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

Практическое задание № 1

по дисциплине «ПиИМРБРСУ»

КОМПЛЕКСНЫЕ СЛАУ. РАСПАРАЛЛЕЛИВАНИЕ ИТЕРАЦИОННЫХ МЕТОДОВ.

Команда СУМИН ЯРОСЛАВ

ПИНЕЛИС КОНСТАНТИН

Группа ПММ-42

Преподаватель ДОМНИКОВ ПЕТР АЛЕКСАНДРОВИЧ

Новосибирск, 2025

СОДЕРЖАНИЕ

Введение 3

1 Теоретическая часть 4
1.1 Комплексные матрицы и векторы 4
1.2 Алгоритм COCG 4
1.3 Алгоритм COCR 5
1.4 Сглаживание невязки 6

2 Исследование 8
2.1 Исследование на сходимость методов 9
2.2 Исследование времени решения СЛАУ в зависимости от количества потоков 14

3 Заключение 16

Приложение А. Текст программы 17

ВВЕДЕНИЕ

Целью данной практической работы является разработка программы решения СЛАУ с комплексными матрицами. Изучение особенности схем предобуславливания с заменой скалярного произведения. Исследование влиянич сглаживания невязки на сходимость итерационных методов на матрицах большой размерности. Приобретение навыков работы с библиотеками для высокопроизводительных вычислений. Выполнение распараллеливания итерационных методов решения СЛАУ с разреженными матрицами в вычислительных системах с общей памятью. Оценка ускорения, получаемого за счёт распараллеливания.

1. ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

1.1. КОМПЛЕКСНЫЕ МАТРИЦЫ И ВЕКТОРЫ

Поле комплексных чисел можно рассматривать как поле действительных матриц вида:

$$\begin{pmatrix} a & -b \\ b & a \end{pmatrix}, \quad a, b \in \mathbb{R}, \tag{1}$$

в котором роль действительных чисел играют матрицы вида $\begin{pmatrix} a & 0 \\ 0 & a \end{pmatrix}$, роль мнимой единицы – матрица $\begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$. Таким образом, комплексные матрицы можно рассматривать как вещественные, состоящие из блок-элементов вида (1).

Комплексные векторы также можно рассматривать как вещественные: вектору $x \in \mathbb{C}^n$ будет соответствовать вектор $x' \in \mathbb{R}^{2n}$, причем
$x' = \left(x_1^{\mathrm{Re}}, x_1^{\mathrm{Im}}, x_2^{\mathrm{Re}}, x_2^{\mathrm{Im}}, \ldots, x_n^{\mathrm{Re}}, x_n^{\mathrm{Im}}\right)$.

Скалярное произведение (x, y) воспринимается как скалярное произведение двух комплексных векторов со всеми их свойствами.

1.2. АЛГОРИТМ COCG

Данный метод может быть рассмотрен как обобщение метода сопряженных градиентов для системы вида:

$$Ax = b, \tag{2}$$

где комплексная матрица A размера $n \times n$ симметричная $\left(A = A^{\top}\right)$, но в общем случае может не быть эрмитовой $\left(A \neq \overline{A}^{\top}\right)$.

Алгоритм метода для предобусловленной системы $M^{-1}Ax = M^{-1}b$ может быть записан следующим образом.

Выбрать x^0.

Вычислить

$$r^0 = b - Ax^0, z^0 = M^{-1}r^0, p^0 = z^0. \tag{3}$$

Для $j = 0, 1, 2, \ldots$

$$\alpha_j = \frac{\left(\overline{r}^j, z^j\right)}{\left(\overline{Ap}^j, p^j\right)}, \tag{4}$$

$$x^{j+1} = x^j + \alpha_j p^j, \tag{5}$$

$$r^{j+1} = r^j - \alpha_j A p^j, \tag{6}$$

$$z^{j+1} = M^{-1} r^{j+1}, \tag{7}$$

$$\beta_j = \frac{\left(\overline{r}^{j+1}, z^{j+1}\right)}{\left(\overline{r}^j, z^j\right)}, \tag{8}$$

$$p^{j+1} = z^{j+1} + \beta_j p^j, \tag{9}$$

где вектор z^j – вектор невязки предобусловленной системы на j-ой (текущей) итерации. В апгоритме (3) - (9) на каждой итерации требуется выполнять только одно матрично-векторное умножение и одно решеин СЛАУ с матрицей предобуславливания M. В векторах x^j и r^j хранятся соответственно решение и невязка исходной (не предобусловленной) СЛАУ на j-ой (текущей) итерации.

1.3. АЛГОРИТМ COCR

Кроме адаптации метода сопряженных градиентов к решению СЛАУ с комплексно-симметричными матрицами существует обобщенный метод сопряженных невязок.

Схема метода с матрицей предобуславливания M (при применении ее к СЛАУ с матрицей $M^{-1}A$ и при замене естественных скалярных произведений на $(x, y)_M = (Mx, y)$) выглядит следующий образом.

Выбрать x^0.

Вычислить

$$r^0 = b - Ax^0, p^0 = s^0 = M^{-1} r^0, a^0 = z^0 = As^0, \omega^0 = M^{-1} z^0. \tag{10}$$

Для $j = 0, 1, 2, \ldots$

$$\alpha_j = \frac{\left(\overline{a}^j, s^j\right)}{\left(\overline{z}^j, \omega^j\right)}, \tag{11}$$

$$x^{j+1} = x^j + \alpha_j p^j, \tag{12}$$

$$r^{j+1} = r^j - \alpha_j z^j, \tag{13}$$

$$s^{j+1} = s^j - \alpha_j \omega^j, \tag{14}$$

$$a^{j+1} = As^{j+1}, \tag{15}$$

$$\beta_j = \frac{\left(\overline{a}^{j+1}, s^{j+1}\right)}{\left(\overline{a}^j, s^j\right)}, \tag{16}$$

$$p^{j+1} = s^{j+1} + \beta_j p^j, \tag{17}$$

$$z^{j+1} = a^{j+1} + \beta_j z^j, \tag{18}$$

$$\omega^{j+1} = M^{-1} z^{j+1}, \tag{19}$$

где вектор s^j – вектор невязки предобусловленной системы на j-ой (текущей) итерации; $a^j = As^j$, $z^j = Ap^j$, $\omega^j = M^{-1}z^j$ – вспомогательные векторы, причем вектор $z^j = Ap^j$, как и ранее, не вычисляется умножением матрицы на вектор, а пересчитывается рекуррентно по формуле (18).

1.4. СГЛАЖИВАНИЕ НЕВЯЗКИ

Невязки методов COCG и COCR могут быть подвержены сильным осцилляциям, что затрудняет контроль сходимости методов и завершение процесса по исчерпанию числа итераций. Поэтому для данных методов целесообразно использовать процедуру сглаживания невязки.

Суть сглаживания невязки заключается в следующем. Пусть некоторый метод генерирует на j-ой итерации приближенное решение x_j и соответствующий вектор невязки r_j. Вводятся два вспомогательных вектора y_j и s_j следующим образом:

$$y_0 = x_0, s_0 = r_0, \tag{20}$$

$$y_j = (1 - \eta_j)y_{j-1} + \eta_j x_j, s_j = (1 - \eta_j)s_{j-1} + \eta_j r_j, j = 1, 2, \ldots \tag{21}$$

Коэффициент $\eta_j \in \mathbb{R}$ выбирается так, чтобы величина $\|f - Ay_j\|^2$ минимизировалась на каждом шаге:

$$\eta_j = -\frac{(s_{j-1}, r_j - s_{j-1})}{(r_j - s_{j-1}, r_j - s_{j-1})}. \tag{22}$$

Для достижения строко монотонной сходимости необходимо обеспечить

выполнение условия $\eta_j \in [0, 1]$, т. е. если $\eta_j < 0$, то следует положить $\eta_j = 0$,если $\eta_j > 1$, то следует положить $\eta_j = 1$.

Решением системы на j-ой итерации считается вектор y_j с соответствующим вектором невязки s_j (при этом выполняется неравенство $\|s_j\| \leq \|r_j\|$). При этом векторы x_j и r_j вычисляются по исходным формулам итерационного метода, в который встроена процедура сглаживания невязки.

2. ИССЛЕДОВАНИЕ

Для тестирования будет использоваться набор СЛАУ с комплексными симметричными матрицами, хранящимися в разреженном строчно–столбцовом формате, выданный преподавателем. Набор состоит из 4-х СЛАУ. Для каждой СЛАУ приведены файлы:

- kuslau.txt – текстовый файл, содержащий 3 числа: size – размер вектора неизвестных, epsilon – порог уменьшения нормы относительной невязки, maximum_iterations – максимальное количество итераций;
- b – двоичный файл из вещественных чисел типа double, содержащий компоненты вектора правой части;
- di – двоичный файл из вещественных чисел типа double, содержащий диагональные элементы матрицы;
- gg – двоичный файл из вещественных чисел типа double, содержащий внедиагональные элементы матрицы;
- ig, jg, idi, ijg – двоичные файлы из целых чисел типа int, содержащие портрет матрицы;

Полученный набор СЛАУ содержит СЛАУ размерности: 88761, 244425, 1134591, 2963318.

У решателей будет диагональное предобуславливание.

В таблице 1 приведены результаты работы решателей для каждой СЛАУ из набора. Видно, что все методы для первых трех СЛАУ сошлись до требуемой точности 1×10^{-5}, а для четвертой СЛАУ ни один из методов не смог и завершил работу по достижении максимального количества итераций.

Таблица 1 – Результаты работы решателей для всех СЛАУ

Метод	Относительная невязка			
	СЛАУ №1	СЛАУ №2	СЛАУ №3	СЛАУ №4
COCG	8.91×10^{-6}	9.76×10^{-6}	6.97×10^{-6}	5.62×10^{-2}
COCG_Smooth	9.99×10^{-6}	9.99×10^{-6}	9.86×10^{-6}	1.37×10^{-2}
COCR	9.84×10^{-6}	9.72×10^{-6}	9.61×10^{-6}	2.96×10^{-2}
COCR_Smooth	9.13×10^{-6}	9.95×10^{-6}	9.88×10^{-6}	7.33×10^{-2}

2.1. ИССЛЕДОВАНИЕ НА СХОДИМОСТЬ МЕТОДОВ

На рисунках 1 - 4 представлены результаты решения метода COCG и COCG со сглаживанием невязки для каждой СЛАУ из набора. Видно, что метод COCG сходится на первых трех СЛАУ, на четвертой СЛАУ не сходится ни COCG, ни COCG со сглаживанием. Также на всех рисунках видно, что COCG без сглаживания невязки сильно осцилирует в процессе решения. Использование алгоритма сглаживания позволило достичь монотонного убывания невязки и небольшого ускорения по итерациям.

