|  |  |  |
| --- | --- | --- |
|  | | |
| Федеральное государственное бюджетное  образовательное учреждение высшего образования «Новосибирский государственный технический университет» | | |
|  | | |
| Кафедра прикладной математики | | |
| Практическое работа № 3 | | |
| по дисциплине «Численные методы» | | |
| **РЕШЕНИЕ РАЗРЕЖЕННЫХ СЛАУ ТРЕХШАГОВЫМИ ИТЕРАЦИОННЫМИ МЕТОДАМИ  С ПРЕДОБУСЛОВЛИВАНИЕМ** | | |
|  | | |
|  |  |  |
| Группа ПМ-01 | будник светлана |
|  | самсонов семён |
| Вариант 11 |  |
|  |  |
| Преподаватель | Задорожный александр геннадьевич |
|  |  |
|  |  |
| Новосибирск, 2022 | | |

# Цели работы

# Изучить особенности реализации трехшаговых итерационных методов для СЛАУ с разреженными матрицами. Исследовать влияние предобусловливания на сходимость изучаемых методов на нескольких матрицах большой (не менее 10000) размерности.

# Входные данные

**Вариант 11:**

Сравнить МСГ и ЛОС для несимметричной матрицы. Факторизация LU(sq) .Формат хранения матрицы: разреженный строчно-столбцовый

# Анализ

**Входные данные:** система линейных алгебраических уравнений **Ax = F**.

**Выходные данные:** вектор неизвестных **.**

Метод сопряжённых градиентов (МСГ) в адаптации для несимметричных матриц:

Поскольку изначальный алгоритм предназначен для решения симметричных матриц, то мы домножим исходное уравнение на транспонированную матрицу А слева, приведя систему к системе с симметричной матрицей:

Выбирается начальное приближение x0 и полагается:

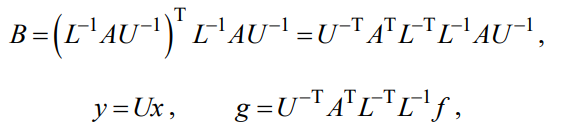
Далее для k = 1, 2, … проводятся следующие вычисления:

Выход из итерационного процесса происходит либо по превышению максимально допустимого числа итераций, либо по условию малости относительной невязки:

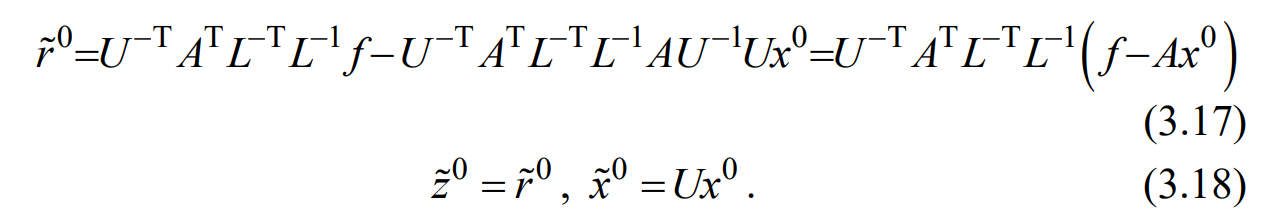
Для ускорения сходимости будем использовать предобуславливание матрицы А неполной факторизацией:

И в таком случае мы будем решать следующую систему:

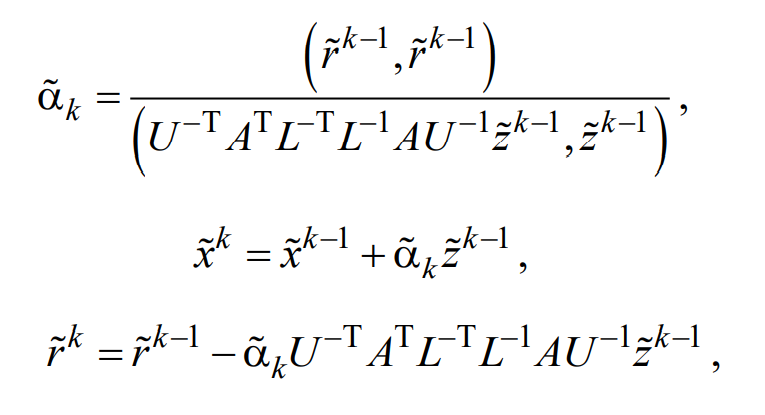
В которой:

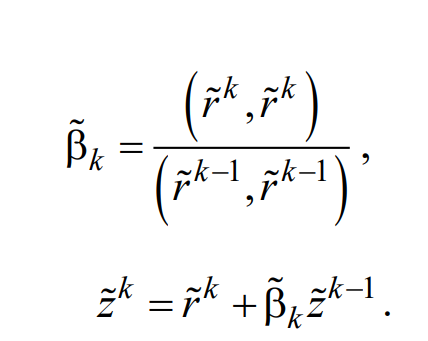


Аналогично выбирается начальное приближение x0 и полагается:

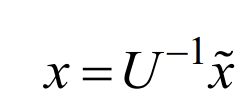


Далее для k = 1, 2, … проводятся следующие вычисления:





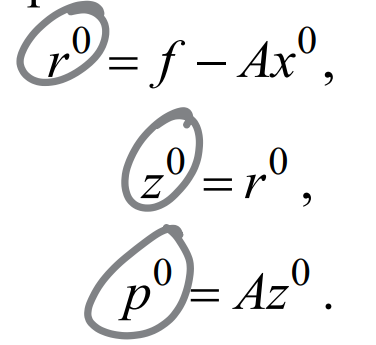
По окончании итерационного процесса вектор решения вычисляется следующим образом:



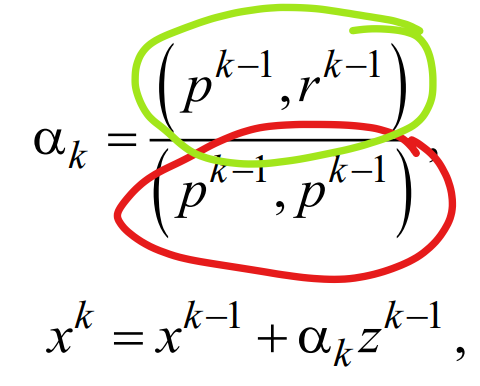
Для неполной факторизации можно принять (диагональ матрицы А).

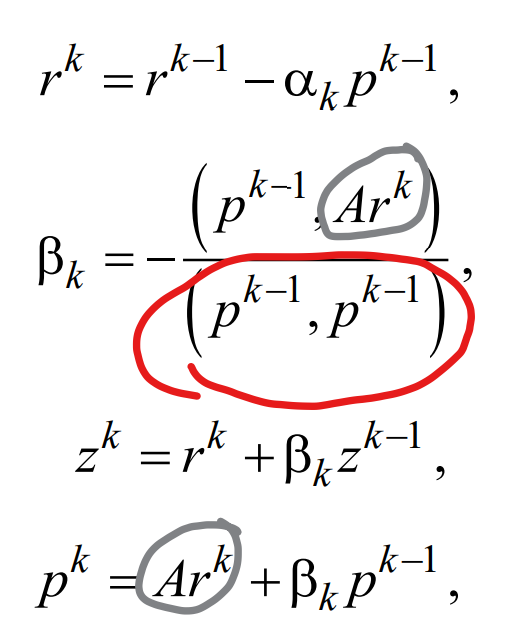
Локально-оптимальная схема (ЛОС):

Выбирается начальное приближение x0 и полагается:



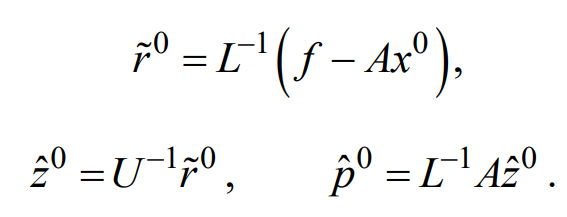
Далее для k = 1, 2, … проводятся следующие вычисления:



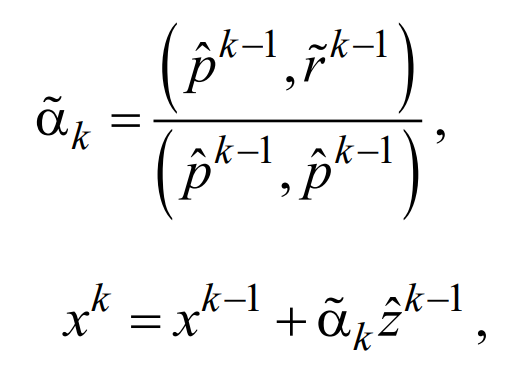


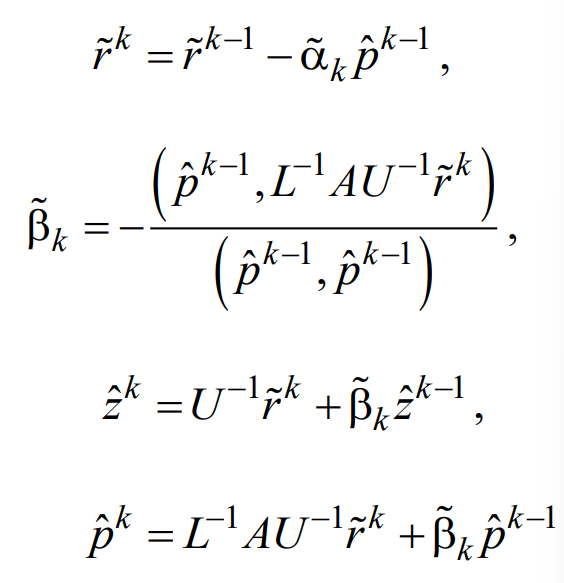
Выход из итерационного процесса происходит либо по превышению максимально допустимого числа итераций, либо по условию малости относительной невязки:

