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

Кафедра прикладной математики

ОТЧЕТ ПО УЧЕБНОЙ ПРАКТИКЕ: ПРАКТИКЕ ПО ПОЛУЧЕНИЮ ПЕРВИЧНЫХ ПРОФЕССИОНАЛЬНЫХ УМЕНИЙ И НАВЫКОВ

Направление подготовки: 01.04.02 Прикладная математика и информатика

Выполнил:

Студент Жиляков Александр Константинович
(Ф.И.О.)

Группа ПММ-72

Факультет ПМИ

_____
подпись

«___» _____ 20__ г.

Проверил:

Руководитель от НГТУ Персова Марина Геннадьевна
(Ф.И.О.)

Балл: _____, ECTS_____,

Оценка _____
«отлично», «хорошо», «удовлетворительно», «неуд.»

_____
подпись

«___» _____ 20__ г.

Новосибирск 2017

Индивидуальное задание на учебную практику: практику по получению первичных профессиональных умений и навыков

Студент группы ПММ-72

Место прохождения практики: НГТУ, Научно-образовательный центр «Моделирование электромагнитных технологий»

Задачи практики:

Реализовать метод ускорения Андерсона для нелинейной задачи магнитостатики (задачи Пуассона с нелинейно зависящим от решения коэффициентом диффузии); сравнить производительность нелинейных методов решение с применением ускорения и без него.

Ожидаемые результаты практики:

Ожидаем, что ускорение Андерсона можно рассматривать как альтернативу более дорогим итерациям Ньютона, которые обычно используются вместе с методом простой итерации (метод Ньютона имеет второй порядок сходимости (локально) и запускается после того, как невязка по нелинейности достаточно мала).

Задание выдал: _____ ФИО руководителя практики от НГТУ

_____ ФИО руководителя практики от профильной организации

Задание принято к исполнению: _____ «___» _____ 2017 г.

(подпись студента)

1 Описание задачи

Расчётная область представлена на рисунке 1. При заданной плотности тока $\mathbf{J} = (0, 0, J_z)$, $J_z := \pm j \left[A\, m^{-2}\right]$, в проводах и магнитной проницаемости $\mu \left[N\, A^{-2}\right]$ железного тела, требуется найти результирующее поле магнитной индукции $\mathbf{B}\,[T]$.

Физика данной задачи описывается системой Максвелла. Введём в рассмотрение вектор-потенциал $\mathbf{A}$ через

$$\mathbf{B} = \nabla \times \mathbf{A};$$

тогда уравнения Максвелла $\nabla \times \frac{1}{\mu}\mathbf{B} = \mathbf{J}$, $\nabla \cdot \mathbf{B} = 0$ можно переписать в терминах вектор-потенциала

$$\nabla \times \frac{1}{\mu} \nabla \times \mathbf{A} = \mathbf{J}.$$

Предполагая, что магнит достаточно длинный в z-направлении и учитывая тот факт, что поле $\mathbf{J}$ имеет только одну ненулевую z-компоненту, $J_z = J_z(x, y)$, мы можем свести последнее уравнение к уравнению Пуассона

$$-\nabla \cdot (\frac{1}{\mu} \nabla A_z) = J_z; \tag{1}$$

из физических соображений задача (1) оснащается однородными краевыми условиями Неймана (на нижней границе) и Дирихле (на остальной части). В результате КЭ дискретизации задача сведётся к решению линейной

$$\mathbf{S}\,\mathbf{x} = \mathbf{b} \tag{2}$$

или нелинейной

$$\mathbf{S}(\mathbf{x})\,\mathbf{x} = \mathbf{b} \tag{3}$$

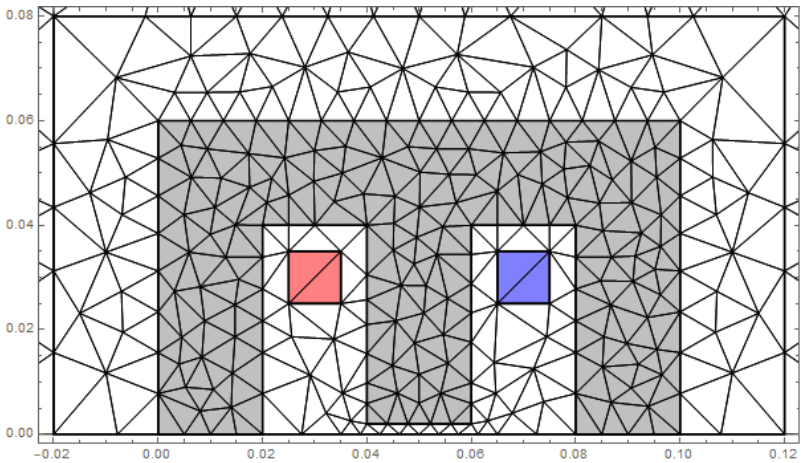
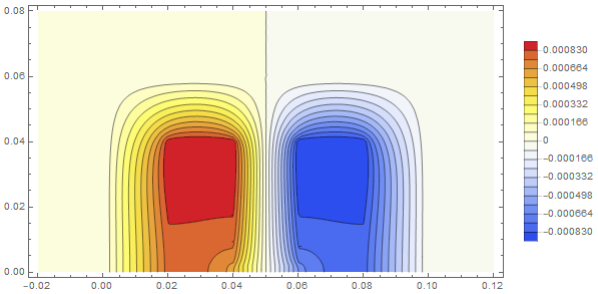
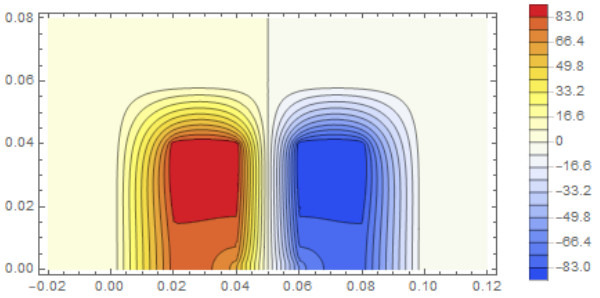


Рис. 1: Сечение в плоскости $x\,O\,y$: железный магнит (выделен серым цветом), медные провода (выделены красным и синим)



(a) $|J_z| = 10^6\, A\, m^{-2}$



(b) $|J_z| = 10^{11}\, A\, m^{-2}$

Рис. 2: Конечно элементные интерполянты вектор-потенциала A_z, полученные при решении линейной задачи (2)

системы уравнений в зависимости от того, каким образом магнитная проницаемость зависит от вектор-потенциала.