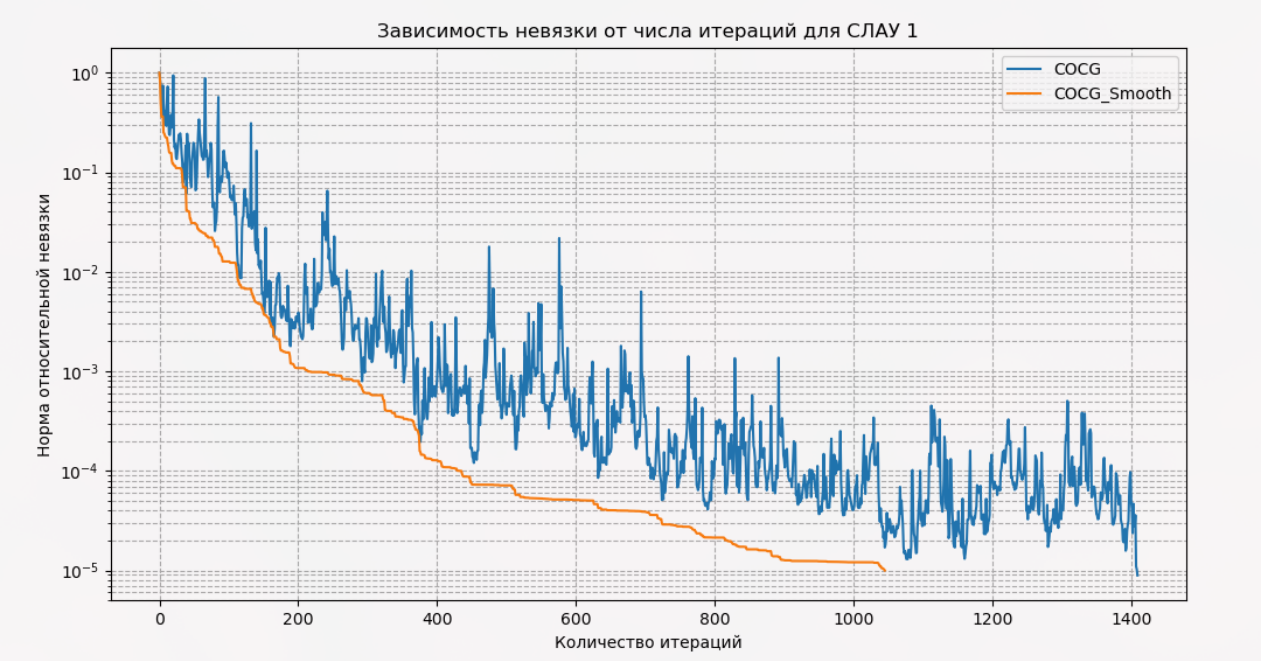

Рисунок 1 – Результаты решения методом COCG для первой СЛАУ

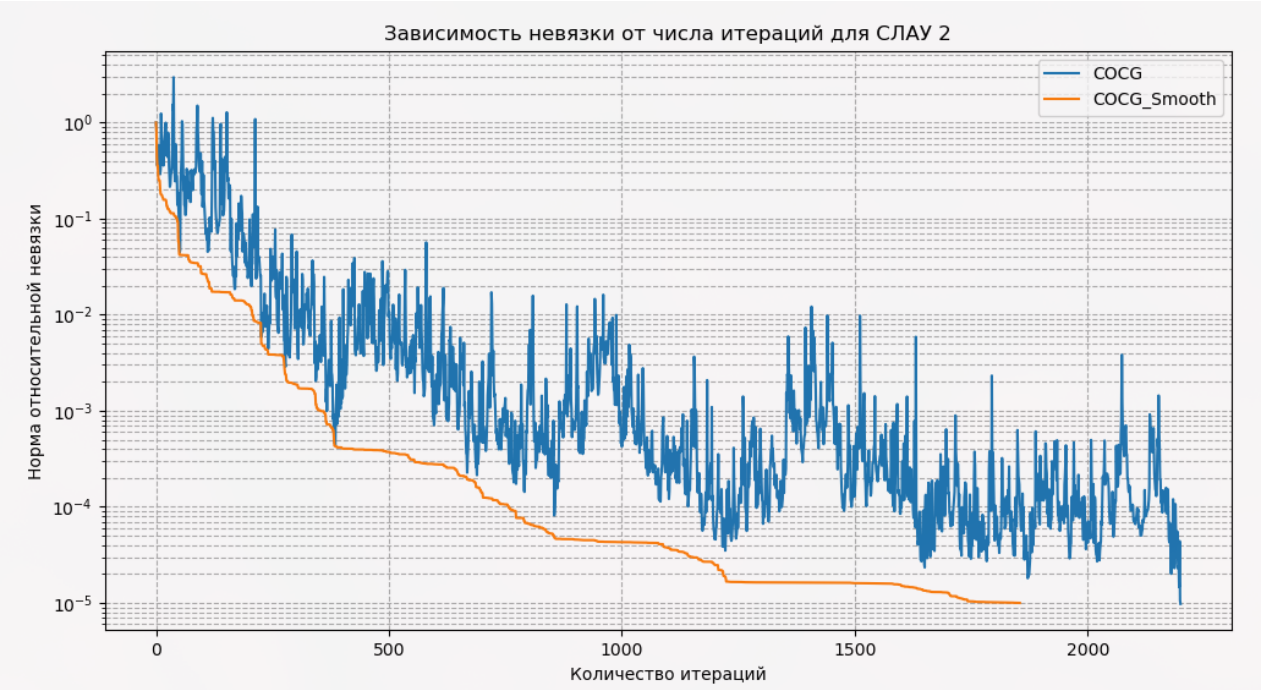

Рисунок 2 – Результаты решения методом COCG для второй СЛАУ

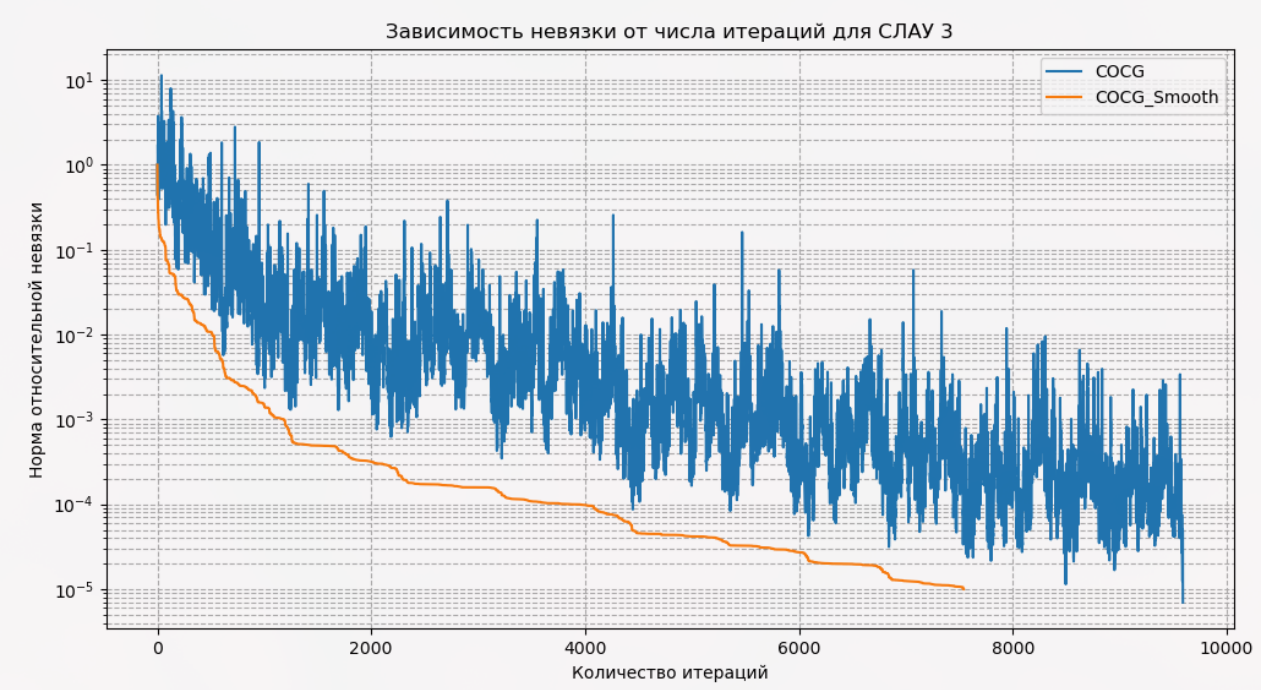

Рисунок 3 – Результаты решения методом COCG для третьей СЛАУ

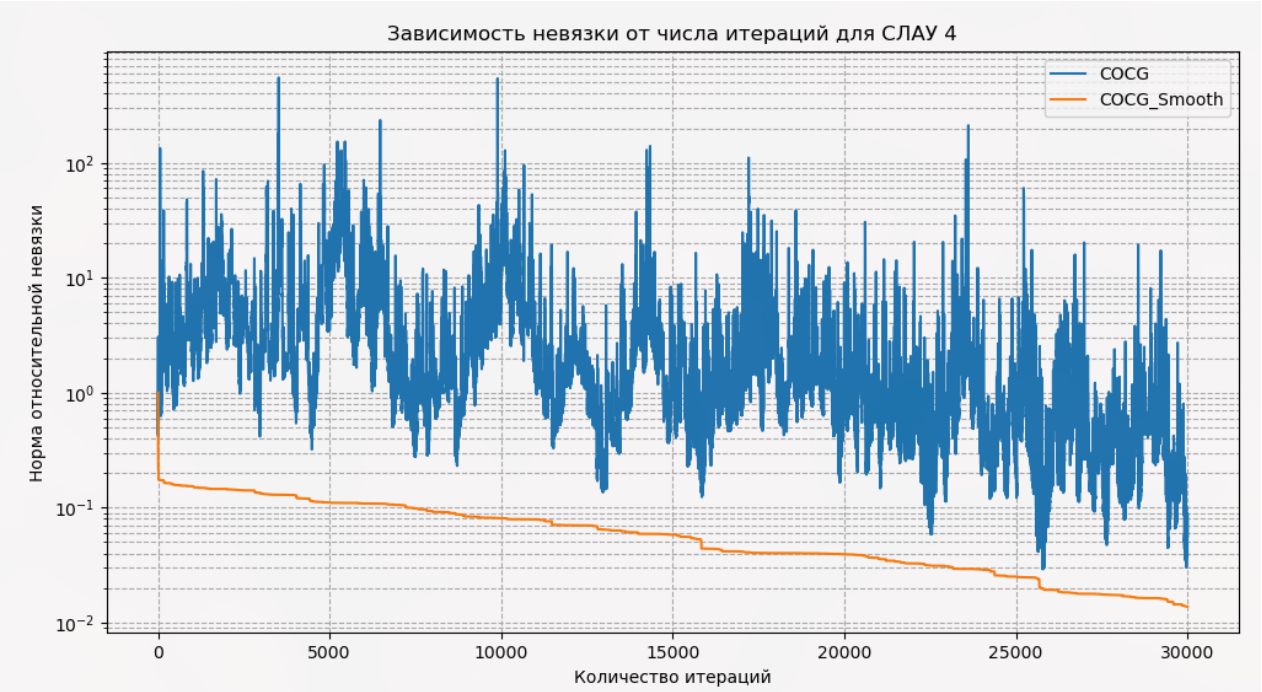

Рисунок 4 – Результаты решения методом COCG для четвертой СЛАУ

На рисунках 5 - 8 представлены результаты решения метода COCR и COCR со сглаживанием невязки для каждой СЛАУ из набора. Видно, что метод COCR сходится на первых трех СЛАУ, на четвертой СЛАУ не сходится ни COCR, ни COCR со сглаживанием. Также, как и для случая с COCG, использование алгоритма сглаживания позволило достичь монотонного убывания невязки и небольшого ускорения по количеству итераций.