При использовании неполной факторизации этот метод преобразуется к следующему виду:



Далее для k = 1, 2, … проводятся следующие вычисления:

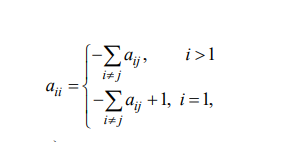




Выход из итерационного процесса происходит либо по превышению максимально допустимого числа итераций, либо по условию малости относительной невязки:

# Исследование

Матрица А строится по правилу:



****

Матрица А:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 16 | -4 | 0 | 0 | 0 | 0 | -4 | -3 | -4 | 0 | 0 | 0 |
| -4 | 10 | -1 | 0 | 0 | 0 | 0 | -2 | -2 | -1 | 0 | 0 |
| 0 | -3 | 7 | -1 | 0 | 0 | 0 | 0 | 0 | -3 | 0 | 0 |
| 0 | 0 | -3 | 14 | -1 | 0 | 0 | 0 | 0 | -3 | -3 | -4 |
| 0 | 0 | 0 | -3 | 9 | -4 | 0 | 0 | 0 | 0 | -2 | 0 |
| 0 | 0 | 0 | 0 | -3 | 8 | -4 | 0 | 0 | 0 | 0 | -1 |
| -4 | 0 | 0 | 0 | 0 | -4 | 9 | -1 | 0 | 0 | 0 | 0 |
| -3 | -2 | 0 | 0 | 0 | 0 | -4 | 9 | 0 | 0 | 0 | 0 |
| 0 | -3 | -1 | 0 | 0 | 0 | 0 | 0 | 7 | -3 | 0 | 0 |
| 0 | -2 | -1 | -1 | 0 | 0 | 0 | 0 | 0 | 5 | -1 | 0 |
| 0 | 0 | -4 | -3 | -4 | 0 | 0 | 0 | 0 | -1 | 14 | -2 |
| 0 | 0 | 0 | -4 | -1 | -1 | 0 | 0 | 0 | 0 | -4 | 10 |

|  |
| --- |
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
| 6 |
| 7 |
| 8 |
| 9 |
| 10 |
| 11 |
| 12 |

|  |
| --- |
| -80 |
| -31 |
| -19 |
| -69 |
| -13 |
| -7 |
| 27 |
| 37 |
| 24 |
| 28 |
| 76 |
| 49 |

Х

|  |
| --- |
| 112 |
| 71 |
| 61 |
| 181 |
| 103 |
| 103 |
| 99 |
| 107 |
| 102 |
| 72 |
| 232 |
| 191 |

Матрица B (матрица А с обратным знаком внедиагональных элементов):

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 16 | 4 | 0 | 0 | 0 | 0 | 4 | 3 | 4 | 0 | 0 | 0 |
| 4 | 10 | 1 | 0 | 0 | 0 | 0 | 2 | 2 | 1 | 0 | 0 |
| 0 | 3 | 7 | 1 | 0 | 0 | 0 | 0 | 0 | 3 | 0 | 0 |
| 0 | 0 | 3 | 14 | 1 | 0 | 0 | 0 | 0 | 3 | 3 | 4 |
| 0 | 0 | 0 | 3 | 9 | 4 | 0 | 0 | 0 | 0 | 2 | 0 |
| 0 | 0 | 0 | 0 | 3 | 8 | 4 | 0 | 0 | 0 | 0 | 1 |
| 4 | 0 | 0 | 0 | 0 | 4 | 9 | 1 | 0 | 0 | 0 | 0 |
| 3 | 2 | 0 | 0 | 0 | 0 | 4 | 9 | 0 | 0 | 0 | 0 |
| 0 | 3 | 1 | 0 | 0 | 0 | 0 | 0 | 7 | 3 | 0 | 0 |
| 0 | 2 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 5 | 1 | 0 |
| 0 | 0 | 4 | 3 | 4 | 0 | 0 | 0 | 0 | 1 | 14 | 2 |
| 0 | 0 | 0 | 4 | 1 | 1 | 0 | 0 | 0 | 0 | 4 | 10 |

|  |
| --- |
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
| 6 |
| 7 |
| 8 |
| 9 |
| 10 |
| 11 |
| 12 |