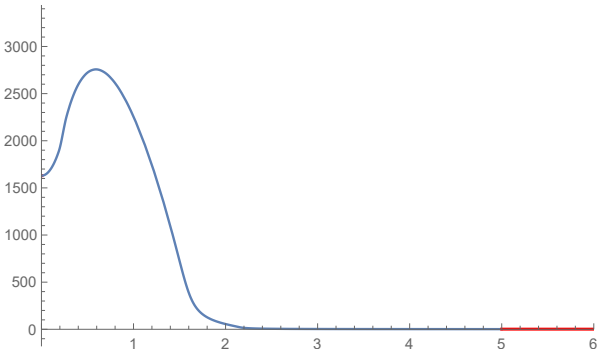
Представим магнитную проницаемость в виде $\mu = \mu_0\, \hat{\mu}$, где $\mu_0 := 4\,\pi\, 10^{-7}\, \left[N\, A^{-2}\right]$ есть проницаемость вакуума. Мы можем положить $\hat{\mu}_{\text{iron}} = 1000$ — тогда (1) сведётся к линейной системе (2). В таком случае при изменении правой части уравнения меняться будет только масштаб решения (см. рисунок 2).

В действительности же $\hat{\mu}_{\text{iron}}$ нелинейно зависит $\mathbf{B}$ (см. рисунок 3) — $\mu_{\text{iron}} = \mu_{\text{iron}}(||\mathbf{B}|| = ||\nabla A_z||)$. При таком раскладе (1) сведётся к нелинейной системе (2). Далее мы рассмотрим методы решения нелинейных систем.

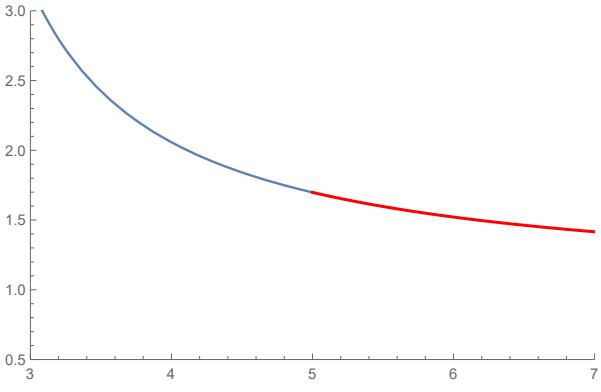
2 Ускорение Андерсона

Пусть требуется решить нелинейную задачу

$$\mathbf{f}(\mathbf{x}) = 0. \tag{4}$$



(a) $\hat{\mu}_{\text{iron}} = \hat{\mu}_{\text{iron}}(||\mathbf{B}||)$



(b) Часть с экстраполяцией

Рис. 3: Интерполянт для магнитной проницаемости, построенный по реальным данным

От задачи поиска корня (4), записанной в терминах оператора невязки $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$, можно перейти к эквивалентной задаче поиска фиксированной точки

$$\mathbf{x} = \mathbf{g}(\mathbf{x}), \tag{5}$$

записанной в терминах оператора итераций $\mathbf{g}$. (Сделать это можно, например, положив $\mathbf{g}(\mathbf{x}) \coloneqq \mathbf{x} - \gamma\,\mathbf{f}(\mathbf{x})$.)

Выбрав начальное приближение $\mathbf{x}^{(0)}$, для решения (5) можно применить **метод простой итерации**

$$\mathbf{x}^{(k+1)} = \mathbf{g}(\mathbf{x}^{(k)}). \tag{6}$$

Для ускорения сходимости (6) можно предложить следующую идею. Будем искать новое приближение используя $(m+1)$ предыдущих приближений:

$$\mathbf{x}^{(k+1)} = \sum_{i=0}^{m} \alpha_i\,\mathbf{g}(\mathbf{x}^{(k-i)}). \tag{7}$$

Метод (7) будет состоятельным тогда и только тогда, когда

$$\sum_{i=0}^{m} \alpha_i = 1.$$

Коэффициенты α_i будем искать решая задачу минимизации

$$\boldsymbol{\alpha} = \underset{\sum_{i=0}^{m}\alpha_i=1}{\arg\min} \left\| \sum_{i=0}^{m} \alpha_i\,\mathbf{f}(\mathbf{x}^{(k-i)}) \right\|_2. \tag{8}$$

Представим коэффициенты α_i в виде

$$\begin{aligned} \alpha_0 &= 1 - \beta_1, \\ \alpha_1 &= \beta_1 - \beta_2, \\ &\vdots \\ \alpha_{m-1} &= \beta_{m-1} - \beta_m, \\ \alpha_m &= \beta_m; \end{aligned} \tag{9}$$

тогда задачу минимизации с ограничениями (8) можно переписать как задачу минимизации без ограничений — свести её к **задаче о наименьших квадратах**

$$\begin{aligned} \boldsymbol{\beta} &= \underset{\boldsymbol{\beta}\in\mathbb{R}^m}{\arg\min} \left\| \mathbf{f}(\mathbf{x}^{(k)}) - \sum_{i=1}^{m} \beta_i \left(\mathbf{f}(\mathbf{x}^{(k-i)}) - \mathbf{f}(\mathbf{x}^{(k-i+1)})\right) \right\|_2 \\ &= \underset{\boldsymbol{\beta}\in\mathbb{R}^m}{\arg\min} \left\| \mathbf{f}(\mathbf{x}^{(k)}) - \mathbf{F}\,\boldsymbol{\beta} \right\|_2, \end{aligned} \tag{10}$$

