 \cdot 10^{-10}$	$3.31 \cdot 10^{-16}$	$1.76 \cdot 10^{-16}$
100	5	42	$3.74 \cdot 10^{-11}$	$1.06 \cdot 10^{-10}$	$1.68 \cdot 10^{-16}$	$1.79 \cdot 10^{-16}$
$1{,}000$	4	45	$9.42 \cdot 10^{-11}$	$1.03 \cdot 10^{-10}$	$2.72 \cdot 10^{-16}$	$1.35 \cdot 10^{-16}$
$10{,}000$	5	46	$2.6 \cdot 10^{-11}$	$1.53 \cdot 10^{-11}$	$1.56 \cdot 10^{-15}$	$1.57 \cdot 10^{-16}$
$1 \cdot 10^5$	4	42	$6.3 \cdot 10^{-12}$	$1.79 \cdot 10^{-12}$	$1.38 \cdot 10^{-14}$	$1.68 \cdot 10^{-16}$
$1 \cdot 10^6$	5	42	$2.47 \cdot 10^{-13}$	$8.41 \cdot 10^{-14}$	$1.54 \cdot 10^{-13}$	$1.53 \cdot 10^{-16}$
$1 \cdot 10^7$	5	42	$3.34 \cdot 10^{-13}$	$3.33 \cdot 10^{-15}$	$1.39 \cdot 10^{-12}$	$1.39 \cdot 10^{-16}$
$1 \cdot 10^8$	7	51	$3.82 \cdot 10^{-12}$	$2.33 \cdot 10^{-16}$	$1.37 \cdot 10^{-11}$	$1.58 \cdot 10^{-16}$

Изменение параметра χ

Коэффициенты:

- $\omega = 1$
- $\sigma = 2$
- $\lambda = 110.$

χ	LOSLU (ms)	LDU (ms)	Погрешность u^s (LOS)	Погрешность u^c (LOS)	Погрешность u^s (LDU)	Погрешность u^c (LDU)
$1 \cdot 10^{-12}$	5	44	$1.52 \cdot 10^{-10}$	$2 \cdot 10^{-10}$	$1.58 \cdot 10^{-16}$	$8.3 \cdot 10^{-17}$
$1 \cdot 10^{-11}$	5	45	$1.52 \cdot 10^{-10}$	$2 \cdot 10^{-10}$	$3.9 \cdot 10^{-16}$	$1.28 \cdot 10^{-16}$
$1 \cdot 10^{-10}$	5	42	$1.52 \cdot 10^{-10}$	$2 \cdot 10^{-10}$	$3.92 \cdot 10^{-16}$	$1.6 \cdot 10^{-16}$

Исследования для сетки с большим количеством узлов

Функция:

- $u^s = 3x - 2y + z$
- $u^c = 2x + y - z.$

Правая часть:

- $f^s = -\omega\sigma\,(2x + y - z) - \omega^2\chi(3x - 2y + z)$
- $f^c = \omega\sigma\,(3x - 2y + z) - \omega^2\chi(2x + y - z).$

Заданы краевые условия первого рода.

Изменение параметра ω

Коэффициенты:

- $\sigma = 2$
- $\chi = 10^{-11}$
- $\lambda = 110.$

ω	LOSLU (ms)	LDU (ms)	Погрешность u^s (LOS)	Погрешность u^c (LOS)	Погрешность u^s (LDU)	Погрешность u^c (LDU)
$1 \cdot 10^{-4}$	69	324	$5.16 \cdot 10^{-11}$	$3.57 \cdot 10^{-11}$	$2.32 \cdot 10^{-16}$	$2.11 \cdot 10^{-16}$
$1 \cdot 10^{-3}$	59	310	$5.14 \cdot 10^{-11}$	$3.75 \cdot 10^{-11}$	$3.51 \cdot 10^{-16}$	$2.6 \cdot 10^{-16}$
$1 \cdot 10^{-2}$	60	312	$1.12 \cdot 10^{-10}$	$3.42 \cdot 10^{-11}$	$4.32 \cdot 10^{-16}$	$3.69 \cdot 10^{-16}$
0.1	64	301	$3.9 \cdot 10^{-11}$	$1.91 \cdot 10^{-11}$	$2 \cdot 10^{-15}$	$1.91 \cdot 10^{-15}$
1	82	296	$1.11 \cdot 10^{-10}$	$4.14 \cdot 10^{-11}$	$9.01 \cdot 10^{-16}$	$6.9 \cdot 10^{-16}$
10	113	281	$9.09 \cdot 10^{-11}$	$1.6 \cdot 10^{-11}$	$8.28 \cdot 10^{-16}$	$3.55 \cdot 10^{-16}$
100	41	321	$1.34 \cdot 10^{-11}$	$1.44 \cdot 10^{-11}$	$2.07 \cdot 10^{-16}$	$2.51 \cdot 10^{-16}$
1,000	39	279	$1.42 \cdot 10^{-11}$	$1.24 \cdot 10^{-11}$	$8.56 \cdot 10^{-16}$	$4.87 \cdot 10^{-16}$
10,000	38	272	$2.35 \cdot 10^{-12}$	$1.97 \cdot 10^{-12}$	$8.28 \cdot 10^{-15}$	$4.98 \cdot 10^{-16}$
$1 \cdot 10^{5}$	47	170	$1.99 \cdot 10^{-11}$	$1.13 \cdot 10^{-11}$	$4.67 \cdot 10^{-16}$	$3.68 \cdot 10^{-16}$
$1 \cdot 10^{6}$	43	248	$1.17 \cdot 10^{-12}$	$2.77 \cdot 10^{-12}$	$8.49 \cdot 10^{-13}$	$4.96 \cdot 10^{-16}$
$1 \cdot 10^{7}$	16	261	$3.68 \cdot 10^{-12}$	$3.02 \cdot 10^{-12}$	$4.61 \cdot 10^{-11}$	$9.76 \cdot 10^{-16}$
$1 \cdot 10^{8}$	20	819	$3.68 \cdot 10^{-12}$	$3.02 \cdot 10^{-12}$	$1.12 \cdot 10^{-12}$	$4.64 \cdot 10^{-16}$
$1 \cdot 10^{9}$	18	794	$3.68 \cdot 10^{-12}$	$3.02 \cdot 10^{-12}$	$1.08 \cdot 10^{-13}$	$4.7 \cdot 10^{-16}$

Изменение параметра λ

Коэффициенты:

- $\omega = 1$
- $\sigma = 2$
- $\chi = 10^{-11}$

λ	LOSLU (ms)	LDU (ms)	Погрешность u^s (LOS)	Погрешность u^c (LOS)	Погрешность u^s (LDU)	Погрешность u^c (LDU)
100	105	809	$1.49 \cdot 10^{-10}$	$7.88 \cdot 10^{-11}$	$1.51 \cdot 10^{-15}$	$1.08 \cdot 10^{-15}$
1,000	87	818	$5.33 \cdot 10^{-11}$	$1.76 \cdot 10^{-11}$	$9.35 \cdot 10^{-16}$	$8.48 \cdot 10^{-16}$
10,000	65	801	$1.17 \cdot 10^{-10}$	$4.01 \cdot 10^{-11}$	$3.44 \cdot 10^{-16}$	$3.14 \cdot 10^{-16}$
$1 \cdot 10^{5}$	66	184	$5.15 \cdot 10^{-11}$	$3.77 \cdot 10^{-11}$	$2.18 \cdot 10^{-16}$	$1.78 \cdot 10^{-16}$
$8 \cdot 10^{5}$	73	163	$5.15 \cdot 10^{-11}$	$3.57 \cdot 10^{-11}$	$4.86 \cdot 10^{-16}$	$3.94 \cdot 10^{-16}$

Изменение параметра σ

Коэффициенты:

- $\omega = 1$
- $\chi = 10^{-11}$
- $\lambda = 110.$

σ	LOSLU (ms)	LDU (ms)	Погрешность u^s (LOS)	Погрешность u^c (LOS)	Погрешность u^s (LDU)	Погрешность u^c (LDU)
0	18	577	0.37	0.3	NaN	NaN
10	127	997	$5.32 \cdot 10^{-11}$	$1.48 \cdot 10^{-10}$	$8.94 \cdot 10^{-16}$	$2.76 \cdot 10^{-16}$
100	58	140	$8.92 \cdot 10^{-12}$	$6.44 \cdot 10^{-12}$	$2.51 \cdot 10^{-16}$	$2.58 \cdot 10^{-16}$
$1{,}000$	40	368	$3.37 \cdot 10^{-12}$	$5.3 \cdot 10^{-12}$	$4.46 \cdot 10^{-16}$	$4.08 \cdot 10^{-16}$
$10{,}000$	39	407	$4.99 \cdot 10^{-12}$	$3.47 \cdot 10^{-12}$	$4.29 \cdot 10^{-15}$	$5.34 \cdot 10^{-16}$
$1 \cdot 10^5$	44	851	$1.6 \cdot 10^{-13}$	$5.46 \cdot 10^{-14}$	$4.26 \cdot 10^{-14}$	$5.02 \cdot 10^{-16}$
$1 \cdot 10^6$	46	189	$5.95 \cdot 10^{-14}$	$2.72 \cdot 10^{-15}$	$4.37 \cdot 10^{-13}$	$4.99 \cdot 10^{-16}$
$1 \cdot 10^7$	49	200	$5.84 \cdot 10^{-13}$	$2.81 \cdot 10^{-15}$	$4.28 \cdot 10^{-12}$	$5.09 \cdot 10^{-16}$
$1 \cdot 10^8$	53	196	$6.07 \cdot 10^{-12}$	$3.04 \cdot 10^{-16}$	$4.14 \cdot 10^{-11}$	$4.8 \cdot 10^{-16}$