Х

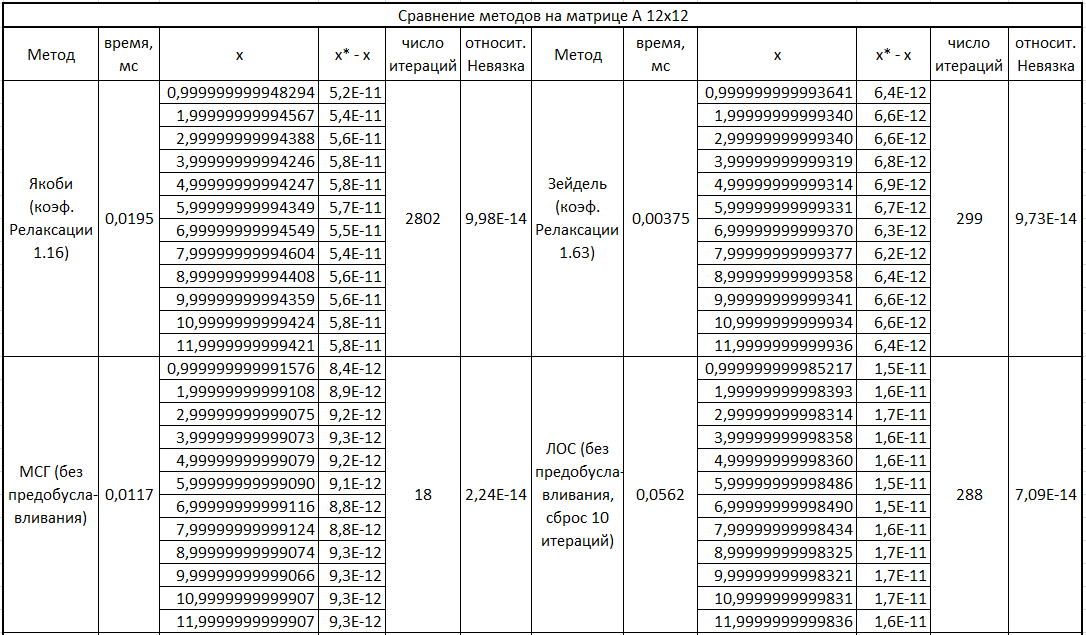
**Параметры тестирования для матриц:**

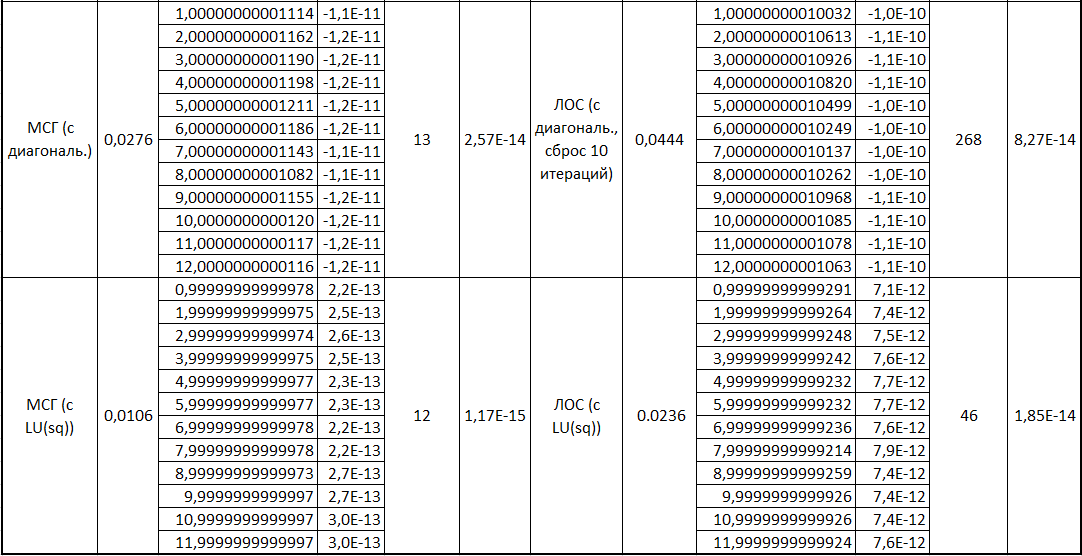
Максимальное число итераций: 2000

Начальное приближение: нулевой вектор

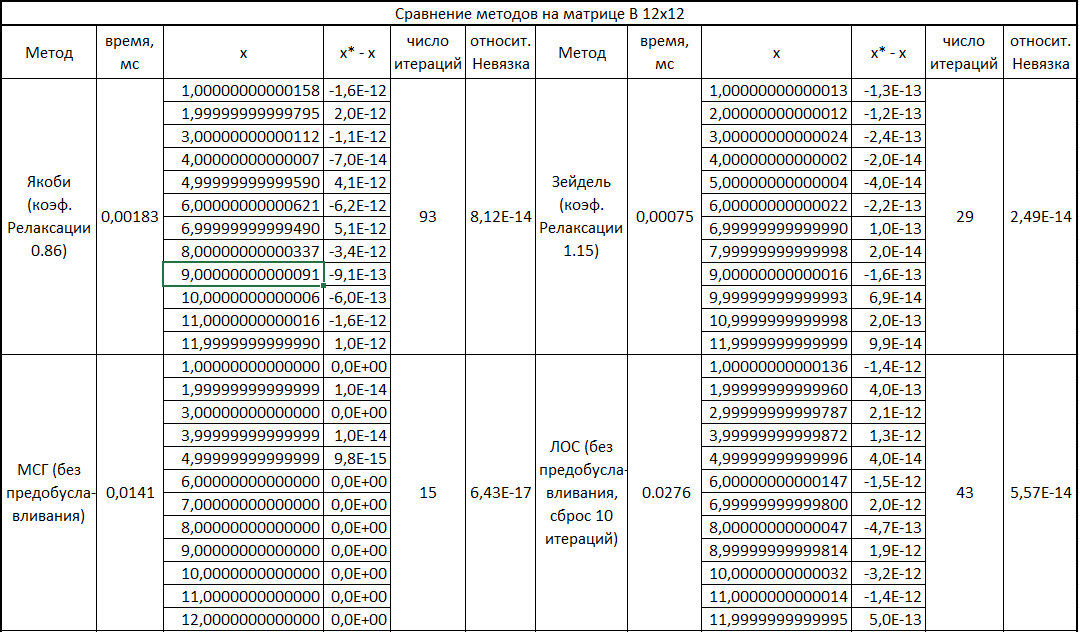
Минимальная относительная невязка: 10-13

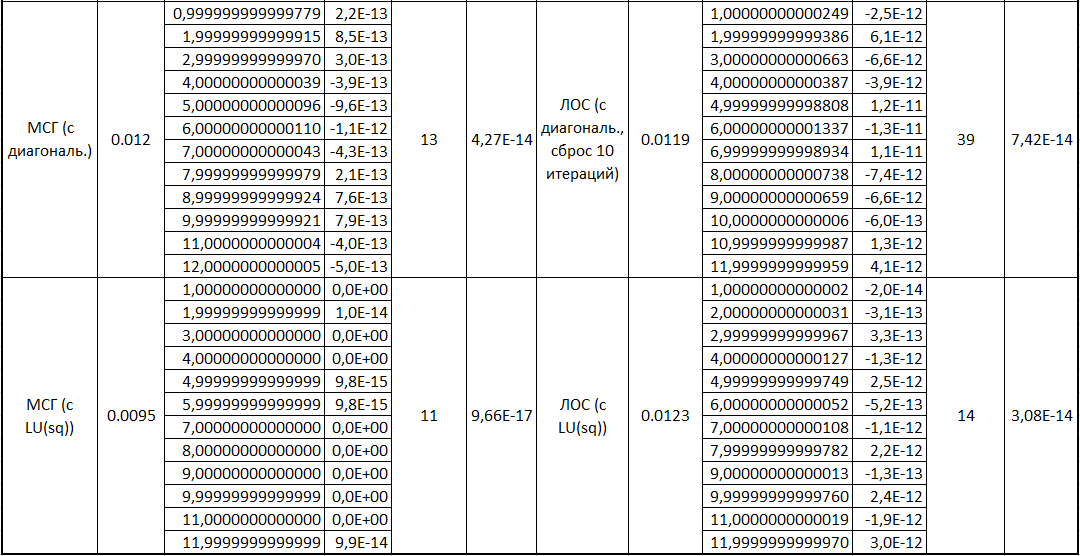
**Матрица А:**



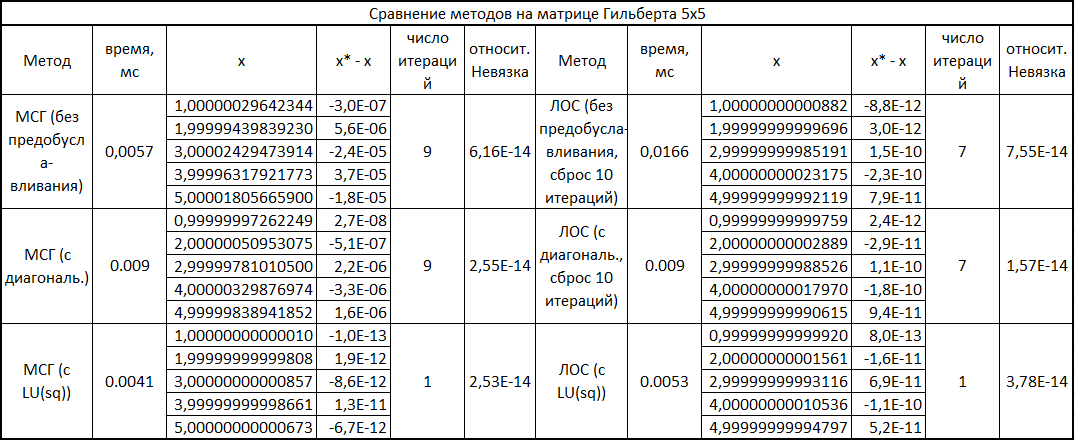


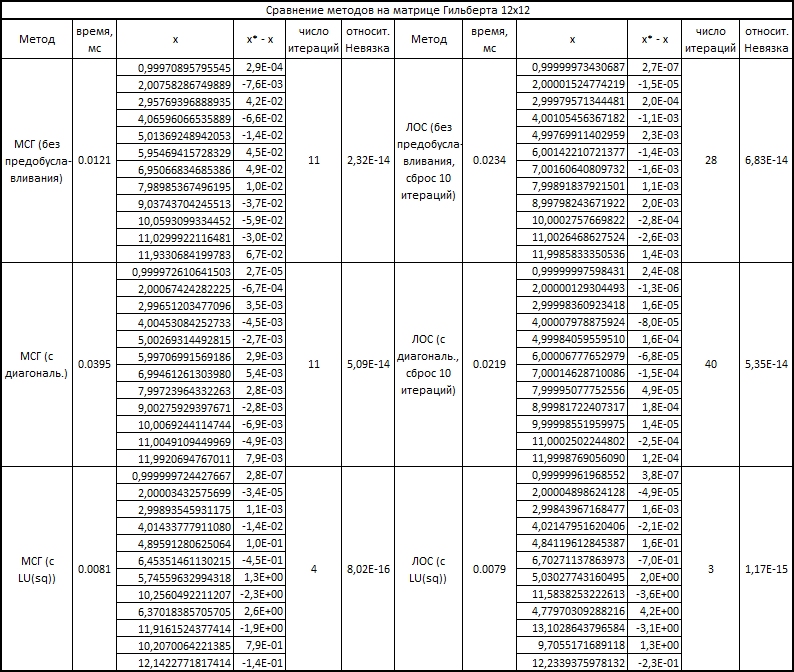
**Матрица В:**



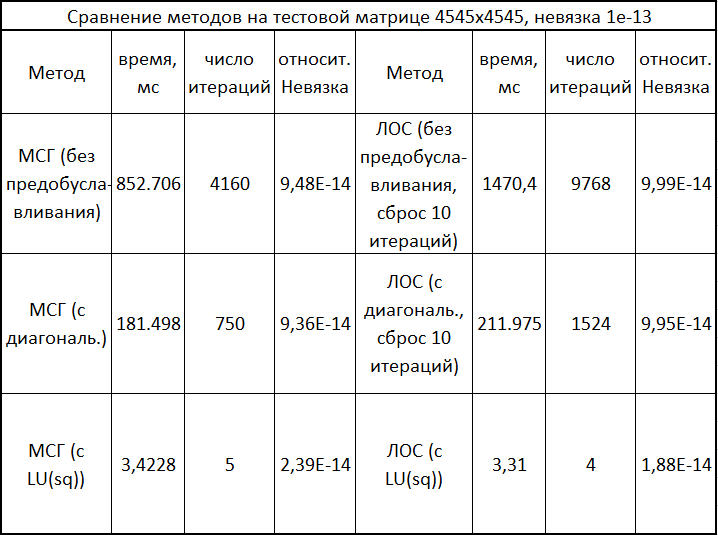


Сравнение на матрицах Гильберта:





Сравнение методов на тестовой матрице 4545х4545, невязка 10е-20:



# Выводы:

1. Выводы по матрице А:

На данной матрице лучше всего себя по скорости показал метод Зейделя. При этом, по количеству итераций, метод Зейделя оказался хуже всех рассмотренных трёхшаговых методов, из чего следует, что метод Зейделя сделал гораздо меньше действий, нежели остальные методы. Методы МСГ показали себя на данном сравнении гораздо лучше методов ЛОС, при этом для всех трёхшаговых методов заметна тенденция сокращения числа итераций при предобуславливании исходной матрицы. Изменение времени работы методов по данным тестам оценить не получится, поскольку для всех методов оно слишком короткое и скорее всего работа системы вносила задержки гораздо большие, нежели работали сами методы.