которая эквивалентна решению системы

$$\left(\mathbf{F}^T\,\mathbf{F}\right)\,\boldsymbol{\beta} = \mathbf{F}^T\,\mathbf{f}(\mathbf{x}^{(k)}). \tag{11}$$

Здесь через $\mathbf{F}$ обозначена матрица из разностей нелинейных невязок

$$\mathbf{F} \coloneqq \left[\mathbf{f}(\mathbf{x}^{(k)}) - \mathbf{f}(\mathbf{x}^{(k-1)}) \;\; |\ldots|\;\; \mathbf{f}(\mathbf{x}^{(k-m+1)}) - \mathbf{f}(\mathbf{x}^{(k-m)})\right] \in \mathbb{R}^{m\times n}.$$

Таким образом, **метод ускорения Андерсона** (или метод смешивания Андерсона) для задачи (6) записывается в следующем виде. Используя информацию об $(m+1)$ предыдущих приближениях, найти новое приближение в виде

$$\mathbf{x}^{(k+1)} = \sum_{i=0}^{m} \alpha_i\,\mathbf{g}(\mathbf{x}^{(k-i)}),$$

где коэффициенты $\boldsymbol{\alpha}$ выражаются через коэффициенты $\boldsymbol{\beta}$ (9), которые находятся из решения нормальной системы (11).

Обычно количество смешиваний (размер матрицы системы (11)) m выбирается в диапазоне $10-50$, $m \ll n$, и для составления матрицы требуется $O(n)$ операций на каждой нелинейной итерации метода (6); для небольшой системы достаточно использовать LU факторизацию / метод исключения Гаусса с partial pivoting (используется в нашей реализации)[1]. Заметим, что в обычном методе (6) мы исполняем как минимум $O(n)$ операций на каждой нелинейной итерации, реассемблируя конечно-элементную матрицу, так что ускоренный метод (7) асимптотически эквивалентен методу (6) в терминах необходимых операций и затрат ОЗУ на каждой нелинейной итерации.

Выбор коэффициентов в (8) обусловлен следующим наблюдением. Предположим, что приближения $\mathbf{x}^{(k)}, \ldots, \mathbf{x}^{(k-m+1)}$ достаточно близки друг к другу и

[1]Другой вариант (более подходящий с точки зрения устойчивости при работе в конечной арифметике) решения задачи наименьших квадратов (10) — метод ортогональных преобразований с использованием отражений Хаусхолдера.

дефекты $||\mathbf{x}^{(i)} - \mathbf{g}(\mathbf{x}^{(i)})||_2$ достаточно малы. Тогда

$$\begin{aligned}
\mathbf{f}(\mathbf{x}^{(k+1)}) &= \mathbf{f}\left(\sum_{i=0}^{m} \alpha_i\, \mathbf{g}(\mathbf{x}^{(k-i)})\right) \\
&= \mathbf{f}\left(\sum_{i=0}^{m} \alpha_i\, \mathbf{x}^{(k-i)} - \sum_{i=0}^{m} \alpha_i\, \left(\mathbf{x}^{(k-i)} - \mathbf{g}(\mathbf{x}^{(k-i)})\right)\right) \\
&\approx \mathbf{f}\left(\sum_{i=0}^{m} \alpha_i\, \mathbf{x}^{(k-i)}\right) \\
&= \mathbf{f}\left(\mathbf{x}^{(k)} - \sum_{i=1}^{m} \beta_i\, \left(\mathbf{x}^{(k-i+1)} - \mathbf{x}^{(k-i)}\right)\right) \\
&\approx \mathbf{f}(\mathbf{x}^{(k)}) - \sum_{i=1}^{m} \beta_i\, \mathbf{f}'(\mathbf{x}^{(k)})\, \left(\mathbf{x}^{(k-i+1)} - \mathbf{x}^{(k-i)}\right) \\
&\approx \mathbf{f}(\mathbf{x}^{(k)}) - \sum_{i=1}^{m} \beta_i\, \left(\mathbf{f}(\mathbf{x}^{(k-i+1)}) - \mathbf{f}(\mathbf{x}^{(k-i)})\right) \\
&= \sum_{i=0}^{m} \alpha_i\, \mathbf{f}(\mathbf{x}^{(k-i)}).
\end{aligned} \tag{12}$$

Таким образом, решая (8) в методе ускорения Андерсона мы надеемся минимизировать невязку $(k+1)$-го приближения. Поэтому метод ускорения Андерсона иногда называют нелинейным аналогом GMRES.

3 Численные результаты

Для задачи (3) оператор невязки имеет вид

$$\mathbf{f}(\mathbf{x}) \coloneqq \mathbf{b} - \mathbf{S}(\mathbf{x})\, \mathbf{x}.$$

В качестве оператора итераций можно выбрать, например,

$$\mathbf{g}(\mathbf{x}) \coloneqq \mathbf{S}^{-1}(\mathbf{x})\, \mathbf{b} \tag{13}$$

или

$$\mathbf{g}(\mathbf{x}) \coloneqq \mathbf{x} - \left[\mathbf{f}'(\mathbf{x})\right]^{-1} \mathbf{f}(\mathbf{x}). \tag{14}$$

Метод (13) называется **методом Пикарда** (так же методом простой итерации или упрощённым методом Ньютона), а метод (14) — **методом Ньютона**.

На каждой итерации обоих методов необходимо вычислять результат действия матрицы, обратной к дискретному эллиптическому оператору (симметричная положительно определённая матрица); мы делаем это с помощью метода сопряжённых градиентов с неполным LDL^T переобуславливателем. Метод

Таблица 1: Количество итераций и время работы для разных нелинейных методов решения системы (3)

Плотность тока $\lvert J_z\rvert$, $\left[A\,m^{-2}\right]$	10^6	10^7	10^8	10^9	10^{10}	10^{11}
Метод Пикарда (13)	1 0.4 s	6 1.3 s	— —	— —	203 25.2 s	7 1 s
Метод Пикарда с релаксацией (15)	1 0.5 s	6 2 s	84 71 s	70 52.4 s	164 54.3 s	7 2.6 s
Ускорение Андерсона (7)	1+0 0.7 s	0+4 1 s	22+20 20 s	21+21 14.5 s	35+6 11.4 s	4+2 1.9 s

Ньютона (14) с вычислительной точки зрения дороже, чем (13), потому что он требует вычисления матрицы Якоби; кроме того метод Ньютона имеет *локальный* квадратичный порядок сходимости.