Изменение параметра χ

Коэффициенты:

- $\omega = 1$
- $\sigma = 2$
- $\lambda = 110.$

χ	LOSLU (ms)	LDU (ms)	Погрешность u^s (LOS)	Погрешность u^c (LOS)	Погрешность u^s (LDU)	Погрешность u^c (LDU)
$1 \cdot 10^{-12}$	137	513	$9.99 \cdot 10^{-11}$	$2.83 \cdot 10^{-11}$	$3.98 \cdot 10^{-16}$	$3.36 \cdot 10^{-16}$
$1 \cdot 10^{-11}$	75	324	$9.99 \cdot 10^{-11}$	$2.83 \cdot 10^{-11}$	$8.8 \cdot 10^{-16}$	$7.43 \cdot 10^{-16}$
$1 \cdot 10^{-10}$	86	254	$9.99 \cdot 10^{-11}$	$2.83 \cdot 10^{-11}$	$1.16 \cdot 10^{-15}$	$9.05 \cdot 10^{-16}$

Проведенные исследования и выводы

В целом, результаты везде одинаковые. При $\sigma = 0$ мы не смогли получить решение. LDU работает в разы дольше, чем LOS с LU предобуславливанием.

Тексты основных модулей

Program.cs

```csharp
using eMP_3;

GridFactory gridFactory = new();
var grid = gridFactory.CreateGrid(GridTypes.Regular,
↪ GridParameters.ReadJson("grid.jsonc")!.Value);

#region Итерационный метод
FEM fem1 = FEM.CreateBuilder().SetSpaceGrid(grid).SetTest(new
↪ Test1()).SetSolverSLAE(new LOSLU(1000, 1E-14));
fem1.Compute();
#endregion

#region Прямой метод
FEM fem2 = FEM.CreateBuilder().SetSpaceGrid(grid).SetTest(new
↪ Test1()).SetDecomposer(new DecomposerLDU());
fem2.Compute();
#endregion
```

FEM.cs

```csharp
namespace eMP_3;

public class FEM {
    public class FEMBuilder {
        private readonly FEM _fem = new();

        public FEMBuilder SetTest(ITest test) {
            _fem._test = test;
            return this;
        }

        public FEMBuilder SetSpaceGrid(Grid grid) {
            _fem._grid = grid;
            return this;
        }

        public FEMBuilder SetSolverSLAE(Solver solver) {
            _fem._solver = solver;
            return this;
        }

        public FEMBuilder SetDecomposer(Decomposer decomposer) {
            _fem._decomposer = decomposer;
            return this;
        }

        public static implicit operator FEM(FEMBuilder builder)
            => builder._fem;
```

```
    }

    // default! указывает на то, что данное поле не может принимать null
    private delegate double Basis(Point3D point);
    private Basis[] _basis = default!;
    private ITest _test = default!;
    private Grid _grid = default!;
    private Solver? _solver;
    private Decomposer? _decomposer;
    private Matrix _massMatrix = default!; // матрица масс
    private Matrix _stiffnessMatrix = default!; // матрица жесткости
    private SparseMatrix _globalMatrix = default!;
    private Vector<double> _localVector1 = default!;
    private Vector<double> _localVector2 = default!;
    private Vector<double> _vector = default!; // вектор правой части

    public void Compute() {
        try {
            ArgumentNullException.ThrowIfNull(_test, $"{nameof(_test)} cannot be null, set the test");

            if (_solver is null && _decomposer is null) {
                throw new ArgumentNullException(nameof(_solver), "Set the method of solving SLAE");
            }

            Init();
            Solve();
        } catch (Exception ex) {
            Console.WriteLine($"We had problem: {ex.Message}");
        }
    }

    private void Init() {
        _massMatrix = new(8);
        _stiffnessMatrix = new(8);
        _localVector1 = new(8);
        _localVector2 = new(8);

        _basis = new Basis[] { LinearBasis.Psi1, LinearBasis.Psi2, LinearBasis.Psi3,
                               LinearBasis.Psi4, LinearBasis.Psi5, LinearBasis.Psi6,
                               LinearBasis.Psi7, LinearBasis.Psi8};
    }

    private void InitSLAE(int sizeOffDiag) {
        _globalMatrix = new(2 * _grid.Points.Length, sizeOffDiag); // resizing in method ConstructPortrait()
        _vector = new(2 * _grid.Points.Length);
    }

    private void Solve() {
        ConstructPortrait();
        AssemblySLAE();

        AccountingDirichletBoundary();
        _globalMatrix.PrintDense("matrix.txt");

        if (_decomposer is not null) {
            _globalMatrix.AsProfileMatrix();
            _decomposer.SetMatrix(_globalMatrix);
```

```
            _decomposer.SetVector(_vector);
            _decomposer.Compute();
            Console.WriteLine(_decomposer.RunningTime);
            ErrForward();
        } else {
            _solver!.SetMatrix(_globalMatrix);
            _solver.SetVector(_vector);
            _solver.Compute();
            Console.WriteLine(_solver.RunningTime);
            ErrIter();
        }
    }

    private void ConstructPortrait() {
        List<int>[] list = new List<int>[_grid.Points.Length].Select(_ => new List<int>()).ToArray();

        for (int ielem = 0; ielem < _grid.Elements.Length; ielem++) {
            for (int i = 0; i < _grid.Elements[ielem].Length; i++) {
                for (int j = i + 1; j < _grid.Elements[ielem].Length; j++) {
                    int pos = _grid.Elements[ielem][j];
                    int elem = _grid.Elements[ielem][i];

                    if (!list[pos].Contains(elem)) {
                        list[pos].Add(elem);
                    }
                }
            }
        }

        list = list.Select(list => list.OrderBy(value => value).ToList()).ToArray();
        int sizeOffDiag = list.Sum(childList => childList.Count);

        InitSLAE(sizeOffDiag);

        _globalMatrix.ig[2] = 1;

        for (int i = 1; i < list.Length; i++) {
            _globalMatrix.ig[(2 * i) + 1] = _globalMatrix.ig[2 * i] + (2 * list[i].Count);
            _globalMatrix.ig[(2 * i) + 2] = _globalMatrix.ig[(2 * i) + 1] + (2 * list[i].Count) + 1;
        }

        _globalMatrix.jg = new int[_globalMatrix.ig[^1]];
        _globalMatrix.ggl = new double[_globalMatrix.ig[^1]];
        _globalMatrix.ggu = new double[_globalMatrix.ig[^1]];

        int index = 1;

        for (int i = 1; i < list.Length; i++) {
            for (int j = 0; j < list[i].Count; j++) {
                _globalMatrix.jg[index] = 2 * list[i][j];
                _globalMatrix.jg[index + 1] = (2 * list[i][j]) + 1;
                index += 2;
            }

            for (int k = 0; k < list[i].Count; k++) {
                _globalMatrix.jg[index] = 2 * list[i][k];
                _globalMatrix.jg[index + 1] = (2 * list[i][k]) + 1;
```

```
                index += 2;
            }

            _globalMatrix.jg[index] = 2 * i;
            index++;
        }
    }

    private void AssemblySLAE() {
        for (int ielem = 0; ielem < _grid.Elements.Length; ielem++) {
            AssemblyLocalMatrices(ielem);
            AssemblyLocalVector(ielem);

            for (int i = 0; i < _grid.Elements[ielem].Length; i++) {
                for (int j = 0; j < _grid.Elements[ielem].Length; j++) {
                    AddElementToGlobalMatrix(2 * _grid.Elements[ielem][i], 2 * _grid.Elements[ielem][j], _stiffnessMatrix[i, j]);
                    AddElementToGlobalMatrix((2 * _grid.Elements[ielem][i]) + 1, (2 * _grid.Elements[ielem][j]) + 1, _stiffnessMatrix[i, j]);
                    AddElementToGlobalMatrix(2 * _grid.Elements[ielem][i], (2 * _grid.Elements[ielem][j]) + 1, -_massMatrix[i, j]);
                    AddElementToGlobalMatrix((2 * _grid.Elements[ielem][i]) + 1, 2 * _grid.Elements[ielem][j], _massMatrix[i, j]);
                }
            }

            for (int i = 0; i < _localVector1.Length; i++) {
                _vector[2 * _grid.Elements[ielem][i]] += _localVector1[i];
                _vector[(2 * _grid.Elements[ielem][i]) + 1] += _localVector2[i];
            }

            _localVector1.Fill(0);
            _localVector2.Fill(0);
        }
    }

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

        if (i < j) {
            for (int index = _globalMatrix.ig[j]; index < _globalMatrix.ig[j + 1]; index++) {
                if (_globalMatrix.jg[index] == i) {
                    _globalMatrix.ggu[index] += value;
                    return;
                }
            }
        } else {
            for (int index = _globalMatrix.ig[i]; index < _globalMatrix.ig[i + 1]; index++) {
                if (_globalMatrix.jg[index] == j) {
                    _globalMatrix.ggl[index] += value;
                    return;
                }
            }
        }
    }
```

```
    private void AssemblyLocalMatrices(int ielem) {
        double hx = Math.Abs(_grid.Points[_grid.Elements[ielem][1]].X - _grid.Points[_grid.Elements[ielem][0]].X);
        double hy = Math.Abs(_grid.Points[_grid.Elements[ielem][2]].Y - _grid.Points[_grid.Elements[ielem][0]].Y);
        double hz = Math.Abs(_grid.Points[_grid.Elements[ielem][4]].Z - _grid.Points[_grid.Elements[ielem][0]].Z);

        for (int i = 0; i < _stiffnessMatrix.Size; i++) {
            for (int j = 0; j < _stiffnessMatrix.Size; j++) {
                _stiffnessMatrix[i, j] = (_grid.Lambda * (Integration.GaussSegmentGrad(_basis[i].Invoke, _basis[j].Invoke, new(0, 0, 0), new(1, 1, 1)) /
                (hx * hy * hz))) - (_grid.Omega * _grid.Omega * _grid.Chi * (Integration.GaussSegment(_basis[i].Invoke, _basis[j].Invoke, new(0, 0, 0), new(1, 1, 1)) * (hx * hy * hz)));
            }
        }

        for (int i = 0; i < _massMatrix.Size; i++) {
            for (int j = 0; j < _massMatrix.Size; j++) {
                _massMatrix[i, j] = _grid.Omega * _grid.Sigma * Integration.GaussSegment(_basis[i].Invoke, _basis[j].Invoke,
                        new(0, 0, 0), new(1, 1, 1)) * (hx * hy * hz);
            }
        }
    }

    private void AssemblyLocalVector(int ielem) {
        for (int i = 0; i < _massMatrix.Size; i++) {
            for (int j = 0; j < _massMatrix.Size; j++) {
                _localVector1[i] += _test.Fs(_grid.Points[_grid.Elements[ielem][j]]) * _massMatrix[i, j] / (_grid.Omega * _grid.Sigma);
                _localVector2[i] += _test.Fc(_grid.Points[_grid.Elements[ielem][j]]) * _massMatrix[i, j] / (_grid.Omega * _grid.Sigma);
            }
        }
    }

    private void AccountingDirichletBoundary() {
        for (int iside = 0; iside < _grid.Sides.Length; iside++) {
            for (int inode = 0; inode < _grid.Sides[iside].Length; inode++) {
                _globalMatrix.di[2 * _grid.Sides[iside][inode]] = 1.0;
                _globalMatrix.di[(2 * _grid.Sides[iside][inode]) + 1] = 1.0;
                _vector[2 * _grid.Sides[iside][inode]] = _test.Us(_grid.Points[_grid.Sides[iside][inode]]);
                _vector[(2 * _grid.Sides[iside][inode]) + 1] = _test.Uc(_grid.Points[_grid.Sides[iside][inode]]);

                int diagonal = 2 * _grid.Sides[iside][inode];

                for (int k = _globalMatrix.ig[diagonal]; k < _globalMatrix.ig[diagonal + 1]; k++) {
                    _globalMatrix.ggl[k] = 0.0;
                }

                for (int i = diagonal + 1; i < _globalMatrix.Size; i++) {
                    for (int k = _globalMatrix.ig[i]; k < _globalMatrix.ig[i + 1]; k++) {
```

```
                        if (_globalMatrix.jg[k] == diagonal) {
                            _globalMatrix.ggu[k] = 0.0;
                            break;
                        }
                    }
                }

                diagonal = (2 * _grid.Sides[iside][inode]) + 1;

                for (int k = _globalMatrix.ig[diagonal]; k <
                ↪   _globalMatrix.ig[diagonal + 1]; k++) {
                    _globalMatrix.ggl[k] = 0.0;
                }

                for (int i = diagonal + 1; i < _globalMatrix.Size; i++) {
                    for (int k = _globalMatrix.ig[i]; k < _globalMatrix.ig[i + 1];
                    ↪   k++) {
                        if (_globalMatrix.jg[k] == diagonal) {
                            _globalMatrix.ggu[k] = 0.0;
                            break;
                        }
                    }
                }
            }
        }
    }

    //for report
    private void ErrIter() {
        using var sw = new StreamWriter("results/errIter.txt");
        for (int i = 0; i < _grid.Points.Length; i++) {
            sw.WriteLine(Math.Abs(_solver!.Solution!.Value[2 * i] -
            ↪   _test.Us(_grid.Points[i])));
            sw.WriteLine(Math.Abs(_solver.Solution.Value[(2 * i) + 1] -
            ↪   _test.Uc(_grid.Points[i])));
        }
    }

    private void ErrForward() {
        using var sw = new StreamWriter("results/errForward.txt");
        for (int i = 0; i < _grid.Points.Length; i++) {
            sw.WriteLine(Math.Abs(_decomposer!.Solution!.Value[2 * i] -
            ↪   _test.Us(_grid.Points[i])));
            sw.WriteLine(Math.Abs(_decomposer.Solution.Value[(2 * i) + 1] -
            ↪   _test.Uc(_grid.Points[i])));
        }
    }

    public static FEMBuilder CreateBuilder()
        => new();
}
```

