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

Практическая работа №2
по дисциплине «Метод конечных элементов»

Решение линейных и нелинейных задач магнитостатики. Построение интерполяционных и сглаживающих сплайнов

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

Новосибирск, 2023

Цель работы

Разработать программу решения линейной и нелинейной задачи магнитостатики методом конечных элементов с использованием билинейных базисных функций на прямоугольниках. Провести сравнение полученных численных результатов с результатами, полученными при решении задачи в готовом конечноэлементном пакете.

Разработать программу построения сглаживающего сплайна для одномерной задачи.

Теоретическая часть

Постановка задачи

$$-\operatorname{div}\left(\frac{1}{\mu}\operatorname{grad} A_z\right) = J_z,$$

$$A_z\big|_{S_1} = 0,\ \left.\frac{\partial A_z}{\partial n}\right|_{S_2} = 0.$$

Задача магнитостатики является линейной, если коэффициент магнитной проницаемости μ зависит только от координат x и y, и нелинейной, если μ зависит от вектора индукции $\vec{B}$ магнитного поля.

Основные положения

Для решения краевой задачи в области Ω, используется сетка с прямоугольными элементами $\Omega_m = [x_p, x_{p+1}] \times [y_s, y_{s+1}]$, где ячейки сетки строятся в виде декартового произведения независимых друг от друга одномерных сеток $\{x_1, \ldots, x_n\}, \{y_1, \ldots, y_m\}$, причем узлы по x и y - координатам расположены так, что они точно попадают в границы прямоугольных подобластей Ω_i. Таким образом, прямоугольная сетка задается в виде двух одномерных массивов узлов по x и y - координатам.

Граница S_1, на которой заданы однородные краевые условия первого рода, задаются набором номером узлов, лежащих на S_1. На границе S_2 заданы однородные краевые условия второго рода, которые дают нулевые влкады в вектор правой части, поэтому данную границу можно не задавать.

Локальные базисные функции $\hat{\psi}_i$ на каждом элементе Ω_m являются билинейными, определенными через одномерные линейные функции:

$$X_1(x) = \frac{x_{p+1} - x}{h_x},\ X_2(x) = \frac{x - x_p}{h_x},\ h_x = x_{p+1} - x_p,$$

$$Y_1(y) = \frac{y_{s+1} - y}{h_y},\ Y_2(y) = \frac{y - y_s}{h_y},\ h_y = y_{s+1} - y_s.$$

Локальные базисные функции на конечном элементе Ω_m представляются в виде произведения одномерных функций:

$$\hat{\psi}_1(x,y) = X_1(x)Y_1(y),\ \hat{\psi}_2(x,y) = X_2(x)Y_1(y),$$
$$\hat{\psi}_3(x,y) = X_1(x)Y_2(y),\ \hat{\psi}_4(x,y) = X_2(x)Y_2(y).$$

Линейная задача

Компоненты локальной матрицы жесткости $\hat{G}$ и локального вектора правой части $\hat{b}$ конечного элемента Ω_m вычисляются как

$$\hat{G}_{ij} = \int\limits_{x_p}^{x_{p+1}} \int\limits_{y_s}^{y_{s+1}} \overline{\lambda} \left(\frac{\partial \hat{\psi}_i}{\partial x} \frac{\partial \hat{\psi}_j}{\partial x} + \frac{\partial \hat{\psi}_i}{\partial y} \frac{\partial \hat{\psi}_j}{\partial y} \right) dxdy,$$

$$\hat{b}_i = \int\limits_{x_p}^{x_{p+1}} \int\limits_{y_s}^{y_{s+1}} J_z \hat{\psi}_i \, dxdy,$$

где $\overline{\lambda}$ постоянное на конечном элементе Ω_m значение $\overline{\lambda} = \dfrac{1}{\mu}$, где μ – магнитная проницаемость, коэффициент J_z постоянен на конечном элементе Ω_m.

Однородные краевые условия первого рода учитываются в глобальной матрице следующим образом: диагональный элемент a_{ii}, где i – номер узла, лежащего на границе S_1, заменяется на некоторое очень большое число $\mathbb{C}$, а i-я компонента вектора правой части заменяется значением 0. Однородные краевые условия второго рода дают нулевые вклады в вектор правой части.

Нелинейная задача

В нелинейной задаче коэффициент магнитной индукции μ, соответствующий материалу "железо"зависит от вектора индукции $\vec{B}$ магнитного поля, точнее от модуля этого вектора $B = \sqrt{B_x^2 + B_y^2}$. Поэтому в результате конечноэлементной аппроксимации необходимо решать систему алгебраических уравнений вида:

$$A(q)q = b,$$

где компоненты матрицы жесткости определяются

$$G_{ij} = \int\limits_{\Omega} \frac{1}{\mu(B)} \operatorname{grad} \psi_i \operatorname{grad} \psi_j d\Omega.$$

Нелинейную задачу необходимо решить методом простой итерации, который заключается в последовательности решении линейных задач

$$A(q^{k-1})q^k = b,$$

где $A(q^{k-1}$ матрица, вычисленная на k-ой итерации с помощью вектора q^{k-1} решения, полученного на $k-1$-ой итерации по нелинейности. Начальное приближение – вектор q^0 – вектор весов, полученных при решении линейной задачи.

На k-ом шаге по нелинейности решается линейная задача с локальной матрицей жесткости $\hat{G}$ и вектором правой части $\hat{b}$, где коэффициент $\overline{\lambda} = \dfrac{1}{\mu}$ будет зависеть от модуля вектора магнитной индукции $\vec{B}$. Зная вектор весов q^{k-1} разложения A_z по базисным функциям, необходимо вычислить $\vec{B}$ и его модуль B следующим образом:

$$\vec{B} = rot\vec{A} = \left(\frac{\partial A_z}{\partial y}, -\frac{\partial A_z}{\partial x}, 0 \right),$$

$$B = \sqrt{\left(\frac{\partial A_z}{\partial y}\right)^2 + \left(\frac{\partial A_z}{\partial x}\right)^2}.$$

Обычно зависимость μ или $\frac{1}{\mu}$ от B задается таблично. Поэтому в процессе решения нелинейной задачи магнитостатики на каждой итерации по нелинейности при вычислении μ или $\frac{1}{\mu}$ в любой точке необходимо использовать кубический сплайн – интерполяционный или сглаживающий. По этому сплайну для любого значения B можно вычислить μ или $\frac{1}{\mu}$.

Для вычисления значения $\frac{1}{\mu}$ по B для $B > B_n$, где B_n – последнее табличное значение, используют следующее соотношение

$$\frac{1}{\mu} = \frac{B_n}{B}\left(\frac{1}{\mu_n} - 1\right) + 1,$$

где μ_n – значение, соответствующее B_n.

Для ускорения сходимости процесса решения нелинейной задачи используется параметр релаксации ω^k. В этом случае каждое последующее приближение строится как

$$q^k = \omega^k \overline{q}^k + (1 - \omega^k) q^{k-1},$$

где ω^k – коэффициент релаксации, а $\overline{q}^k$ – решение системы.

Выход из итерационного процесса осуществляется либо по достижении максимального числа итераций ("аварийный"выход), либо при выполнении условия

$$\frac{\|A(q^k)q^k - b\|}{\|b\|} < \varepsilon,$$

где ε – некоторое малое число, выбранное как требуемая точность решения нелинейной задачи.

Построение кубического сглаживающего сплайна

Под построением сглаживающего сплайна подразумевается задача сглаживания, под которой понимается построение достаточно гладкой функции $S(x)$, значения которой в точках интерполяции максимально близки к значениями исходной функции.

Метод наименьших квадратов

Данную задачу удобно решать на основе метода наименьших квадратов:

$$\sum_{i=0}^{m} (S(x) - f_i)^2 \to min.$$

Разобьем область определения на $m \ll n$ конечнных элементов и выпишем сплайн $S(x)$ с использованием эрмитовых базисных функций $\psi_i(x)$ на этих элементах в виде

$$S(x) = \sum_{i=0}^{m} [\tilde{f}_i \cdot \psi_{2i}(x) + \tilde{f}'_i \cdot \psi_{2i+1}(x)] = \sum_{i=0}^{2m} q_i \cdot \psi_i(x),$$

причем коэффициенты q_i находятся из условия минимизации функционала

$$\Phi(q) = \sum_{k=0}^{n} \omega_k \cdot \left(f_k - \sum_{i=0}^{2m} q_i \cdot \psi_i(x_k) \right)^2,$$

где ω_j – это заданные веса, определяющие меру близости сплайна $S(x)$ к значениям функции $f(x)$ в узлах x_i. В результате подстановки и минимизации, можно получить следующую СЛАУ размерности $2m$:

$$M \cdot q = b,$$

где компоненты матрицы M и вектора b рассчитываются по следующим формулам:

$$m_{i,j} = \sum_{k=1}^{n} \omega_k \cdot \psi_i(x_k) \cdot \psi_j(x_k),$$

$$f_i = \sum_{k=1}^{n} \omega_k \cdot f_k \cdot \psi_i(x_k).$$

Матрица M в некоторых случаях может оказаться вырожденной, что означает существование неединственного сплайна с минимальной суммой квадратов отклонений в узлах x_k. Это может произойти, когда в интервал $[\tilde{x}_i,\ \tilde{x}_{i+1}]$ попадает недостаточное количество узлов x_k. В этом случае можно изменить либо сглаживающую сетку, либо функционал $\Phi(q)$ путем добавления регуляризирующих слагаемых $\Phi^{\alpha}(q)$ и $\Phi^{\beta}(q)$:

$$\bar{\Phi}(q) = \sum_{k=0}^{n} \omega_k \cdot (S(x_k) - f_k)^2 + \Phi^{\alpha}(q) + \Phi^{\beta}(q),$$

где:

$$\Phi^{\alpha}(q) = \int_{\tilde{x}_0}^{\tilde{x}_m} \alpha \cdot \left(\frac{\partial S(x)}{\partial x} \right)^2 dx,$$

$$\Phi^{\beta}(q) = \int_{\tilde{x}_0}^{\tilde{x}_m} \beta \cdot \left(\frac{\partial^2 S(x)}{\partial x^2} \right)^2 dx.$$

Минимизация функционала $\bar{\Phi}(q)$ при $\alpha > 0$ и $\beta > 0$ определяет единственность сплайна $S(x)$. При этом увеличение коэффициента β уменьшает вторую производную сплайна, приближая его к линейной функции. Аналогично, увеличение коэффициента α уменьшает первую производную сплайна, приближая его к константной функции.
При переходе от $\Phi(q)$ к $\bar{\Phi}(q)$ вектор правой части не изменится, в отличие от компонент матрицы СЛАУ, которые перерассчитываются с учетом добавок $m^{\alpha}_{i,j}$ и $m^{\beta}_{i,j}$:

$$m_{i,j} = \sum_{k=0}^{n} \omega_k \cdot \psi_i(x_k) \cdot \psi_j(x_k) + m^{\alpha}_{i,j} + m^{\beta}_{i,j},$$

где:

$$m^{\alpha}_{i,j} = \int_{\tilde{x}_0}^{\tilde{x}_m} \alpha \cdot \frac{\partial \psi_i(x)}{\partial x} \cdot \frac{\partial \psi_j(x)}{\partial x}\, dx,$$

$$m_{i,j}^{\alpha} = \int\limits_{\tilde{x}_0}^{\tilde{x}_m} \alpha \cdot \frac{\partial^2 \psi_i(x)}{\partial x^2} \cdot \frac{\partial^2 \psi_j(x)}{\partial x^2}\, dx.$$

Расчет коэффициентов $m_{i,j}^{\alpha}$ и $m_{i,j}^{\beta}$ удобно организовать через интегрирование локальных эрмитовых базисных функций $\phi_i\left(\xi(x)\right)$ на элементах $[\tilde{x}_i, \tilde{x}_{i+1}]$ длиной h:

$$\hat{m}_{i,j}^{\alpha} = \int\limits_{\tilde{x}_i}^{\tilde{x}_{i+1}} \alpha \cdot \frac{\partial \phi_i(x)}{\partial x} \cdot \frac{\partial \phi_j(x)}{\partial x}\, dx,$$

$$\hat{m}_{i,j}^{\beta} = \int\limits_{\tilde{x}_i}^{\tilde{x}_{i+1}} \beta \cdot \frac{\partial^2 \phi_i(x)}{\partial x^2} \cdot \frac{\partial^2 \phi_j(x)}{\partial x^2}\, dx.$$

Эрмитовы базисные функции

Введем следующую замену для преобразования исходного интервала $[x_i, x_{i+1}]$ длиной h_i в шаблонный $[0, 1]$:

$$\xi(x) = \frac{x - x_i}{h_i}.$$

Зададим локальные эрмитовы базисные функции функции $\phi_i(\xi)$ следующим образом:

$$\begin{aligned}
\phi_0(\xi) &= \psi_{2i+0}(\xi) = 1 - 3 \cdot \xi^2 + 2 \cdot \xi^3, \\
\phi_1(\xi) &= \psi_{2i+1}(\xi) = h_i \cdot (\xi - 2 \cdot \xi^2 + \xi^3), \\
\phi_2(\xi) &= \psi_{2i+2}(\xi) = 3 \cdot \xi^2 - 2 \cdot \xi^3, \\
\phi_3(\xi) &= \psi_{2i+3}(\xi) = h_i \cdot (-\xi^2 + \xi^3).
\end{aligned}$$

Практическая часть

1. Выполнить конечноэлементную аппроксимацию исходного уравнения при решении линейной задачи на прямоугольных конечных элементах.
2. Разработать программа генерации прямоугольной сетки для дискретизации расчетной области, сборки матрицы и вектора правой части СЛАУ при решении линейной задачи на прямоугольной сетке с учетом следующих требований:
    - прямоугольная сетка должна быть сохранена в двух форматах: первый – два одномерных массива узлов по x и y - координатам и размерность данных массивов, второй – хранение элементов четырьмя узлами, хранение всех узлов, хранение для каждого элемента номера в каталоге;
    - портрет матрицы СЛАУ в разреженном строчно-столбцовом формате должен быть сгенерирован оптимальным образом;
    - для решения СЛАУ использовать метод сопряженных градиентов или локально-оптимальную схему с предобуславливанием Холесского.
3. Разработать программу выдачи модуля B вектора магнитной индукции $\vec{B}$ на элементе.
4. Провести сравнение полученных численных результатов решения линейной задачи с результатами, полученными в предыдущей лабораторной работе.
5. Реализовать программу решения нелинейной задачи магнитостатики методом простой итерации.
6. Провести расчеты нелинейной задачи при изменении плотности тока J в 10, 10^2 и 10^3 раз.
7. Провести сравнение полученных численных результатов решения нелинейной задачи с результатами, полученными в предыдущей лабораторной работе.