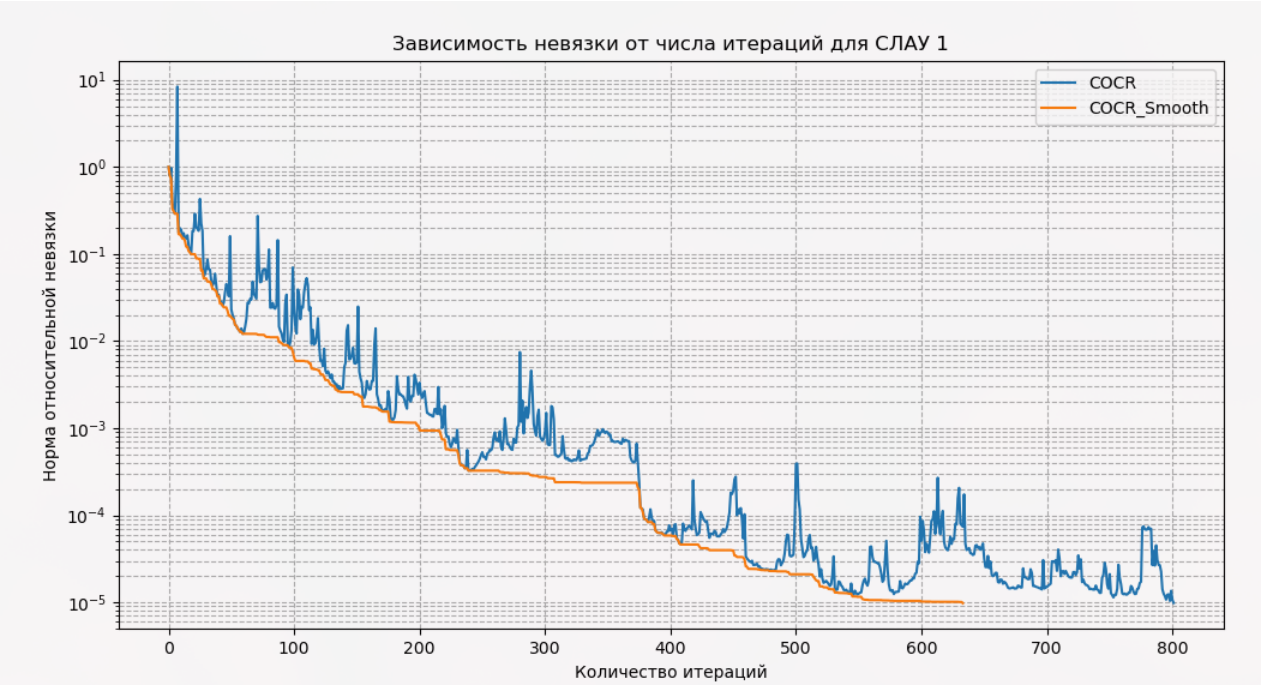

Рисунок 5 – Результаты решения методом COCR для первой СЛАУ

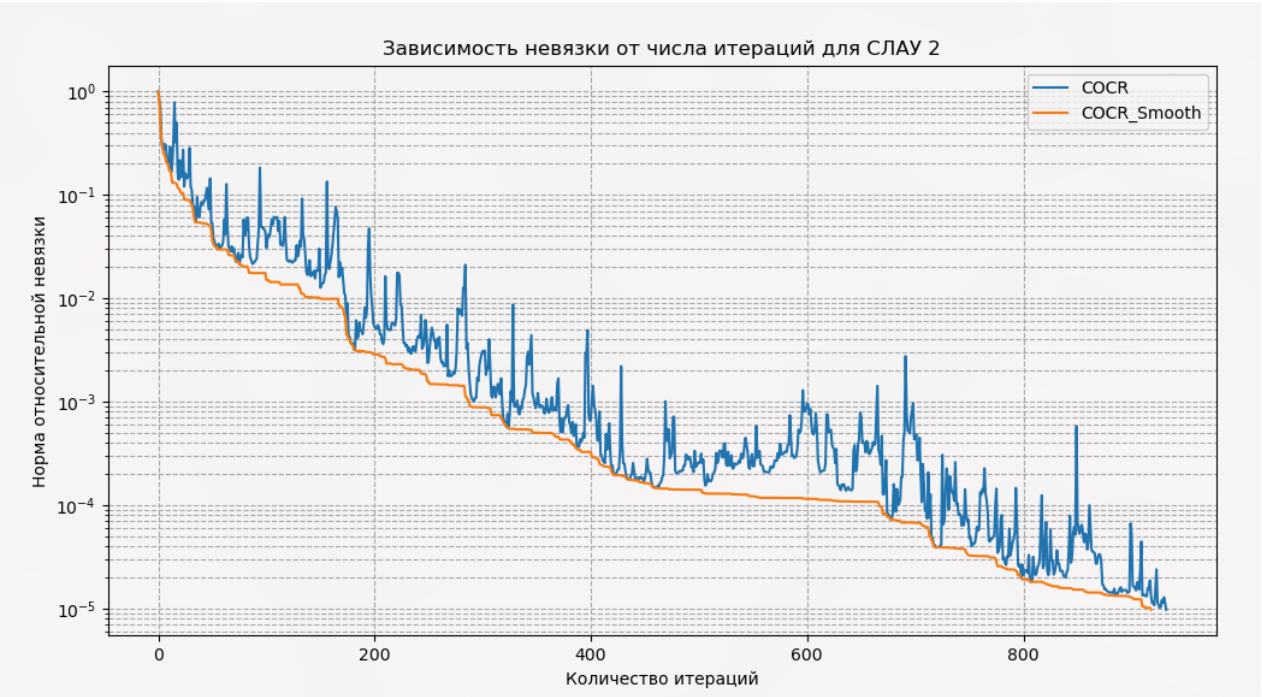

Рисунок 6 – Результаты решения методом COCR для второй СЛАУ

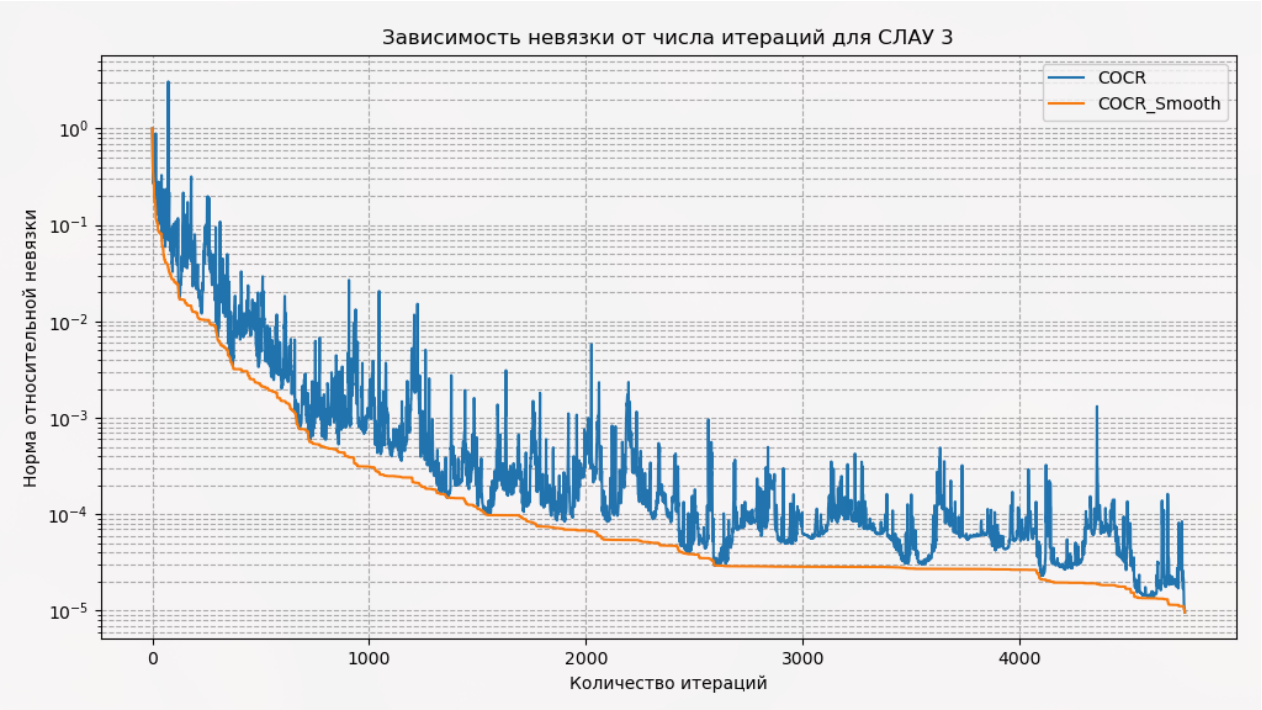

Рисунок 7 – Результаты решения методом COCR для третьей СЛАУ

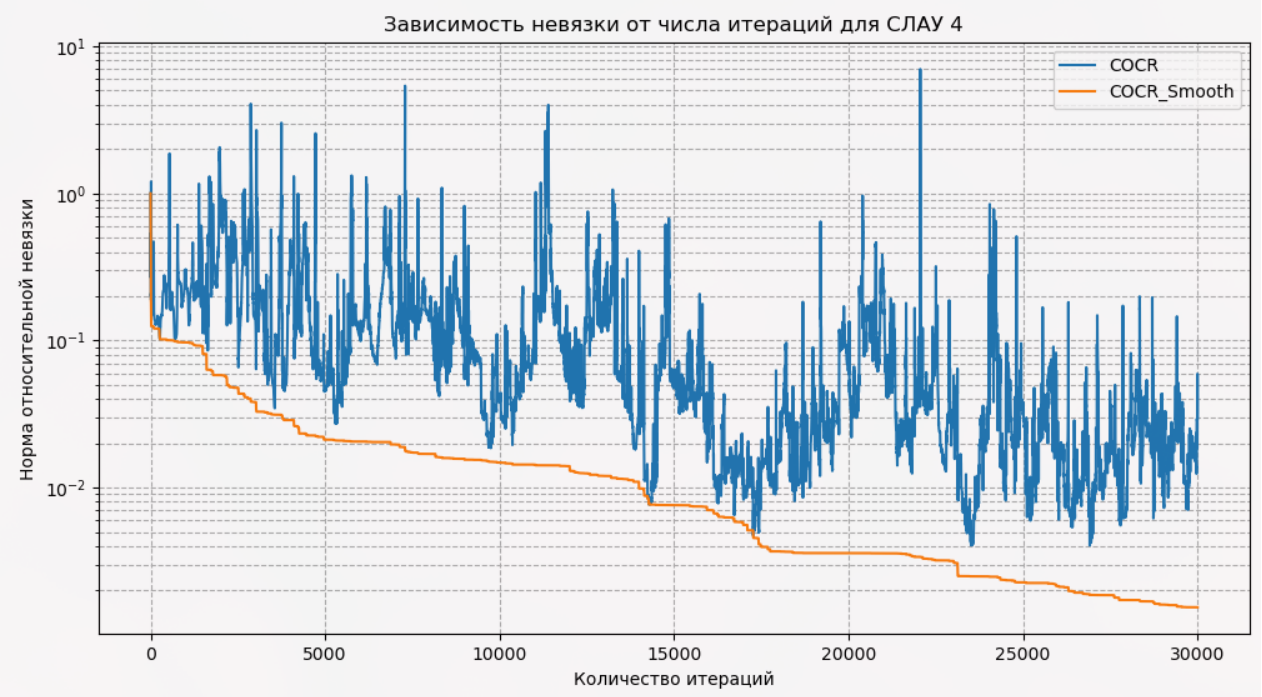

Рисунок 8 – Результаты решения методом COCR для четвертой СЛАУ

2.2. ИССЛЕДОВАНИЕ ВРЕМЕНИ РЕШЕНИЯ СЛАУ В ЗАВИСИМОСТИ ОТ КОЛИЧЕСТВА ПОТОКОВ

В таблице 2 представлены результаты тестирования на 1, 2, 4 и 8 вычислительных потоках. Видно, что наилучшее ускорение достигается при 4 потоках. При переходе на 8 потоков, на всех СЛАУ, кроме первой происходит снижение скорости. Разница в ускорении между методами COCG и COCR незначительна.

Таблица 2 – Результаты работы решателей на всех потоках для всех СЛАУ

1 поток								
Метод	СЛАУ №1		СЛАУ №2		СЛАУ №3		СЛАУ №4	
	Время	Ускорение	Время	Ускорение	Время	Ускорение	Время	Ускорение
COCG	4.65	1.0	19.79	1.0	412.71	1.0	3534.51	1.0
COCG_Smooth	3.70	1.0	18.62	1.0	375.41	1.0	3910.75	1.0
COCR	2.85	1.0	9.15	1.0	226.17	1.0	3702.72	1.0
COCR_Smooth	2.43	1.0	9.77	1.0	245.40	1.0	4139.35	1.0
2 потока								
Метод	СЛАУ №1		СЛАУ №2		СЛАУ №3		СЛАУ №4	
	Время	Ускорение	Время	Ускорение	Время	Ускорение	Время	Ускорение
COCG	2.67	1.74	11.51	1.72	254.94	1.62	2169.47	1.63
COCG_Smooth	2.19	1.68	11.29	1.65	237.11	1.58	2541.19	1.54
COCR	1.63	1.74	5.32	1.72	139.34	1.62	2344.80	1.58
COCR_Smooth	1.43	1.69	6.13	1.59	155.50	1.57	2670.54	1.55
4 потока								
Метод	СЛАУ №1		СЛАУ №2		СЛАУ №3		СЛАУ №4	
	Время	Ускорение	Время	Ускорение	Время	Ускорение	Время	Ускорение
COCG	1.68	2.76	8.60	2.30	220.79	1.87	1857.41	1.90
COCG_Smooth	1.39	2.66	9.44	1.97	205.55	1.83	2221.84	1.76
COCR	1.05	2.71	4.07	2.25	120.06	1.88	2028.56	1.83
COCR_Smooth	0.93	2.61	5.10	1.91	134.88	1.81	2365.34	1.75
8 потоков								
Метод	СЛАУ №1		СЛАУ №2		СЛАУ №3		СЛАУ №4	
	Время	Ускорение	Время	Ускорение	Время	Ускорение	Время	Ускорение
COCG	1.32	3.52	10.50	1.88	255.28	1.62	2303.34	1.53