SparseMatrix.cs

```
namespace eMP_3;

public class SparseMatrix {
    // public fields - its bad, but the readability is better
    public int[] ig = default!;
    public int[] jg = default!;
```

```
    public double[] di = default!;
    public double[] ggl = default!;
    public double[] ggu = default!;
    public int Size { get; init; }

    public SparseMatrix(int size, int sizeOffDiag) {
        Size = size;
        ig = new int[size + 1];
        jg = new int[sizeOffDiag];
        ggl = new double[sizeOffDiag];
        ggu = new double[sizeOffDiag];
        di = new double[size];
    }

    public static Vector<double> operator *(SparseMatrix matrix, Vector<double>
        vector) {
        Vector<double> product = new(vector.Length);

        for (int i = 0; i < vector.Length; i++) {
            product[i] = matrix.di[i] * vector[i];

            for (int j = matrix.ig[i]; j < matrix.ig[i + 1]; j++) {
                product[i] += matrix.ggl[j] * vector[matrix.jg[j]];
                product[matrix.jg[j]] += matrix.ggu[j] * vector[i];
            }
        }

        return product;
    }

    public void PrintDense(string path) {
        double[,] A = new double[Size, Size];

        for (int i = 0; i < Size; i++) {
            A[i, i] = di[i];

            for (int j = ig[i]; j < ig[i + 1]; j++) {
                A[i, jg[j]] = ggl[j];
                A[jg[j], i] = ggu[j];
            }
        }

        using var sw = new StreamWriter(path);
        for (int i = 0; i < Size; i++) {
            for (int j = 0; j < Size; j++) {
                sw.Write(A[i, j].ToString("0.0000") + "\t");
            }

            sw.WriteLine();
        }
    }

    public void AsProfileMatrix() {
        int[] ignew = ig.ToArray();

        for (int i = 0; i < Size; i++) {
            int i0 = ig[i];
            int i1 = ig[i + 1];

            int profile = i1 - i0;
```

```
                if (profile > 0) {
                    int count = i - jg[i0];
                    ignew[i + 1] = ignew[i] + count;
                } else {
                    ignew[i + 1] = ignew[i];
                }
            }

            double[] gglnew = new double[ignew[^1]];
            double[] ggunew = new double[ignew[^1]];

            for (int i = 0; i < Size; i++) {
                int i0 = ignew[i];
                int i1 = ignew[i + 1];

                int j = i - (i1 - i0);

                int i0Old = ig[i];

                for (int ik = i0; ik < i1; ik++, j++) {
                    if (j == jg[i0Old]) {
                        gglnew[ik] = ggl[i0Old];
                        ggunew[ik] = ggu[i0Old];
                        i0Old++;
                    } else {
                        gglnew[ik] = 0.0;
                        ggunew[ik] = 0.0;
                    }
                }
            }

            ig = ignew;
            ggl = gglnew;
            ggu = ggunew;
        }
    }
```

Matrix.cs

```
namespace eMP_3;

public class Matrix {
    private readonly double[,] storage;
    public int Size { get; init; }

    public double this[int i, int j] {
        get => storage[i, j];
        set => storage[i, j] = value;
    }

    public Matrix(int size) {
        storage = new double[size, size];
        Size = size;
    }

    public void Clear()
        => Array.Clear(storage, 0, storage.Length);

    public static Matrix operator +(Matrix fstMatrix, Matrix sndMatrix) {
```

```
        Matrix resultMatrix = new(fstMatrix.Size);

        for (int i = 0; i < resultMatrix.Size; i++) {
            for (int j = 0; j < resultMatrix.Size; j++) {
                resultMatrix[i, j] = fstMatrix[i, j] + sndMatrix[i, j];
            }
        }

        return resultMatrix;
    }
}
```