Тестирование линейной задачи

Расчетная область

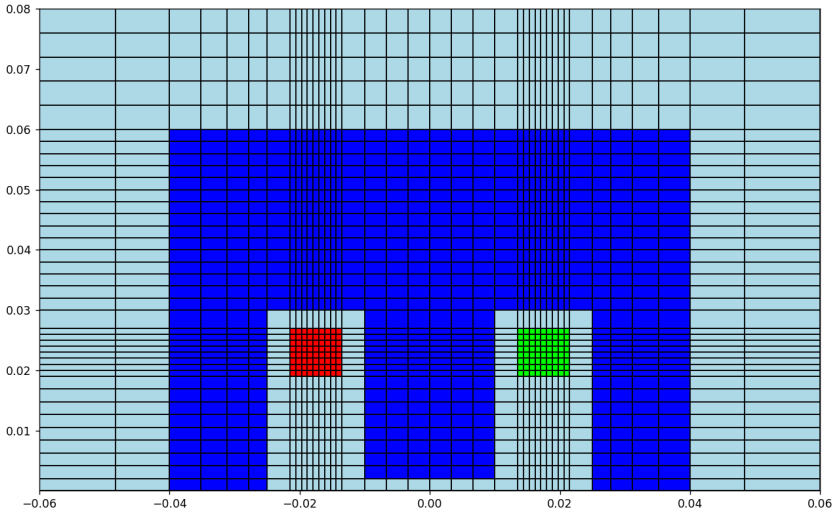

Рис. 1: Расчетная область в программе

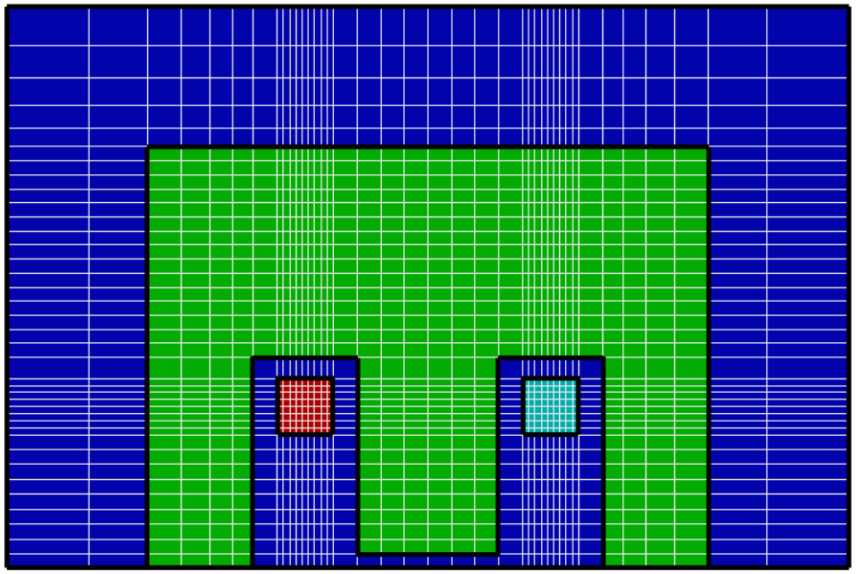

Рис. 2: Расчетная область в TELMA

```
-6e-2 -4e-2 -2.5e-2 -2.15e-2 -1.35e-2 -1e-2 1e-2 1.35e-2
 ↪  2.15e-2 2.5e-2 4e-2 6e-2
0.00001 0.2e-2 1.9e-2 2.7e-2 3e-2 6e-2 8e-2
2 4 1 9 1 6 1 9 1 4 2
1 8 8 1 15 5
-1.4 -1.2 -1. -1. 1. 1. 1. 1. 1. 1.2 1.4
-1. 1. 1. 1. 1. 1.
0 0
0 1. 0. 0 11 0 6
1 1000. 0. 1 2 0 4
1 1000. 0. 1 10 4 5
1 1000. 0. 9 10 0 4
1 1000. 0. 5 6 1 4
2 1.0 1eб 3 4 2 3
3 1.0 -1eб 7 8 2 3
```

Рис. 3: Входной файл среды в программе

```
7
-6e-2 6e-2 0.00001 8e-2 1. 0. 1
-4e-2 -2.5e-2 0.00001 3e-2 1000. 0. 2
-4e-2 4e-2 3e-2 6e-2 1000. 0. 2
2.5e-2 4e-2 0.00001 3e-2 1000. 0. 2
-1e-2 1.0e-2 0.2e-2 3e-2 1000. 0. 2
1.35e-2 2.15e-2 1.9e-2 2.7e-2 1.0 -1eб 3
-2.15e-2 -1.35e-2 1.9e-2 2.7e-2 1.0 1eб 4
-6e-2 11
-4e-2 -2.5e-2 -2.15e-2 -1.35e-2 -1e-2 1e-2 1.35e-2 2.15e-2
 ↪  2.5e-2 4e-2 6e-2
0.01 0.003 0.002 0.0008 0.002 0.003 0.002 0.0008 0.002
 ↪  0.003 0.01
1.4 1.2 1. 1. 1. 1. 1. 1. 1. 1.2 1.4
-1 -1 -1 -1 1 1 1 1 1 1 1
0.00001 6
0.2e-2 1.9e-2 2.7e-2 3e-2 6e-2 8e-2
0.003 0.002 0.001 0.002 0.002 0.003
1.2 1. 1. 1. 1. 1.2
-1 1 1 1 1 1
0 0
```

Рис. 4: Входной файл среды в TELMA

Сравнение результатов

x	y	z	A_z^T	A_z	$\frac{\lvert A_z^T - A_z\rvert}{A_z^T}$	$\lvert B\rvert^T$	$\lvert B\rvert$	$\frac{\lvert\lvert B\rvert^T - \lvert B\rvert\rvert}{\lvert B\rvert^T}$
$-7.8 \cdot 10^{-3}$	$1.6 \cdot 10^{-3}$	0	0.000302281710	0.000302281711	$3.42 \cdot 10^{-9}$	0.0368431690	0.0388904106	$5.26 \cdot 10^{-2}$
$-3.7 \cdot 10^{-3}$	$1.7 \cdot 10^{-3}$	0	0.000142982770	0.000142982771	$5.85 \cdot 10^{-9}$	0.0390201230	0.0386290750	$1.01 \cdot 10^{-2}$
$-1.3 \cdot 10^{-3}$	$1.9 \cdot 10^{-3}$	0	0.000050323491	0.000050323491	$8.07 \cdot 10^{-9}$	0.0385401930	0.0386268966	$2.24 \cdot 10^{-3}$
$4 \cdot 10^{-4}$	$1.5 \cdot 10^{-3}$	0	-0.000154461590	-0.000154461593	$2.09 \cdot 10^{-8}$	0.0389100770	0.0386290750	$7.27 \cdot 10^{-3}$
$8.2 \cdot 10^{-3}$	$4 \cdot 10^{-4}$	0	-0.000337283020	-0.000337283022	$5.78 \cdot 10^{-9}$	0.0379676180	0.0388904106	$2.37 \cdot 10^{-2}$

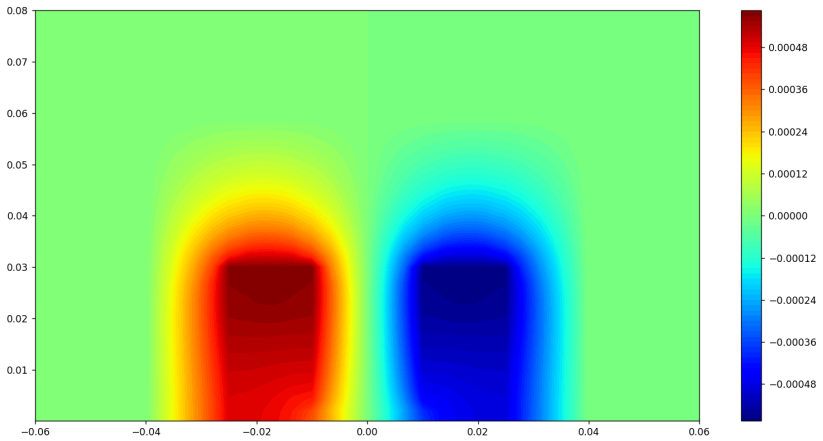

Рис. 5: Решение линейной задачи в программе

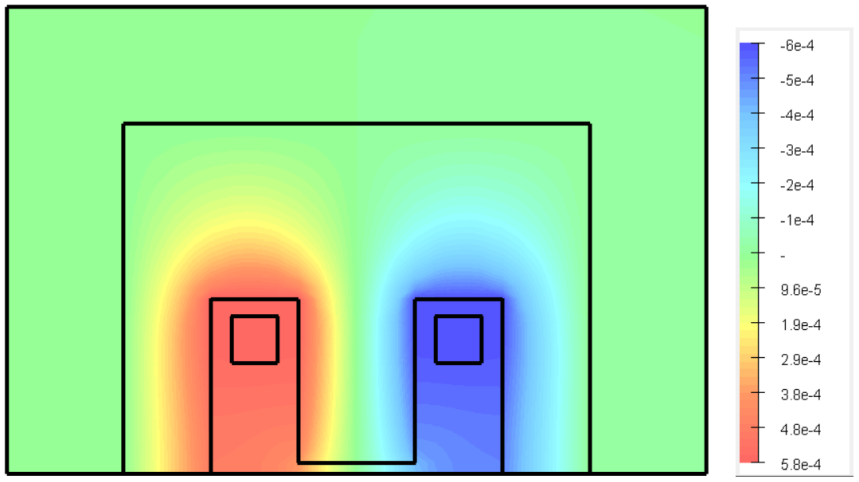

Рис. 6: Решение линейной задаче в TELMA

A_z^T – это значения потенциала в готовом конечноэлементном пакете TELMA, A_z – значения, полученные в программе. Аналогичное именование переменных с модулем индукции $|B|$.

Тестирование нелинейной задачи

График сглаживающего сплайна

Рис. 7: Зависимость $\mu(B)$

Сравнение результатов

- $J = 10^{11}$

x	y	z	A_z^T	A_z	$\frac{\lvert A_z^T - A_z\rvert}{A_z^T}$	$\lvert B\rvert^T$	$\lvert B\rvert$	$\frac{\lvert\lvert B\rvert^T - \lvert B\rvert\rvert}{\lvert B\rvert^T}$
$-7.8 \cdot 10^{-3}$	$1.6 \cdot 10^{-3}$	0	0.669749860000	0.669748471554	$2.07 \cdot 10^{-6}$	75.6693111100	75.9797056720	$4.1 \cdot 10^{-3}$
$-3.7 \cdot 10^{-3}$	$1.7 \cdot 10^{-3}$	0	0.330647490000	0.330646813826	$2.04 \cdot 10^{-6}$	84.7532571700	85.0363463261	$3.34 \cdot 10^{-3}$
$-1.3 \cdot 10^{-3}$	$1.9 \cdot 10^{-3}$	0	0.116862020000	0.116861779759	$2.06 \cdot 10^{-6}$	89.1715620200	89.5340469739	$4.07 \cdot 10^{-3}$
$4 \cdot 10^{-4}$	$1.5 \cdot 10^{-3}$	0	−0.355910100000	−0.355909372553	$2.04 \cdot 10^{-6}$	84.8354308300	85.0363463261	$2.37 \cdot 10^{-3}$
$8.2 \cdot 10^{-3}$	$4 \cdot 10^{-4}$	0	−0.737217720000	−0.737216180497	$2.09 \cdot 10^{-6}$	75.8395788800	75.9797056720	$1.85 \cdot 10^{-3}$

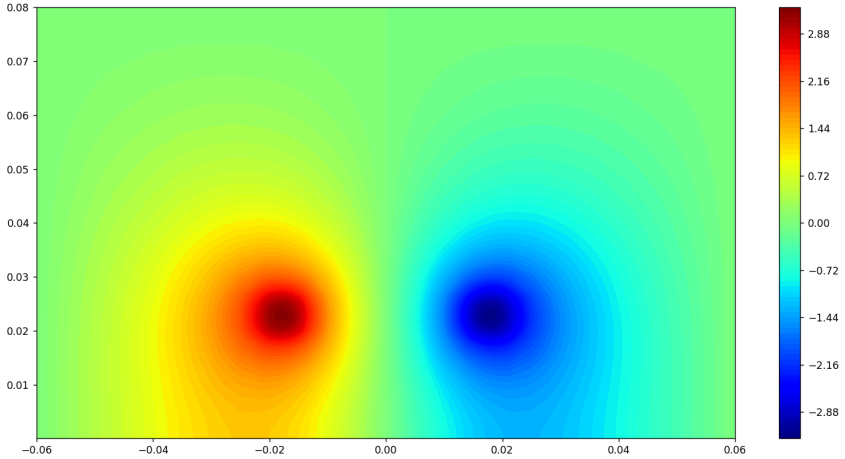

Рис. 8: Решение нелинейной задачи в программе при $J = 10^{11}$

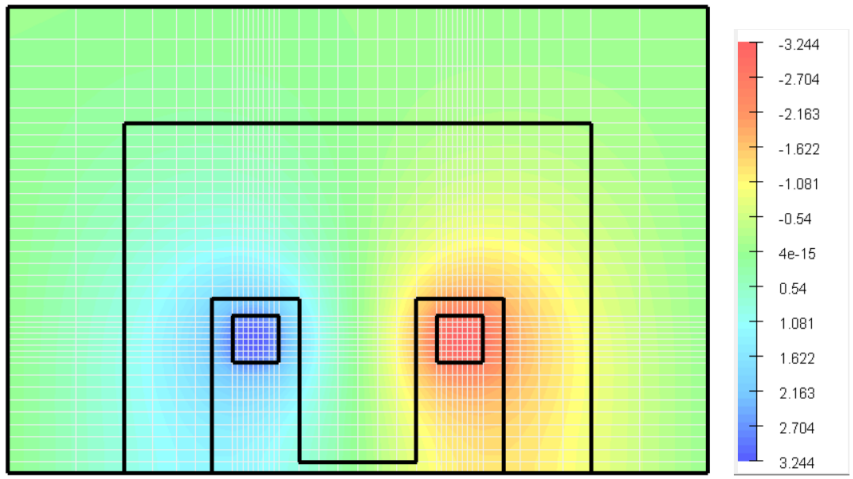

Рис. 9: Решение нелинейной задачи в TELMA при $J = 10^{11}$

Расчетная область взята с тестирования линейной задачи, так как в программе задается количество разбиений интервала, а не предварительный шаг как в конечноэлементном пакете TELMA, поэтому проблематично задать идентичные сетки, сделанные в прошлой лабораторной работе.