1. Выводы по матрице B:

На данной матрице, по сравнению с матрицей А, все методы сошлись за гораздо меньшее число итераций. В целом, как и в матрице А, лучше всего себя проявил метод Зейделя, а МСГ оказались лучше соответствующих им ЛОС.

1. Матрица Гильберта 12х12:

На данной матрице меньше всего итераций показал метод ЛОС(с ЛУ-предобуславливанием - 3 итерации).

Т.к. матрица Гильберта полная, метод с ЛУ-предобуславливанием на первой итерации решает СЛАУ полностью методом ЛУ, но поскольку матрица плохо обусловлена и метод ЛУ даёт большую погрешность, итерационный процесс продолжается на ещё несколько итераций, чтобы добиться требуемой точности. При этом погрешность результата всё ещё оказалась достаточно большая, за счёт накопления погрешности в матрицах ЛУ.

4.) Матрица Гильберта 4545х4545:

На данной размерности матрицы Гильберта лучше всего себя показали методы МСГ и ЛОС с ЛУ-предобуславливанием, оба эти метода работают во много раз быстрее, чем все остальные.

Метод МСГ без предобуславливания был гораздо быстрее аналогичного ЛОС, но в методе ЛОС фактическая точность решения вышла гораздо выше, чем фактическая точность в МСГ.

# Код программы

Файл Chrono\_timer.h:

#include <chrono>

class Timer

{

private:

using clock\_t = std::chrono::high\_resolution\_clock;

using second\_t = std::chrono::duration<double, std::ratio<1> >;

std::chrono::time\_point<clock\_t> m\_beg;

public:

Timer() : m\_beg(clock\_t::now())

{

}

void reset()

{

m\_beg = clock\_t::now();

}

double elapsedValue = 0.0;

double elapsed()

{

elapsedValue = std::chrono::duration\_cast<second\_t>(clock\_t::now() - m\_beg).count();

return elapsedValue;

}

};

Файл Matrix.h:

#pragma once

#include <vector>

#include <cmath>

#include <stdexcept>

#include <fstream>

#include <string>

std::vector<double> ReadVecFromFile(size\_t size, const std::string& path);

/// <summary>

/// Класс объектов матриц, хранящихся в разреженном строчно-столбцовом виде

/// <para> Точность хранения элементов - double </para>

/// </summary>

class Matrix {

// Переменные матрицы

public:

/// <summary>

/// Массив индексов строк/столбцов, вида 0, 0, 0 + k2, ..., 0+k2+...+kn, где ki - число элементов в i cтроке/столбце

/// <para> Помимо этого первый элемент i строки можно найти как ggl[ig[i]] </para>

/// <para> Пример массива для матрицы 3х3: </para>

///

/// <para> Матрица: </para>

/// <para> | 1 2 0 | </para>

/// <para> | 3 8 1 | </para>

/// <para> | 0 2 4 | </para>

/// <para> ig: { 0, 0, 1, 2 } </para>

/// </summary>

std::vector<uint32\_t> ig;

/// <summary>

/// Массив индексов столбцов/строк элементов (ставит индекс в соответствие элементу)

/// <para> Пример массива для матрицы 3х3: </para>

///

/// <para> Матрица: </para>

/// <para> | 1 2 0 | </para>

/// <para> | 3 8 1 | </para>

/// <para> | 0 2 4 | </para>

/// <para> jg: { 0, 1 } </para>

/// </summary>

std::vector<uint16\_t> jg;

/// <summary>

/// Массив элементов нижнего треугольника матрицы

/// <para> Пример массива для матрицы 3х3: </para>

///

/// <para> Матрица: </para>

/// <para> | 1 2 0 | </para>

/// <para> | 3 8 1 | </para>

/// <para> | 0 2 4 | </para>

/// <para> ggl: { 3, 2 } </para>

/// </summary>

std::vector<double> ggl;

/// <summary>

/// Массив элементов верхнего треугольника матрицы

/// <para> Пример массива для матрицы 3х3: </para>

///

/// <para> Матрица: </para>

/// <para> | 1 2 0 | </para>

/// <para> | 3 8 1 | </para>

/// <para> | 0 2 4 | </para>

/// <para> ggu: { 2, 1 } </para>

/// </summary>

std::vector<double> ggu;

/// <summary>

/// Массив элементов диагонали матрицы

/// <para> Пример массива для матрицы 3х3: </para>

///

/// <para> Матрица: </para>

/// <para> | 1 2 0 | </para>

/// <para> | 3 8 1 | </para>

/// <para> | 0 2 4 | </para>

/// <para> di: { 1, 8, 4 } </para>

/// </summary>

std::vector<double> di;

// Методы матрицы

public:

uint16\_t Size() const;

/// <summary>

/// Умножение матрицы на вектор

/// </summary>

std::vector<double> MultToVec(const std::vector<double>& right) const;

std::vector<double>& MultToVec(const std::vector<double>& right, std::vector<double>& result) const;

std::vector<double> operator\*(const std::vector<double>& right) const;

/// <summary>

/// Умножение транспонированной матрицы на вектор

/// </summary>

std::vector<double> TranspMultToVec(const std::vector<double>& right) const;

std::vector<double>& TranspMultToVec(const std::vector<double>& right, std::vector<double>& result) const;

Matrix& operator= (Matrix&& right) noexcept;

// Конструкторы матрицы

public:

Matrix();

Matrix(Matrix& right);

// Конструктор перемещения (нужен для метода ReadFromFiles)

Matrix(Matrix&& right) noexcept;

// Статические методы матрицы

public:

static Matrix ReadFromFiles(uint16\_t matrixSize, const std::string& igP, const std::string& jgP, const std::string& gglP, const std::string& gguP, const std::string& diP);

};

Файл Matrix.cpp:

#include "Matrix.h"

using namespace std;

vector<double> ReadVecFromFile(size\_t size, const string& path) {

vector<double> vec(size);

auto file = ifstream(path);

if (!file.is\_open())

{

throw runtime\_error("Файл " + path + " отсутствует в директории");

}

for (size\_t i = 0; i < size; i++)

{

file >> vec[i];

}

file.close();

return vec;

}

// Методы матрицы

uint16\_t Matrix::Size() const { return di.size(); }

/// <summary>

/// Умножение матрицы на вектор

/// </summary>

vector<double> Matrix::MultToVec(const vector<double>& right) const {

vector<double> result(right.size());

return MultToVec(right, result);

}

/// <summary>

/// Умножение матрицы на вектор

/// </summary>

vector<double>& Matrix::MultToVec(const vector<double>& right, vector<double>& result) const {

if (right.size() != di.size()) throw runtime\_error("Размеры матрицы и вектора не совпадают.");

if (right.size() != result.size()) throw runtime\_error("Размеры матрицы и результирующего вектора не совпадают.");

for (uint16\_t i = 0; i < result.size(); i++)

{

// Умножаем диагональ

result[i] = di[i] \* right[i];

// Умножаем нижний и верхний треугольники

for (uint32\_t j = ig[i]; j < ig[i + 1]; j++)

{

result[i] += ggl[j] \* right[jg[j]];

result[jg[j]] += ggu[j] \* right[i];

}

}

return result;

}

vector<double> Matrix::operator\*(const vector<double>& right) const {

return MultToVec(right);

}

/// <summary>

/// Умножение транспонированной матрицы на вектор

/// </summary>

vector<double>& Matrix::TranspMultToVec(const vector<double>& right, vector<double>& result) const {

if (right.size() != di.size()) throw runtime\_error("Размеры матрицы и вектора не совпадают.");

if (right.size() != result.size()) throw runtime\_error("Размеры матрицы и результирующего вектора не совпадают.");

for (uint16\_t i = 0; i < result.size(); i++)

{

// Умножаем диагональ

result[i] = di[i] \* right[i];