Vector.cs

```
namespace eMP_3;

public class Vector<T> where T : INumber<T> {
    private readonly T[] vec;
    public int Length { get; init; }

    public T this[int index] {
        get => vec[index];
        set => vec[index] = value;
    }

    public Vector(int dim) {
        vec = new T[dim];
        Length = dim;
    }

    public static T operator *(Vector<T> firstVec, Vector<T> secondVec) {
        T result = T.Zero;

        for (int i = 0; i < firstVec.Length; i++) {
            result += firstVec[i] * secondVec[i];
        }

        return result;
    }

    public static Vector<T> operator *(double constant, Vector<T> vector) {
        Vector<T> result = new(vector.Length);

        for (int i = 0; i < vector.Length; i++) {
            result[i] = vector[i] * T.Create(constant);
        }

        return result;
    }

    public static Vector<T> operator +(Vector<T> firstVec, Vector<T> secondVec) {
        Vector<T> result = new(firstVec.Length);

        for (int i = 0; i < firstVec.Length; i++) {
            result[i] = firstVec[i] + secondVec[i];
        }

        return result;
    }

```

```
    public static Vector<T> operator -(Vector<T> firstVec, Vector<T> secondVec) {
        Vector<T> result = new(firstVec.Length);

        for (int i = 0; i < firstVec.Length; i++) {
            result[i] = firstVec[i] - secondVec[i];
        }

        return result;
    }

    public static void Copy(Vector<T> source, Vector<T> destination) {
        for (int i = 0; i < source.Length; i++) {
            destination[i] = source[i];
        }
    }

    public void Fill(double value) {
        for (int i = 0; i < Length; i++) {
            vec[i] = T.Create(value);
        }
    }

    public double Norm() {
        T result = T.Zero;

        for (int i = 0; i < Length; i++) {
            result += vec[i] * vec[i];
        }

        return Math.Sqrt(Convert.ToDouble(result));
    }

    public ImmutableArray<T> ToImmutableArray()
        => ImmutableArray.Create(vec);
}
```

Solvers.cs

```
namespace eMP_3;

public abstract class Solver {
    protected TimeSpan _runningTime;
    protected SparseMatrix _matrix = default!;
    protected Vector<double> _vector = default!;
    protected Vector<double>? _solution;
    public int MaxIters { get; init; }
    public double Eps { get; init; }
    public TimeSpan? RunningTime => _runningTime;
    public ImmutableArray<double>? Solution => _solution?.ToImmutableArray();

    protected Solver(int maxIters, double eps)
        => (MaxIters, Eps) = (maxIters, eps);

    public void SetMatrix(SparseMatrix matrix)
        => _matrix = matrix;

    public void SetVector(Vector<double> vector)
        => _vector = vector;

    public abstract void Compute();
```

```
}

public class LOS : Solver {
    public LOS(int maxIters, double eps) : base(maxIters, eps) { }

    public override void Compute() {
        try {
            ArgumentNullException.ThrowIfNull(_matrix, $"{nameof(_matrix)} cannot be null, set the matrix");
            ArgumentNullException.ThrowIfNull(_vector, $"{nameof(_vector)} cannot be null, set the vector");

            double alpha, beta;
            double squareNorm;

            _solution = new(_vector.Length);

            Vector<double> r = new(_vector.Length);
            Vector<double> z = new(_vector.Length);
            Vector<double> p = new(_vector.Length);
            Vector<double> tmp = new(_vector.Length);

            Stopwatch sw = Stopwatch.StartNew();

            r = _vector - (_matrix * _solution);

            Vector<double>.Copy(r, z);

            p = _matrix * z;

            squareNorm = r * r;

            for (int index = 0; index < MaxIters && squareNorm > Eps; index++) {
                alpha = p * r / (p * p);
                _solution += alpha * z;
                squareNorm = (r * r) - (alpha * alpha * (p * p));
                r -= alpha * p;

                tmp = _matrix * r;

                beta = -(p * tmp) / (p * p);
                z = r + (beta * z);
                p = tmp + (beta * p);
            }

            sw.Stop();

            _runningTime = sw.Elapsed;
        } catch (Exception ex) {
            Console.WriteLine($"We had problem: {ex.Message}");
        }
    }
}

public class LOSLU : Solver {
    public LOSLU(int maxIters, double eps) : base(maxIters, eps) { }

    public override void Compute() {
        try {
            ArgumentNullException.ThrowIfNull(_matrix, $"{nameof(_matrix)} cannot be null, set the matrix");
```

```
            ArgumentNullException.ThrowIfNull(_vector, $"{nameof(_vector)} cannot be null, set the vector");

            double alpha, beta;
            double squareNorm;

            _solution = new(_vector.Length);

            double[] gglnew = new double[_matrix.ggl.Length];
            double[] ggunew = new double[_matrix.ggu.Length];
            double[] dinew = new double[_matrix.di.Length];

            _matrix.ggl.Copy(gglnew);
            _matrix.ggu.Copy(ggunew);
            _matrix.di.Copy(dinew);

            Vector<double> r = new(_vector.Length);
            Vector<double> z = new(_vector.Length);
            Vector<double> p = new(_vector.Length);
            Vector<double> tmp = new(_vector.Length);

            Stopwatch sw = Stopwatch.StartNew();

            LU(gglnew, ggunew, dinew);

            r = Direct(_vector - MultDi(_solution), gglnew, dinew);
            z = Reverse(r, ggunew);
            p = Direct(_matrix * z, gglnew, dinew);

            squareNorm = r * r;

            for (int iter = 0; iter < MaxIters && squareNorm > Eps; iter++) {
                alpha = p * r / (p * p);
                squareNorm = (r * r) - (alpha * alpha * (p * p));
                _solution += alpha * z;
                r -= alpha * p;

                tmp = Direct(_matrix * Reverse(r, ggunew), gglnew, dinew);

                beta = -(p * tmp) / (p * p);
                z = Reverse(r, ggunew) + (beta * z);
                p = tmp + (beta * p);
            }

            sw.Stop();

            _runningTime = sw.Elapsed;
        } catch (Exception ex) {
            Console.WriteLine($"We had problem: {ex.Message}");
        }
    }

    private Vector<double> Direct(Vector<double> vector, double[] gglnew, double[] dinew) {
        Vector<double> y = new(vector.Length);
        Vector<double>.Copy(vector, y);

        double sum = 0.0;

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

```
            int i0 = _matrix.ig[i];
            int i1 = _matrix.ig[i + 1];

            for (int k = i0; k < i1; k++)
                sum += gglnew[k] * y[_matrix.jg[k]];

            y[i] = (y[i] - sum) / dinew[i];
            sum = 0.0;
        }

        return y;
    }

    private Vector<double> Reverse(Vector<double> vector, double[] ggunew) {
        Vector<double> result = new(vector.Length);
        Vector<double>.Copy(vector, result);

        for (int i = _matrix.Size - 1; i >= 0; i--) {
            int i0 = _matrix.ig[i];
            int i1 = _matrix.ig[i + 1];

            for (int k = i0; k < i1; k++)
                result[_matrix.jg[k]] -= ggunew[k] * result[i];
        }

        return result;
    }

    private void LU(double[] gglnew, double[] ggunew, double[] dinew) {
        double suml = 0.0;
        double sumu = 0.0;
        double sumdi = 0.0;

        for (int i = 0; i < _matrix.Size; i++) {
            int i0 = _matrix.ig[i];
            int i1 = _matrix.ig[i + 1];

            for (int k = i0; k < i1; k++) {
                int j = _matrix.jg[k];
                int j0 = _matrix.ig[j];
                int j1 = _matrix.ig[j + 1];
                int ik = i0;
                int kj = j0;

                while (ik < k && kj < j1) {
                    if (_matrix.jg[ik] == _matrix.jg[kj]) {
                        suml += gglnew[ik] * ggunew[kj];
                        sumu += ggunew[ik] * gglnew[kj];
                        ik++;
                        kj++;
                    } else if (_matrix.jg[ik] > _matrix.jg[kj]) {
                        kj++;
                    } else {
                        ik++;
                    }
                }

                gglnew[k] -= suml;
                ggunew[k] = (ggunew[k] - sumu) / dinew[j];
                sumdi += gglnew[k] * ggunew[k];
```