- $J = 10^{12}$

x	y	z	A_z^T	A_z	$\frac{\lvert A_z^T - A_z\rvert}{A_z^T}$	$\lvert B\rvert^T$	$\lvert B\rvert$	$\frac{\lvert\lvert B\rvert^T - \lvert B\rvert\rvert}{\lvert B\rvert^T}$
$-7.8 \cdot 10^{-3}$	$1.6 \cdot 10^{-3}$	0	6.562283600000	6.562269475923	$2.15 \cdot 10^{-6}$	741.2598552000	743.6463955337	$3.22 \cdot 10^{-3}$
$-3.7 \cdot 10^{-3}$	$1.7 \cdot 10^{-3}$	0	3.241772300000	3.241765392350	$2.13 \cdot 10^{-6}$	830.0346454000	832.9590397677	$3.52 \cdot 10^{-3}$
$-1.3 \cdot 10^{-3}$	$1.9 \cdot 10^{-3}$	0	1.145866300000	1.145863887357	$2.11 \cdot 10^{-6}$	874.2158003000	877.8526546222	$4.16 \cdot 10^{-3}$
$4 \cdot 10^{-4}$	$1.5 \cdot 10^{-3}$	0	−3.489190100000	−3.489182741073	$2.11 \cdot 10^{-6}$	830.8571692000	832.9590397677	$2.53 \cdot 10^{-3}$
$8.2 \cdot 10^{-3}$	$4 \cdot 10^{-4}$	0	−7.222531300000	−7.222515682423	$2.16 \cdot 10^{-6}$	742.5145405000	743.6463955337	$1.52 \cdot 10^{-3}$

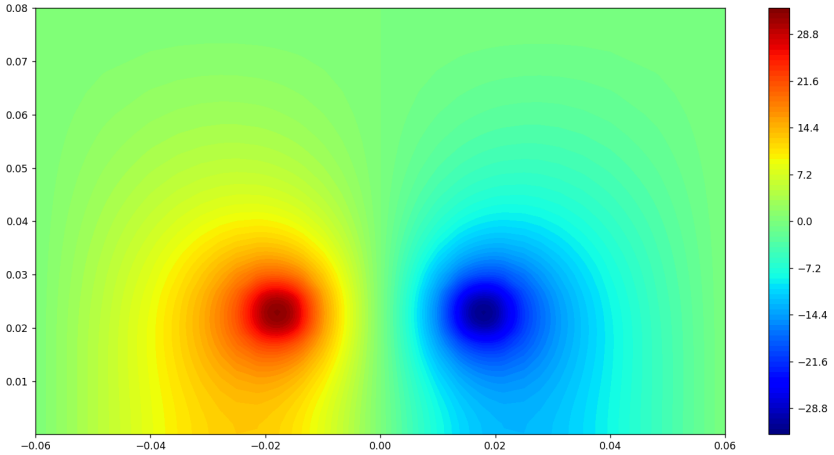

Рис. 10: Решение нелинейной задачи в программе при $J = 10^{12}$

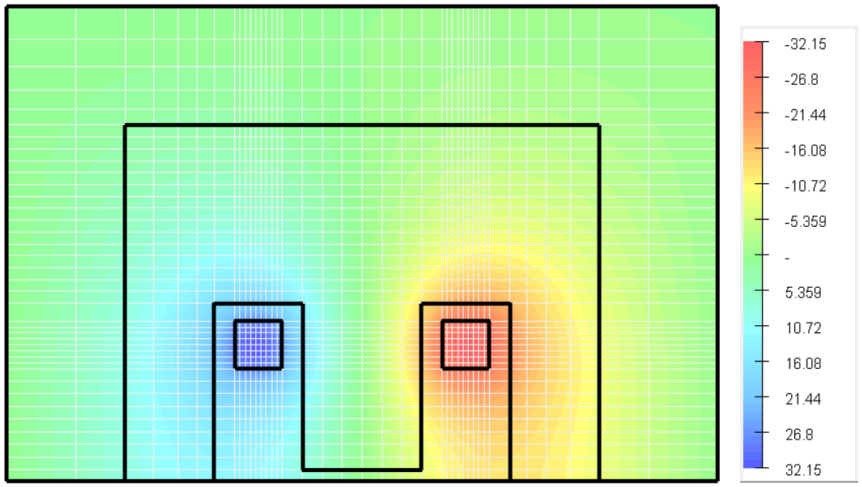

Рис. 11: Решение нелинейной задачи в TELMA при $J = 10^{12}$

- $J = 10^{13}$

x	y	z	A_z^T	A_z	$\frac{\lvert A_z^T - A_z\rvert}{A_z^T}$	$\lvert B\rvert^T$	$\lvert B\rvert$	$\frac{\lvert\lvert B\rvert^T - \lvert B\rvert\rvert}{\lvert B\rvert^T}$
$-7.8 \cdot 10^{-3}$	$1.6 \cdot 10^{-3}$	0	65.487695000000	65.487553521534	$2.16 \cdot 10^{-6}$	7,396.5664030000	7,420.3182597975	$3.21 \cdot 10^{-3}$
$-3.7 \cdot 10^{-3}$	$1.7 \cdot 10^{-3}$	0	32.353065000000	32.352996633609	$2.11 \cdot 10^{-6}$	8,283.3492170000	8,312.1947137964	$3.48 \cdot 10^{-3}$
$-1.3 \cdot 10^{-3}$	$1.9 \cdot 10^{-3}$	0	11.435926000000	11.435901505432	$2.14 \cdot 10^{-6}$	8,725.1581910000	8,761.0511799597	$4.11 \cdot 10^{-3}$
$4 \cdot 10^{-4}$	$1.5 \cdot 10^{-3}$	0	−34.822038000000	−34.821964264976	$2.12 \cdot 10^{-6}$	8,291.5729660000	8,312.1947137964	$2.49 \cdot 10^{-3}$
$8.2 \cdot 10^{-3}$	$4 \cdot 10^{-4}$	0	−72.075743000000	−72.075587144077	$2.16 \cdot 10^{-6}$	7,410.0636540000	7,420.3182597974	$1.38 \cdot 10^{-3}$

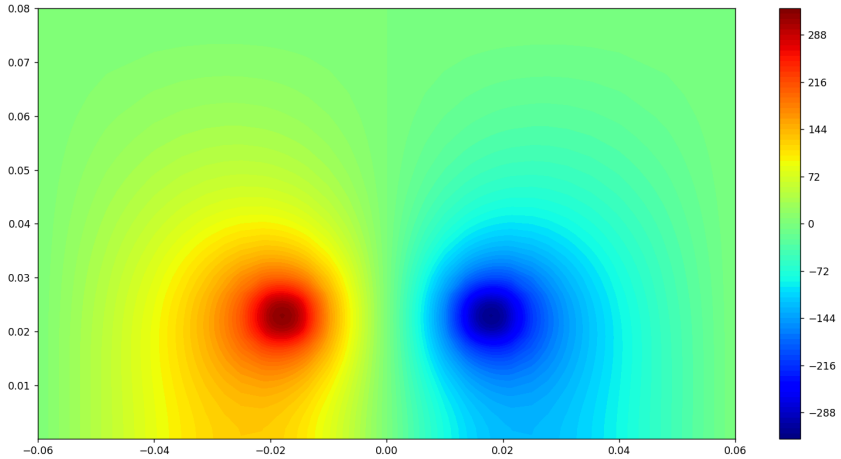

Рис. 12: Решение нелинейной задачи в программе при $J = 10^{13}$

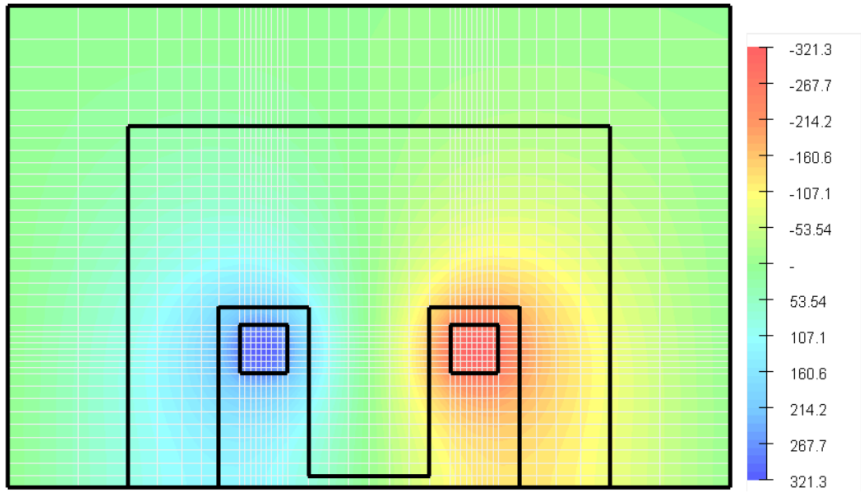

Рис. 13: Решение нелинейной задачи в TELMA при $J = 10^{13}$

- $J = 10^{14}$

x	y	z	A_z^T	A_z	$\frac{\lvert A_z^T - A_z\rvert}{A_z^T}$	$\lvert B\rvert^T$	$\lvert B\rvert$	$\frac{\lvert\lvert B\rvert^T - \lvert B\rvert\rvert}{\lvert B\rvert^T}$
$-7.8 \cdot 10^{-3}$	$1.6 \cdot 10^{-3}$	0	654.741810000000	654.740401773940	$2.15 \cdot 10^{-6}$	73,955.6307900000	74,187.0374404319	$3.13 \cdot 10^{-3}$
$-3.7 \cdot 10^{-3}$	$1.7 \cdot 10^{-3}$	0	323.466000000000	323.465313789554	$2.12 \cdot 10^{-6}$	82,813.4954300000	83,104.5523807320	$3.51 \cdot 10^{-3}$
$-1.3 \cdot 10^{-3}$	$1.9 \cdot 10^{-3}$	0	114.336520000000	114.336279409746	$2.1 \cdot 10^{-6}$	87,231.5822700000	87,593.0377318163	$4.14 \cdot 10^{-3}$
$4 \cdot 10^{-4}$	$1.5 \cdot 10^{-3}$	0	-348.150530000000	-348.149784501621	$2.14 \cdot 10^{-6}$	82,895.7334500000	83,104.5523807318	$2.52 \cdot 10^{-3}$
$8.2 \cdot 10^{-3}$	$4 \cdot 10^{-4}$	0	-720.607870000000	-720.606309832396	$2.17 \cdot 10^{-6}$	74,080.5650400000	74,187.0374404317	$1.44 \cdot 10^{-3}$