COCG_Smooth	1.10	3.36	10.91	1.71	235.91	1.59	2670.89	1.46
COCR	0.82	3.47	4.94	1.85	141.24	1.60	2488.01	1.49
COCR_Smooth	0.72	3.37	5.85	1.67	155.08	1.58	2815.88	1.47

3. ЗАКЛЮЧЕНИЕ

В ходе практической работы были реализованы решатели COCG и COCR, оба с диагональной предобусловленностью и применением сглаживания невязки.

На первых трёх СЛАУ из набора тестирования все методы успешно сошлись до требуемой точности. Однако на четвёртой СЛАУ было достигнуто максимальное количество итераций без достижения нужной точности.

Применение сглаживания невязки позволило добиться монотонного убывания невязки и некоторого ускорения по количеству итераций.

В результате исследования по многопоточности было определено, что наилучшее количество вычислительных потоков для данной задачи – 4.

ПРИЛОЖЕНИЕ А. ТЕКСТ ПРОГРАММЫ

Листинг 1: main.cpp

```
#include "../include/slae.h"

#include <array>
#include <string>

int main() {
  for (const std::string &number :
       std::array<std::string, 3>{"1", "2", "3"}) {
    const std::string folder_index = number;

    SLAE slae;

    slae.inputComplexSLAEData(folder_index);
    slae.printComplexSLAEDataInformation();
    slae.solve(folder_index);
  }
}
```

Листинг 2: slae.h

```
#ifndef SLAE_H
#define SLAE_H

#include "../include/COCG_Di_solver.h"
#include "../include/COCR_Di_solver.h"
#include "../include/symmetric_matrix.h"
#include "../include/vector.h"

#include <filesystem>
#include <string>

class SLAE {
private:
  int size = 0;
  double epsilon = 0.0;
  int maximum_iterations = 0;

  SymmetricMatrix matrix;
  Vector x;
  Vector b;

  COCGDiSolver COCG_Di_solver;
  COCRDiSolver COCR_Di_solver;

  template <typename T>
  void readBinaryFileOfData(const std::filesystem::path &file_name, T *result,
                            int number_of_records, int len_of_record);

public:
  void solve(const std::string &folder_index);

  void inputComplexSLAEData(const std::string &folder_index);
```

```
  void printComplexSLAEDataInformation();
};

#endif
```