// Умножаем нижний и верхний треугольники

for (uint32\_t j = ig[i]; j < ig[i + 1]; j++)

{

result[i] += ggu[j] \* right[jg[j]];

result[jg[j]] += ggl[j] \* right[i];

}

}

return result;

}

/// <summary>

/// Умножение транспонированной матрицы на вектор

/// </summary>

vector<double> Matrix::TranspMultToVec(const vector<double>& right) const {

vector<double> result(right.size());

return TranspMultToVec(right, result);

}

Matrix& Matrix::operator= (Matrix&& right) noexcept {

ig = std::move(right.ig);

jg = std::move(right.jg);

ggl = std::move(right.ggl);

ggu = std::move(right.ggu);

di = std::move(right.di);

return \*this;

}

// Конструкторы матрицы

Matrix::Matrix() {}

Matrix::Matrix(Matrix& right) :

ig{ right.ig },

jg{ right.jg },

ggl{ right.ggl },

ggu{ right.ggu },

di{ right.di }

{}

// Конструктор перемещения (нужен для метода ReadFromFiles)

Matrix::Matrix(Matrix&& right) noexcept

{

ig = std::move(right.ig);

jg = std::move(right.jg);

ggl = std::move(right.ggl);

ggu = std::move(right.ggu);

di = std::move(right.di);

}

// Статические методы матрицы

Matrix Matrix::ReadFromFiles(uint16\_t matrixSize, const string& igP, const string& jgP, const string& gglP, const string& gguP, const string& diP) {

Matrix mat;

bool isStartFromOne = false;

{

mat.ig.resize(matrixSize + 1);

auto igS = ifstream(igP);

if (!igS.is\_open()) throw runtime\_error("Файл " + igP + " отсутствует в директории.");

for (uint16\_t i = 0; i <= matrixSize; i++)

{

igS >> mat.ig[i];

}

// Если массив ig в файле начинался с 1, то меняем его под наши параметры (под 0)

if (isStartFromOne = mat.ig[0])

{

for (uint16\_t i = 0; i <= matrixSize; i++)

{

mat.ig[i]--;

}

}

}

{

auto jgS = ifstream(jgP);

if (!jgS.is\_open()) throw runtime\_error("Файл " + jgP + " отсутствует в директории.");

mat.jg.resize(mat.ig.back());

for (uint32\_t i = 0; i < mat.jg.size(); i++)

{

jgS >> mat.jg[i];

if (isStartFromOne)

{

mat.jg[i]--;

}

}

}

try

{

mat.di = ReadVecFromFile(matrixSize, diP);

mat.ggl = ReadVecFromFile(mat.jg.size(), gglP);

mat.ggu = ReadVecFromFile(mat.jg.size(), gguP);

}

catch (exception& e)

{

throw e;

}

return mat;

}

Файл LU.h:

#pragma once

#include <vector>

#include "Matrix.h"

// Неполное разложение LU(sq) матрицы разреженного строчно-столбцового формата Matrix

// Не хранит портрет матрицы, но использует портрет исходной матрицы (а также ссылается на неё)

class LU {

// Блок внутренних переменных разложения LU

public:

const Matrix\* parent = nullptr;

// Вектор диагональных элементов LU разложения. В данном случае диагонали L и U совпадают

std::vector<double> di;

// Вектор элементов нижнего треугольника L

std::vector<double> ggl;

// Вектор элементов верхнего треугольника U

std::vector<double> ggu;

// Блок основных конструкторов класса

public:

/// <summary>

/// Конструктор с резервированием памяти под разложение

/// </summary>

/// <param name="diSize"> - размер диагонали,</param>

/// <param name="luSize"> - размер массивов нижнего и верхнего треугольника</param>

LU(size\_t diSize, size\_t luSize);

/// <summary>

/// Конструктор с построением неполного LU(sq)-разложения по матрице mat

/// </summary>

/// <param name="mat"> - матрица, по которой построится LU-разложение, с привязкой этой матрицы к объекту</param>

LU(const Matrix& mat);

// Блок основных нестатических методов класса

public:

/// <summary>

/// Разложить матрицу mat в неполное LU(sq) - разложение

/// </summary>

/// <param name="mat"> - матрица, которую требуется разложить. Она же будет использоваться для просмотра портрета матриц</param>

void MakeLuFor(const Matrix& mat);

/// <summary>

/// Метод изменения размера разложения

/// </summary>

/// <param name="diSize"> - размер диагонали,</param>

/// <param name="luSize"> - размер массивов нижнего и верхнего треугольника</param>

void Resize(size\_t diSize, size\_t luSize);

// Умножение матриц на вектор

/// <summary>

/// Умножение нижней матрицы L на вектор vec. Выделяет память под вектор ответа, не меняет матрицу LU

/// </summary>

/// <param name="vec"> - вектор, на который будет происходить умножение матрицы L</param>

/// <returns>Вектор с результатом перемножения (выделяется в памяти)</returns>

std::vector<double> LMultToVec(const std::vector<double>& vec) const;

/// <summary>

/// Умножение нижней матрицы L на вектор vec. Ответ записывается в вектор ans, не меняет матрицу LU

/// </summary>

/// <param name="vec"> - вектор, на который будет происходить умножение матрицы L;</param>

/// <param name="ans"> - вектор, куда запишется ответ без выделения памяти (должен отличаться от vec!)</param>

/// <returns>Ссылка на вектор ans</returns>

std::vector<double>& LMultToVec(const std::vector<double>& vec, std::vector<double>& ans) const;

/// <summary>

/// Умножение нижней матрицы L^T на вектор vec. Выделяет память под вектор ответа, не меняет матрицу LU

/// </summary>

/// <param name="vec"> - вектор, на который будет происходить умножение матрицы L</param>

/// <returns>Вектор с результатом перемножения (выделяется в памяти)</returns>

std::vector<double> LTranspMultToVec(const std::vector<double>& vec) const;

/// <summary>

/// Умножение нижней матрицы L^T на вектор vec. Ответ записывается в вектор ans, не меняет матрицу LU

/// </summary>

/// <param name="vec"> - вектор, на который будет происходить умножение матрицы L;</param>

/// <param name="ans"> - вектор, куда запишется ответ без выделения памяти (должен отличаться от vec!)</param>

/// <returns>Ссылка на вектор ans</returns>

std::vector<double>& LTranspMultToVec(const std::vector<double>& vec, std::vector<double>& ans) const;

/// <summary>

/// Умножение верхней матрицы U на вектор vec. Выделяет память под вектор ответа, не меняет матрицу LU

/// </summary>

/// <param name="vec"> - вектор, на который будет происходить умножение матрицы U</param>

/// <returns>Вектор с результатом перемножения (выделяется в памяти)</returns>

std::vector<double> UMultToVec(const std::vector<double>& vec) const;

/// <summary>

/// Умножение верхней матрицы U на вектор vec. Ответ записывается в вектор ans, не меняет матрицу LU

/// </summary>

/// <param name="vec"> - вектор, на который будет происходить умножение матрицы U;</param>

/// <param name="ans"> - вектор, куда запишется ответ без выделения памяти (должен отличаться от vec!)</param>

/// <returns>Ссылка на вектор ans</returns>

std::vector<double>& UMultToVec(const std::vector<double>& vec, std::vector<double>& ans) const;

/// <summary>

/// Умножение верхней матрицы U^T на вектор vec. Выделяет память под вектор ответа, не меняет матрицу LU

/// </summary>

/// <param name="vec"> - вектор, на который будет происходить умножение матрицы U</param>

/// <returns>Вектор с результатом перемножения (выделяется в памяти)</returns>

std::vector<double> UTranspMultToVec(const std::vector<double>& vec) const;

/// <summary>

/// Умножение верхней матрицы U^T на вектор vec. Ответ записывается в вектор ans, не меняет матрицу LU

/// </summary>

/// <param name="vec"> - вектор, на который будет происходить умножение матрицы U;</param>

/// <param name="ans"> - вектор, куда запишется ответ без выделения памяти (должен отличаться от vec!)</param>

/// <returns>Ссылка на вектор ans</returns>

std::vector<double>& UTranspMultToVec(const std::vector<double>& vec, std::vector<double>& ans) const;

// Решение слау с использованием матриц и вектора правой части

/// <summary>

/// Решение слау вида Lx = right. Не выделяет память под вектор x, не меняет матрицы LU

/// </summary>

/// <param name="right"> - вектор правой части уравнения;</param>

/// <param name="x"> - вектор, куда будет записан ответ. Должен быть с уже выделенной памятью. Должен отличаться от right!</param>

/// <returns>ссылка на вектор x</returns>

std::vector<double>& LSlauSolve(const std::vector<double>& right, std::vector<double>& x) const;

/// <summary>

/// Решение слау вида Lx = right. Выделяет память под вектор x, не меняет матрицы LU

/// </summary>

/// <param name="right"> - вектор правой части уравнения;</param>

/// <returns>Полученный вектор x</returns>

std::vector<double> LSlauSolve(const std::vector<double>& right) const;