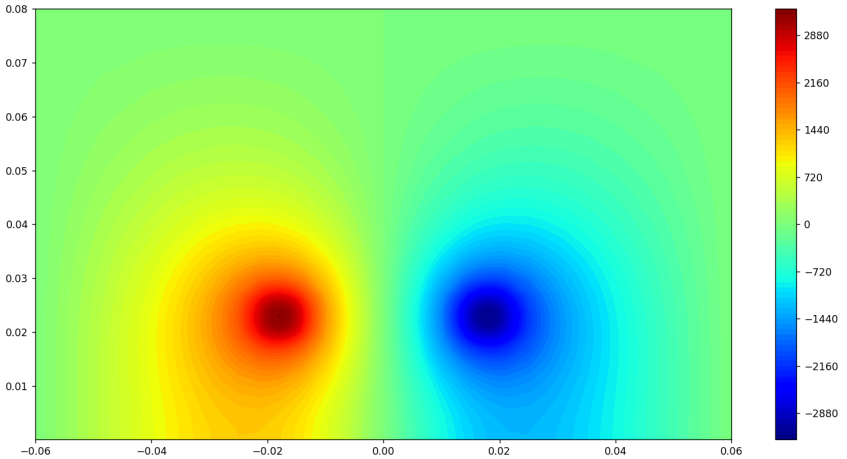

Рис. 14: Решение нелинейной задачи в программе при $J = 10^{14}$

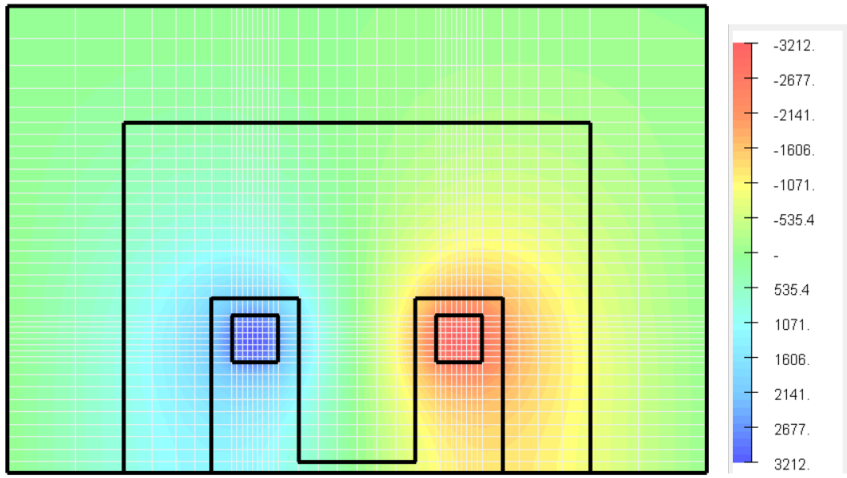

Рис. 15: Решение нелинейной задачи в TELMA при $J = 10^{14}$

Тексты основных модулей

Program.cs

```
var boundariesParameters =
↪  BoundaryParameters.ReadJson("input/boundaryParameters.json");
var meshParameters = new MeshParameters("input/meshParameters");
var mesh = new SuperMesh(meshParameters, new LinearMeshBuilder());
var boundaryHandler = new LinearBoundaryHandler(boundariesParameters, meshParameters);
var muB = new MathDependence("mu", "B");
muB.LoadData("input/mu(B)");
var integrator = new Integration(Quadratures.SegmentGaussOrder5());
Spline spline = Spline.CreateBuilder()
    .SetBasis(new HermiteBasis())
    .SetIntegrator(integrator)
    .SetParameters((1E-07, 1E-07))
    .SetPartitions(200)
    .SetPoints(muB.Data!.Select(data => new Point2D(data.Argument,
↪  data.Function)).ToArray());
var assembler = new BiMatrixAssembler(new LinearBasis(), integrator, mesh);
FemSolver problem = FemSolver.CreateBuilder()
    .SetMesh(mesh)
    .SetSolverSlae(new CGMCholesky(1000, 1E-15))
    .SetAssembler(assembler)
    .SetDependence(muB)
    .SetNonLinearParameters((1E-10, 50))
    .SetSpline(spline)
    .SetBoundaries(boundaryHandler.Process());

problem.Received += assembler.ReceivePermeability;

problem.Compute();

problem.CalculateAzAtPoint((-.0078, 0.0016));
problem.CalculateAzAtPoint((-.0037, 0.0017));
problem.CalculateAzAtPoint((-.0013, 0.0019));
problem.CalculateAzAtPoint((.004, 0.0015));
problem.CalculateAzAtPoint((.0087, 0.0013));

problem.CalculateBAtPoint((-.0078, 0.0016));
problem.CalculateBAtPoint((-.0037, 0.0017));
problem.CalculateBAtPoint((-.0013, 0.0019));
problem.CalculateBAtPoint((.004, 0.0015));
problem.CalculateBAtPoint((.0087, 0.0013));
```

Mesh.cs

```
namespace Magnetostatics.Mesh;

public interface IBaseMesh
{
    IReadOnlyList<Point2D> Points { get; }
    IReadOnlyList<FiniteElement> Elements { get; }
    IReadOnlyList<Area> Areas { get; }
}

public abstract class MeshBuilder
{
```

```
    protected abstract int ElementSize { get; }

    public abstract (IReadOnlyList<Point2D>, FiniteElement[]) Build(MeshParameters
↪   meshParameters);

    protected (IReadOnlyList<Point2D>, FiniteElement[]) BaseBuild(MeshParameters
↪   parameters)
    {
        var result = new
        {
            Points = new Point2D[(parameters.SplitsX.Sum() + 1) *
↪   (parameters.SplitsY.Sum() + 1)],
            Elements = new FiniteElement[parameters.SplitsX.Sum() *
↪   parameters.SplitsY.Sum()]
        };

        double[] pointsX = new double[parameters.SplitsX.Sum() + 1];
        double[] pointsY = new double[parameters.SplitsY.Sum() + 1];

        pointsX[0] = parameters.LinesX[0];
        pointsY[0] = parameters.LinesY[0];

        var idx = 1;

        for (int i = 0; i < parameters.LinesX.Length - 1; i++)
        {
            var sum = 0.0;
            var sign = Math.Sign(parameters.Kx[i]);

            for (int k = 0; k < parameters.SplitsX[i]; k++)
            {
                sum += Math.Pow(sign * parameters.Kx[i], sign * k);
            }

            var hx = (parameters.LinesX[i + 1] - parameters.LinesX[i]) / sum;

            for (int j = 0, k = idx; j < parameters.SplitsX[i] - 1; j++, k++)
            {
                pointsX[idx++] = pointsX[k - 1] + hx;
                hx = sign == 1 ? hx * parameters.Kx[i] : hx / (sign *
↪   parameters.Kx[i]);
            }

            pointsX[idx++] = parameters.LinesX[i + 1];
        }

        idx = 1;

        for (int i = 0; i < parameters.LinesY.Length - 1; i++)
        {
            var sum = 0.0;
            var sign = Math.Sign(parameters.Ky[i]);

            for (int k = 0; k < parameters.SplitsY[i]; k++)
            {
                sum += Math.Pow(sign * parameters.Ky[i], sign * k);
            }

            var hy = (parameters.LinesY[i + 1] - parameters.LinesY[i]) / sum;

```

```
            for (int j = 0, k = idx; j < parameters.SplitsY[i] - 1; j++, k++)
            {
                pointsY[idx++] = pointsY[k - 1] + hy;
                hy = sign == 1 ? hy * parameters.Ky[i] : hy / (sign *
↪ parameters.Ky[i]);
            }

            pointsY[idx++] = parameters.LinesY[i + 1];
        }

        idx = 0;

        foreach (var y in pointsY)
        {
            foreach (var x in pointsX)
            {
                result.Points[idx++] = new(x, y);
            }
        }

        int nx = pointsX.Length;
        idx = 0;

        var nodes = new int[ElementSize];

        for (int j = 0; j < pointsY.Length - 1; j++)
        {
            for (int i = 0; i < pointsX.Length - 1; i++)
            {
                nodes[0] = i + j * nx;
                nodes[1] = i + 1 + j * nx;
                nodes[2] = i + (j + 1) * nx;
                nodes[3] = i + 1 + (j + 1) * nx;

                result.Elements[idx++] = new(nodes.ToArray(),
↪ FindAreaNumber(result.Points, nodes, parameters));
            }
        }

        using StreamWriter sw1 = new("output/elements.txt"), sw2 =
↪ new("output/points.txt");

        foreach (var element in result.Elements)
        {
            foreach (var node in element.Nodes)
            {
                sw1.Write(node + " ");
            }

            sw1.Write(element.AreaNumber);
            sw1.WriteLine();
        }

        foreach (var point in result.Points)
        {
            sw2.WriteLine($"{point.X} {point.Y}");
        }

        return (result.Points, result.Elements);
    }
```

```
124
125     protected static int FindAreaNumber(Point2D[] points, IEnumerable<int> nodes,
    ↪  MeshParameters parameters)
126     {
127         var localPoints = nodes.Select(node => points[node]).ToArray();
128
129         foreach (var area in from area in parameters.Areas
130                  let massCenter = new Point2D(localPoints.Sum(p => p.X) / 4.0,
    ↪  localPoints.Sum(p => p.Y) / 4.0)
131                  where massCenter.X >= parameters.LinesX[area.X1] && massCenter.X <=
    ↪  parameters.LinesX[area.X2] &&
132                        massCenter.Y >= parameters.LinesY[area.Y1] && massCenter.Y <=
    ↪  parameters.LinesY[area.Y2]
133                  select area)
134         {
135             return area.Number;
136         }
137
138         throw new("Incorrect area parameters!");
139     }
140 }
141
142 public class LinearMeshBuilder : MeshBuilder
143 {
144     protected override int ElementSize => 4;
145
146     public override (IReadOnlyList<Point2D>, FiniteElement[]) Build(MeshParameters
    ↪  parameters) => BaseBuild(parameters);
147 }
148
149 public class SuperMesh : IBaseMesh
150 {
151     public IReadOnlyList<Point2D> Points { get; }
152     public IReadOnlyList<FiniteElement> Elements { get; }
153     public IReadOnlyList<Area> Areas { get; }
154
155     public SuperMesh(MeshParameters parameters, MeshBuilder meshBuilder)
156         => ((Points, Elements), Areas) = (meshBuilder.Build(parameters),
    ↪  parameters.Areas);
157 }
158
159 public static class PhysicsConstants
160 {
161     public const double VacuumPermeability = 4.0 * Math.PI * 1E-07;
162 }
163
164 public readonly record struct Area(int Number, double Permeability, double Current,
    ↪  int X1, int X2, int Y1, int Y2)
165 {
166     public static Area Parse(string line)
167     {
168         if (!TryParse(line, out var area))
169         {
170             throw new FormatException("Cant parse Area!");
171         }
172
173         return area;
174     }
175
176     public static bool TryParse(string line, out Area area)
```

```
    {
        var data = line.Split(new[] { ' ', ',' },
↪   StringSplitOptions.RemoveEmptyEntries);

        if (data.Length != 7 || !int.TryParse(data[0], out var number) ||
↪   !double.TryParse(data[1], out var mu)
            || !double.TryParse(data[2], out var current) ||
            !int.TryParse(data[3], out var x1) || !int.TryParse(data[4], out var x2)
↪   ||
            !int.TryParse(data[5], out var y1) || !int.TryParse(data[6], out var y2))
        {
            area = default;
            return false;
        }

        area = new(number, PhysicsConstants.VacuumPermeability * mu, current, x1, x2,
↪   y1, y2);
        return true;
    }
}

public class MeshParameters
{
    private readonly Area[] _areas;
    private int[] _splitsX;
    private int[] _splitsY;
    private double[] _kx;
    private double[] _ky;

    public ImmutableArray<double> LinesX { get; init; }
    public ImmutableArray<double> LinesY { get; init; }
    public ImmutableArray<int> SplitsX => _splitsX.ToImmutableArray();
    public ImmutableArray<int> SplitsY => _splitsY.ToImmutableArray();
    public ImmutableArray<double> Kx => _kx.ToImmutableArray();
    public ImmutableArray<double> Ky => _ky.ToImmutableArray();
    public (int, int) Nesting { get; init; }
    public ImmutableArray<Area> Areas => _areas.ToImmutableArray();

    public MeshParameters(IEnumerable<double> linesX, IEnumerable<double> linesY,
↪   IEnumerable<int> splitsX,
        IEnumerable<int> splitsY, IEnumerable<double> kx, IEnumerable<double> ky,
↪   (int, int) nesting,
        IEnumerable<Area> areas)
    {
        LinesX = linesX.ToImmutableArray();
        LinesY = linesY.ToImmutableArray();
        _splitsX = splitsX.ToArray();
        _splitsY = splitsY.ToArray();
        _kx = kx.ToArray();
        _ky = ky.ToArray();
        Nesting = nesting;
        _areas = areas.ToArray();
    }


    public MeshParameters(string path)
    {
        if (!File.Exists(path)) throw new("File does not exist");

        using var sr = new StreamReader(path);
```

```
        LinesX = sr.ReadLine()!.Split().Where(line =>
↪   !string.IsNullOrWhiteSpace(line)).Select(double.Parse)
            .ToImmutableArray();
        LinesY = sr.ReadLine()!.Split().Where(line =>
↪   !string.IsNullOrWhiteSpace(line)).Select(double.Parse)
            .ToImmutableArray();
        _splitsX = sr.ReadLine()!.Split().Where(line =>
↪   !string.IsNullOrWhiteSpace(line)).Select(int.Parse)
            .ToArray();
        _splitsY = sr.ReadLine()!.Split().Where(line =>
↪   !string.IsNullOrWhiteSpace(line)).Select(int.Parse)
            .ToArray();
        _kx = sr.ReadLine()!.Split().Where(line =>
↪   !string.IsNullOrWhiteSpace(line)).Select(double.Parse)
            .ToArray();
        _ky = sr.ReadLine()!.Split().Where(line =>
↪   !string.IsNullOrWhiteSpace(line)).Select(double.Parse)
            .ToArray();
        var line = sr.ReadLine()!.Split(new[] { ' ', ',' },
↪   StringSplitOptions.RemoveEmptyEntries);
        Nesting = (int.Parse(line[0]), int.Parse(line[1]));
        _areas = sr.ReadToEnd().Split("\n").Select(Area.Parse).ToArray();

        var expectedResult = Areas.OrderBy(area => area.Number);

        if (Nesting.Item1 > 2 || Nesting.Item2 > 2 || Nesting.Item1 < 0 ||
↪   Nesting.Item2 < 0)
        {
            // maybe TODO any number of nesting mesh
            throw new("Nesting parameters should be from 0 to 2!");
        }

        if (!expectedResult.SequenceEqual(Areas)) throw new("Area numbers must be
↪   sorted by ascending!");

        _areas = _areas.OrderByDescending(area => area.Number).ToArray();

        if (Nesting.Item1 != 0 || Nesting.Item2 != 0) RecalculateParameters();
    }

    private void RecalculateParameters()
    {
        _splitsX = _splitsX.Select(x => Nesting.Item1 == 1 ? x * 2 : x * 4).ToArray();
        _splitsY = _splitsY.Select(y => Nesting.Item1 == 1 ? y * 2 : y * 4).ToArray();
        _kx = _kx.Select(k =>
            {
                var sign = Math.Sign(k);
                return Nesting.Item1 == 1 ? sign * Math.Sqrt(sign * k) : sign *
↪   Math.Sqrt(Math.Sqrt(sign * k));
            })
            .ToArray();
        _ky = _ky.Select(k =>
            {
                var sign = Math.Sign(k);
                return Nesting.Item1 == 1 ? sign * Math.Sqrt(sign * k) : sign *
↪   Math.Sqrt(Math.Sqrt(sign * k));
            })
            .ToArray();
    }
}
```