Листинг 3: slae.cpp

```
#include "../include/slae.h"

#include <filesystem>
#include <fstream>
#include <iostream>
#include <string>

void SLAE::solve(const std::string &folder_index) {
  COCG_Di_solver.setSolverParameters(maximum_iterations, epsilon, 10);
  COCR_Di_solver.setSolverParameters(maximum_iterations, epsilon, 10);

  // auto start_x_for_COCG = x;
  // auto COCG_results = COCG_Di_solver.solve(matrix, b, start_x_for_COCG, false
      );
  // auto end_x_for_COCG = x;
  // SymmetricMatrix::multiplyMatrixByVector(matrix, start_x_for_COCG,
  //                                         &end_x_for_COCG);

  // auto start_x_for_COCG_smooth = x;
  // auto COCG_smooth_results =
  //     COCG_Di_solver.solve(matrix, b, start_x_for_COCG_smooth, true);
  // auto end_x_for_COCG_smooth = x;
  // SymmetricMatrix::multiplyMatrixByVector(matrix, start_x_for_COCG_smooth,
  //                                         &end_x_for_COCG_smooth);

  // auto start_x_for_COCR = x;
  // auto COCR_results = COCR_Di_solver.solve(matrix, b, start_x_for_COCR, false
      );
  // auto end_x_for_COCR = x;
  // SymmetricMatrix::multiplyMatrixByVector(matrix, start_x_for_COCR,
  //                                         &end_x_for_COCR);

  auto start_x_for_COCR_smooth = x;
  auto COCR_smooth_results =
      COCR_Di_solver.solve(matrix, b, start_x_for_COCR_smooth, true);
  auto end_x_for_COCR_smooth = x;
  SymmetricMatrix::multiplyMatrixByVector(matrix, start_x_for_COCR_smooth,
                                          &end_x_for_COCR_smooth);

  // {
  //   auto out = std::ofstream("output/omp8/COCG_Di_" + folder_index + ".txt");
  //   out << COCG_results.dump() << "\n";
  //   out << COCG_results.dumpEps() << "\n";
  // }

  // {
  //   auto out =
  //       std::ofstream("output/omp8/COCG_Di_Smooth_" + folder_index + ".txt");
  //   out << COCG_smooth_results.dump() << "\n";
  //   out << COCG_smooth_results.dumpEps() << "\n";
  // }
```

```
  // {
  //   auto out = std::ofstream("output/omp8/COCR_Di_" + folder_index + ".txt");
  //   out << COCR_results.dump() << "\n";
  //   out << COCR_results.dumpEps() << "\n";
  // }

  {
    auto out =
        std::ofstream("output/omp8/COCR_Di_Smooth_" + folder_index + ".txt");
    out << COCR_smooth_results.dump() << "\n";
    out << COCR_smooth_results.dumpEps() << "\n";
  }
}

template <typename T>
void SLAE::readBinaryFileOfData(const std::filesystem::path &file_name,
                                T *result, int number_of_records,
                                int len_of_record) {
  std::ifstream in(file_name, std::ios::binary);

  in.read(reinterpret_cast<char *>(result),
          static_cast<std::streamsize>(sizeof(T) * len_of_record *
                                       number_of_records));
}

void SLAE::printComplexSLAEDataInformation() {
  std::cout << "SLAE information:" << "\n";
  std::cout << "\tdi: " << matrix.di.size() << "\n";
  std::cout << "\tgg: " << matrix.gg.size() << "\n";
  std::cout << "\tidi: " << matrix.idi.size() << "\n";
  std::cout << "\tig: " << matrix.ig.size() << "\n";
  std::cout << "\tigg: " << matrix.igg.size() << "\n";
  std::cout << "\tjg: " << matrix.jg.size() << "\n";
  std::cout << "\tx: " << x.size() << "\n";
  std::cout << "\tb: " << b.size() << "\n";
}

void SLAE::inputComplexSLAEData(const std::string &folder_index) {
  const std::filesystem::path input_folder =
      std::filesystem::path("data") / "complex" / folder_index;

  std::vector<int> ig;
  std::vector<int> igg;
  std::vector<int> jg;
  std::vector<int> idi;
  std::vector<double> di;
  std::vector<double> gg;
  std::vector<double> pr;

  {
    std::ifstream kuslau(input_folder / "kuslau");
    kuslau >> size >> epsilon >> maximum_iterations;
  }

  ig.resize(size + 1);
  readBinaryFileOfData(input_folder / "ig", ig.data(), ig.size(), 1);

  for (auto &i : ig) {
    --i;
```

```
  }

  idi.resize(size + 1);
  readBinaryFileOfData(input_folder / "idi", idi.data(), idi.size(), 1);

  for (auto &i : idi) {
    --i;
  }

  int ig_n_1 = ig[size];
  int di_n_1 = idi[size];

  jg.resize(ig_n_1);
  readBinaryFileOfData(input_folder / "jg", jg.data(), jg.size(), 1);

  for (auto &j : jg) {
    --j;
  }

  igg.resize(ig_n_1 + 1);
  readBinaryFileOfData(input_folder / "ijg", igg.data(), igg.size(), 1);

  for (auto &i : igg) {
    --i;
  }

  di.resize(di_n_1);
  readBinaryFileOfData(input_folder / "di", di.data(), di.size(), 1);

  int gg_count = igg[ig_n_1];

  gg.resize(gg_count);
  readBinaryFileOfData(input_folder / "gg", gg.data(), gg.size(), 1);

  pr.resize(size * 2);
  readBinaryFileOfData(input_folder / "pr", pr.data(), pr.size(), 1);

  matrix.di = std::move(di);
  matrix.gg = std::move(gg);
  matrix.idi = std::move(idi);
  matrix.ig = std::move(ig);
  matrix.jg = std::move(jg);
  matrix.igg = std::move(igg);

  b.data = std::move(pr);
  x.resize(b.size());
}
```

Листинг 4: symmetric_matrix.h

```
#ifndef SYMMETRIC_MATRIX_H
#define SYMMETRIC_MATRIX_H

#include "../include/cspan.h"
#include "../include/span.h"
#include "../include/vector.h"

#include <complex>
#include <optional>
```

```cpp
#include <vector>

class SymmetricMatrix {
public:
  std::vector<double> di;
  std::vector<double> gg;

  std::vector<int> idi;
  std::vector<int> igg;
  std::vector<int> ig;
  std::vector<int> jg;

  size_t size() const { return (idi.size() == 0) ? 0 : idi.size() - 1; }

  size_t totalItems() const { return (size() == 0) ? 0 : *ig.rbegin(); }

  std::optional<std::complex<double>> tryGetEntry(size_t row, size_t col) const;

  std::complex<double> at(size_t row, size_t col) const {
    auto val = tryGetEntry(row, col);
    return val.has_value() ? val.value() : std::complex<double>{};
  }

  static void multiplyMatrixBlockToVector(const CSpan &matSpan,
                                          const CSpan &vecSpan, Span resSpan);

  static void multiplyMatrixByVector(const SymmetricMatrix &mat,
                                     const Vector &x, Vector *res);

  static void getDiagonalPrecondition(const SymmetricMatrix &mat,
                                      Vector *diPrec);
};

#endif
```

Листинг 5: symmetric_matrix.cpp

```cpp
#include "../include/symmetric_matrix.h"

#include <algorithm>
#include <cstddef>
#include <omp.h>

static constexpr bool IGNORE_OMP = false;
static constexpr size_t NUM_THREADS = 8;
static constexpr size_t MIN_SIZE = 1000;

#define VEC_SCHEDULE schedule(static)

std::optional<std::complex<double>>
SymmetricMatrix::tryGetEntry(size_t row, size_t col) const {
  size_t size = this->size();

  if (row == col) {
    int blkBeg = idi[row];
    int blkSize = idi[row + 1] - blkBeg;

    double real = di[blkBeg];
    double imag = (blkSize == 2) ? di[blkBeg + 1] : 0.0;
```

```
    return std::complex<double>{real, imag};
  }

  if (col > row) {
    std::swap(row, col);
  }

  size_t start = static_cast<size_t>(ig[row]);
  size_t end = static_cast<size_t>(ig[row + 1]);

  if (start >= end)
    return std::nullopt;

  int target = static_cast<int>(col);
  auto it_begin = jg.begin() + static_cast<std::ptrdiff_t>(start);
  auto it_end = jg.begin() + static_cast<std::ptrdiff_t>(end);

  auto it = std::lower_bound(it_begin, it_end, target);

  if (it == it_end || *it != target) {
    return std::nullopt;
  }

  size_t el_id = static_cast<size_t>(std::distance(jg.begin(), it));

  int blkBeg = igg[el_id];
  int blkSize = igg[el_id + 1] - blkBeg;

  double real = gg[blkBeg];
  double imag = (blkSize == 2) ? gg[blkBeg + 1] : 0.0;

  return std::complex<double>{real, imag};
}

void SymmetricMatrix::multiplyMatrixBlockToVector(const CSpan &matSpan,
                                                  const CSpan &vecSpan,
                                                  Span resSpan) {
  auto mat = matSpan.begin();
  auto vec = vecSpan.begin();
  auto res = resSpan.begin();

  auto y0 = mat[0] * vec[0];
  auto y1 = mat[0] * vec[1];

  if (matSpan.size == 2) {
    y0 -= mat[1] * vec[1];
    y1 += mat[1] * vec[0];
  }

  res[0] += y0;
  res[1] += y1;
}

void SymmetricMatrix::multiplyMatrixByVector(const SymmetricMatrix &mat,
                                             const Vector &x, Vector *res) {
  res->clear();

  size_t size = mat.size();
  if (IGNORE_OMP || size < MIN_SIZE) {
```

```
    for (size_t i = 0; i < size; ++i) {
      size_t sz = mat.idi[i + 1] - mat.idi[i];
      auto matSpan = CSpan(mat.di, mat.idi[i], sz);
      SymmetricMatrix::multiplyMatrixBlockToVector(matSpan, x.cspan(i * 2, 2),
                                                   res->span(i * 2, 2));

      for (size_t j = mat.ig[i]; j < mat.ig[i + 1]; ++j) {
        size_t k = mat.jg[j];
        size_t sz = mat.igg[j + 1] - mat.igg[j];

        auto matSpan = CSpan(mat.gg, mat.igg[j], sz);
        SymmetricMatrix::multiplyMatrixBlockToVector(matSpan, x.cspan(k * 2, 2),
                                                     res->span(i * 2, 2));
        SymmetricMatrix::multiplyMatrixBlockToVector(matSpan, x.cspan(i * 2, 2),
                                                     res->span(k * 2, 2));
      }
    }
  } else {
    std::vector<Vector> thread_results(NUM_THREADS);

#pragma omp parallel num_threads(NUM_THREADS)
    {
      auto t_num = omp_get_thread_num();
      auto &t_res = thread_results[t_num];
      t_res.resize(size * 2);

#pragma omp for schedule(dynamic, 300)
      for (size_t i = 0; i < size; ++i) {
        size_t sz = mat.idi[i + 1] - mat.idi[i];
        auto matSpan = CSpan(mat.di, mat.idi[i], sz);
        SymmetricMatrix::multiplyMatrixBlockToVector(matSpan, x.cspan(i * 2, 2),
                                                     t_res.span(i * 2, 2));

        for (size_t j = mat.ig[i]; j < mat.ig[i + 1]; ++j) {
          size_t k = mat.jg[j];
          size_t sz = mat.igg[j + 1] - mat.igg[j];
          auto matSpan = CSpan(mat.gg, mat.igg[j], sz);
          SymmetricMatrix::multiplyMatrixBlockToVector(
              matSpan, x.cspan(k * 2, 2), t_res.span(i * 2, 2));
          SymmetricMatrix::multiplyMatrixBlockToVector(
              matSpan, x.cspan(i * 2, 2), t_res.span(k * 2, 2));
        }
      }
    }

#pragma omp parallel num_threads(NUM_THREADS)
    {
#pragma omp for VEC_SCHEDULE
      for (size_t j = 0; j < size * 2; ++j) {
        for (size_t i = 0; i < NUM_THREADS; ++i) {
          auto &t_res = thread_results[i].data;
          res->data[j] += t_res[j];
        }
      }
    }
  }
}

void SymmetricMatrix::getDiagonalPrecondition(const SymmetricMatrix &mat,
                                              Vector *diPrec) {
```

```
  auto size = mat.size();
  if (IGNORE_OMP || size < NUM_THREADS) {
    for (size_t i = 0; i < size; ++i) {
      int countItems = mat.idi[i + 1] - mat.idi[i];
      auto comp = std::complex<double>();

      comp.real(mat.di[mat.idi[i]]);

      if (countItems == 2) {
        comp.imag(mat.di[mat.idi[i]]);
      }

      comp = 1.0 / comp;

      diPrec->data[i * 2] = comp.real();
      diPrec->data[i * 2 + 1] = comp.imag();
    }
  } else {
#pragma omp parallel num_threads(NUM_THREADS)
    {
#pragma omp for VEC_SCHEDULE
      for (size_t i = 0; i < size; ++i) {
        int countItems = mat.idi[i + 1] - mat.idi[i];
        auto comp = std::complex<double>();

        comp.real(mat.di[mat.idi[i]]);

        if (countItems == 2) {
          comp.imag(mat.di[mat.idi[i]]);
        }

        comp = 1.0 / comp;

        diPrec->data[i * 2] = comp.real();
        diPrec->data[i * 2 + 1] = comp.imag();
      }
    }
  }
}
```