```
                suml = 0.0;
                sumu = 0.0;
            }

            dinew[i] -= sumdi;
            sumdi = 0.0;
        }
    }

    private Vector<double> MultDi(Vector<double> vector) {
        Vector<double> product = new(vector.Length);

        for (int i = 0; i < _matrix.Size; i++) {
            product[i] = 1 / Math.Sqrt(_matrix.di[i]) * vector[i];
        }

        return product;
    }
}

public class BCGSTABLU : Solver {
    public BCGSTABLU(int maxIters, double eps) : base(maxIters, eps) { }

    public override void Compute() {
        try {
            ArgumentNullException.ThrowIfNull(_matrix, $"{nameof(_matrix)} cannot be
             ↪  null, set the matrix");
            ArgumentNullException.ThrowIfNull(_vector, $"{nameof(_vector)} cannot be
             ↪  null, set the vector");

            double alpha = 1.0;
            double omega = 1.0;
            double rho = 1.0;
            double beta, temp;

            double vectorNorm = _vector.Norm();

            _solution = new(_vector.Length);

            double[] gglnew = new double[_matrix.ggl.Length];
            double[] ggunew = new double[_matrix.ggu.Length];
            double[] dinew = new double[_matrix.di.Length];

            _matrix.ggl.Copy(gglnew);
            _matrix.ggu.Copy(ggunew);
            _matrix.di.Copy(dinew);

            Vector<double> r = new(_vector.Length);
            Vector<double> r0 = new(_vector.Length);
            Vector<double> z = new(_vector.Length);
            Vector<double> p = new(_vector.Length);
            Vector<double> v = new(_vector.Length);
            Vector<double> s = new(_vector.Length);
            Vector<double> t = new(_vector.Length);

            Stopwatch sw = Stopwatch.StartNew();

            LU(gglnew, ggunew, dinew);

            r = Direct(_vector - (_matrix * _solution), gglnew, dinew);
```

```
            Vector<double>.Copy(r, r0);

            for (int iter = 0; iter < MaxIters && r.Norm() / vectorNorm >= Eps; iter++) {
                temp = rho;
                rho = r0 * r;
                beta = rho / temp * (alpha / omega);
                p = r + (beta * (p - (omega * v)));
                v = Direct(_matrix * Reverse(p, ggunew), gglnew, dinew);
                alpha = rho / (r0 * v);
                s = r - (alpha * v);
                t = Direct(_matrix * Reverse(s, ggunew), gglnew, dinew);
                omega = t * s / (t * t);
                _solution += (omega * s) + (alpha * p);
                r = s - (omega * t);
            }

            _solution = Reverse(_solution, ggunew);

            sw.Stop();

            _runningTime = sw.Elapsed;
        } catch (Exception ex) {
            Console.WriteLine($"We had problem: {ex.Message}");
        }
    }

    private Vector<double> Direct(Vector<double> vector, double[] gglnew, double[] dinew) {
        Vector<double> y = new(vector.Length);
        Vector<double>.Copy(vector, y);

        double sum = 0.0;

        for (int i = 0; i < _matrix.Size; i++) {
            int i0 = _matrix.ig[i];
            int i1 = _matrix.ig[i + 1];

            for (int k = i0; k < i1; k++)
                sum += gglnew[k] * y[_matrix.jg[k]];

            y[i] = (y[i] - sum) / dinew[i];
            sum = 0.0;
        }

        return y;
    }

    private Vector<double> Reverse(Vector<double> vector, double[] ggunew) {
        Vector<double> result = new(vector.Length);
        Vector<double>.Copy(vector, result);

        for (int i = _matrix.Size - 1; i >= 0; i--) {
            int i0 = _matrix.ig[i];
            int i1 = _matrix.ig[i + 1];

            for (int k = i0; k < i1; k++)
                result[_matrix.jg[k]] -= ggunew[k] * result[i];
        }
```

```
        return result;
    }

    private void LU(double[] gglnew, double[] ggunew, double[] dinew) {
        double suml = 0.0;
        double sumu = 0.0;
        double sumdi = 0.0;

        for (int i = 0; i < _matrix.Size; i++) {
            int i0 = _matrix.ig[i];
            int i1 = _matrix.ig[i + 1];

            for (int k = i0; k < i1; k++) {
                int j = _matrix.jg[k];
                int j0 = _matrix.ig[j];
                int j1 = _matrix.ig[j + 1];
                int ik = i0;
                int kj = j0;

                while (ik < k && kj < j1) {
                    if (_matrix.jg[ik] == _matrix.jg[kj]) {
                        suml += gglnew[ik] * ggunew[kj];
                        sumu += ggunew[ik] * gglnew[kj];
                        ik++;
                        kj++;
                    } else if (_matrix.jg[ik] > _matrix.jg[kj]) {
                        kj++;
                    } else {
                        ik++;
                    }
                }

                gglnew[k] -= suml;
                ggunew[k] = (ggunew[k] - sumu) / dinew[j];
                sumdi += gglnew[k] * ggunew[k];
                suml = 0.0;
                sumu = 0.0;
            }

            dinew[i] -= sumdi;
            sumdi = 0.0;
        }
    }
}
```

Decomposers.cs

```
namespace eMP_3;

public abstract class Decomposer {
    protected TimeSpan _runningTime;
    protected SparseMatrix _matrix = default!;
    protected Vector<double> _vector = default!;
    protected Vector<double>? _solution;
    public TimeSpan? RunningTime => _runningTime;
    public ImmutableArray<double>? Solution => _solution?.ToImmutableArray();

    public void SetMatrix(SparseMatrix matrix)
        => _matrix = matrix;
```

```
    public void SetVector(Vector<double> vector)
        => _vector = vector;

    public abstract void Compute();
}

public class DecomposerLDU : Decomposer {
    public override void Compute() {
        _solution = new(_matrix.Size);
        Vector<double>.Copy(_vector, _solution);

        Stopwatch sw = Stopwatch.StartNew();

        LDU();
        ForwardElimination();
        DiagonalStroke();
        BackSubstitution();

        sw.Stop();

        _runningTime = sw.Elapsed;
    }

    private void LDU() {
        for (int i = 0; i < _matrix.Size; i++) {
            double sumdi = 0.0;

            int i0 = _matrix.ig[i];
            int i1 = _matrix.ig[i + 1];

            int j = i - (i1 - i0);

            for (int ij = i0; ij < i1; ij++, j++) {
                double suml = 0.0;
                double sumu = 0.0;

                int j0 = _matrix.ig[j];
                int j1 = _matrix.ig[j + 1];

                int ik = i0;
                int jk = j0;

                int k = i - (i1 - i0);

                int ci = ij - i0;
                int cj = j1 - j0;

                int cij = ci - cj;

                if (cij > 0) {
                    ik += cij;
                    k += cij;
                } else {
                    jk -= cij;
                }

                for (; ik < ij; ik++, jk++, k++) {
                    suml += _matrix.ggl[ik] * _matrix.di[k] * _matrix.ggu[jk];
                    sumu += _matrix.ggu[ik] * _matrix.di[k] * _matrix.ggl[jk];
```

```
                }

                _matrix.ggl[ij] = (_matrix.ggl[ij] - suml) / _matrix.di[j];
                _matrix.ggu[ij] = (_matrix.ggu[ij] - sumu) / _matrix.di[j];

                sumdi += _matrix.ggl[ij] * _matrix.ggu[ij] * _matrix.di[k];
            }

            _matrix.di[i] -= sumdi;
        }
    }

    private void DiagonalStroke() {
        for (int i = 0; i < _matrix.Size; i++) {
            _solution![i] /= _matrix.di[i];
        }
    }

    private void ForwardElimination() {
        for (int i = 0; i < _matrix.Size; i++) {
            double sum = 0.0;

            int i0 = _matrix.ig[i];
            int i1 = _matrix.ig[i + 1];

            int j = i - (i1 - i0);

            for (int ij = i0; ij < i1; ij++, j++) {
                sum += _matrix.ggl[ij] * _solution![j];
            }

            _solution![i] -= sum;
        }
    }

    private void BackSubstitution() {
        for (int i = _matrix.Size - 1; i >= 0; i--) {
            int i0 = _matrix.ig[i];
            int i1 = _matrix.ig[i + 1];

            for (int j = i - 1, ij = i1 - 1; ij >= i0; ij--, j--) {
                _solution![j] -= _matrix.ggu[ij] * _solution[i];
            }
        }
    }
}
```

Interval.cs

```
namespace eMP_3;

public readonly record struct Interval {
    [JsonProperty("Left Border")]
    public double LeftBorder { get; init; }

    [JsonProperty("Right Border")]
    public double RightBorder { get; init; }

    [JsonIgnore]
    public double Lenght { get; init; }
```

```

    [JsonConstructor]
    public Interval(double leftBorder, double rightBorder) {
        LeftBorder = leftBorder;
        RightBorder = rightBorder;
        Lenght = Math.Abs(rightBorder - leftBorder);
    }
}
```

Integration.cs