FEM.cs

```
namespace Magnetostatics.src.FEM;

using Spline;

public sealed class FemSolver
{
    public class FemSolverBuilder
    {
        private readonly FemSolver _femSolver = new();

        public FemSolverBuilder SetTest(ITest test)
        {
            _femSolver._test = test;
            return this;
        }

        public FemSolverBuilder SetMesh(IBaseMesh mesh)
        {
            _femSolver._mesh = mesh;
            return this;
        }

        public FemSolverBuilder SetSolverSlae(IterativeSolver iterativeSolver)
        {
            _femSolver._iterativeSolver = iterativeSolver;
            return this;
        }

        public FemSolverBuilder SetBoundaries(IEnumerable<IBoundary> boundaries)
        {
            _femSolver._boundaries = boundaries.DistinctBy(b => b.Node);
            return this;
        }

        public FemSolverBuilder SetAssembler(BaseMatrixAssembler matrixAssembler)
        {
            _femSolver._matrixAssembler = matrixAssembler;
            return this;
        }

        public FemSolverBuilder SetDependence(MathDependence dependence)
        {
            _femSolver._dependence = dependence;
            return this;
        }

        public FemSolverBuilder SetNonLinearParameters((double Residual, int
↪   MaxIters) nonLinearParameters)
        {
            _femSolver._nonLinearParameters = nonLinearParameters;
            return this;
        }

        public FemSolverBuilder SetSpline(Spline spline)
        {
```

```
            _femSolver._spline = spline;
            return this;
        }

        public static implicit operator FemSolver(FemSolverBuilder builder)
            => builder._femSolver;
    }

    private IBaseMesh _mesh = default!;
    private ITest _test = default!;
    private IterativeSolver _iterativeSolver = default!;
    private IEnumerable<IBoundary> _boundaries = default!;
    private Vector<double> _localVector = default!;
    private Vector<double> _globalVector = default!;
    private BaseMatrixAssembler _matrixAssembler = default!;
    private MathDependence? _dependence;
    private (double Residual, int MaxIters)? _nonLinearParameters;
    private Spline? _spline;
    private bool _isInit;

    public event EventHandler<double>? Received;

    public void Compute()
    {
        Initialize();

        int iter;
        var iters = _nonLinearParameters?.MaxIters ?? 1;

        AssemblySystem();
        AccountingDirichletBoundary();

        var qk = new Vector<double>(_globalVector.Length);

        _isInit = true;

        for (iter = 0; iter < iters; iter++)
        {
            _iterativeSolver.SetMatrix(_matrixAssembler.GlobalMatrix!);
            _iterativeSolver.SetVector(_globalVector);
            _iterativeSolver.Compute();

            qk.Add(_iterativeSolver.Solution!.Value);
            _matrixAssembler.GlobalMatrix!.Clear();
            _globalVector.Fill(0.0);

            AssemblySystem();
            AccountingDirichletBoundary();

            var residual = (_matrixAssembler.GlobalMatrix! * qk -
↪  _globalVector).Norm() / _globalVector.Norm();

            Console.WriteLine(residual);

            if (residual < _nonLinearParameters?.Residual) break;
        }

        using var sw = new StreamWriter("output/q.txt");

        foreach (var value in _iterativeSolver.Solution!.Value)
```

```
        {
            sw.WriteLine(value);
        }

        Console.WriteLine($"Iterations = {iter}");

        // CalculateError();
    }

    private void OnReceived(double value) => Received?.Invoke(this, value);

    private void Initialize()
    {
        PortraitBuilder.Build(_mesh, out var ig, out var jg);
        _spline?.Compute();
        _matrixAssembler.GlobalMatrix = new(ig.Length - 1, jg.Length)
        {
            Ig = ig,
            Jg = jg
        };

        _globalVector = new(ig.Length - 1);
        _localVector = new(_matrixAssembler.Basis.Size);
    }

    private void AssemblySystem()
    {
        for (int ielem = 0; ielem < _mesh.Elements.Count; ielem++)
        {
            var element = _mesh.Elements[ielem];

            if (!_isInit && element.AreaNumber == 1 && _dependence is not null)
            {
                OnReceived(_dependence.Data![0].Function);
            }

            if (_isInit && element.AreaNumber == 1 && _dependence is not null)
            {
                double mu;
                var localPoints = _mesh.Elements[ielem].Nodes.Select(node =>
↪   _mesh.Points[node]).ToArray();
                var massCenter = new Point2D(localPoints.Sum(p => p.X) / 4.0,
↪   localPoints.Sum(p => p.Y) / 4.0);
                var module = CalculateBAtPoint(massCenter);
                var firstDependenceValue = _dependence!.Data!.First();
                var lastDependenceValue = _dependence.Data!.Last();
                // Console.WriteLine($"|B| = {module}");

                if (module > lastDependenceValue.Argument)
                {
                    mu = 1.0 / (lastDependenceValue.Argument / module * (1.0 /
↪   lastDependenceValue.Function - 1.0) +
                                1.0);
                    OnReceived(mu);
                    // Console.WriteLine($"> table value, mu = {mu}");
                }
                else if (module <= firstDependenceValue.Argument)
                {
                    mu = firstDependenceValue.Function;
                    OnReceived(mu);
```

```
                    // Console.WriteLine($"< table value, mu = {mu}");
                }
                else
                {
                    OnReceived(_spline!.ValueAtPoint(module));
                    // Console.WriteLine($"inside, mu =
↪ {_spline!.ValueAtPoint(module)}");
                }
            }

            _matrixAssembler.BuildLocalMatrices(ielem);
            BuildLocalVector(ielem);

            for (int i = 0; i < _matrixAssembler.Basis.Size; i++)
            {
                _globalVector[element.Nodes[i]] += _localVector[i];

                for (int j = 0; j < _matrixAssembler.Basis.Size; j++)
                {
                    _matrixAssembler.FillGlobalMatrix(element.Nodes[i],
↪ element.Nodes[j],
                        _matrixAssembler.StiffnessMatrix[i, j]);
                }
            }
        }
    }

    private void BuildLocalVector(int ielem)
    {
        _localVector.Fill(0.0);
        var jCurrent = _mesh.Areas.First(area => area.Number ==
↪ _mesh.Elements[ielem].AreaNumber).Current;

        for (int i = 0; i < _matrixAssembler.Basis.Size; i++)
        {
            for (int j = 0; j < _matrixAssembler.Basis.Size; j++)
            {
                // _localVector[i] += _matrixAssembler.MassMatrix[i, j] *
                //
↪ _test.J(_mesh.Points[_mesh.Elements[ielem].Nodes[j]]);
                _localVector[i] += _matrixAssembler.MassMatrix[i, j] * jCurrent;
            }
        }
    }

    private void AccountingDirichletBoundary()
    {
        // int[] checkBc = new int[_mesh.Points.Count];
        //
        // checkBc.Fill(-1);
        // var boundariesArray = _boundaries.ToArray();
        //
        // // foreach (var b in boundariesArray)
        // // {
        // //     _matrixAssembler.GlobalMatrix.Di[b.Node] = 1E+32;
        // //     _globalVector[b.Node] = 1E+32 * b.Value;
        // // }
        //
        // for (var i = 0; i < boundariesArray.Length; i++)
        // {
```

```
        //     boundariesArray[i].Value =
↪ _test.Az(_mesh.Points[boundariesArray[i].Node]);
        //     checkBc[boundariesArray[i].Node] = i;
        // }
        //
        // for (int i = 0; i < _mesh.Points.Count; i++)
        // {
        //     int index;
        //     if (checkBc[i] != -1)
        //     {
        //         _matrixAssembler.GlobalMatrix!.Di[i] = 1.0;
        //         _globalVector[i] = boundariesArray[checkBc[i]].Value;
        //
        //         for (int k = _matrixAssembler.GlobalMatrix.Ig[i]; k <
↪ _matrixAssembler.GlobalMatrix.Ig[i + 1]; k++)
        //         {
        //             index = _matrixAssembler.GlobalMatrix.Jg[k];
        //
        //             if (checkBc[index] == -1)
        //             {
        //                 _globalVector[index] -=
↪ _matrixAssembler.GlobalMatrix.Gg[k] * _globalVector[i];
        //             }
        //
        //             _matrixAssembler.GlobalMatrix.Gg[k] = 0.0;
        //         }
        //     }
        //     else
        //     {
        //         for (int k = _matrixAssembler.GlobalMatrix!.Ig[i]; k <
↪ _matrixAssembler.GlobalMatrix.Ig[i + 1]; k++)
        //         {
        //             index = _matrixAssembler.GlobalMatrix.Jg[k];
        //
        //             if (checkBc[index] == -1) continue;
        //             _globalVector[i] -= _matrixAssembler.GlobalMatrix.Gg[k] *
↪ _globalVector[index];
        //             _matrixAssembler.GlobalMatrix.Gg[k] = 0.0;
        //         }
        //     }
        // }

        var boundariesArray = _boundaries.ToArray();

        foreach (var boundary in boundariesArray)
        {
            _matrixAssembler.GlobalMatrix!.Di[boundary.Node] = 1E+32;
            _globalVector[boundary.Node] = 0.0;
        }
    }

    private void CalculateError()
    {
        var error = new double[_mesh.Points.Count];

        for (int i = 0; i < error.Length; i++)
        {
            error[i] = Math.Abs(_iterativeSolver.Solution!.Value[i] -
↪ _test.Az(_mesh.Points[i]));
        }
```

```
281
282         Array.ForEach(error, Console.WriteLine);
283
284         var sum = error.Sum(t => t * t);
285
286         sum = Math.Sqrt(sum / _mesh.Points.Count);
287
288         Console.WriteLine($"rms = {sum}");
289
290         // using var sw = new StreamWriter("output/3.csv");
291         //
292         // for (int i = 0; i < error.Length; i++)
293         // {
294         //     if (i == 0)
295         //     {
296         //         sw.WriteLine("$i$, $u_i^*$, $u_i$, $|u^* - u|$, Погрешность");
297         //         sw.WriteLine(
298         //             $"{i}, {_test.U(_mesh.Points[i])},
    ↪ {_iterativeSolver.Solution!.Value[i]}, {error[i]}, {sum}");
299         //         continue;
300         //     }
301         //
302         //     sw.WriteLine($"{i}, {_test.U(_mesh.Points[i])},
    ↪ {_iterativeSolver.Solution!.Value[i]}, {error[i]},");
303         // }
304     }
305
306     public double CalculateAzAtPoint(Point2D point)
307     {
308         var res = 0.0;
309
310         var ielem = FindElementNumber(point);
311
312         var element = _mesh.Elements[ielem];
313         var bPoint = _mesh.Points[element.Nodes[0]];
314         var ePoint = _mesh.Points[element.Nodes[^1]];
315
316         double hx = ePoint.X - bPoint.X;
317         double hy = ePoint.Y - bPoint.Y;
318
319         var ksi = (point.X - bPoint.X) / hx;
320         var eta = (point.Y - bPoint.Y) / hy;
321
322         var templatePoint = new Point2D(ksi, eta);
323
324         for (int i = 0; i < _matrixAssembler.Basis.Size; i++)
325         {
326             res += _iterativeSolver.Solution!.Value[_mesh.Elements[ielem].Nodes[i]] *
327                    _matrixAssembler.Basis.GetPsi(i, templatePoint);
328         }
329
330         Console.WriteLine($"Az at {point} = {res}");
331         return res;
332     }
333
334     public double CalculateBAtPoint(Point2D point)
335     {
336         var ielem = FindElementNumber(point);
337         OnReceived(1.0 / PhysicsConstants.VacuumPermeability);
338         _matrixAssembler.BuildLocalMatrices(ielem);
```

```
339
340         var sqrModule = 0.0;
341
342         for (int i = 0; i < _matrixAssembler.Basis.Size; i++)
343         {
344             for (int j = 0; j < _matrixAssembler.Basis.Size; j++)
345             {
346                 sqrModule += _matrixAssembler.StiffnessMatrix[i, j] *
347
    ↪ _iterativeSolver.Solution!.Value[_mesh.Elements[ielem].Nodes[i]] *
348
    ↪ _iterativeSolver.Solution.Value[_mesh.Elements[ielem].Nodes[j]];
349             }
350         }
351
352         var elementArea = (_mesh.Points[_mesh.Elements[ielem].Nodes[1]].X -
353                            _mesh.Points[_mesh.Elements[ielem].Nodes[0]].X) *
354                           (_mesh.Points[_mesh.Elements[ielem].Nodes[2]].Y -
355                            _mesh.Points[_mesh.Elements[ielem].Nodes[0]].Y);
356
357         sqrModule /= elementArea;
358
359         var module = Math.Sqrt(sqrModule);
360
361         // Console.WriteLine($"|B| at ({point.X}; {point.Y}) = {module}");
362         return module;
363
364         // var ielem = FindElementNumber(point);
365         //
366         // var element = _mesh.Elements[ielem];
367         // var bPoint = _mesh.Points[element.Nodes[0]];
368         // var ePoint = _mesh.Points[element.Nodes[^1]];
369         //
370         // double hx = ePoint.X - bPoint.X;
371         // double hy = ePoint.Y - bPoint.Y;
372         //
373         // var ksi = (point.X - bPoint.X) / hx;
374         // var eta = (point.Y - bPoint.Y) / hy;
375         //
376         // var templatePoint = (ksi, eta);
377         //
378         // double dx = 0.0;
379         // double dy = 0.0;
380         //
381         // for (int i = 0; i < _matrixAssembler.Basis.Size; i++)
382         // {
383         //     dx += _iterativeSolver.Solution!.Value[element.Nodes[i]] *
384         //           _matrixAssembler.Basis.GetDPsi(i, 0, templatePoint);
385         //     dy += _iterativeSolver.Solution!.Value[element.Nodes[i]] *
386         //           _matrixAssembler.Basis.GetDPsi(i, 1, templatePoint);
387         // }
388         //
389         // dx = -dx / hx;
390         // dy /= hy;
391         //
392         // return Math.Sqrt(dx * dx + dy * dy); // calculate with B_x and B_y
    ↪ components of rotor A
393     }
394
395     private int FindElementNumber(Point2D point)
```

```
396     {
397         foreach (var (element, idx) in _mesh.Elements.Select((element, idx) =>
    ↪ (element, idx)))
398         {
399             if (point.X >= _mesh.Points[element.Nodes[0]].X && point.X <=
    ↪ _mesh.Points[element.Nodes[1]].X &&
400                 point.Y >= _mesh.Points[element.Nodes[0]].Y && point.Y <=
    ↪ _mesh.Points[element.Nodes[2]].Y)
401             {
402                 return idx;
403             }
404         }
405 
406         throw new("Not supported exception!");
407     }
408 
409     public static FemSolverBuilder CreateBuilder() => new();
410 }
```