/// <summary>

/// Решение слау вида L^T \* x = right. Не выделяет память под вектор x, не меняет матрицы LU

/// </summary>

/// <param name="right"> - вектор правой части уравнения;</param>

/// <param name="x"> - вектор, куда будет записан ответ. Должен быть с уже выделенной памятью. Должен отличаться от right!</param>

/// <returns>ссылка на вектор x</returns>

std::vector<double>& LTranspSlauSolve(const std::vector<double>& right, std::vector<double>& x) const;

/// <summary>

/// Решение слау вида L^T \* x = right. Выделяет память под вектор x, не меняет матрицы LU

/// </summary>

/// <param name="right"> - вектор правой части уравнения;</param>

/// <returns>Полученный вектор x</returns>

std::vector<double> LTranspSlauSolve(const std::vector<double>& right) const;

/// <summary>

/// Решение слау вида Ux = right. Не выделяет память под вектор x, не меняет матрицы LU

/// </summary>

/// <param name="right"> - вектор правой части уравнения;</param>

/// <param name="x"> - вектор, куда будет записан ответ. Должен быть с уже выделенной памятью. Должен отличаться от right!</param>

/// <returns>ссылка на вектор x</returns>

std::vector<double>& USlauSolve(const std::vector<double>& right, std::vector<double>& x) const;

/// <summary>

/// Решение слау вида Ux = right. Выделяет память под вектор x, не меняет матрицы LU

/// </summary>

/// <param name="right"> - вектор правой части уравнения;</param>

/// <returns>Полученный вектор x</returns>

std::vector<double> USlauSolve(const std::vector<double>& right) const;

/// <summary>

/// Решение слау вида U^T \* x = right. Не выделяет память под вектор x, не меняет матрицы LU

/// </summary>

/// <param name="right"> - вектор правой части уравнения;</param>

/// <param name="x"> - вектор, куда будет записан ответ. Должен быть с уже выделенной памятью. Должен отличаться от right!</param>

/// <returns>ссылка на вектор x</returns>

std::vector<double>& UTranspSlauSolve(const std::vector<double>& right, std::vector<double>& x) const;

/// <summary>

/// Решение слау вида U^T \* x = right. Выделяет память под вектор x, не меняет матрицы LU

/// </summary>

/// <param name="right"> - вектор правой части уравнения;</param>

/// <returns>Полученный вектор x</returns>

std::vector<double> UTranspSlauSolve(const std::vector<double>& right) const;

};

Файл LU.cpp:

#include "LU.h"

/// <summary>

/// Конструктор с резервированием памяти под разложение

/// </summary>

/// <param name="diSize"> - размер диагонали,</param>

/// <param name="luSize"> - размер массивов нижнего и верхнего треугольника</param>

LU::LU(size\_t diSize, size\_t luSize) {

Resize(diSize, luSize);

}

/// <summary>

/// Конструктор с построением неполного LU(sq)-разложения по матрице mat

/// </summary>

/// <param name="mat"> - матрица, по которой построится LU-разложение, с привязкой этой матрицы к объекту</param>

LU::LU(const Matrix& mat)

{

MakeLuFor(mat);

}

/// <summary>

/// Разложить матрицу mat в неполное LU(sq) - разложение

/// </summary>

/// <param name="mat"> - матрица, которую требуется разложить. Она же будет использоваться для просмотра портрета матриц</param>

void LU::MakeLuFor(const Matrix& mat) {

parent = &mat;

if (di.size() != mat.di.size())

di.resize(mat.di.size());

if (ggl.size() != mat.ggl.size())

ggl.resize(mat.ggl.size());

if (ggu.size() != mat.ggu.size())

ggu.resize(mat.ggu.size());

const auto& ig = mat.ig;

const auto& jg = mat.jg;

di[0] = sqrt(mat.di[0]);

for (size\_t i = 1; i < mat.Size(); i++)

{

di[i] = 0;

for (size\_t j = ig[i]; j < ig[i + 1]; j++)

{

size\_t k = ig[i];

size\_t v = ig[jg[j]];

ggl[j] = ggu[j] = 0;

while (k < j && v < ig[jg[j] + 1])

{

if (jg[k] > jg[v]) v++;

else if (jg[k] < jg[v]) k++;

else

{

ggl[j] += ggl[k] \* ggu[v];

ggu[j] += ggl[v] \* ggu[k];

k++;

v++;

}

}

ggl[j] = (mat.ggl[j] - ggl[j]) / di[jg[j]];

ggu[j] = (mat.ggu[j] - ggu[j]) / di[jg[j]];

di[i] += ggl[j] \* ggu[j];

}

di[i] = sqrt(mat.di[i] - di[i]);

}

}

void LU::Resize(size\_t diSize, size\_t luSize) {

di.resize(diSize);

ggl.resize(luSize);

ggu.resize(luSize);

}

/// <summary>

/// Умножение нижней матрицы L на вектор vec. Выделяет память под вектор ответа, не меняет матрицу LU

/// </summary>

/// <param name="vec"> - вектор, на который будет происходить умножение матрицы L</param>

/// <returns>Вектор с результатом перемножения (выделяется в памяти)</returns>

std::vector<double> LU::LMultToVec(const std::vector<double>& vec) const

{

std::vector<double> ans(vec.size());

return LMultToVec(vec, ans);

}

/// <summary>

/// Умножение нижней матрицы L на вектор vec. Ответ записывается в вектор ans, не меняет матрицу LU

/// </summary>

/// <param name="vec"> - вектор, на который будет происходить умножение матрицы L;</param>

/// <param name="ans"> - вектор, куда запишется ответ без выделения памяти (должен отличаться от vec!)</param>

/// <returns>Ссылка на вектор ans</returns>

std::vector<double>& LU::LMultToVec(const std::vector<double>& vec, std::vector<double>& ans) const

{

if (vec.size() != di.size()) throw std::runtime\_error("Размеры матрицы и вектора не совпадают.");

if (vec.size() != ans.size()) throw std::runtime\_error("Размеры матрицы и результирующего вектора не совпадают.");

for (uint16\_t i = 0; i < ans.size(); i++)

{

// Умножаем диагональ

ans[i] = di[i] \* vec[i];

// Умножаем нижний треугольник

for (uint32\_t j = parent->ig[i]; j < parent->ig[i + 1]; j++)

{

ans[i] += ggl[j] \* vec[parent->jg[j]];

}

}

return ans;

}

std::vector<double> LU::LTranspMultToVec(const std::vector<double>& vec) const

{

std::vector<double> ans(vec.size());

return LTranspMultToVec(vec, ans);

}

std::vector<double>& LU::LTranspMultToVec(const std::vector<double>& vec, std::vector<double>& ans) const

{

if (vec.size() != di.size()) throw std::runtime\_error("Размеры матрицы и вектора не совпадают.");

if (vec.size() != ans.size()) throw std::runtime\_error("Размеры матрицы и результирующего вектора не совпадают.");

for (uint16\_t i = 0; i < ans.size(); i++)

{

// Умножаем диагональ

ans[i] = di[i] \* vec[i];

// Умножаем на верхний треугольник с данными нижнего

for (uint32\_t j = parent->ig[i]; j < parent->ig[i + 1]; j++)

{

ans[parent->jg[j]] += ggl[j] \* vec[i];

}

}

return ans;

}

/// <summary>

/// Умножение верхней матрицы U на вектор vec. Выделяет память под вектор ответа, не меняет матрицу LU

/// </summary>

/// <param name="vec"> - вектор, на который будет происходить умножение матрицы U</param>

/// <returns>Вектор с результатом перемножения (выделяется в памяти)</returns>

std::vector<double> LU::UMultToVec(const std::vector<double>& vec) const

{

std::vector<double> ans(vec.size());

return UMultToVec(vec, ans);

}

/// <summary>

/// Умножение верхней матрицы U на вектор vec. Ответ записывается в вектор ans, не меняет матрицу LU

/// </summary>

/// <param name="vec"> - вектор, на который будет происходить умножение матрицы U;</param>

/// <param name="ans"> - вектор, куда запишется ответ без выделения памяти (должен отличаться от vec!)</param>

/// <returns>Ссылка на вектор ans</returns>

std::vector<double>& LU::UMultToVec(const std::vector<double>& vec, std::vector<double>& ans) const

{

if (vec.size() != di.size()) throw std::runtime\_error("Размеры матрицы и вектора не совпадают.");

if (vec.size() != ans.size()) throw std::runtime\_error("Размеры матрицы и результирующего вектора не совпадают.");

for (uint16\_t i = 0; i < ans.size(); i++)

{

// Умножаем диагональ

ans[i] = di[i] \* vec[i];

// Умножаем верхний треугольник

for (uint32\_t j = parent->ig[i]; j < parent->ig[i + 1]; j++)

{

ans[parent->jg[j]] += ggu[j] \* vec[i];

}

}

return ans;