```
namespace eMP_3;

public static class Integration {
    public static double GaussSegment(Func<Point3D, double> psiI, Func<Point3D,
    ↪ double> psiJ, Point3D firstPoint, Point3D secondPoint) {
        var quadratures = Quadratures.SegmentGaussOrder9();

        double hx = secondPoint.X - firstPoint.X;
        double hy = secondPoint.Y - firstPoint.Y;
        double hz = secondPoint.Z - firstPoint.Z;
        double result = 0.0;

        foreach (var qi in quadratures) {
            foreach (var qj in quadratures) {
                foreach (var qk in quadratures) {
                    Point3D point = new(((qi.Node * hx) + firstPoint.X +
                    ↪ secondPoint.X) / 2.0,
                                        ((qj.Node * hy) + firstPoint.Y + secondPoint.Y)
                                        ↪ / 2.0,
                                        ((qk.Node * hz) + firstPoint.Z + secondPoint.Z)
                                        ↪ / 2.0);

                    result += psiI(point) * psiJ(point) * qi.Weight * qj.Weight *
                    ↪ qk.Weight;
                }
            }
        }

        return result * hx * hy * hz / 8.0;
    }

    public static double GaussSegmentGrad(Func<Point3D, double> psiI, Func<Point3D,
    ↪ double> psiJ, Point3D firstPoint, Point3D secondPoint) {
        var quadratures = Quadratures.SegmentGaussOrder9();

        double hx = secondPoint.X - firstPoint.X;
        double hy = secondPoint.Y - firstPoint.Y;
        double hz = secondPoint.Z - firstPoint.Z;
        double resultX = 0.0;
        double resultY = 0.0;
        double resultZ = 0.0;

        foreach (var qi in quadratures) {
            foreach (var qj in quadratures) {
                foreach (var qk in quadratures) {
                    Point3D point = new(((qi.Node * hx) + firstPoint.X +
                    ↪ secondPoint.X) / 2.0,
                                        ((qj.Node * hy) + firstPoint.Y + secondPoint.Y)
                                        ↪ / 2.0,
```

```
                                     ((qk.Node * hz) + firstPoint.Z + secondPoint.Z) / 2.0);

                    resultX += DerivativeX(point, hx) * qi.Weight * qj.Weight * qk.Weight;
                }
            }
        }

        resultX *= hx * hy * hz / 8.0;

        foreach (var qi in quadratures) {
            foreach (var qj in quadratures) {
                foreach (var qk in quadratures) {
                    Point3D point = new(((qi.Node * hx) + firstPoint.X + secondPoint.X) / 2.0,
                                     ((qj.Node * hy) + firstPoint.Y + secondPoint.Y) / 2.0,
                                     ((qk.Node * hz) + firstPoint.Z + secondPoint.Z) / 2.0);

                    resultY += DerivativeY(point, hy) * qi.Weight * qj.Weight * qk.Weight;
                }
            }
        }

        resultY *= hx * hy * hz / 8.0;

        foreach (var qi in quadratures) {
            foreach (var qj in quadratures) {
                foreach (var qk in quadratures) {
                    Point3D point = new(((qi.Node * hx) + firstPoint.X + secondPoint.X) / 2.0,
                                     ((qj.Node * hy) + firstPoint.Y + secondPoint.Y) / 2.0,
                                     ((qk.Node * hz) + firstPoint.Z + secondPoint.Z) / 2.0);

                    resultZ += DerivativeZ(point, hz) * qi.Weight * qj.Weight * qk.Weight;
                }
            }
        }

        resultZ *= hx * hy * hz / 8.0;

        return resultX + resultY + resultZ;

        double DerivativeX(Point3D point, double h)
            => (psiI(point + (h, 0, 0)) - psiI(point - (h, 0, 0))) / (2 * h) *
               ((psiJ(point + (h, 0, 0)) - psiJ(point - (h, 0, 0))) / (2 * h));

        double DerivativeY(Point3D point, double h)
           => (psiI(point + (0, h, 0)) - psiI(point - (0, h, 0))) / (2 * h) *
              ((psiJ(point + (0, h, 0)) - psiJ(point - (0, h, 0))) / (2 * h));

        double DerivativeZ(Point3D point, double h)
           => (psiI(point + (0, 0, h)) - psiI(point - (0, 0, h))) / (2 * h) *
              ((psiJ(point + (0, 0, h)) - psiJ(point - (0, 0, h))) / (2 * h));
```

```
    }
}
```

Quadratures.cs

```
namespace eMP_3;

public class QuadratureNode<T> where T : notnull {
    public T Node { get; init; }
    public double Weight { get; init; }

    public QuadratureNode(T node, double weight) {
        Node = node;
        Weight = weight;
    }
}

public class Quadrature<T> where T : notnull {
    private readonly QuadratureNode<T>[] _nodes = default!;
    public ImmutableArray<QuadratureNode<T>> Nodes => _nodes.ToImmutableArray();

    public Quadrature(QuadratureNode<T>[] nodes)
        => _nodes = nodes;
}

public static class Quadratures {
    public static IEnumerable<QuadratureNode<double>> SegmentGaussOrder9() {
        const int n = 5;
        double[] points = { -1.0 / 3.0 * Math.Sqrt(5 + (2 * Math.Sqrt(10.0 / 7.0))),
                            1.0 / 3.0 * Math.Sqrt(5 - (2 * Math.Sqrt(10.0 / 7.0))),
                            0.0,
                            -1.0 / 3.0 * Math.Sqrt(5 - (2 * Math.Sqrt(10.0 / 7.0))),
                            1.0 / 3.0 * Math.Sqrt(5 + (2 * Math.Sqrt(10.0 / 7.0)))};

        double[] weights = { (322.0 - (13.0 * Math.Sqrt(70.0))) / 900.0, (322.0 +
        ↪ (13.0 * Math.Sqrt(70.0))) / 900.0,
                             128.0 / 225.0,
                            (322.0 + (13.0 * Math.Sqrt(70.0))) / 900.0,
                            (322.0 - (13.0 * Math.Sqrt(70.0))) / 900.0 };

        for (int i = 0; i < n; i++) {
            yield return new QuadratureNode<double>(points[i], weights[i]);
        }
    }
}
```

LinearBasis.cs

```
namespace eMP_3;

public static class LinearBasis {
    public static double Psi1(Point3D point)
        => (1 - point.X) * (1 - point.Y) * (1 - point.Z);

    public static double Psi2(Point3D point)
        => point.X * (1 - point.Y) * (1 - point.Z);

    public static double Psi3(Point3D point)
        => point.Y * (1 - point.X) * (1 - point.Z);
```

```
    public static double Psi4(Point3D point)
        => point.Z * (1 - point.X) * (1 - point.Y);

    public static double Psi5(Point3D point)
        => (1 - point.X) * point.Y * point.Z;

    public static double Psi6(Point3D point)
        => point.X * point.Y * (1 - point.Z);

    public static double Psi7(Point3D point)
        => point.X * point.Z * (1 - point.Y);

    public static double Psi8(Point3D point)
        => point.X * point.Z * point.Y;
}
```

ArrayHelper.cs

```
namespace eMP_3;

public static class ArrayHelper {
    public static T[] Copy<T>(this T[] source, T[] destination) {
        for (int i = 0; i < source.Length; i++) {
            destination[i] = source[i];
        }