Assembler.cs

```
1  namespace Magnetostatics.src.FEM;
2  
3  public abstract class BaseMatrixAssembler
4  {
5      protected readonly IBaseMesh _mesh;
6      protected readonly Integration _integrator;
7      protected Matrix[]? _baseStiffnessMatrix;
8      protected Matrix? _baseMassMatrix;
9      protected double? _mu;
10 
11     public SparseMatrix? GlobalMatrix { get; set; } // need initialize with portrait
    ↪ builder
12     public Matrix StiffnessMatrix { get; }
13     public Matrix MassMatrix { get; }
14     public IBasis2D Basis { get; }
15 
16     protected BaseMatrixAssembler(IBasis2D basis, Integration integrator, IBaseMesh
    ↪ mesh)
17     {
18         Basis = basis;
19         _integrator = integrator;
20         _mesh = mesh;
21         StiffnessMatrix = new(basis.Size);
22         MassMatrix = new(basis.Size);
23     }
24 
25     public abstract void BuildLocalMatrices(int ielem);
26 
27     public void FillGlobalMatrix(int i, int j, double value)
28     {
29         if (GlobalMatrix is null)
30         {
31             throw new("Initialize the global matrix (use portrait builder)!");
32         }
33 
34         if (i == j)
35         {
```

```
            GlobalMatrix.Di[i] += value;
            return;
        }

        if (i <= j) return;
        for (int ind = GlobalMatrix.Ig[i]; ind < GlobalMatrix.Ig[i + 1]; ind++)
        {
            if (GlobalMatrix.Jg[ind] != j) continue;
            GlobalMatrix.Gg[ind] += value;
            return;
        }
    }
}

public class BiMatrixAssembler : BaseMatrixAssembler
{
    public BiMatrixAssembler(IBasis2D basis, Integration integrator, IBaseMesh mesh)
    : base(basis, integrator, mesh)
    {
    }

    public override void BuildLocalMatrices(int ielem)
    {
        var element = _mesh.Elements[ielem];
        var bPoint = _mesh.Points[element.Nodes[0]];
        var ePoint = _mesh.Points[element.Nodes[^1]];

        double hx = ePoint.X - bPoint.X;
        double hy = ePoint.Y - bPoint.Y;

        if (_baseStiffnessMatrix is null)
        {
            _baseStiffnessMatrix = new Matrix[] { new(Basis.Size), new(Basis.Size) };
            _baseMassMatrix = new(Basis.Size);
            var templateElement = new Rectangle(new(0.0, 0.0), new(1.0, 1.0));

            for (int i = 0; i < Basis.Size; i++)
            {
                for (int j = 0; j <= i; j++)
                {
                    Func<Point2D, double> function;

                    for (int k = 0; k < 2; k++)
                    {
                        var ik = i;
                        var jk = j;
                        var k1 = k;
                        function = p =>
                        {
                            var dFi1 = Basis.GetDPsi(ik, k1, p);
                            var dFi2 = Basis.GetDPsi(jk, k1, p);

                            return dFi1 * dFi2;
                        };

                        _baseStiffnessMatrix[k][i, j] = _baseStiffnessMatrix[k][j, i]
    =
                            _integrator.Gauss2D(function, templateElement);
                    }

```

```
                    var i1 = i;
                    var j1 = j;
                    function = p =>
                    {
                        var fi1 = Basis.GetPsi(i1, p);
                        var fi2 = Basis.GetPsi(j1, p);

                        return fi1 * fi2;
                    };
                    _baseMassMatrix[i, j] = _baseMassMatrix[j, i] =
↪   _integrator.Gauss2D(function, templateElement);
                }
            }
        }

        var mu = _mu ?? _mesh.Areas.First(area => area.Number ==
↪   _mesh.Elements[ielem].AreaNumber).Permeability;

        for (int i = 0; i < Basis.Size; i++)
        {
            for (int j = 0; j <= i; j++)
            {
                StiffnessMatrix[i, j] = StiffnessMatrix[j, i] =
                    1.0 / mu * (hy / hx * _baseStiffnessMatrix[0][i, j] +
                                hx / hy * _baseStiffnessMatrix[1][i, j]);
            }
        }

        _mu = null;

        for (int i = 0; i < Basis.Size; i++)
        {
            for (int j = 0; j <= i; j++)
            {
                MassMatrix[i, j] = MassMatrix[j, i] = hx * hy * _baseMassMatrix![i,
↪   j];
            }
        }
    }

    public void ReceivePermeability(object? sender, double value) => _mu =
↪   PhysicsConstants.VacuumPermeability * value;
}
```

PortraitBuilder.cs

```
namespace Magnetostatics.src.FEM;

public static class PortraitBuilder
{
    public static void Build(IBaseMesh mesh, out int[] ig, out int[] jg)
    {
        var connectivityList = new List<HashSet<int>>();

        for (int i = 0; i < mesh.Points.Count; i++)
        {
            connectivityList.Add(new());
        }
```

```
        int localSize = mesh.Elements[0].Nodes.Count;

        foreach (var element in mesh.Elements.Select(element =>
↪   element.Nodes.OrderBy(node => node).ToArray()))
        {
            for (int i = 0; i < localSize - 1; i++)
            {
                int nodeToInsert = element[i];

                for (int j = i + 1; j < localSize; j++)
                {
                    int posToInsert = element[j];

                    connectivityList[posToInsert].Add(nodeToInsert);
                }
            }
        }

        var orderedList = connectivityList.Select(list => list.OrderBy(val =>
↪   val)).ToList();

        ig = new int[connectivityList.Count + 1];

        ig[0] = 0;
        ig[1] = 0;

        for (int i = 1; i < connectivityList.Count; i++)
        {
            ig[i + 1] = ig[i] + connectivityList[i].Count;
        }

        jg = new int[ig[^1]];

        for (int i = 1, j = 0; i < connectivityList.Count; i++)
        {
            foreach (var it in orderedList[i])
            {
                jg[j++] = it;
            }
        }
    }
}
```

Integration.cs

```
namespace Magnetostatics.src.FEM;

public class Integration
{
    private readonly IEnumerable<QuadratureNode<double>> _quadratures;

    public Integration(IEnumerable<QuadratureNode<double>> quadratures) =>
↪   _quadratures = quadratures;

    public double Gauss2D(Func<Point2D, double> psi, Rectangle element)
    {
        double hx = element.RightTop.X - element.LeftTop.X;
```

```
        double hy = element.RightTop.Y - element.RightBottom.Y;

        var result = (from qi in _quadratures
            from qj in _quadratures
            let point = new Point2D((qi.Node * hx + element.LeftBottom.X +
↪  element.RightBottom.X) / 2.0,
                (qj.Node * hy + element.RightBottom.Y + element.RightTop.Y) / 2.0)
            select psi(point) * qi.Weight * qj.Weight).Sum();

        return result * hx * hy / 4.0;
    }

    public double Gauss1D(Func<double, double, double> psi, Interval interval)
    {
        double h = interval.Length;
        double result = Quadratures.SegmentGaussOrder5()
            .Sum(q => q.Weight *
                      psi((interval.LeftBorder + interval.RightBorder + q.Node * h) /
↪  2.0, h));

        return result * h / 2.0;
    }
}
```

Quadratures.cs

```
namespace Magnetostatics.src.FEM;

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

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

public static class Quadratures
{
    public static IEnumerable<QuadratureNode<double>> SegmentGaussOrder5()
    {
        const int n = 3;
        double[] points =
        {
            0,
            -Math.Sqrt(3.0 / 5.0),
            Math.Sqrt(3.0 / 5.0)
        };
        double[] weights =
        {
            8.0 / 9.0,
            5.0 / 9.0,
            5.0 / 9.0
        };

```

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

    public static IEnumerable<QuadratureNode<double>> SegmentGaussOrder9()
    {
        const int n = 5;
        double[] points =
        {
            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)),
            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 =
        {
            128.0 / 225.0,
            (322.0 + 13.0 * Math.Sqrt(70.0)) / 900.0,
            (322.0 + 13.0 * Math.Sqrt(70.0)) / 900.0,
            (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(points[i], weights[i]);
        }
    }
}
```

Solvers.cs

```
namespace Magnetostatics.src.FEM;

public abstract class IterativeSolver
{
    protected TimeSpan? _runningTime;
    protected SparseMatrix _matrix = default!;
    protected Vector<double> _vector = default!;
    protected Vector<double>? _solution;

    public int MaxIters { get; }
    public double Eps { get; }
    public TimeSpan? RunningTime => _runningTime;
    public ImmutableArray<double>? Solution => _solution?.ToImmutableArray();

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

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

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

```
23
24      public abstract void Compute();
25
26      protected void Cholesky(double[] ggnew, double[] dinew)
27      {
28          double suml = 0.0;
29          double sumdi = 0.0;
30
31          for (int i = 0; i < _matrix.Size; i++)
32          {
33              int i0 = _matrix.Ig[i];
34              int i1 = _matrix.Ig[i + 1];
35
36              for (int k = i0; k < i1; k++)
37              {
38                  int j = _matrix.Jg[k];
39                  int j0 = _matrix.Ig[j];
40                  int j1 = _matrix.Ig[j + 1];
41                  int ik = i0;
42                  int kj = j0;
43
44                  while (ik < k && kj < j1)
45                  {
46                      if (_matrix.Jg[ik] == _matrix.Jg[kj])
47                      {
48                          suml += ggnew[ik] * ggnew[kj];
49                          ik++;
50                          kj++;
51                      }
52                      else
53                      {
54                          if (_matrix.Jg[ik] > _matrix.Jg[kj])
55                              kj++;
56                          else
57                              ik++;
58                      }
59                  }
60
61                  ggnew[k] = (ggnew[k] - suml) / dinew[j];
62                  sumdi += ggnew[k] * ggnew[k];
63                  suml = 0.0;
64              }
65
66              dinew[i] = Math.Sqrt(dinew[i] - sumdi);
67              sumdi = 0.0;
68          }
69      }
70
71      protected Vector<double> MoveForCholesky(Vector<double> vector, double[] ggnew,
    ↪  double[] dinew)
72      {
73          Vector<double> y = new(vector.Length);
74          Vector<double> x = new(vector.Length);
75          Vector<double>.Copy(vector, y);
76
77          double sum = 0.0;
78
79          for (int i = 0; i < _matrix.Size; i++) // Прямой ход
80          {
81              int i0 = _matrix.Ig[i];
```

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

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

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

        Vector<double>.Copy(y, x);

        for (int i = _matrix.Size - 1; i >= 0; i--) // Обратный ход
        {
            int i0 = _matrix.Ig[i];
            int i1 = _matrix.Ig[i + 1];
            x[i] = y[i] / dinew[i];

            for (int k = i0; k < i1; k++)
                y[_matrix.Jg[k]] -= ggnew[k] * x[i];
        }

        return x;
    }
}

public class CGM : IterativeSolver
{
    public CGM(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 vectorNorm = _vector.Norm();

            _solution = new(_vector.Length);

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

            Stopwatch sw = Stopwatch.StartNew();

            var r = _vector - (_matrix * _solution);

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

            for (int iter = 0; iter < MaxIters && r.Norm() / vectorNorm >= Eps;
↪  iter++)
            {
                var tmp = _matrix * z;
                var alpha = r * r / (tmp * z);
                _solution += alpha * z;
                var squareNorm = r * r;
                r -= alpha * tmp;
```

```
                var beta = r * r / squareNorm;
                z = r + beta * z;
            }

            sw.Stop();

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

public class CGMCholesky : IterativeSolver
{
    public CGMCholesky(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 vectorNorm = _vector.Norm();

            _solution = new(_vector.Length);

            double[] ggnew = new double[_matrix.Gg.Length];
            double[] dinew = new double[_matrix.Di.Length];

            _matrix.Gg.Copy(ggnew);
            _matrix.Di.Copy(dinew);

            Stopwatch sw = Stopwatch.StartNew();

            Cholesky(ggnew, dinew);

            var r = _vector - _matrix * _solution;
            var z = MoveForCholesky(r, ggnew, dinew);

            for (int iter = 0; iter < MaxIters && r.Norm() / vectorNorm >= Eps;
↪   iter++)
            {
                var tmp = MoveForCholesky(r, ggnew, dinew) * r;
                var sndTemp = _matrix * z;
                var alpha = tmp / (sndTemp * z);
                _solution += alpha * z;
                r -= alpha * sndTemp;
```

```
                var fstTemp = MoveForCholesky(r, ggnew, dinew);
                var beta = fstTemp * r / tmp;
                z = fstTemp + beta * z;
            }

            sw.Stop();