Для решения больших нелинейных систем обычно используют следующий подход: обрушают нелинейную невязку с помощью относительно дешёвого метода, — например, (13), — а затем запускают метод Ньютона.

В данной работе мы вместо метода Ньютона для ускорения сходимости использовали более дешёвый с вычислительной точки зрения метод смешивания Андерсона[2] для ускорения сходимости метода Пикарда (13).

На практике отображение $\mathbf{g}$ редко является сжимающим, поэтому для обрушения невязки мы использовали метод Пикарда с релаксацией

$$\mathbf{g}_\omega(\mathbf{x}) \coloneqq \omega\,\mathbf{S}^{-1}(\mathbf{x})\,\mathbf{b} + (1-\omega)\,\mathbf{x}, \tag{15}$$

$0 < \omega \leq 1$. Здесь параметр ω выбирался адаптивно — начиная с единичного значения, мы делили его на 10 до тех пор, пока норма новой нелинейной невязки не становилась меньше текущей. Метод (15) дороже (13): на одной итерации невязку приходится вычислять несколько раз.

Результаты приведены в таблице 1. Использовались линейные Лагранжевы элементы первого порядка (сетка изображена на рисунке 1, 362 степени свободы). Критерий остановки: $\mathbf{f}(\mathbf{x}^{(k)}) < 10^{-8}$. Для метода смешивания Андерсона $k = i+j$ означает i итераций Пикарда с *релаксацией*[3] ($\mathbf{f}(\mathbf{x}^{(i)}) < 10^{-4}$) и j *ускоренных* итераций Пикарда. Мы использовали количество смешиваний $m = 10$.

Из таблицы видно, что степень «сложности» нелинейной задачи сильно зависит от применяемой плотности тока: при плотности вне промежутка $10^8 - 10^{10}\left[A\,m^{-2}\right]$ оказывается достаточным использовать простой метод Пикарда; внутри промежутка метод оказывается неработоспособным.

Применение метода Пикарда с релаксацией позволяет решить задачу, однако

[2]Ускорение Андерсона подходит для любого метода, записанного в форме (5) — в том числе и для метода Ньютона (14). В данной работе мы рассматриваем только производительность ускоренного метода Пикарда (13).

[3]Из (12) видно, что ускорение Андерсона применимо при определённых предположениях, поэтому для обрушения невязки мы использовали метод Пикарда с релаксацией.

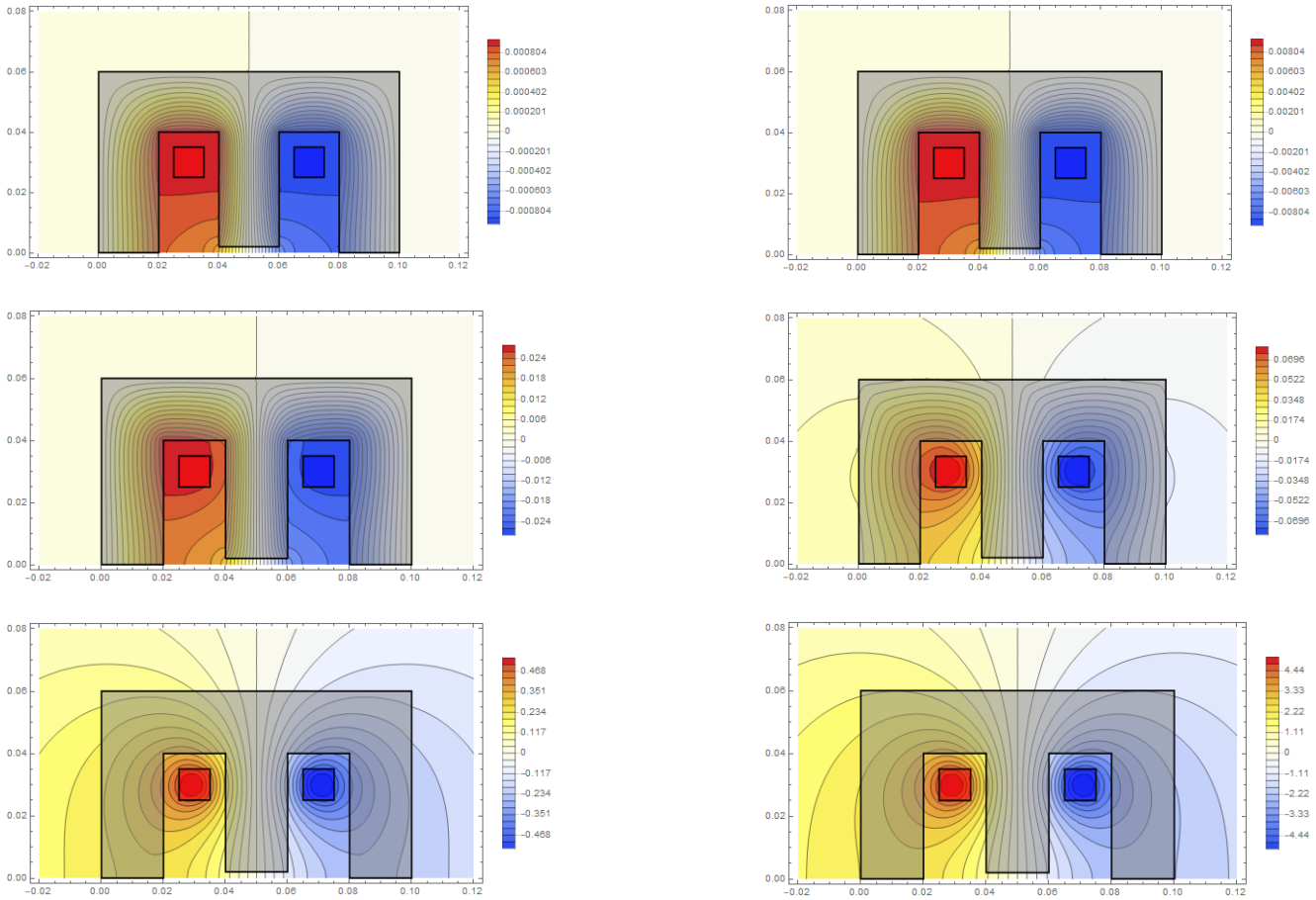


Рис. 4: Конечно элементные интерполянты вектор-потенциала A_z, полученные при решении нелинейной задачи (3): $|J_z| = 10^6, 10^7, \ldots, 10^{11}\, A\, m^{-2}$

временные затраты получается колоссальными (с учётом скромного размера системы, $n = 362$). Из таблицы видно, что применение ускорения Андерсона позволяет методу сойтись в > 3 раза быстрее.