        return destination;
    }

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

GridParameters.cs

```
namespace eMP_3;

public class GridParametersJsonConverter : JsonConverter {
    public override void WriteJson(JsonWriter writer, object? value, JsonSerializer
     ↪ serializer) {
        if (value is null) {
            writer.WriteNull();
            return;
        }

        var gridParameters = (GridParameters)value;

        writer.WriteStartObject();
        writer.WritePropertyName("Initial area in X");
        serializer.Serialize(writer, gridParameters.IntervalX);
        writer.WritePropertyName("Splits by X");
        writer.WriteValue(gridParameters.SplitsX);
        writer.WriteWhitespace("\n");

        writer.WritePropertyName("Initial area in Y");
```

```
        serializer.Serialize(writer, gridParameters.IntervalY);
        writer.WritePropertyName("Splits by Y");
        writer.WriteValue(gridParameters.SplitsY);
        writer.WriteWhitespace("\n");

        writer.WritePropertyName("Initial area in Z");
        serializer.Serialize(writer, gridParameters.IntervalZ);
        writer.WritePropertyName("Splits by Z");
        writer.WriteValue(gridParameters.SplitsZ);
        writer.WriteWhitespace("\n");

        writer.WriteComment("Коэффициент разрядки");
        writer.WritePropertyName("Coef");
        writer.WriteValue(gridParameters.K);

        writer.WriteComment("Коэффициенты уравнения");
        writer.WritePropertyName("Lambda");
        writer.WriteValue(gridParameters.Lambda);
        writer.WritePropertyName("Omega");
        writer.WriteValue(gridParameters.Omega);
        writer.WritePropertyName("Chi");
        writer.WriteValue(gridParameters.Chi);
        writer.WriteEndObject();
    }

    public override object? ReadJson(JsonReader reader, Type objectType, object? existingValue, JsonSerializer serializer) {
        if (reader.TokenType == JsonToken.Null || reader.TokenType != JsonToken.StartObject)
            return null;

        Interval intervalX;
        Interval intervalY;
        Interval intervalZ;
        int splitsX;
        int splitsY;
        int splitsZ;
        double? coef;
        double lambda, omega, sigma, chi;

        var maintoken = JObject.Load(reader);

        var token = maintoken["Initial area in X"];
        intervalX = serializer.Deserialize<Interval>(token!.CreateReader());
        token = maintoken["Splits by X"];
        splitsX = Convert.ToInt32(token);

        token = maintoken["Initial area in Y"];
        intervalY = serializer.Deserialize<Interval>(token!.CreateReader());
        token = maintoken["Splits by Y"];
        splitsY = Convert.ToInt32(token);

        token = maintoken["Initial area in Z"];
        intervalZ = serializer.Deserialize<Interval>(token!.CreateReader());
        token = maintoken["Splits by Z"];
        splitsZ = Convert.ToInt32(token);

        token = maintoken["Coef"];
        if (token is not null) {
            coef = double.TryParse(token.ToString(), out double res) ? res : null;
```

```
        } else {
            coef = null;
        }

        token = maintoken["Lambda"];
        lambda = Convert.ToDouble(token);

        token = maintoken["Omega"];
        omega = Convert.ToDouble(token);

        token = maintoken["Sigma"];
        sigma = Convert.ToDouble(token);

        token = maintoken["Chi"];
        chi = Convert.ToDouble(token);

        return new GridParameters(intervalX, splitsX, intervalY, splitsY, intervalZ, splitsZ, coef, lambda, omega, sigma, chi);
    }

    public override bool CanConvert(Type objectType)
        => objectType == typeof(GridParameters);
}

[JsonConverter(typeof(GridParametersJsonConverter))]
public readonly record struct GridParameters {
    public Interval IntervalX { get; init; }
    public int SplitsX { get; init; }
    public Interval IntervalY { get; init; }
    public int SplitsY { get; init; }
    public Interval IntervalZ { get; init; }
    public int SplitsZ { get; init; }
    public double? K { get; init; }
    public double Lambda { get; init; }
    public double Omega { get; init; }
    public double Sigma { get; init; }
    public double Chi { get; init; }

    public GridParameters(Interval intervalX, int splitsX, Interval intervalY, int splitsY, Interval intervalZ, int splitsZ,
                          double? k, double lambda, double omega, double sigma, double chi) {
        IntervalX = intervalX;
        SplitsX = splitsX;
        IntervalY = intervalY;
        SplitsY = splitsY;
        IntervalZ = intervalZ;
        SplitsZ = splitsZ;
        K = k;
        Lambda = lambda;
        Omega = omega;
        Sigma = sigma;
        Chi = chi;
    }

    public static GridParameters? ReadJson(string jsonPath) {
        try {
            if (!File.Exists(jsonPath))
                throw new Exception("File does not exist");

```

```
            var sr = new StreamReader(jsonPath);
            using (sr) {
                return JsonConvert.DeserializeObject<GridParameters>(sr.ReadToEnd());
            }
        } catch (Exception ex) {
            Console.WriteLine($"We had problem: {ex.Message}");
            return null;
        }
    }
}
```

GridFactory.cs

```
namespace eMP_3;

public enum GridTypes {
    Regular,
    Irregular
}

public interface IFactory {
    public Grid CreateGrid(GridTypes gridType, GridParameters gridParameters);
}

public class GridFactory : IFactory {
    public Grid CreateGrid(GridTypes gridType, GridParameters gridParameters) {
        return gridType switch {
            GridTypes.Regular => new RegularGrid(gridParameters),
            GridTypes.Irregular => new IrregularGrid(gridParameters),

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

Grid.cs

```
namespace eMP_3;

public abstract class Grid {
    public abstract ImmutableArray<Point3D> Points { get; }
    public abstract ImmutableArray<ImmutableArray<int>> Elements { get; }
    public abstract ImmutableArray<ImmutableArray<int>> Sides { get; }
    public double Lambda { get; init; }
    public double Omega { get; init; }
    public double Sigma { get; init; }
    public double Chi { get; init; }

    protected Grid(GridParameters gridParameters)
        => (Lambda, Omega, Sigma, Chi) = (gridParameters.Lambda, gridParameters.Omega, gridParameters.Sigma, gridParameters.Chi);
}
```

RegularGrid.cs

```
namespace eMP_3;

public class RegularGrid : Grid {
```

```
    private readonly Point3D[] _points = default!;
    private readonly int[][] _elements = default!;
    private readonly int[][] _sides = default!;
    public override ImmutableArray<Point3D> Points => _points.ToImmutableArray();
    public override ImmutableArray<ImmutableArray<int>> Elements => _elements.Select(item => item.ToImmutableArray()).ToImmutableArray();
    public override ImmutableArray<ImmutableArray<int>> Sides => _sides.Select(item => item.ToImmutableArray()).ToImmutableArray();

    public RegularGrid(GridParameters gridParameters) : base (gridParameters)  {
        _points = new Point3D[(gridParameters.SplitsX + 1) * (gridParameters.SplitsY + 1) * (gridParameters.SplitsZ + 1)];
        _elements = new double[gridParameters.SplitsX * gridParameters.SplitsY * gridParameters.SplitsZ].Select(_ => new int[8]).ToArray();
        _sides = new int[6][];
        _sides[0] = new int[(gridParameters.SplitsX + 1) * (gridParameters.SplitsZ + 1)]; // front
        _sides[1] = new int[(gridParameters.SplitsX + 1) * (gridParameters.SplitsZ + 1)]; // back
        _sides[2] = new int[(gridParameters.SplitsY + 1) * (gridParameters.SplitsZ + 1)]; // left
        _sides[3] = new int[(gridParameters.SplitsY + 1) * (gridParameters.SplitsZ + 1)]; // right
        _sides[4] = new int[(gridParameters.SplitsX + 1) * (gridParameters.SplitsY + 1)]; // bottom
        _sides[5] = new int[(gridParameters.SplitsX + 1) * (gridParameters.SplitsY + 1)]; // top
        Build(gridParameters);
    }

    private void Build(GridParameters gridParameters) {
        try {
            if (gridParameters.SplitsX < 1 || gridParameters.SplitsY < 1 || gridParameters.SplitsZ < 1) {
                throw new Exception("The number of splits must be greater than or equal to 1");
            }

            double hx = gridParameters.IntervalX.Lenght / gridParameters.SplitsX;
            double hy = gridParameters.IntervalY.Lenght / gridParameters.SplitsY;
            double hz = gridParameters.IntervalZ.Lenght / gridParameters.SplitsZ;

            double[] pointsX = new double[gridParameters.SplitsX + 1];
            double[] pointsY = new double[gridParameters.SplitsY + 1];
            double[] pointsZ = new double[gridParameters.SplitsZ + 1];

            pointsX[0] = gridParameters.IntervalX.LeftBorder;
            pointsY[0] = gridParameters.IntervalY.LeftBorder;
            pointsZ[0] = gridParameters.IntervalZ.LeftBorder;

            for (int i = 1; i < pointsX.Length; i++) {
                pointsX[i] = pointsX[i - 1] + hx;
            }

            for (int i = 1; i < pointsY.Length; i++) {
                pointsY[i] = pointsY[i - 1] + hy;
            }

            for (int i = 1; i < pointsZ.Length; i++) {
                pointsZ[i] = pointsZ[i - 1] + hz;
```

```
        }

        int index = 0;

        for (int k = 0; k < pointsZ.Length; k++) {
            for (int j = 0; j < pointsY.Length; j++) {
                for (int i = 0; i < pointsX.Length; i++) {
                    _points[index++] = new(pointsX[i], pointsY[j], pointsZ[k]);
                }
            }
        }

        // k по z, j по y, i по x

        int nx = pointsX.Length;
        int ny = pointsY.Length;
        int nz = pointsZ.Length;

        int Nx = pointsX.Length - 1;
        int Ny = pointsY.Length - 1;
        int Nz = pointsZ.Length - 1;

        index = 0;

        for (int k = 0; k < Nz; k++) {
            for (int j = 0; j < Ny; j++) {
                for (int i = 0; i < Nx; i++) {
                    _elements[index][0] = i + (j * nx) + (k * nx * ny);
                    _elements[index][1] = i + 1 + (j * nx) + (k * nx * ny);
                    _elements[index][2] = i + ((j + 1) * nx) + (k * nx * ny);
                    _elements[index][3] = i + 1 + ((j + 1) * nx) + (k * nx * ny);
                    _elements[index][4] = i + (j * nx) + ((k + 1) * nx * ny);
                    _elements[index][5] = i + 1 + (j * nx) + ((k + 1) * nx * ny);
                    _elements[index][6] = i + ((j + 1) * nx) + ((k + 1) * nx * ny);
                    _elements[index++][7] = i + 1 + ((j + 1) * nx) + ((k + 1) * nx * ny);
                }
            }
        }

        // front and back
        for (int k = 0; k < nz; k++) {
            for (int i = 0; i < nx; i++) {
                _sides[0][i + (k * nx)] = i + 0 + (k * nx * ny);
                _sides[1][i + (k * nx)] = i + (nx * (ny - 1)) + (k * nx * ny);
            }
        }