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

public record GaussSeidel(int MaxIters, double Eps, double W)
{
    public Vector<double> Compute(SevenDiagonalMatrix diagMatrix, Vector<double> pr)
    {
        Vector<double> qk = new(diagMatrix.Size);
        Vector<double> qk1 = new(diagMatrix.Size);
        Vector<double> residual = new(diagMatrix.Size);
        double prNorm = pr.Norm();

        for (int i = 0; i < MaxIters; i++)
        {
            for (int k = 0; k < diagMatrix.Size; k++)
            {
                double sum1 = MultLine(diagMatrix, k, qk1, 1);
                double sum2 = MultLine(diagMatrix, k, qk, 2);

                residual[k] = pr[k] - (sum1 + sum2);
                qk1[k] = qk[k] + W * residual[k] / diagMatrix[0, k];
            }


            Vector<double>.Copy(qk1, qk);
            qk1.Fill(0.0);

            if (residual.Norm() / prNorm < Eps) break;
        }

        return qk;
    }

    private static double MultLine(SevenDiagonalMatrix diagMatrix, int i,
↪   Vector<double> vector, int method)
    {
        double sum = 0.0;

        if (method is 0 or 1)
        {
            if (i > 0) sum += diagMatrix[4, i - 1] * vector[i - 1];

```

```
            if (i > 1) sum += diagMatrix[5, i - 2] * vector[i - 2];

            if (i > 2) sum += diagMatrix[6, i - 3] * vector[i - 3];
        }

        if (method is not (0 or 2)) return sum;

        sum += diagMatrix[0, i] * vector[i];

        if (i < diagMatrix.Size - 1) sum += diagMatrix[1, i] * vector[i + 1];

        if (i < diagMatrix.Size - 2) sum += diagMatrix[2, i] * vector[i + 2];

        if (i < diagMatrix.Size - 3) sum += diagMatrix[3, i] * vector[i + 3];

        return sum;
    }
}
```

Matrices.cs

```
namespace Magnetostatics.src.FEM;

public class SparseMatrix
{
    public int[] Ig { get; init; }
    public int[] Jg { get; init; }
    public double[] Di { get; }
    public double[] Gg { get; }
    public int Size { get; }

    public SparseMatrix(int size, int sizeOffDiag)
    {
        Size = size;
        Ig = new int[size + 1];
        Jg = new int[sizeOffDiag];
        Gg = 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.Gg[j] * vector[matrix.Jg[j]];
                product[matrix.Jg[j]] += matrix.Gg[j] * vector[i];
            }
        }

        return product;
    }
```

```
37
38      public void PrintDense(string path)
39      {
40          double[,] A = new double[Size, Size];
41
42          for (int i = 0; i < Size; i++)
43          {
44              A[i, i] = Di[i];
45
46              for (int j = Ig[i]; j < Ig[i + 1]; j++)
47              {
48                  A[i, Jg[j]] = Gg[j];
49                  A[Jg[j], i] = Gg[j];
50              }
51          }
52
53          using var sw = new StreamWriter(path);
54          for (int i = 0; i < Size; i++)
55          {
56              for (int j = 0; j < Size; j++)
57              {
58                  sw.Write(A[i, j].ToString("0.00") + "\t");
59              }
60
61              sw.WriteLine();
62          }
63      }
64
65      public void Clear()
66      {
67          for (int i = 0; i < Size; i++)
68          {
69              Di[i] = 0.0;
70
71              for (int k = Ig[i]; k < Ig[i + 1]; k++)
72              {
73                  Gg[k] = 0.0;
74              }
75          }
76      }
77  }
78
79  public class Matrix
80  {
81      private readonly double[,] _storage;
82      public int Size { get; }
83
84      public double this[int i, int j]
85      {
86          get => _storage[i, j];
87          set => _storage[i, j] = value;
88      }
89
90      public Matrix(int size)
91      {
92          _storage = new double[size, size];
93          Size = size;
94      }
95
96      public void Clear() => Array.Clear(_storage, 0, _storage.Length);
```

```
97
98      public void Copy(Matrix destination)
99      {
100         for (int i = 0; i < destination.Size; i++)
101         {
102             for (int j = 0; j < destination.Size; j++)
103             {
104                 destination[i, j] = _storage[i, j];
105             }
106         }
107     }
108
109     public void LU()
110     {
111         for (int i = 0; i < Size; i++)
112         {
113             for (int j = 0; j < Size; j++)
114             {
115                 double suml = 0.0;
116                 double sumu = 0.0;
117
118                 if (i < j)
119                 {
120                     for (int k = 0; k < i; k++)
121                     {
122                         sumu += _storage[i, k] * _storage[k, j];
123                     }
124
125                     _storage[i, j] = (_storage[i, j] - sumu) / _storage[i, i];
126                 }
127                 else
128                 {
129                     for (int k = 0; k < j; k++)
130                     {
131                         suml += _storage[i, k] * _storage[k, j];
132                     }
133
134                     _storage[i, j] -= suml;
135                 }
136             }
137         }
138     }
139
140     public static Matrix operator +(Matrix fstMatrix, Matrix sndMatrix)
141     {
142         Matrix resultMatrix = new(fstMatrix.Size);
143
144         for (int i = 0; i < resultMatrix.Size; i++)
145         {
146             for (int j = 0; j < resultMatrix.Size; j++)
147             {
148                 resultMatrix[i, j] = fstMatrix[i, j] + sndMatrix[i, j];
149             }
150         }
151
152         return resultMatrix;
153     }
154
155     public static Matrix operator *(double value, Matrix matrix)
156     {
```

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

        for (int i = 0; i < resultMatrix.Size; i++)
        {
            for (int j = 0; j < resultMatrix.Size; j++)
            {
                resultMatrix[i, j] = value * matrix[i, j];
            }
        }

        return resultMatrix;
    }
}
```

Basis.cs

```
namespace Magnetostatics.src.FEM;

public interface IBasis2D
{
    int Size { get; }

    double GetPsi(int number, Point2D point);

    double GetDPsi(int number, int varNumber, Point2D point);
}

public interface IBasis1D
{
    int Size { get; }

    double GetPsi(int number, double point, double h = 1.0);

    double GetDPsi(int number, double point, double h = 1.0);

    double GetDdPsi(int number, double point, double h = 1.0);
}

public class HermiteBasis : IBasis1D
{
    public int Size => 4;

    public double GetPsi(int number, double point, double h = 1.0)
        => number switch
        {
            0 => 1.0 - 3.0 * point * point + 2.0 * point * point * point, // 1 - 3ξ^2 + 2ξ^3
            1 => h * (point - 2.0 * point * point + point * point * point), // h_i · (ξ - 2ξ^2 + ξ^3)
            2 => 3.0 * point * point - 2.0 * point * point * point, // 3ξ^2 - 2ξ^3
            3 => h * (-point * point + point * point * point), // h_i · (-ξ^2 + ξ^3)
            _ => throw new ArgumentOutOfRangeException(nameof(number), number, "Not expected function number!")
        };

    public double GetDPsi(int number, double point, double h = 1.0)
        => number switch
        {
```

```
            0 => -6.0 * (point - point * point) / h, // $\frac{-6 \cdot (\xi - \xi^2)}{h_i}$
            1 => 1.0 - 4.0 * point + 3.0 * point * point, // $1 - 4\xi + 3\xi^2$
            2 => 6.0 * (point - point * point) / h, // $\frac{6 \cdot (\xi - \xi^2)}{h_i}$
            3 => -2.0 * point + 3.0 * point * point, // $-2\xi + 3\xi^2$
            _ => throw new ArgumentOutOfRangeException(nameof(number), number, "Not expected function number!")
        };

    public double GetDdPsi(int number, double point, double h = 1.0)
        => number switch
        {
            0 => -6.0 * (1.0 - 2.0 * point) / (h * h), // $\frac{-6 \cdot (1 - 2\xi)}{h_i^2}$,
            1 => (-4.0 + 6.0 * point) / h, // $\frac{-4 + 6\xi}{h_i}$
            2 => 6.0 * (1.0 - 2.0 * point) / (h * h), // $\frac{6 \cdot (1 - 2\xi)}{h_i^2}$
            3 => (-2.0 + 6.0 * point) / h, // $\frac{-2 + 6\xi}{h_i}$
            _ => throw new ArgumentOutOfRangeException(nameof(number), number, "Not expected function number!")
        };
}

public readonly record struct LinearBasis : IBasis2D
{
    public int Size => 4;

    public double GetPsi(int number, Point2D point)
        => number switch
        {
            0 => (1.0 - point.X) * (1.0 - point.Y),
            1 => point.X * (1.0 - point.Y),
            2 => (1.0 - point.X) * point.Y,
            3 => point.X * point.Y,
            _ => throw new ArgumentOutOfRangeException(nameof(number), number, "Not expected function number!")
        };

    public double GetDPsi(int number, int varNumber, Point2D point)
        => varNumber switch
        {
            0 => number switch
            {
                0 => point.Y - 1.0,
                1 => 1.0 - point.Y,
                2 => -point.Y,
                3 => point.Y,
                _ => throw new ArgumentOutOfRangeException(nameof(number), number, "Not expected function number!")
            },
            1 => number switch
            {
                0 => point.X - 1.0,
                1 => -point.X,
                2 => 1.0 - point.X,
                3 => point.X,
                _ => throw new ArgumentOutOfRangeException(nameof(number), number, "Not expected function number!")
```

```
            },
            _ => throw new ArgumentOutOfRangeException(nameof(varNumber), varNumber,
↪   "Not expected var number!")
        };
}
```

Boundaries.cs

```
namespace Magnetostatics.src.FEM;

public interface IBoundary
{
    int Node { get; }
    double Value { get; set; }
}

public class DirichletBoundary : IBoundary
{
    public int Node { get; }
    public double Value { get; set; }

    public DirichletBoundary(int node, double value) => (Node, Value) = (node, value);
}

public readonly record struct BoundaryParameters
{
    [JsonProperty("Left border"), JsonRequired]
    public required byte LeftBorder { get; init; }
    [JsonProperty("Right border"), JsonRequired]
    public required byte RightBorder { get; init; }
    [JsonProperty("Bottom border"), JsonRequired]
    public required byte BottomBorder { get; init; }
    [JsonProperty("Top border"), JsonRequired]
    public required byte TopBorder { get; init; }

    public static BoundaryParameters ReadJson(string jsonPath)
    {
        if (!File.Exists(jsonPath))
        {
            throw new("File does not exist");
        }

        using var sr = new StreamReader(jsonPath);
        return JsonConvert.DeserializeObject<BoundaryParameters>(sr.ReadToEnd());
    }
}

public interface IBoundaryHandler
{
    IEnumerable<IBoundary> Process();
}

public class LinearBoundaryHandler : IBoundaryHandler
{
    private readonly BoundaryParameters _parameters;
    private readonly MeshParameters _meshParameters;

    public LinearBoundaryHandler(BoundaryParameters parameters, MeshParameters
↪   meshParameters)
        => (_parameters, _meshParameters) = (parameters, meshParameters);
```

```
52
53      public IEnumerable<IBoundary> Process() // for now only Dirichlet
54      {
55          if (_parameters.TopBorder == 1)
56          {
57              int startingNode = (_meshParameters.SplitsX.Sum() + 1) *
    ↪  _meshParameters.SplitsY.Sum();
58
59              for (int i = 0; i < _meshParameters.SplitsX.Sum() + 1; i++)
60              {
61                  yield return new DirichletBoundary(startingNode + i, 0.0);
62              }
63          }
64
65          if (_parameters.BottomBorder == 1)
66          {
67              for (int i = 0; i < _meshParameters.SplitsX.Sum() + 1; i++)
68              {
69                  yield return new DirichletBoundary(i, 0.0);
70              }
71          }
72
73          if (_parameters.LeftBorder == 1)
74          {
75              for (int i = 0; i < _meshParameters.SplitsY.Sum() + 1; i++)
76              {
77                  yield return new DirichletBoundary(i * (_meshParameters.SplitsX.Sum()
    ↪  + 1), 0.0);
78              }
79          }
80
81          if (_parameters.RightBorder != 1) yield break;
82          {
83              for (int i = 0; i < _meshParameters.SplitsY.Sum() + 1; i++)
84              {
85                  yield return new DirichletBoundary(
86                      i * _meshParameters.SplitsX.Sum() + _meshParameters.SplitsX.Sum()
    ↪  + i, 0.0);
87              }
88          }
89      }
90  }
```