Листинг 6: vector.h

```
#ifndef VECTOR_H
#define VECTOR_H

#include <complex>
#include <vector>

#include "../include/cspan.h"
#include "../include/span.h"

class Vector {
public:
  std::vector<double> data;

  Vector() = default;
  explicit Vector(size_t size) { resize(size); }

  void resize(size_t size) { data.resize(size); }
```

```
  size_t size() const { return data.size(); }
  size_t blocks_count() const { return size() / 2; }

  std::complex<double> at(size_t ind) const {
    return {data.at(ind * 2), data.at(ind * 2 + 1)};
  }

  void setAt(size_t ind, const std::complex<double> &num) {
    data.at(ind * 2) = num.real();
    data.at(ind * 2 + 1) = num.imag();
  }

  void clear() {
    for (double &v : data) {
      v = 0.0;
    }
  }

  Span span(size_t start, size_t size) { return Span(data, start, size); }

  CSpan cspan(size_t start, size_t size) const {
    return CSpan(data, start, size);
  }

  static double getRealPartOfVectorsProduct(const Vector &x, const Vector &y);

  static std::complex<double> calculateVectorsProduct(const Vector &x,
                                                      const Vector &y);

  static std::complex<double>
  calculateVectorsProductWithoutConjugation(const Vector &x, const Vector &y);

  static double calculateNormOfRealPartOfVectorsProduct(const Vector &x);

  static void multiplyVectorByScalar(const Vector &x, double scal, Vector *y);

  static void multiplyVectorByVector(const Vector &x, const Vector &y,
                                     Vector *z);

  static void calculateLinearCombinationOfVectors(double mX, const Vector &x,
                                                  double mY, const Vector &y,
                                                  Vector *z);

  static void calculateLinearCombinationOfVectors(std::complex<double> mX,
                                                  const Vector &x,
                                                  std::complex<double> mY,
                                                  const Vector &y, Vector *z);

  static void copyVector(const Vector &x, Vector *y);
};

#endif
```

Листинг 7: vector.cpp

```
#include "../include/vector.h"

#include <omp.h>
```

```
static constexpr bool IGNORE_OMP = false;
static constexpr size_t NUM_THREADS = 8;
static constexpr size_t MIN_SIZE = 1000;

#define VEC_SCHEDULE schedule(static)

double Vector::getRealPartOfVectorsProduct(const Vector &x, const Vector &y) {
  return Vector::calculateVectorsProduct(x, y).real();
}

std::complex<double> Vector::calculateVectorsProduct(const Vector &x,
                                                     const Vector &y) {
  size_t size = x.blocks_count();
  std::complex<double> res;

  if (IGNORE_OMP || size * size < MIN_SIZE) {
    for (size_t i = 0; i < size; ++i) {
      std::complex<double> comp_l = {
          x.data[i * 2],
          -x.data[i * 2 + 1],
      };

      std::complex<double> comp_r = {
          y.data[i * 2],
          y.data[i * 2 + 1],
      };

      res += comp_l * comp_r;
    }

    return res;
  }
  double real_acc = 0.0;
  double imag_acc = 0.0;

#pragma omp parallel num_threads(NUM_THREADS)
  {
#pragma omp for reduction(+ : real_acc, imag_acc) VEC_SCHEDULE
    for (size_t i = 0; i < size; ++i) {
      const double xr = x.data[2 * i];
      const double xi = x.data[2 * i + 1];
      const double yr = y.data[2 * i];
      const double yi = y.data[2 * i + 1];

      real_acc += xr * yr + xi * yi;
      imag_acc += xr * yi - xi * yr;
    }
  }

  return std::complex<double>(real_acc, imag_acc);
}

std::complex<double>
Vector::calculateVectorsProductWithoutConjugation(const Vector &x,
                                                  const Vector &y) {
  auto size = x.blocks_count();
  std::complex<double> res;

  if (IGNORE_OMP || size * size < MIN_SIZE) {
    for (size_t i = 0; i < size; ++i) {
```

```
      std::complex<double> comp_l = {
          x.data[i * 2],
          x.data[i * 2 + 1],
      };

      std::complex<double> comp_r = {
          y.data[i * 2],
          y.data[i * 2 + 1],
      };

      res += comp_l * comp_r;
    }

    return res;
  }

  double real_acc = 0.0;
  double imag_acc = 0.0;

#pragma omp parallel num_threads(NUM_THREADS)
  {
#pragma omp for reduction(+ : real_acc, imag_acc) VEC_SCHEDULE
    for (size_t i = 0; i < size; ++i) {
      const double xr = x.data[2 * i];
      const double xi = x.data[2 * i + 1];
      const double yr = y.data[2 * i];
      const double yi = y.data[2 * i + 1];

      real_acc += xr * yr - xi * yi;
      imag_acc += xr * yi + xi * yr;
    }
  }

  return std::complex<double>(real_acc, imag_acc);
}

double Vector::calculateNormOfRealPartOfVectorsProduct(const Vector &x) {
  return std::sqrt(getRealPartOfVectorsProduct(x, x));
}

void Vector::multiplyVectorByScalar(const Vector &x, double scal, Vector *y) {
  size_t size = x.size();
  std::vector<double> &yy = y->data;
  const std::vector<double> &xx = x.data;

  if (IGNORE_OMP || size * size < MIN_SIZE) {
    for (size_t i = 0; i < size; ++i) {
      yy[i] = xx[i] * scal;
    }
  } else {
#pragma omp parallel num_threads(NUM_THREADS)
    {
#pragma omp for VEC_SCHEDULE
      for (size_t i = 0; i < size; ++i) {
        yy[i] = xx[i] * scal;
      }
    }
  }
}
```

```
void Vector::multiplyVectorByVector(const Vector &x, const Vector &y,
                                    Vector *z) {
  size_t size = x.blocks_count();

  if (IGNORE_OMP || size * size < MIN_SIZE) {
    for (size_t i = 0; i < size; ++i) {
      std::complex<double> xx = {x.data[i * 2], x.data[i * 2 + 1]};
      std::complex<double> yy = {y.data[i * 2], y.data[i * 2 + 1]};
      std::complex<double> res = xx * yy;

      z->data[i * 2] = res.real();
      z->data[i * 2 + 1] = res.imag();
    }
  } else {
#pragma omp parallel num_threads(NUM_THREADS)
    {
#pragma omp for VEC_SCHEDULE
      for (size_t i = 0; i < size; ++i) {
        std::complex<double> xx = {x.data[i * 2], x.data[i * 2 + 1]};
        std::complex<double> yy = {y.data[i * 2], y.data[i * 2 + 1]};
        std::complex<double> res = xx * yy;

        z->data[i * 2] = res.real();
        z->data[i * 2 + 1] = res.imag();
      }
    }
  }
}

void Vector::calculateLinearCombinationOfVectors(double mX, const Vector &x,
                                                 double mY, const Vector &y,
                                                 Vector *z) {

  size_t size = x.size();
  const std::vector<double> &xx = x.data;
  const std::vector<double> &yy = y.data;
  std::vector<double> &zz = z->data;

  if (IGNORE_OMP || size * size < MIN_SIZE) {
    for (size_t i = 0; i < size; ++i) {
      zz[i] = mX * xx[i] + mY * yy[i];
    }
  } else {
#pragma omp parallel num_threads(NUM_THREADS)
    {
#pragma omp for VEC_SCHEDULE
      for (size_t i = 0; i < size; ++i) {
        zz[i] = mX * xx[i] + mY * yy[i];
      }
    }
  }
}

void Vector::calculateLinearCombinationOfVectors(std::complex<double> mX,
                                                 const Vector &x,
                                                 std::complex<double> mY,
                                                 const Vector &y, Vector *z) {
  size_t size = x.blocks_count();
  const std::vector<double> &xx = x.data;
  const std::vector<double> &yy = y.data;
```

```
  std::vector<double> &zz = z->data;

  if (IGNORE_OMP || size * size < MIN_SIZE) {
    for (size_t i = 0; i < size; ++i) {
      std::complex<double> x_comp = {xx[i * 2], xx[i * 2 + 1]};
      std::complex<double> y_comp = {yy[i * 2], yy[i * 2 + 1]};
      std::complex<double> res = mX * x_comp + mY * y_comp;

      zz[i * 2] = res.real();
      zz[i * 2 + 1] = res.imag();
    }
  } else {
#pragma omp parallel num_threads(NUM_THREADS)
    {
#pragma omp for VEC_SCHEDULE
      for (size_t i = 0; i < size; ++i) {
        std::complex<double> x_comp = {xx[i * 2], xx[i * 2 + 1]};
        std::complex<double> y_comp = {yy[i * 2], yy[i * 2 + 1]};
        std::complex<double> res = mX * x_comp + mY * y_comp;

        zz[i * 2] = res.real();
        zz[i * 2 + 1] = res.imag();
      }
    }
  }
}

void Vector::copyVector(const Vector &x, Vector *y) {
  Vector::multiplyVectorByScalar(x, 1.0, y);
}
```

Листинг 8: COCR_Di_solver.h

```
#ifndef COCR_DI_SOLVER_H
#define COCR_DI_SOLVER_H

#include "../include/solver_result.h"
#include "../include/symmetric_matrix.h"
#include "../include/vector.h"

class COCRDiSolver {
public:
  SolverResult solve(const SymmetricMatrix &A, const Vector &b, Vector &x,
                     const bool isSmoothSolving);

  void setSolverParameters(int maximum_iterations_s, double epsilon_s,
                           int print_interval_s);

private:
  int maximum_iterations = 0;
  double epsilon = 0.0;
  int print_interval = 0;

  SolverResult solveByBasicMethod(const SymmetricMatrix &A, const Vector &b,
                                  Vector &x);

  SolverResult solveBySmoothMethod(const SymmetricMatrix &A, const Vector &b,
                                   Vector &x);
};
```

```
#endif
```