        // left and right
        for (int k = 0; k < nz; k++) {
            for (int j = 0; j < ny; j++) {
                _sides[2][j + (k * ny)] = 0 + (j * nx) + (k * nx * ny);
                _sides[3][j + (k * ny)] = nx - 1 + (j * nx) + (k * nx * ny);
            }
        }

        // bottom and top
        for (int j = 0; j < ny; j++) {
            for (int i = 0; i < nx; i++) {
```

```
                    _sides[4][i + (j * nx)] = i + (j * nx) + 0;
                    _sides[5][i + (j * nx)] = i + (j * nx) + (nx * ny * (nz - 1));
                }
            }

            // var res = _points.Except(_internalPoints);

            WritePoints();
        } catch (Exception ex) {
            Console.WriteLine($"We had problem: {ex.Message}");
        }
    }

    private void WritePoints() {
        var sw = new StreamWriter("points.txt");
        using (sw) {
            for (int i = 0; i < _points.Length; i++) {
                sw.WriteLine(_points[i]);
            }
        }
    }
}
```

IrregularGrid.cs

```
namespace eMP_3;

public class IrregularGrid : Grid {
    private readonly Point3D[] _points = default!;
    private readonly int[][] _elements = default!;
    private readonly int[][] _sides = default!;
    public override ImmutableArray<Point3D> Points => _points.ToImmutableArray();
    public override ImmutableArray<ImmutableArray<int>> Elements =>
    ↪ _elements.Select(item => item.ToImmutableArray()).ToImmutableArray();
    public override ImmutableArray<ImmutableArray<int>> Sides => _sides.Select(item
    ↪ => item.ToImmutableArray()).ToImmutableArray();

    public IrregularGrid(GridParameters gridParameters) : base(gridParameters) {
        _points = new Point3D[(gridParameters.SplitsX + 1) * (gridParameters.SplitsY
        ↪ + 1) * (gridParameters.SplitsZ + 1)];
        _sides = new int[6][];
        _sides[0] = new int[(gridParameters.SplitsX + 1) * (gridParameters.SplitsZ +
        ↪ 1)]; // front
        _sides[1] = new int[(gridParameters.SplitsX + 1) * (gridParameters.SplitsZ +
        ↪ 1)]; // back
        _sides[2] = new int[(gridParameters.SplitsY + 1) * (gridParameters.SplitsZ +
        ↪ 1)]; // left
        _sides[3] = new int[(gridParameters.SplitsY + 1) * (gridParameters.SplitsZ +
        ↪ 1)]; // right
        _sides[4] = new int[(gridParameters.SplitsX + 1) * (gridParameters.SplitsY +
        ↪ 1)]; // bottom
        _sides[5] = new int[(gridParameters.SplitsX + 1) * (gridParameters.SplitsY +
        ↪ 1)]; // top
        Build(gridParameters);
    }

    private void Build(GridParameters gridParameters) {
        try {
            if (gridParameters.SplitsX < 1 || gridParameters.SplitsY < 1 ||
            ↪ gridParameters.SplitsZ < 1) {
```

```
                throw new Exception("The number of splits must be greater than or
                ↪  equal to 1");
            }

            ArgumentNullException.ThrowIfNull(gridParameters.K,
            ↪  $"{nameof(gridParameters.K)} cannot be null");

            if (gridParameters.K <= 0) {
                throw new Exception("The coefficient must be greater than 0");
            }

            double[] pointsX = new double[gridParameters.SplitsX + 1];
            double[] pointsY = new double[gridParameters.SplitsY + 1];
            double[] pointsZ = new double[gridParameters.SplitsZ + 1];

            pointsX[0] = gridParameters.IntervalX.LeftBorder;
            pointsY[0] = gridParameters.IntervalY.LeftBorder;
            pointsZ[0] = gridParameters.IntervalZ.LeftBorder;

            double hx, hy, hz;
            double sum = 0.0;

            for (int k = 0; k < gridParameters.SplitsX; k++) {
                sum += Math.Pow(gridParameters.K.Value, k);
            }

            hx = gridParameters.IntervalX.Lenght / sum;
            sum = 0.0;

            for (int k = 0; k < gridParameters.SplitsY; k++) {
                sum += Math.Pow(gridParameters.K.Value, k);
            }

            hy = gridParameters.IntervalY.Lenght / sum;
            sum = 0.0;

            for (int k = 0; k < gridParameters.SplitsZ; k++) {
                sum += Math.Pow(gridParameters.K.Value, k);
            }

            hz = gridParameters.IntervalZ.Lenght / sum;

            for (int i = 1; i < pointsX.Length; i++) {
                pointsX[i] = pointsX[i - 1] + hx;
            }

            for (int i = 1; i < pointsY.Length; i++) {
                pointsY[i] = pointsY[i - 1] + hy;
            }

            for (int i = 1; i < pointsZ.Length; i++) {
                pointsZ[i] = pointsZ[i - 1] + hz;
            }

            int index = 0;

            for (int i = 0; i < pointsZ.Length; i++) {
                for (int j = 0; j < pointsY.Length; j++) {
                    for (int k = 0; k < pointsX.Length; k++) {
                        _points[index++] = new(pointsX[k], pointsY[j], pointsZ[i]);
```

```
                    }
                }
            }

            // k по z, j по y, i по x

            int nx = pointsX.Length;
            int ny = pointsY.Length;
            int nz = pointsZ.Length;

            int Nx = pointsX.Length - 1;
            int Ny = pointsY.Length - 1;
            int Nz = pointsZ.Length - 1;

            index = 0;

            for (int k = 0; k < Nz; k++) {
                for (int j = 0; j < Ny; j++) {
                    for (int i = 0; i < Nx; i++) {
                        _elements[index][0] = i + (j * nx) + (k * nx * ny);
                        _elements[index][1] = i + 1 + (j * nx) + (k * nx * ny);
                        _elements[index][2] = i + ((j + 1) * nx) + (k * nx * ny);
                        _elements[index][3] = i + 1 + ((j + 1) * nx) + (k * nx * ny);
                        _elements[index][4] = i + (j * nx) + ((k + 1) * nx * ny);
                        _elements[index][5] = i + 1 + (j * nx) + ((k + 1) * nx * ny);
                        _elements[index][6] = i + ((j + 1) * nx) + ((k + 1) * nx *
                        ↪  ny);
                        _elements[index++][7] = i + 1 + ((j + 1) * nx) + ((k + 1) *
                        ↪  nx * ny);
                    }
                }
            }

            // front and back
            for (int k = 0; k < nz; k++) {
                for (int i = 0; i < nx; i++) {
                    _sides[0][i + (k * nx)] = i + 0 + (k * nx * ny);
                    _sides[1][i + (k * nx)] = i + (nx * (ny - 1)) + (k * nx * ny);
                }
            }

            // left and right
            for (int k = 0; k < nz; k++) {
                for (int j = 0; j < ny; j++) {
                    _sides[2][j + (k * ny)] = 0 + (j * nx) + (k * nx * ny);
                    _sides[3][j + (k * ny)] = nx - 1 + (j * nx) + (k * nx * ny);
                }
            }

            // bottom and top
            for (int j = 0; j < ny; j++) {
                for (int i = 0; i < nx; i++) {
                    _sides[4][i + (j * nx)] = i + (j * nx) + 0;
                    _sides[5][i + (j * nx)] = i + (j * nx) + (nx * ny * (nz - 1));
                }
            }

            WritePoints();
        } catch (Exception ex) {
            Console.WriteLine($"We had problem: {ex.Message}");
```

```
        }
    }

    private void WritePoints() {
        var sw = new StreamWriter("points.txt");
        using (sw) {
            for (int i = 0; i < _points.Length; i++) {
                sw.WriteLine(_points[i]);
            }
        }
    }
}
```

Point3D.cs

```
namespace eMP_3;

public readonly record struct Point3D(double X, double Y, double Z) {
    public static Point3D operator +(Point3D point, (double, double, double) value)
        => new(point.X + value.Item1, point.Y + value.Item2, point.Z + value.Item3);

    public static Point3D operator -(Point3D point, (double, double, double) value)
        => new(point.X - value.Item1, point.Y - value.Item2, point.Z - value.Item3);

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

Tests.cs

```
namespace eMP_3;

public interface ITest {
    public double Us(Point3D point);

    public double Fs(Point3D point);

    public double Uc(Point3D point);

    public double Fc(Point3D point);
}

public class Test1 : ITest { // все коэффициенты равны 1
    public double Fc(Point3D point)
       => -point.X + point.Y;

    public double Fs(Point3D point)
        => (-3 * point.X) - (3 * point.Y) - (2 * point.Z);

    public double Us(Point3D point)
        => point.X + (2 * point.Y) + point.Z;

    public double Uc(Point3D point)
        => (2 * point.X) + point.Y + point.Z;
}

public class Test2 : ITest { // λ =, ω =, χ =
    public double Fc(Point3D point)
      => 0;
```

```

    public double Fs(Point3D point)
        => 0;

    public double Us(Point3D point)
        => point.X * point.Y * point.Z;

    public double Uc(Point3D point)
        => 2 * point.X * 2 * point.Y * 2 * point.Z;
}
```