Решения представлены на рисунке 4.

4 Исходный текст программы

4.1 NonlinearSolvers.hpp

Модуль с имплементацией общего нелинейного решателя в форме (6) и метода ускорения Андерсона (7).

```
1  #pragma once
2  #include <boost/circular_buffer.hpp>
3  #include "DenseMatrix.hpp"
4  #include "Mapping.hpp"
5
6  namespace NonlinearSolvers {
7
8    struct FixedPointData {
9      Operator f; // residual operator
10     Operator g; // iteration operator
11   };
```

```
12
13  inline void logFixedPoint(double r, Index i, Index max = 1) {
14    auto& logger = SingletonLogger::instance();
15    auto& w = std::setw(std::to_string(max).length());
16    logger.buf << "|| f(x_" << std::setfill('0') << w << i << ") || = " <<
          std::scientific << r;
17    logger.log();
18  }
19
20  inline std::vector<double> FixedPointMethod(
21    FixedPointData const & fp,
22    std::vector<double> const & x_0,
23    Index n = 100, // max numb of iters
24    double eps = 1e-7
25  ) {
26    auto& logger = SingletonLogger::instance();
27    logger.log("start Fixed Point method");
28    auto x = x_0;
29    auto r_norm = norm(fp.f(x));
30    logFixedPoint(r_norm, 0, n);
31    if (r_norm < eps) {
32      logger.log("stop Fixed Point method");
33      return x;
34    }
35    Index i;
36    for (i = 1; i <= n; ++i) {
37      x = fp.g(x);
38      r_norm = norm(fp.f(x));
39      logFixedPoint(r_norm, i, n);
40      if (r_norm < eps) break;
41    }
42    logger.log("stop Fixed Point method");
43    if (i > n) logger.wrn("Fixed Point Method exceeded max numb of
          iterations");
44    return x;
45  }
46
47  inline std::vector<double> AndersonMixingMethod(
48      FixedPointData const & fp,
49      std::vector<double> const & x_0,
50      Index m = 0,
51      Index n = 100, // max numb of iters
52      double eps = 1e-7
53    ) {
54    auto& logger = SingletonLogger::instance();
55    if (m == 0) {
56      logger.log("m = 0 -> Fixed Point Method");
57      return FixedPointMethod(fp, x_0, n, eps);
58    }
59    boost::circular_buffer<std::vector<double>> F(m), G(m + 1);
60    logger.log("start Fixed Point method");
61    // i = 0
62    auto x = x_0;
63    auto r = fp.f(x);
64    auto r_norm = norm(r);
65    logFixedPoint(r_norm, 0, n);
66    if (r_norm < eps) {
67      logger.log("stop Fixed Point method");
68      return x;
69    }
70    G.push_front(fp.g(x));
71    // i = 1
72    x = G[0];
```

```
73      auto r_new = fp.f(x);
74      r_norm = norm(r_new);
75      logFixedPoint(r_norm, 1, n);
76      if (r_norm < eps) {
77        logger.log("stop Fixed Point method");
78        return x;
79      }
80      G.push_front(fp.g(x));
81      F.push_front(r_new - r);
82      r = r_new;
83      logger.log("stop Fixed Point method");
84      // i = 2, 3, . . .
85      logger.log("start Anderson Mixing method");
86      Index i;
87      for (i = 2; i <= n; ++i) {
88        m = F.size();
89        DenseMatrix<double> A(m);
90        std::vector<double> b(m);
91        for (Index k = 0; k < m; ++k) {
92          b[k] = F[k] * r;
93          for (Index j = 0; j < m; ++j)
94            A(k, j) = F[k] * F[j];
95        }
96        auto beta = A.GaussElimination(b);
97        decltype(beta) alpha(m + 1);
98        alpha[0] = 1. - beta[0];
99        for (Index k = 1; k < m; ++k) alpha[k] = beta[k - 1] - beta[k];
100       alpha[m] = beta[m - 1];
101       std::fill(x.begin(), x.end(), 0.);
102       for (Index k = 0; k < m + 1; ++k) x += alpha[k] * G[k];
103       r_new = fp.f(x);
104       r_norm = norm(r_new);
105       logFixedPoint(r_norm, i, n);
106       if (r_norm < eps) break;
107       G.push_front(fp.g(x));
108       F.push_front(r_new - r);
109       r = r_new;
110     }
111     logger.log("stop Anderson Mixing method");
112     if (i > n) logger.wrn("Anderson Mixing Method exceeded max numb of
            iterations");
113     return x;
114   }
115
116 }
```