Geometry.cs

```
1   namespace Magnetostatics;
2
3   public class Point2DJsonConverter : JsonConverter
4   {
5       public override bool CanConvert(Type objectType) => typeof(Point2D) == objectType;
6
7       public override object ReadJson(JsonReader reader, Type objectType, object?
    ↪  existingValue,
8           JsonSerializer serializer)
9       {
10          if (reader.TokenType == JsonToken.StartArray)
11          {
12              var array = JArray.Load(reader);
```

```
            if (array.Count == 2) return new Point2D(array[0].Value<double>(),
↪   array[1].Value<double>());
            throw new FormatException($"Wrong vector length({array.Count})!");
        }

        if (Point2D.TryParse((string?)reader.Value ?? "", out var point)) return
↪   point;
        throw new FormatException($"Can't parse({(string?)reader.Value}) as
↪   Vector2D!");
    }

    public override void WriteJson(JsonWriter writer, object? value, JsonSerializer
↪   serializer)
    {
        value ??= new Point2D();
        var p = (Point2D)value;
        writer.WriteRawValue($"[{p.X}, {p.Y}]");
        // [[0, 0],[0, 0]] // runtime exception if use method WriteRaw()
        // [[0, 0][0, 0]]
    }
}

[JsonConverter(typeof(Point2DJsonConverter))]
public readonly record struct Point2D(double X, double Y)
{
    public static bool TryParse(string line, out Point2D point)
    {
        var words = line.Split(new[] { ' ', ',', '(', ')' },
↪   StringSplitOptions.RemoveEmptyEntries);
        if (words.Length != 3 || !float.TryParse(words[1], out var x) ||
↪   !float.TryParse(words[2], out var y))
        {
            point = default;
            return false;
        }

        point = new(x, y);
        return true;
    }

    public static Point2D operator +(Point2D a, Point2D b) => new(a.X + b.X, a.Y +
↪   b.Y);

    public static Point2D operator -(Point2D a, Point2D b) => new(a.X - b.X, a.Y -
↪   b.Y);

    public static Point2D operator *(Point2D p, double value) => new(p.X * value, p.Y
↪   * value);

    public static Point2D operator /(Point2D p, double value) => new(p.X / value, p.Y
↪   / value);

    public static Point2D operator +(Point2D p, (double, double) value) => new(p.X +
↪   value.Item1, p.Y + value.Item2);

    public static Point2D operator -(Point2D p, (double, double) value) => new(p.X -
↪   value.Item1, p.Y - value.Item2);

    public static implicit operator Point2D((double, double) value) =>
↪   new(value.Item1, value.Item2);
```

```
}

public class IntervalJsonConverter : JsonConverter
{
    public override bool CanConvert(Type objectType) => typeof(Interval) ==
    ↪ objectType;

    public override object ReadJson(JsonReader reader, Type objectType, object?
    ↪ existingValue,
        JsonSerializer serializer)
    {
        if (reader.TokenType == JsonToken.StartArray)
        {
            var array = JArray.Load(reader);
            if (array.Count == 2) return new Interval(array[0].Value<double>(),
    ↪ array[1].Value<double>());
            throw new FormatException($"Wrong vector length({array.Count})!");
        }

        if (Interval.TryParse((string?)reader.Value ?? "", out var interval)) return
    ↪ interval;
        throw new FormatException($"Can't parse({(string?)reader.Value}) as
    ↪ Interval!");
    }

    public override void WriteJson(JsonWriter writer, object? value, JsonSerializer
    ↪ serializer)
    {
        value ??= new Interval();
        var interval = (Interval)value;
        serializer.Serialize(writer, interval);
    }
}

[JsonConverter(typeof(IntervalJsonConverter))]
public readonly record struct Interval(
    [property: JsonProperty("Left border"), JsonRequired]
    double LeftBorder,
    [property: JsonProperty("Right border"), JsonRequired]
    double RightBorder)
{
    [JsonIgnore] public double Center => (LeftBorder + RightBorder) / 2.0;
    [JsonIgnore] public double Length => Math.Abs(RightBorder - LeftBorder);

    public static bool TryParse(string line, out Interval interval)
    {
        var words = line.Split(new[] { ' ', ',', '[', ']' },
    ↪ StringSplitOptions.RemoveEmptyEntries);
        if (words.Length != 2 || !float.TryParse(words[0], out var x) ||
    ↪ !float.TryParse(words[1], out var y))
        {
            interval = default;
            return false;
        }

        interval = new(x, y);
        return true;
    }

    public bool IsContain(double point)
```

```
        => point >= LeftBorder && point <= RightBorder;
}

public readonly record struct Rectangle(Point2D LeftBottom, Point2D RightTop)
{
    public Point2D LeftTop { get; } = new(LeftBottom.X, RightTop.Y);
    public Point2D RightBottom { get; } = new(RightTop.X, LeftBottom.Y);
}

public class FiniteElement
{
    public IReadOnlyList<int> Nodes { get; }
    public int AreaNumber { get; }

    public FiniteElement(IReadOnlyList<int> nodes, int areaNumber)
        => (Nodes, AreaNumber) = (nodes, areaNumber);
}
```

Spline.cs

```
namespace Magnetostatics.src.Spline;

public class Spline
{
    public class SplineBuilder
    {
        private readonly Spline _spline = new();

        public SplineBuilder SetParameters((double Alpha, double Beta) parameters)
        {
            _spline._parameters = parameters;
            return this;
        }

        public SplineBuilder SetPoints(Point2D[] points)
        {
            _spline._points = points;
            return this;
        }

        public SplineBuilder SetPartitions(int partitions)
        {
            _spline._partitions = partitions;
            return this;
        }

        public SplineBuilder SetBasis(IBasis1D basis)
        {
            _spline._basis = basis;
            return this;
        }

        public SplineBuilder SetIntegrator(Integration integrator)
        {
            _spline._integrator = integrator;
            return this;
        }

```

```
        public static implicit operator Spline(SplineBuilder builder)
            => builder._spline;
    }

    private Interval[] _elements = default!;
    private Point2D[] _points = default!;
    private SevenDiagonalMatrix _globalMatrix = default!;
    private Matrix _localMatrix = default!;
    private FEM.Vector<double> _vector = default!;
    private List<Point2D> _result = default!;
    private IBasis1D _basis = default!;
    private Integration _integrator = default!;
    private (double Alpha, double Beta) _parameters;
    private int _partitions;

    public IReadOnlyList<Point2D> InitialPoints => _points;
    public IReadOnlyList<Point2D> Result => _result;

    private void Init()
    {
        _globalMatrix = new(_elements.Length * 2 + 2);
        _localMatrix = new(_basis.Size);
        _vector = new(_globalMatrix.Size);
        _result = new();
    }

    private void BuildMesh()
    {
        _elements = new Interval[_partitions];
        _points = _points.OrderBy(p => p.X).ToArray();

        if (_partitions == 1)
        {
            _elements[0] = new(_points[0].X, _points[^1].X);
            return;
        }

        double step = (_points.MaxBy(p => p.X)!.X - _points.MinBy(p => p.X)!.X) /
↪   _partitions;
        _elements[0] = new(_points[0].X, _points[0].X + step);

        for (int i = 1; i < _elements.Length; i++)
        {
            _elements[i] = new(_elements[i - 1].RightBorder, _elements[i -
↪   1].RightBorder + step);
        }
    }

    public void Compute()
    {
        BuildMesh();
        Init();
        AssemblyMatrix();
        var gaussSeidel = new GaussSeidel(1000, 1E-15, 1.23);
        _vector = gaussSeidel.Compute(_globalMatrix, _vector);
        // ValuesAtPoints();
    }

    public double ValueAtPoint(double point)
    {
```

```
        int ielem = -1;
        double result = 0.0;

        for (int i = 0; i < _elements.Length; i++)
        {
            if (!_elements[i].IsContain(point)) continue;
            ielem = i;
            break;
        }

        if (ielem == -1) throw new("Not supported exception!");

        double x = (point - _elements[ielem].LeftBorder) / _elements[ielem].Length;

        for (int i = 0; i < _basis.Size; i++)
        {
            result += _vector[2 * ielem + i] * _basis.GetPsi(i, x,
↪  _elements[ielem].Length);
        }

        return result;
    }

    private void AssemblyMatrix()
    {
        int[] checker = new int[_points.Length];
        checker.Fill(1);

        for (int ielem = 0; ielem < _elements.Length; ielem++)
        {
            for (int ipoint = 0; ipoint < _points.Length; ipoint++)
            {
                if (!_elements[ielem].IsContain(_points[ipoint].X) || checker[ipoint]
↪  != 1) continue;

                checker[ipoint] = -1;
                double x = (_points[ipoint].X - _elements[ielem].LeftBorder) /
↪  _elements[ielem].Length;

                for (int i = 0; i < _basis.Size; i++)
                {
                    _vector[2 * ielem + i] += _points[ipoint].Y * _basis.GetPsi(i, x,
↪  _elements[ielem].Length);

                    for (int j = 0; j < _basis.Size; j++)
                    {
                        double Function1(double point, double h)
                        {
                            var dFi1 = _basis.GetDPsi(i, point, h);
                            var dFi2 = _basis.GetDPsi(j, point, h);

                            return dFi1 * dFi2;
                        }

                        double Function2(double point, double h)
                        {
                            var ddFi1 = _basis.GetDdPsi(i, point, h);
                            var ddFi2 = _basis.GetDdPsi(j, point, h);

                            return ddFi1 * ddFi2;
```

```
                        }

                        _localMatrix[i, j] += _basis.GetPsi(i, x,
↪ _elements[ielem].Length) *
                                              _basis.GetPsi(j, x,
↪ _elements[ielem].Length) +
                                              _parameters.Alpha *
↪ _integrator.Gauss1D(Function1, _elements[ielem]) +
                                              _parameters.Beta *
↪ _integrator.Gauss1D(Function2, _elements[ielem]);
                    }
                }
            }

            for (int i = 0; i < _localMatrix.Size; i++)
            {
                _globalMatrix[0, 2 * ielem + i] += _localMatrix[i, i];
            }

            for (int i = 0; i < _localMatrix.Size - 1; i++)
            {
                _globalMatrix[1, 2 * ielem + i] += _localMatrix[i, i + 1];
                _globalMatrix[4, 2 * ielem + i] += _localMatrix[i, i + 1];
            }

            for (int i = 0; i < _localMatrix.Size - 2; i++)
            {
                _globalMatrix[2, 2 * ielem + i] = _localMatrix[i, i + 2];
                _globalMatrix[5, 2 * ielem + i] = _localMatrix[i, i + 2];
            }

            for (int i = 0; i < _localMatrix.Size - 3; i++)
            {
                _globalMatrix[3, 2 * ielem + i] = _localMatrix[i, i + 3];
                _globalMatrix[6, 2 * ielem + i] = _localMatrix[i, i + 3];
            }

            _localMatrix.Clear();
        }
    }

    private void ValuesAtPoints()
    {
        double sum = 0.0;

        for (int ielem = 0; ielem < _elements.Length; ielem++)
        {
            Point2D changedPoint = new(_elements[ielem].LeftBorder, 0.0);

            do
            {
                double x = (changedPoint.X - _elements[ielem].LeftBorder) /
↪ _elements[ielem].Length;

                for (int i = 0; i < _basis.Size; i++)
                {
                    sum += _vector[2 * ielem + i] * _basis.GetPsi(i, x,
↪ _elements[ielem].Length);
                }

```

```
                _result.Add(changedPoint with { Y = sum });

                changedPoint += (0.0005, 0.0);
                sum = 0.0;
            } while (_elements[ielem].IsContain(changedPoint.X));
        }
    }

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

Vector.cs

```
namespace Magnetostatics.src.FEM;

public class Vector<T> : IEnumerable<T> where T : INumber<T>, IRootFunctions<T>
{
    private readonly T[] _storage;
    public int Length { get; }

    public T this[int idx]
    {
        get => _storage[idx];
        set => _storage[idx] = value;
    }

    public Vector(int length)
        => (Length, _storage) = (length, new T[length]);

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

        for (int i = 0; i < a.Length; i++)
        {
            result += a[i] * b[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.CreateChecked(constant);
        }

        return result;
    }

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

```

```
        for (int i = 0; i < a.Length; i++)
        {
            result[i] = a[i] + b[i];
        }

        return result;
    }

    public static Vector<T> operator -(Vector<T> a, Vector<T> b)
    {
        Vector<T> result = new(a.Length);

        for (int i = 0; i < a.Length; i++)
        {
            result[i] = a[i] - b[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 static Vector<T> Copy(Vector<T> otherVector)
    {
        Vector<T> newVector = new(otherVector.Length);

        Array.Copy(otherVector._storage, newVector._storage, otherVector.Length);

        return newVector;
    }

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

    public T Norm() => T.Sqrt(_storage.Aggregate(T.Zero, (current, t) => current + t
↪   * t));

    public ImmutableArray<T> ToImmutableArray() => ImmutableArray.Create(_storage);

    public IEnumerator<T> GetEnumerator() =>
↪   ((IEnumerable<T>)_storage).GetEnumerator();

    IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();

    public void Add(IEnumerable<T> collection)
    {
        var enumerable = collection as T[] ?? collection.ToArray();

        if (Length != enumerable.Length)
```

```
        {
            throw new ArgumentOutOfRangeException(nameof(collection), "Sizes of
↪   vector and collection not equal");
        }

        for (int i = 0; i < Length; i++)
        {
            _storage[i] = enumerable[i];
        }
    }
}
```

SLAE.cs

```
namespace Magnetostatics.src.Spline;

using FEM;

public static class SLAE
{
    public static Vector<T> Compute<T>(Matrix matrix, Vector<T> f)
        where T : INumber<T>, IRootFunctions<T>
    {
        Vector<T> x = new(f.Length);
        Vector<T>.Copy(f, x);

        for (int i = 0; i < f.Length; i++)
        {
            T sum = T.Zero;

            for (int k = 0; k < i; k++)
                sum += T.CreateChecked(matrix[i, k]) * x[k];

            x[i] = (f[i] - sum) / T.CreateChecked(matrix[i, i]);
        }

        for (int i = x.Length - 1; i >= 0; i--)
        {
            T sum = T.Zero;

            for (int k = i + 1; k < x.Length; k++)
            {
                sum += T.CreateChecked(matrix[i, k]) * x[k];
            }

            x[i] -= sum;
        }

        return x;
    }
}
```

MathDependence.cs

```
namespace Magnetostatics;

public class MathDependence
```

```
{
    public string Argument { get; }
    public string Function { get; }
    public (double Function, double Argument)[]? Data { get; private set; }

    public MathDependence(string function, string argument)
        => (Function, Argument) = (function, argument);

    public void LoadData(string path)
    {
        if (!File.Exists(path)) throw new("File does not exist");

        using var sr = new StreamReader(path);

        Data = sr.ReadToEnd().Split("\n").Select(line => line.Split(" ",
↪   StringSplitOptions.RemoveEmptyEntries))
            .Select(line => (double.Parse(line[0]), double.Parse(line[1]))).ToArray();
    }

    public override string ToString() => $"Dependence is {Function}({Argument})";
}
```

ArrayHelper.cs

```
namespace Magnetostatics;

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;
        }
    }
}
```