}

std::vector<double> LU::UTranspMultToVec(const std::vector<double>& vec) const

{

std::vector<double> ans(vec.size());

return UTranspMultToVec(vec, ans);

}

std::vector<double>& LU::UTranspMultToVec(const std::vector<double>& vec, std::vector<double>& ans) const

{

if (vec.size() != di.size()) throw std::runtime\_error("Размеры матрицы и вектора не совпадают.");

if (vec.size() != ans.size()) throw std::runtime\_error("Размеры матрицы и результирующего вектора не совпадают.");

for (uint16\_t i = 0; i < ans.size(); i++)

{

// Умножаем диагональ

ans[i] = di[i] \* vec[i];

// Умножаем нижний треугольник с данными верхнего треугольника

for (uint32\_t j = parent->ig[i]; j < parent->ig[i + 1]; j++)

{

ans[i] += ggu[j] \* vec[parent->jg[j]];

}

}

return ans;

}

/// <summary>

/// Решение слау вида Lx = right. Не выделяет память под вектор x, не меняет матрицы LU

/// </summary>

/// <param name="right"> - вектор правой части уравнения;</param>

/// <param name="x"> - вектор, куда будет записан ответ. Должен быть с уже выделенной памятью. Должен отличаться от right!</param>

/// <returns>ссылка на вектор x</returns>

std::vector<double>& LU::LSlauSolve(const std::vector<double>& right, std::vector<double>& x) const

{

if (right.size() != di.size()) throw std::runtime\_error("Размеры матрицы и вектора не совпадают.");

if (right.size() != x.size()) throw std::runtime\_error("Размеры матрицы и результирующего вектора не совпадают.");

size\_t size = x.size();

for (size\_t i = 0; i < size; i++)

{

x[i] = 0;

for (size\_t j = parent->ig[i]; j < parent->ig[i + 1]; j++)

{

x[i] += x[parent->jg[j]] \* ggl[j];

}

x[i] = (right[i] - x[i]) / di[i];

}

return x;

}

/// <summary>

/// Решение слау вида Lx = right. Выделяет память под вектор x, не меняет матрицы LU

/// </summary>

/// <param name="right"> - вектор правой части уравнения;</param>

/// <returns>Полученный вектор x</returns>

std::vector<double> LU::LSlauSolve(const std::vector<double>& right) const

{

std::vector<double> x(right.size());

return LSlauSolve(right, x);

}

std::vector<double>& LU::LTranspSlauSolve(const std::vector<double>& right, std::vector<double>& x) const

{

if (right.size() != di.size()) throw std::runtime\_error("Размеры матрицы и вектора не совпадают.");

if (right.size() != x.size()) throw std::runtime\_error("Размеры матрицы и результирующего вектора не совпадают.");

size\_t size = x.size();

for (size\_t i = 0; i < size; i++)

x[i] = 0;

for (size\_t it = 0, i = size - 1; it < size; it++, i--)

{

x[i] = (right[i] - x[i]) / di[i];

for (size\_t j = parent->ig[i]; j < parent->ig[i + 1]; j++)

{

x[parent->jg[j]] += ggl[j] \* x[i];

}

}

return x;

}

std::vector<double> LU::LTranspSlauSolve(const std::vector<double>& right) const

{

std::vector<double> x(right.size());

return LTranspSlauSolve(right, x);

}

/// <summary>

/// Решение слау вида Ux = right. Не выделяет память под вектор x, не меняет матрицы LU

/// </summary>

/// <param name="right"> - вектор правой части уравнения;</param>

/// <param name="x"> - вектор, куда будет записан ответ. Должен быть с уже выделенной памятью. Должен отличаться от right!</param>

/// <returns>ссылка на вектор x</returns>

std::vector<double>& LU::USlauSolve(const std::vector<double>& right, std::vector<double>& x) const

{

if (right.size() != di.size()) throw std::runtime\_error("Размеры матрицы и вектора не совпадают.");

if (right.size() != x.size()) throw std::runtime\_error("Размеры матрицы и результирующего вектора не совпадают.");

size\_t size = x.size();

for (size\_t i = 0; i < size; i++)

x[i] = 0;

for (size\_t it = 0, i = size - 1; it < size; it++, i--)

{

x[i] = (right[i] - x[i]) / di[i];

for (size\_t j = parent->ig[i]; j < parent->ig[i + 1]; j++)

{

x[parent->jg[j]] += ggu[j] \* x[i];

}

}

return x;

}

/// <summary>

/// Решение слау вида Ux = right. Выделяет память под вектор x, не меняет матрицы LU

/// </summary>

/// <param name="right"> - вектор правой части уравнения;</param>

/// <returns>Полученный вектор x</returns>

std::vector<double> LU::USlauSolve(const std::vector<double>& right) const

{

std::vector<double> x(right.size());

return USlauSolve(right, x);

}

std::vector<double>& LU::UTranspSlauSolve(const std::vector<double>& right, std::vector<double>& x) const

{

if (right.size() != di.size()) throw std::runtime\_error("Размеры матрицы и вектора не совпадают.");

if (right.size() != x.size()) throw std::runtime\_error("Размеры матрицы и результирующего вектора не совпадают.");

size\_t size = x.size();

for (size\_t i = 0; i < size; i++)

{

x[i] = 0;

for (size\_t j = parent->ig[i]; j < parent->ig[i + 1]; j++)

{

x[i] += x[parent->jg[j]] \* ggu[j];

}

x[i] = (right[i] - x[i]) / di[i];

}

return x;

}

std::vector<double> LU::UTranspSlauSolve(const std::vector<double>& right) const

{

std::vector<double> x(right.size());

return UTranspSlauSolve(right, x);

}

Файл IterSolvers.h:

#pragma once

#include <vector>

#include <stdexcept>

#include <format>

#include <iostream>

#include "Matrix.h"

#include "LU.h"

namespace Vec {

inline double Scalar(const std::vector<double>& l, const std::vector<double>& r);

// l or r may be similar vectors to ans

inline void Mult(const std::vector<double>& l, const std::vector<double>& r, std::vector<double>& ans);

inline std::vector<double> Mult(const std::vector<double>& l, const std::vector<double>& r);

}

namespace IterSolvers {

extern double minEps;

extern size\_t maxIter;

extern bool globalDebugOutput;

extern std::vector<double>\* \_tmp1, \* \_tmp2,

\* \_tmp3, \* \_tmp4, \* \_tmp5, \* \_tmp6;

extern LU\* \_lu\_mat;

inline void VecInit(std::vector<double>\*& vec, size\_t size);

namespace MSG\_Assimetric {

inline void Init\_Default(size\_t size);

size\_t Default(const Matrix& A, const std::vector<double>& f, std::vector<double>& x, double& eps, bool debugOutput = globalDebugOutput);

inline void Init\_DiagPrecond(size\_t size);

size\_t DiagPrecond(const Matrix& A, const std::vector<double>& f, std::vector<double>& x, double& eps, bool debugOutput = globalDebugOutput);

inline void Init\_LuPrecond(size\_t diSize, size\_t luSize);

size\_t LuPrecond(const Matrix& A, const std::vector<double>& f, std::vector<double>& x, double& eps, bool debugOutput = globalDebugOutput);

}

namespace LOS {

extern size\_t resetIter;

inline void Init\_Default(size\_t size);

size\_t Default(const Matrix& A, const std::vector<double>& f, std::vector<double>& x, double& eps, bool debugOutput = globalDebugOutput);

inline void Init\_DiagPrecond(size\_t size);

size\_t DiagPrecond(const Matrix& A, const std::vector<double>& f, std::vector<double>& x, double& eps, bool debugOutput = globalDebugOutput);

inline void Init\_LuPrecond(size\_t diSize, size\_t luSize);

size\_t LuPrecond(const Matrix& A, const std::vector<double>& f, std::vector<double>& x, double& eps, bool debugOutput = globalDebugOutput);

}

void Destruct();

};

Файл IterSolvers.cpp:

#include "IterSolvers.h"

using namespace std;

namespace Vec {

inline double Scalar(const vector<double>& l, const vector<double>& r) {

if (l.size() != r.size()) throw runtime\_error("Размеры векторов не совпадают");

double res = 0.0;

for (size\_t i = 0; i < l.size(); i++)

{

res += l[i] \* r[i];

}

return res;

}

// l or r may be similar vectors to ans

inline void Mult(const vector<double>& l, const vector<double>& r, vector<double>& ans) {

if (ans.size() != l.size() || ans.size() != r.size()) throw runtime\_error("Ошибка: размеры векторов должны совпадать.");

for (size\_t i = 0; i < ans.size(); i++)

{

ans[i] = l[i] \* r[i];

}

}

inline vector<double> Mult(const vector<double>& l, const vector<double>& r) {

if (r.size() != l.size()) throw runtime\_error("Ошибка: размеры векторов должны совпадать.");

vector<double> ans(l.size());

for (size\_t i = 0; i < ans.size(); i++)