4.2 user.cpp

Модуль с имплементацией конечно-элементного решения линейной и нелинейной задач магнитостатики (1).

```
1  #include <memory>
2  // finite elements to use
3  #include "Triangle_P0_Lagrange.hpp"
4  #include "Triangle_P1_Lagrange.hpp"
5  #include "Triangle_P2_Lagrange.hpp"
6  #include "Triangle_P1_CrouzeixRaviart.hpp"
7  // interpolant for diffusion
8  #include "FEInterpolant.hpp"
9  // assembler
10 #include "DivGradFEM.hpp"
```

```
11 // solvers
12 #include "ProjectionSolvers.hpp"
13 #include "NonlinearSolvers.hpp"
14 // preconditioner
15 #include "Multigrid.hpp"
16 // pi
17 #include "constants.hpp"
18 // for small rotation matrix
19 #include "DenseMatrix.hpp"
20 // for 1D interpolant of mu_bar_iron
21 #include "Segment_P1_Lagrange.hpp"
22
23 using std::vector;
24 using std::string;
25 using boost::get;
26 using namespace FEM::DivGrad;
27 using namespace ProjectionSolvers;
28 using namespace NonlinearSolvers;
29
30 int main() {
31   string iPath("Mathematica/preprocessing/"),
        oPath("Mathematica/postprocessing/");
32   auto& logger = SingletonLogger::instance();
33   try {
34     logger.beg("set up mesh");
35       vector<string> meshType { "mesh_good.nt", "mesh_coarse.nt",
            "mesh_nonconform.nt" };
36       auto meshTypeIndex = logger.opt("mesh type", meshType);
37       Triangulation Omega;
38       Omega.import(iPath + meshType[meshTypeIndex]);
39       Omega.computeNeighbors().enumerateRibs();
40       // special subdomains of Omega
41       Quadrilateral2D
42         // parts of magnet
43         magnetLeft {
44           { { 0., 0. }, { .02, 0. }, { .02, .06 }, { 0., .06 } }
45         },
46         magnetMiddle {
47           { { .04, .04 }, { .04, .002 }, { .06, .002 }, { .06, .04 } }
48         },
49         magnetRight {
50           { { .08, .06 }, { .08, 0. }, { .1, 0. }, { .1, .06 } }
51         },
52         magnetTop {
53           { { 0.1, 0.06 }, { 0, 0.06 }, { 0, 0.04 }, { 0.1, 0.04 } }
54         },
55         // wires
56         left { // left wire
57           { { .025, .025 }, { .035, .025 }, { .035, .035 }, { .025, .035 } }
58         },
59         right { // right ...
60           { { .065, 0.025 }, { .075, .025 }, { .075, .035 }, { .065, .035 } }
61         };
62       Predicate2D magnet = [&](Node2D const & p) {
63         return nodeInElement(magnetLeft, p)
64             || nodeInElement(magnetMiddle, p)
65             || nodeInElement(magnetRight, p)
66             || nodeInElement(magnetTop, p);
67       };
68     logger.end();
69     logger.beg("set up PDE and BCs");
70       // current density
71       double J_z = 1e+06;
```

```
72        logger.inp("set current density J_z", J_z);
73        // iron permeability
74        double mu_roof_iron = 1000.;
75        logger.inp("set iron permeability mu_roof_iron", mu_roof_iron);
76        // vacuum permeability
77        double const mu_0 = 4. * PI * 1e-7;
78        ScalarField2D
79          diffusion = [&](Node2D const & p) { // linear case
80            double mu_roof = magnet(p) ? mu_roof_iron : 1.;
81            return 1. / mu_roof;
82          },
83          force = [&](Node2D const & p) {
84            if (nodeInElement(left, p)) return mu_0 * J_z;
85            if (nodeInElement(right, p)) return - mu_0 * J_z;
86            return 0.;
87          },
88          reaction = [](Node2D const) { return 0.; },
89            NeumannValue = reaction, RobinCoefficient = reaction,
                DirichletCondition = reaction;
90          Predicate2D naturalBCsPredicate = [](Node2D const p) { return p[1] ==
               0.; };
91        // PDE
92        DiffusionReactionEqn2D PoissonEqn { diffusion, reaction, force };
93        // BCs
94        ScalarBoundaryCondition2D
95          RobinBC {
96            { RobinCoefficient, NeumannValue }, // n . (a nabla u) + R u = N
97            naturalBCsPredicate
98          },
99          DirichletBC { DirichletCondition };
100     logger.end();
101     logger.beg("build and export interpolants for diffusion and force");
102       TriangularScalarFEInterpolant
103         diffisionInterp { diffusion, Triangle_P0_Lagrange::instance(), Omega },
104         forceInterp { force, Triangle_P0_Lagrange::instance(), Omega };
105       export(diffisionInterp.DOFs(), oPath + "diffusion.dat");
106       export(forceInterp.DOFs(), oPath + "force.dat");
107       Omega.export(oPath + "mesh_0.ntnr", { { "format", "NTNR" } });
108     logger.end();
109     logger.beg("set solver data");
110       // method of solving SLAE
111       auto method = logger.opt("choose solving technique", {
112         "MG",
113         "P_MG CG",
114         "P_ILU(0) CG",
115         "CG"
116       });
117       // FE
118       vector<TriangularScalarFiniteElement*> FEs {
119         &Triangle_P1_Lagrange::instance(),
120         &Triangle_P2_Lagrange::instance(),
121         &Triangle_P1_CrouzeixRaviart::instance()
122       };
123       auto FEIndex = logger.opt("choose finite element", { "Lagrange P1",
             "Lagrange P2", "Crouzeix-Raviart P1" });
124       auto& FE = *FEs[FEIndex];
125       // numb of refinements
126       Index numbOfMeshLevels;
127       logger.inp("numb of mesh levels (refinements)", numbOfMeshLevels);
128       // iterations logger
129       Index i_log;
130       logger.inp("log every nth iteration of the solver, n", i_log);
131       // max numb of iters
```

```
132       Index maxNumbOfIterations;
133       logger.inp("max numb of iterations", maxNumbOfIterations);
134       // tolerance
135       double eps;
136       logger.inp("set eps for solver", eps);
137       // stopping criterion
138       auto stop = (StoppingCriterion)logger.opt("stopping criterion", {
              "absolute", "relative" });
139       logger.beg("set MG data");
140         logger.wrn("these values make sense for (MG) and (P_MG CG) solvers
              only");
141         // smoothing
142         Index nu;
143         logger.inp("numb of pre- and post-smoothing iterations", nu);
144         // cycle type
145         auto gamma = logger.opt("set recursive calls type", { "V-cycle",