Листинг 9: COCR_Di_solver.cpp

```
#include "../include/COCR_Di_solver.h"
#include "../include/solver_result.h"
#include "../include/timer.h"

#include <iomanip>
#include <iostream>

void COCRDiSolver::setSolverParameters(int maximum_iterations_s,
                                       double epsilon_s, int print_interval_s) {
  maximum_iterations = maximum_iterations_s;
  print_interval = print_interval_s;
  epsilon = epsilon_s;
}

SolverResult COCRDiSolver::solve(const SymmetricMatrix &A, const Vector &b,
                                 Vector &x, const bool isSmoothSolving) {
  if (isSmoothSolving) {
    return solveBySmoothMethod(A, b, x);
  }

  return solveByBasicMethod(A, b, x);
}

SolverResult COCRDiSolver::solveByBasicMethod(const SymmetricMatrix &A,
                                              const Vector &b, Vector &x) {
  SolverResult _res;
  _res.eps.reserve(maximum_iterations);

  Vector r(b.size());
  Vector r0(b.size());
  Vector p(b.size());
  Vector di(b.size());
  Vector Ap(b.size());
  Vector z(b.size());
  Vector s(b.size());
  Vector a(b.size());
  Vector w(b.size());

  std::complex<double> scal_a_s;
  double norm_r;
  double norm_b;

  Timer t;

  t.start();

  SymmetricMatrix::getDiagonalPrecondition(A, &di); // di = 1.0 / A.di

  SymmetricMatrix::multiplyMatrixByVector(A, x, &r); // r = b - A*x
  Vector::calculateLinearCombinationOfVectors(1.0, b, -1.0, r,
                                              &r); // r = b - A*x

  Vector::copyVector(r, &r0);                    // r0 = r
  Vector::multiplyVectorByVector(di, r, &s); // s = di^-1 * r
```

```
Vector::copyVector(s, &p);                          // p = s

SymmetricMatrix::multiplyMatrixByVector(A, s, &z); // z = A*s
Vector::copyVector(z, &a);                          // a = z

Vector::multiplyVectorByVector(di, z, &w); // w = di^-1 * z

scal_a_s = Vector::calculateVectorsProductWithoutConjugation(a, s);
norm_r = Vector::calculateNormOfRealPartOfVectorsProduct(r);
norm_b = Vector::calculateNormOfRealPartOfVectorsProduct(b);

double eps = norm_r / norm_b;
size_t iteration = 0;

std::cout << "Iter: " << std::setw(5) << iteration
          << ", eps: " << std::setprecision(14) << std::scientific << eps
          << '\n';

_res.eps.push_back(eps);

for (iteration = 1; iteration <= maximum_iterations && eps > epsilon;
     ++iteration) {

  auto alpha = Vector::calculateVectorsProductWithoutConjugation(a, s) /
               Vector::calculateVectorsProductWithoutConjugation(z, w);

  Vector::calculateLinearCombinationOfVectors({1.0, 0.0}, x, alpha, p, &x);
  Vector::calculateLinearCombinationOfVectors({1.0, 0.0}, r, -alpha, z, &r);
  Vector::calculateLinearCombinationOfVectors({1.0, 0.0}, s, -alpha, w, &s);

  SymmetricMatrix::multiplyMatrixByVector(A, s, &a);

  auto norm_r = Vector::calculateNormOfRealPartOfVectorsProduct(r);
  eps = norm_r / norm_b;
  _res.eps.push_back(eps);

  auto next_scal_a_s =
      Vector::calculateVectorsProductWithoutConjugation(a, s);

  auto beta = next_scal_a_s / scal_a_s;

  scal_a_s = next_scal_a_s;

  Vector::calculateLinearCombinationOfVectors({1.0, 0.0}, s, beta, p, &p);
  Vector::calculateLinearCombinationOfVectors({1.0, 0.0}, a, beta, z, &z);

  Vector::multiplyVectorByVector(di, z, &w);

  if (iteration % print_interval == 0) {
    std::cout << "Iter: " << std::setw(5) << iteration
              << ", eps: " << std::setprecision(14) << std::scientific << eps
              << '\n';
  }
}

t.stop();

_res.reachedEps = eps;
_res.totalIterations = iteration - 1;
_res.totalTimeSeconds = t.elapsedSeconds();
```

```
  _res.isSolved = true;

  std::cout << "COCG diag precond solver finished\n";
  std::cout << " - Elapsed time: " << t.elapsedSeconds() << " seconds\n";
  std::cout << " - Total iterations: " << std::setw(6) << (iteration - 1)
            << " (limit: " << std::setw(6) << maximum_iterations << ")\n";
  std::cout << " - Reached epsilon: " << std::setw(6) << std::scientific
            << std::setprecision(2) << eps << " (target: " << std::setw(6)
            << epsilon << ")\n";

  SymmetricMatrix::multiplyMatrixByVector(A, x, &r);
  Vector::calculateLinearCombinationOfVectors(1.0, b, -1.0, r, &r);
  auto norm_residual = Vector::calculateNormOfRealPartOfVectorsProduct(r) /
                       Vector::calculateNormOfRealPartOfVectorsProduct(r0);
  std::cout << " - Norm of relative residual: " << std::scientific
            << std::setprecision(2) << norm_residual << "\n";

  if (iteration - 1 == maximum_iterations && std::abs(eps / epsilon) > 5) {
    std::cout << "Warning: Solver didn't reach target epsilon.\n";
    _res.isSolved = false;
  }

  return _res;
}

SolverResult COCRDiSolver::solveBySmoothMethod(const SymmetricMatrix &A,
                                               const Vector &b, Vector &x) {
  SolverResult _res;
  _res.eps.reserve(maximum_iterations);

  Vector r(b.size());
  Vector r0(b.size());
  Vector p(b.size());
  Vector di(b.size());
  Vector Ap(b.size());
  Vector z(b.size());
  Vector s(b.size());
  Vector a(b.size());
  Vector w(b.size());

  Vector resid_y(b.size());
  Vector resid_s(b.size());

  Vector r_resid_s(b.size());

  std::complex<double> scal_a_s;
  double norm_r;
  double norm_b;

  Timer t;

  t.start();

  SymmetricMatrix::getDiagonalPrecondition(A, &di); // di = 1.0 / A.di

  SymmetricMatrix::multiplyMatrixByVector(A, x, &r); // r = b - A*x
  Vector::calculateLinearCombinationOfVectors(1.0, b, -1.0, r,
                                              &r); // r = b - A*x

  Vector::copyVector(r, &r0);                    // r0 = r
```

```
Vector::multiplyVectorByVector(di, r, &s); // s = di^-1 * r
Vector::copyVector(s, &p);                 // p = s

SymmetricMatrix::multiplyMatrixByVector(A, s, &z); // z = A*s
Vector::copyVector(z, &a);                         // a = z

Vector::multiplyVectorByVector(di, z, &w); // w = di^-1 * z

Vector::copyVector(x, &resid_y);
Vector::copyVector(r, &resid_s);

scal_a_s = Vector::calculateVectorsProductWithoutConjugation(a, s);
norm_r = Vector::calculateNormOfRealPartOfVectorsProduct(resid_s);
norm_b = Vector::calculateNormOfRealPartOfVectorsProduct(b);

double eps = norm_r / norm_b;
size_t iteration = 0;

std::cout << "Iter: " << std::setw(5) << iteration
          << ", eps: " << std::setprecision(14) << std::scientific << eps
          << '\n';

_res.eps.push_back(eps);

for (iteration = 1; iteration <= maximum_iterations && eps > epsilon;
     ++iteration) {

  auto alpha =
      scal_a_s / Vector::calculateVectorsProductWithoutConjugation(z, w);

  Vector::calculateLinearCombinationOfVectors({1.0, 0.0}, x, alpha, p, &x);
  Vector::calculateLinearCombinationOfVectors({1.0, 0.0}, r, -alpha, z, &r);
  Vector::calculateLinearCombinationOfVectors({1.0, 0.0}, s, -alpha, w, &s);

  Vector::calculateLinearCombinationOfVectors(1.0, r, -1.0, resid_s,
                                              &r_resid_s);
  double eta = -Vector::getRealPartOfVectorsProduct(resid_s, r_resid_s) /
               Vector::getRealPartOfVectorsProduct(r_resid_s, r_resid_s);
  eta = std::min(std::max(eta, 0.0), 1.0);
  Vector::calculateLinearCombinationOfVectors((1 - eta), resid_y, eta, x,
                                              &resid_y);
  Vector::calculateLinearCombinationOfVectors((1 - eta), resid_s, eta, r,
                                              &resid_s);

  SymmetricMatrix::multiplyMatrixByVector(A, s, &a);

  auto norm_s = Vector::calculateNormOfRealPartOfVectorsProduct(resid_s);
  eps = norm_s / norm_b;
  _res.eps.push_back(eps);

  auto next_scal_a_s =
      Vector::calculateVectorsProductWithoutConjugation(a, s);

  auto beta = next_scal_a_s / scal_a_s;

  scal_a_s = next_scal_a_s;

  Vector::calculateLinearCombinationOfVectors({1.0, 0.0}, s, beta, p, &p);
  Vector::calculateLinearCombinationOfVectors({1.0, 0.0}, a, beta, z, &z);
```

```
    Vector::multiplyVectorByVector(di, z, &w);

    if (iteration % print_interval == 0) {
      std::cout << "Iter: " << std::setw(5) << iteration
                << ", eps: " << std::setprecision(14) << std::scientific << eps
                << '\n';
    }
  }

  t.stop();

  Vector::copyVector(resid_y, &x);

  _res.reachedEps = eps;
  _res.totalIterations = iteration - 1;
  _res.totalTimeSeconds = t.elapsedSeconds();
  _res.isSolved = true;

  std::cout << "COCG diag precond solver finished\n";
  std::cout << " - Elapsed time: " << t.elapsedSeconds() << " seconds\n";
  std::cout << " - Total iterations: " << std::setw(6) << (iteration - 1)
            << " (limit: " << std::setw(6) << maximum_iterations << ")\n";
  std::cout << " - Reached epsilon: " << std::setw(6) << std::scientific
            << std::setprecision(2) << eps << " (target: " << std::setw(6)
            << epsilon << ")\n";

  SymmetricMatrix::multiplyMatrixByVector(A, x, &r);
  Vector::calculateLinearCombinationOfVectors(1.0, b, -1.0, r, &r);
  auto norm_residual = Vector::calculateNormOfRealPartOfVectorsProduct(r) /
                       Vector::calculateNormOfRealPartOfVectorsProduct(r0);
  std::cout << " - Norm of relative residual: " << std::scientific
            << std::setprecision(2) << norm_residual << "\n";

  if (iteration - 1 == maximum_iterations && std::abs(eps / epsilon) > 5) {
    std::cout << "Warning: Solver didn't reach target epsilon.\n";
    _res.isSolved = false;
  }

  return _res;
}
```