              "W-cycle" });
146         ++gamma;
147         // iters for precond
148         Index numbOfInnerIterations;
149         logger.inp("numb of iterations for inner solver",
              numbOfInnerIterations);
150       logger.end();
151     logger.end();
152     logger.beg("run method");
153       vector<double> x_linear;
154       if      (method == 0) /* MG */ {
155         logger.beg("set up mg");
156           Multigrid<SymmetricCSlCMatrix<double>> MG {
157               FE, Omega, numbOfMeshLevels,
158               [&](Triangulation const & Omega) {
159                 return assembleSystem(PoissonEqn, Omega, RobinBC, DirichletBC,
                        FE);
160               },
161               TransferType::canonical
162           };
163           auto smoother = [&](SymmetricCSlCMatrix<double>& A, vector<double>
                const & b, vector<double> const & x_0, Index) {
164             return Smoothers::SSOR(A, b, x_0, 1., nu, 0.,
                      StoppingCriterion::absolute, 0);
165           };
166         logger.end();
167         logger.beg("solve w/ MG");
168           auto& A = MG.A();
169           auto& b = MG.b();
170           // initial residual
171           double r_0 = norm(b - A * x_linear), r = r_0, r_new = r_0;
172           logInitialResidual(r_0, maxNumbOfIterations);
173           // MG as a stand-alone iteration
174           Index i;
175           for (i = 1; i <= maxNumbOfIterations; ++i) {
176             logger.mute = true;
177             x_linear = MG(numbOfMeshLevels, b, x_linear, smoother, gamma);
178             logger.mute = false;
179             r_new = norm(b - A * x_linear);
180             if (i_log && i % i_log == 0) logResidualReduction(r, r_new, i,
                    maxNumbOfIterations);
181             r = r_new;
182             if (stop == StoppingCriterion::absolute && r < eps) break;
183             else if (stop == StoppingCriterion::relative && r / r_0 < eps)
                    break;
184           }
```

```
185           logFinalResidual(r_0, r_new, i > maxNumbOfIterations ?
                  maxNumbOfIterations : i, maxNumbOfIterations);
186           if (i > maxNumbOfIterations) logger.wrn("MG failed");
187         logger.end();
188       }
189       else if (method == 1) /* P_MG CG */ {
190         logger.beg("build preconditioner");
191           Multigrid<SymmetricCSlCMatrix<double>> MG {
192               FE, Omega, numbOfMeshLevels,
193               [&](Triangulation const & Omega) {
194                 return assembleSystem(PoissonEqn, Omega, RobinBC, DirichletBC,
                        FE);
195               },
196               TransferType::canonical
197           };
198           auto smoother = [&](SymmetricCSlCMatrix<double>& A, vector<double>
                  const & b, vector<double> const & x_0, Index) {
199             return Smoothers::SSOR(A, b, x_0, 1., nu, 0.,
                      StoppingCriterion::absolute, 0);
200           };
201           Index n = MG.A().size();
202           Preconditioner P = [&](vector<double> const & x) {
203             vector<double> y(n);
204             auto cond = logger.mute;
205             logger.mute = true;
206             for (Index i = 0; i < numbOfInnerIterations; ++i)
207               y = MG(numbOfMeshLevels, x, y, smoother, gamma);
208             logger.mute = cond;
209             return y;
210           };
211         logger.end();
212         logger.beg("solve linear problem");
213           auto& A = MG.A();
214           auto& b = MG.b();
215           x_linear = Krylov::PCG(P, A, b, boost::none, maxNumbOfIterations,
                  eps, stop, i_log);
216         logger.end();
217       }
218       else if (method == 2) /* P_ILU(0) CG */ {
219         logger.beg("refine mesh");
220           Omega.refine(numbOfMeshLevels);
221         logger.end();
222         logger.beg("assemble system");
223           auto system = assembleSystem(PoissonEqn, Omega, RobinBC,
                  DirichletBC, FE);
224           auto& A = get<0>(system);
225           auto& b = get<1>(system);
226         logger.end();
227         logger.beg("build ILU(0)");
228           auto LDLT = A;
229           LDLT.decompose();
230           auto P = [&](vector<double> const & x) {
231             return LDLT.backSubst(LDLT.diagSubst(LDLT.forwSubst(x, 0.)), 0.);
232           };
233         logger.end();
234         logger.beg("solve linear problem");
235           x_linear = Krylov::PCG(P, A, b, boost::none, maxNumbOfIterations,
                  eps, stop, i_log);
236         logger.end();
237       }
238       else if (method == 3) /* CG */ {
239         logger.beg("refine mesh");
240           Omega.refine(numbOfMeshLevels);
```

```
241           logger.end();
242           logger.beg("assemble system");
243             auto system = assembleSystem(PoissonEqn, Omega, RobinBC,
                    DirichletBC, FE);
244             auto& A = get<0>(system);
245             auto& b = get<1>(system);
246           logger.end();
247           logger.beg("solve linear problem");
248             x_linear = Krylov::CG(A, b, boost::none, maxNumbOfIterations, eps,
                    stop, i_log);
249           logger.end();
250         }
251       logger.end();
252       logger.beg("compute interpolant for magnetic field");
253         TriangularScalarFEInterpolant x_linear_interp { x_linear, FE, Omega };
254         Index activeElementIndex;
255         DenseMatrix<double> rotate {
256           {  0., 1. },
257           { -1., 0. }
258         };
259         auto B_linear = [&](Node2D const & p) -> Node2D {
260           return rotate * x_linear_interp.grad(p, activeElementIndex);
261         };
262         auto Bx_linear = [&](Node2D const & p) { return B_linear(p)[0]; };
263         auto By_linear = [&](Node2D const & p) { return B_linear(p)[1]; };
264         auto Bn_linear = [&](Node2D const & p) { return norm(B_linear(p)); };
265         TriangularScalarFEInterpolant
266           Bx_linear_interp { Bx_linear, FE/*Triangle_P0_Lagrange::instance()*/,
                Omega, activeElementIndex },
267           By_linear_interp { By_linear, FE/*Triangle_P0_Lagrange::instance()*/,
                Omega, activeElementIndex },
268           Bn_linear_interp { Bn_linear, FE/*Triangle_P0_Lagrange::instance()*/,
                Omega, activeElementIndex };
269       logger.end();
270       logger.beg("export soln vector");
271         export(x_linear, oPath + "x_linear.dat");
272         export(Bx_linear_interp.DOFs(), oPath + "Bx_linear.dat");
273         export(By_linear_interp.DOFs(), oPath + "By_linear.dat");
274         export(Bn_linear_interp.DOFs(), oPath + "Bn_linear.dat");
275       logger.end();
276       /* */
277       if (logger.yes("solve non-linear problem")) {
278         logger.beg("build interpolant for mu_roof_iron");
279           SegmentMesh BnValues;
280           BnValues.import(iPath + "Bn_values.dat");
281           double Bn_min = BnValues.getNodes().front(), Bn_max =
                  BnValues.getNodes().back();
282           vector<double> muValues(BnValues.numbOfNodes());
283           import(muValues, iPath + "mu_values.dat");
284           SegmentFEInterpolant mu_roof_iron_interp { muValues,
                  Segment_P1_Lagrange::instance(), BnValues };
285           // relative permeability of iron as a function of ||B||
286           auto mu_roof_iron = [&](double Bn) {
287             // (1) extrapolation
288             if (Bn < Bn_min)
289               return muValues.front();
290             if (Bn > Bn_max)
291               return Bn * muValues.back() / (Bn_max + Bn * muValues.back() -
                      Bn_max * muValues.back());
292             // (2) healthy case
293             return mu_roof_iron_interp(Bn);
294           };
295         logger.end();
```

```
296       logger.beg("solve non-linear problem");
297         FixedPointData fp;
298         SymmetricCSlCMatrix<double> A;
299         vector<double> b, r;
300         // residual, f(x) := b - A(x).x
301         fp.f = [&](vector<double> const & x) {
302           logger.mute = true;
303           Index activeElementIndex;
304           TriangularScalarFEInterpolant x_interp { x, FE, Omega };
305           PoissonEqn.diffusionTerm() = [&](Node2D const & p) {
306             double mu_roof = magnet(p)
307               ? mu_roof_iron(norm(x_interp.grad(p, activeElementIndex)))
308               : 1.;
309             return 1. / mu_roof;
310           };
311           logger.beg("assemble system");
312             auto system = assembleSystem(PoissonEqn, Omega, RobinBC,
                    DirichletBC, FE, activeElementIndex);
313             A = get<0>(system);
314             b = get<1>(system);
315           logger.end();
316           logger.mute = false;
317           r = b - A * x;
318           return r;
319         };
320         // iteration operator, g(x) := [A(x)]^(-1) . b
321         auto g = [&](vector<double> const & x) {
322           logger.mute = true;
323           logger.beg("build ILU(0)");
324             auto LDLT = A;
325             LDLT.decompose();
326             auto P = [&](vector<double> const & x) {
327               return LDLT.backSubst(LDLT.diagSubst(LDLT.forwSubst(x, 0.)), 0.);
328             };
329           logger.end();
330           logger.beg("compute new approximation");
331             auto x_new = Krylov::PCG(P, A, b, x, 0, 10e-9,
                    StoppingCriterion::relative);
332           logger.end();
333           logger.mute = false;
334           return x_new;
335         };
336         if (logger.yes("use relaxation"))
337           fp.g = [&](vector<double> const & x) {
338             auto r_norm = norm(r), r_norm_new = r_norm + 1., omega = 1.;
339             auto g_x = g(x), x_new = g_x;
340             while (r_norm_new >= r_norm && omega > 1e-10) {
341               x_new = omega * g_x + (1. - omega) * x;
342               r_norm_new = norm(fp.f(x_new));
343               omega /= 2.;
344             }
345             return x_new;
346           };
347         else fp.g = g;
348         // solve
349         decltype(x_linear) x_nonlinear;
350         if (logger.yes("use Anderson Mixing")) {
351           auto x_initial = FixedPointMethod(fp, x_linear, 50, 1e-4);
352           fp.g = g;
353           x_nonlinear = AndersonMixingMethod(fp, x_initial, 10, 200, 1e-8);
354         }
355           else x_nonlinear = FixedPointMethod(fp, x_linear, 300, 1e-8);
356       logger.end();
```

```
357         logger.beg("compute interpolant for magnetic field");
358           TriangularScalarFEInterpolant x_nonlinear_interp { x_nonlinear, FE,
                Omega };
359           auto B_nonlinear = [&](Node2D const & p) -> Node2D {
360             return rotate * x_nonlinear_interp.grad(p, activeElementIndex);
361           };
362           auto Bx_nonlinear = [&](Node2D const & p) { return B_nonlinear(p)[0];
                };
363           auto By_nonlinear = [&](Node2D const & p) { return B_nonlinear(p)[1];
                };
364           auto Bn_nonlinear = [&](Node2D const & p) { return
                norm(B_nonlinear(p)); };
365           TriangularScalarFEInterpolant
366             Bx_nonlinear_interp { Bx_nonlinear, FE, Omega, activeElementIndex },
367             By_nonlinear_interp { By_nonlinear, FE, Omega, activeElementIndex },
368             Bn_nonlinear_interp { Bn_nonlinear, FE, Omega, activeElementIndex };
369         logger.end();
370         logger.beg("export soln vector");
371           export(x_nonlinear, oPath + "x_nonlinear.dat");
372           export(Bx_nonlinear_interp.DOFs(), oPath + "Bx_nonlinear.dat");
373           export(By_nonlinear_interp.DOFs(), oPath + "By_nonlinear.dat");
374           export(Bn_nonlinear_interp.DOFs(), oPath + "Bn_nonlinear.dat");
375         logger.end();
376       }
377       /* */
378       logger.beg("export mesh");
379         Omega.export(oPath + "mesh.ntnr", { { "format", "NTNR" } });
380       logger.end();
381       logger.exp("stdin.txt");
382     }
383     catch (std::exception const & e) {
384       logger.err(e.what());
385     }
386     return 0;
387   }
```