Листинг 10: COCG_Di_solver.h

```
#ifndef COCG_DI_SOLVER_H
#define COCG_DI_SOLVER_H

#include "../include/solver_result.h"
#include "../include/symmetric_matrix.h"
#include "../include/vector.h"

class COCGDiSolver {
public:
  SolverResult solve(const SymmetricMatrix &A, const Vector &b, Vector &x,
                     const bool isSmoothSolving);

  void setSolverParameters(int maximum_iterations_s, double epsilon_s,
                           int print_interval_s);

private:
```

```
  int maximum_iterations = 0;
  double epsilon = 0.0;
  int print_interval = 0;

  SolverResult solveByBasicMethod(const SymmetricMatrix &A, const Vector &b,
                                  Vector &x);

  SolverResult solveBySmoothMethod(const SymmetricMatrix &A, const Vector &b,
                                   Vector &x);
};

#endif
```

Листинг 11: COCG_Di_solver.cpp

```
#include "../include/COCG_Di_solver.h"
#include "../include/solver_result.h"
#include "../include/timer.h"

#include <iomanip>
#include <iostream>

void COCGDiSolver::setSolverParameters(int maximum_iterations_s,
                                       double epsilon_s, int print_interval_s) {
  maximum_iterations = maximum_iterations_s;
  print_interval = print_interval_s;
  epsilon = epsilon_s;
}

SolverResult COCGDiSolver::solve(const SymmetricMatrix &A, const Vector &b,
                                 Vector &x, const bool isSmoothSolving) {
  if (isSmoothSolving) {
    return solveBySmoothMethod(A, b, x);
  }

  return solveByBasicMethod(A, b, x);
}

SolverResult COCGDiSolver::solveByBasicMethod(const SymmetricMatrix &A,
                                              const Vector &b, Vector &x) {
  SolverResult _res;
  _res.eps.reserve(maximum_iterations);

  Vector r(b.size());
  Vector r0(b.size());
  Vector p(b.size());
  Vector di(b.size());
  Vector Ap(b.size());
  Vector z(b.size());

  std::complex<double> scal_r_z;
  double norm_r;
  double norm_b;

  Timer t;

  t.start();

  SymmetricMatrix::getDiagonalPrecondition(A, &di);
```

```
SymmetricMatrix::multiplyMatrixByVector(A, x, &r);
Vector::calculateLinearCombinationOfVectors(1.0, b, -1.0, r, &r);
Vector::copyVector(r, &r0);
Vector::multiplyVectorByVector(di, r, &z);
Vector::copyVector(z, &p);

scal_r_z = Vector::calculateVectorsProductWithoutConjugation(r, z);
norm_r = Vector::calculateNormOfRealPartOfVectorsProduct(r);
norm_b = Vector::calculateNormOfRealPartOfVectorsProduct(b);

double eps = norm_r / norm_b;
size_t iteration = 0;

std::cout << "Iter: " << std::setw(5) << iteration
          << ", eps: " << std::setprecision(14) << std::scientific << eps
          << '\n';

_res.eps.push_back(eps);

for (iteration = 1; iteration <= maximum_iterations && eps > epsilon;
     ++iteration) {
  SymmetricMatrix::multiplyMatrixByVector(A, p, &Ap);

  auto alpha = Vector::calculateVectorsProductWithoutConjugation(r, z) /
               Vector::calculateVectorsProductWithoutConjugation(Ap, p);

  Vector::calculateLinearCombinationOfVectors({1.0, 0.0}, x, alpha, p, &x);
  Vector::calculateLinearCombinationOfVectors({1.0, 0.0}, r, -alpha, Ap, &r);
  auto norm_r = Vector::calculateNormOfRealPartOfVectorsProduct(r);
  eps = norm_r / norm_b;
  _res.eps.push_back(eps);

  Vector::multiplyVectorByVector(di, r, &z);

  auto next_scal_r_z =
      Vector::calculateVectorsProductWithoutConjugation(r, z);
  auto beta = next_scal_r_z / scal_r_z;
  scal_r_z = next_scal_r_z;

  Vector::calculateLinearCombinationOfVectors({1.0, 0.0}, z, beta, p, &p);

  if (iteration % print_interval == 0) {
    std::cout << "Iter: " << std::setw(5) << iteration
              << ", eps: " << std::setprecision(14) << std::scientific << eps
              << '\n';
  }
}

t.stop();

_res.reachedEps = eps;
_res.totalIterations = iteration - 1;
_res.totalTimeSeconds = t.elapsedSeconds();
_res.isSolved = true;

std::cout << "COCG diag precond solver finished\n";
std::cout << " - Elapsed time: " << t.elapsedSeconds() << " seconds\n";
std::cout << " - Total iterations: " << std::setw(6) << (iteration - 1)
          << " (limit: " << std::setw(6) << maximum_iterations << ")\n";
std::cout << " - Reached epsilon: " << std::setw(6) << std::scientific
```

```
            << std::setprecision(2) << eps << " (target: " << std::setw(6)
            << epsilon << ")\n";

  SymmetricMatrix::multiplyMatrixByVector(A, x, &r);
  Vector::calculateLinearCombinationOfVectors(1.0, b, -1.0, r, &r);
  auto norm_residual = Vector::calculateNormOfRealPartOfVectorsProduct(r) /
                       Vector::calculateNormOfRealPartOfVectorsProduct(r0);
  std::cout << " - Norm of relative residual: " << std::scientific
            << std::setprecision(2) << norm_residual << "\n";

  if (iteration - 1 == maximum_iterations && std::abs(eps / epsilon) > 5) {
    std::cout << "Warning: Solver didn't reach target epsilon.\n";
    _res.isSolved = false;
  }

  return _res;
}

SolverResult COCGDiSolver::solveBySmoothMethod(const SymmetricMatrix &A,
                                               const Vector &b, Vector &x) {
  SolverResult _res;
  _res.eps.reserve(maximum_iterations);

  Timer t;
  t.start();

  Vector r(b.size());
  Vector r0(b.size());
  Vector p(b.size());
  Vector di(b.size());
  Vector Ap(b.size());
  Vector z(b.size());
  Vector y(x.size());
  Vector s(r.size());
  Vector rs(r.size());

  std::complex<double> scal_r_z;
  double norm_s;
  double norm_b;

  SymmetricMatrix::getDiagonalPrecondition(A, &di);
  SymmetricMatrix::multiplyMatrixByVector(A, x, &r);
  Vector::calculateLinearCombinationOfVectors(1.0, b, -1.0, r, &r);
  Vector::copyVector(r, &r0);
  Vector::multiplyVectorByVector(di, r, &z);
  Vector::copyVector(z, &p);
  Vector::copyVector(x, &y);
  Vector::copyVector(r, &s);

  scal_r_z = Vector::calculateVectorsProductWithoutConjugation(r, z);
  norm_s = Vector::calculateNormOfRealPartOfVectorsProduct(s);
  norm_b = Vector::calculateNormOfRealPartOfVectorsProduct(b);

  double eps = norm_s / norm_b;
  size_t iteration = 0;

  std::cout << "Iter: " << std::setw(5) << iteration
            << ", eps: " << std::setprecision(14) << std::scientific << eps
            << '\n';
```

```
  _res.eps.push_back(eps);

  for (iteration = 1; iteration <= maximum_iterations && eps > epsilon;
       ++iteration) {
    SymmetricMatrix::multiplyMatrixByVector(A, p, &Ap);

    auto alpha = Vector::calculateVectorsProductWithoutConjugation(r, z) /
                 Vector::calculateVectorsProductWithoutConjugation(Ap, p);

    Vector::calculateLinearCombinationOfVectors({1.0, 0.0}, x, alpha, p, &x);
    Vector::calculateLinearCombinationOfVectors({1.0, 0.0}, r, -alpha, Ap, &r);

    Vector::calculateLinearCombinationOfVectors(1.0, r, -1.0, s, &rs);
    double eta = -Vector::getRealPartOfVectorsProduct(s, rs) /
                 Vector::getRealPartOfVectorsProduct(rs, rs);
    eta = std::min(std::max(eta, 0.0), 1.0);
    Vector::calculateLinearCombinationOfVectors((1 - eta), y, eta, x, &y);
    Vector::calculateLinearCombinationOfVectors((1 - eta), s, eta, r, &s);

    auto norm_s = Vector::calculateNormOfRealPartOfVectorsProduct(s);
    eps = norm_s / norm_b;
    _res.eps.push_back(eps);

    Vector::multiplyVectorByVector(di, r, &z);

    auto next_scal_r_z =
        Vector::calculateVectorsProductWithoutConjugation(r, z);
    auto beta = next_scal_r_z / scal_r_z;
    scal_r_z = next_scal_r_z;

    Vector::calculateLinearCombinationOfVectors({1.0, 0.0}, z, beta, p, &p);

    if (iteration % print_interval == 0) {
      std::cout << "Iter: " << std::setw(5) << iteration
                << ", eps: " << std::setprecision(14) << std::scientific << eps
                << '\n';
    }
  }

  Vector::copyVector(y, &x);

  t.stop();
  _res.reachedEps = eps;
  _res.totalIterations = iteration - 1;
  _res.totalTimeSeconds = t.elapsedSeconds();
  _res.isSolved = true;

  std::cout << "COCG diag precond solver finished\n";
  std::cout << " - Elapsed time: " << t.elapsedSeconds() << " seconds\n";
  std::cout << " - Total iterations: " << std::setw(6) << (iteration - 1)
            << " (limit: " << std::setw(6) << maximum_iterations << ")\n";
  std::cout << " - Reached epsilon: " << std::setw(6) << std::scientific
            << std::setprecision(2) << eps << " (target: " << std::setw(6)
            << epsilon << ")\n";

  SymmetricMatrix::multiplyMatrixByVector(A, x, &r);
  Vector::calculateLinearCombinationOfVectors(1.0, b, -1.0, r, &r);
  auto norm_residual = Vector::calculateNormOfRealPartOfVectorsProduct(r) /
                       Vector::calculateNormOfRealPartOfVectorsProduct(r0);
  std::cout << " - Norm of relative residual: " << std::scientific
```

```
            << std::setprecision(2) << norm_residual << "\n";

  if (iteration - 1 == maximum_iterations && std::abs(eps / epsilon) > 5) {
    std::cout << "Warning: Solver didn't reach target epsilon.\n";
    _res.isSolved = false;
  }

  std::cout << "\n";

  return _res;
}
```