{

ans[i] = l[i] \* r[i];

}

return ans;

}

}

namespace IterSolvers {

double minEps = 1e-8;

size\_t maxIter = 2000;

bool globalDebugOutput = false;

std::vector<double>\* \_tmp1 = nullptr, \* \_tmp2 = nullptr,

\* \_tmp3 = nullptr, \* \_tmp4 = nullptr, \* \_tmp5 = nullptr, \* \_tmp6 = nullptr;

LU\* \_lu\_mat = nullptr;

inline void VecInit(vector<double>\*& vec, size\_t size) {

if (vec == nullptr)

{

vec = new vector<double>(size);

}

else if (vec->size() != size)

{

vec->resize(size);

}

}

namespace MSG\_Assimetric {

inline void Init\_Default(size\_t size) {

VecInit(\_tmp1, size); // Массив для вектора r метода

VecInit(\_tmp2, size); // Массив для вектора z

VecInit(\_tmp3, size); // Массив для вектора t

VecInit(\_tmp4, size); // Массив для временного вектора

}

size\_t Default(const Matrix& A, const vector<double>& f, vector<double>& x, double& eps, bool debugOutput) {

size\_t size = x.size();

Init\_Default(size);

vector<double>& tmp = \*\_tmp4;

vector<double>& r = \*\_tmp1; // r0 = A^t \* (f - A \* x)

A.MultToVec(x, tmp);

for (uint16\_t i = 0; i < size; i++) tmp[i] = f[i] - tmp[i];

A.TranspMultToVec(tmp, r);

vector<double>& z = \*\_tmp2;

z = r; // z0

vector<double>& t = \*\_tmp3;

double rPrevScalar = Vec::Scalar(r, r);

double rScalar = 0;

double a = 0; // alpha\_k

double b = 0; // beta\_k

double normF = Vec::Scalar(f, f); // (f, f)

eps = DBL\_MAX;

size\_t iter = 0;

for (iter = 1; iter <= maxIter && eps > minEps; iter++)

{

A.MultToVec(z, tmp);

A.TranspMultToVec(tmp, t); // t = A^t \* A \* z\_k-1

a = rPrevScalar / Vec::Scalar(t, z); // a\_k = (r\_k-1, r\_k-1) / (t\_k-1, z\_k-1)

for (uint16\_t i = 0; i < size; i++)

{

x[i] += a \* z[i]; // x\_k = x\_k-1 + a \* z\_k-1

r[i] -= a \* t[i]; // r\_k = r\_k-1 - a \* t\_k-1

}

rScalar = Vec::Scalar(r, r);

b = rScalar / rPrevScalar; // b = (r\_k, r\_k) / (r\_k-1, r\_k-1)

for (uint16\_t i = 0; i < size; i++)

{

z[i] = r[i] + b \* z[i]; // z\_k = r\_k + b \* z\_k-1

}

rPrevScalar = rScalar;

eps = sqrt(rScalar / normF);

// Выводим на то же место, что и раньше (со сдвигом каретки)

if (debugOutput)

{

cout << format("\rИтерация: {0:<10} относительная невязка: {1:<15.3e}", iter, eps);

}

if (isinf(eps))

{

break;

}

}

if (debugOutput)

{

cout << endl;

if (isinf(eps))

{

cout << "Выход по переполнению метода" << endl << endl;

}

else if (iter > maxIter)

{

cout << "Выход по числу итераций" << endl << endl;

}

else

{

cout << "Выход по относительной невязке" << endl << endl;

}

}

return iter - 1;

}

inline void Init\_DiagPrecond(size\_t size) {

Init\_Default(size);

VecInit(\_tmp5, size); // Массив для вектора D

}

size\_t DiagPrecond(const Matrix& A, const vector<double>& f, vector<double>& x, double& eps, bool debugOutput) {

size\_t size = x.size();

Init\_DiagPrecond(size);

vector<double>& D = \*\_tmp5; // D = обратный корень от диагонали матрицы

for (uint16\_t i = 0; i < size; i++) D[i] = 1 / sqrt(A.di[i]);

for (uint16\_t i = 0; i < size; i++) x[i] /= D[i]; // local\_x

vector<double>& r = \*\_tmp1; // r = U^-t \* A^t \* L^-t \* L^-1 (f - A \* x)

vector<double>& tmp = \*\_tmp4;

A.MultToVec(x, tmp);

for (uint16\_t i = 0; i < size; i++) tmp[i] = f[i] - tmp[i];

Vec::Mult(D, tmp, tmp);

Vec::Mult(D, tmp, tmp);

A.TranspMultToVec(tmp, r);

Vec::Mult(D, r, r);

vector<double>& z = \*\_tmp2;

z = r;

vector<double>& t = \*\_tmp3; // t = U^-1 \* A^t \* L^-t \* L^-1 \* A \* U^-1 \* z

double rPrevScalar = Vec::Scalar(r, r); // (r\_k-1, r\_k-1)

double rScalar = 0;

double a = 0; // alpha\_k,

double b = 0; // beta\_k

double normF = Vec::Scalar(f, f); // ||f||

eps = sqrt(rPrevScalar / normF);

size\_t iter;

for (iter = 1; iter <= maxIter && eps > minEps; iter++)

{

Vec::Mult(D, z, t);

A.MultToVec(t, tmp);

Vec::Mult(D, tmp, tmp);

Vec::Mult(D, tmp, tmp);

A.TranspMultToVec(tmp, t);

Vec::Mult(D, t, t);

a = rPrevScalar / Vec::Scalar(t, z); // a\_k = (r\_k-1, r\_k-1) / (t\_k-1, z\_k-1)

for (uint16\_t i = 0; i < size; i++)

{

x[i] += a \* z[i]; // local\_x\_k = local\_x\_k-1 + a \* z\_k-1

r[i] -= a \* t[i]; // r\_k = r\_k-1 - a \* t\_k-1

}

rScalar = Vec::Scalar(r, r);

b = rScalar / rPrevScalar; // b = (r\_k, r\_k) / (r\_k-1, r\_k-1)

for (uint16\_t i = 0; i < size; i++)

{

z[i] = r[i] + b \* z[i]; // z\_k = r\_k + b \* z\_k-1

}

rPrevScalar = rScalar;

eps = sqrt(rPrevScalar / normF);

// Выводим на то же место, что и раньше (со сдвигом каретки)

if (debugOutput)

{

cout << format("\rИтерация: {0:<10} относительная невязка: {1:<15.3e}", iter, eps);

}

if (isinf(eps))

{

break;

}

}

Vec::Mult(D, x, x); // x = U^-1 \* local\_x

if (debugOutput)

{

cout << endl;

if (isinf(eps))

{

cout << "Выход по переполнению метода" << endl << endl;

}

else if (iter > maxIter)

{

cout << "Выход по числу итераций" << endl << endl;

}

else

{

cout << "Выход по относительной невязке" << endl << endl;

}

}

return iter - 1;

}

inline void Init\_LuPrecond(size\_t diSize, size\_t luSize) {

VecInit(\_tmp1, diSize); // Массив для вектора r метода

VecInit(\_tmp2, diSize); // Массив для вектора z

VecInit(\_tmp3, diSize); // Массив для вектора t

VecInit(\_tmp4, diSize); // Массив для временного вектора

VecInit(\_tmp5, diSize); // Массив для вектора local\_x

if (\_lu\_mat == nullptr)

{

\_lu\_mat = new LU(diSize, luSize);

}

else if (\_lu\_mat->ggl.size() != luSize || \_lu\_mat->di.size() != diSize)

{

\_lu\_mat->Resize(diSize, luSize);

}

}

size\_t LuPrecond(const Matrix& A, const vector<double>& f, vector<double>& x, double& eps, bool debugOutput) {

size\_t size = x.size();

Init\_LuPrecond(size, A.ggl.size());

LU& lu = \*\_lu\_mat;

lu.MakeLuFor(A); // неполное LU(sq) разложение для матрицы A

vector<double>& local\_x = \*\_tmp5; // local\_x

lu.UMultToVec(x, local\_x);

vector<double>& r = \*\_tmp1; // r = U^-t \* A^t \* L^-t \* L^-1 (f - A \* x)

vector<double>& tmp = \*\_tmp4;

A.MultToVec(x, r);

for (uint16\_t i = 0; i < size; i++) r[i] = f[i] - r[i];

lu.LSlauSolve(r, tmp);

lu.LTranspSlauSolve(tmp, r);

A.TranspMultToVec(r, tmp);

lu.UTranspSlauSolve(tmp, r);

vector<double>& z = \*\_tmp2;

z = r;

vector<double>& t = \*\_tmp3; // t = U^-1 \* A^t \* L^-t \* L^-1 \* A \* U^-1 \* z

double rPrevScalar = Vec::Scalar(r, r); // (r\_k-1, r\_k-1)

double rScalar = 0;

double a = 0; // alpha\_k,

double b = 0; // beta\_k

double normF = Vec::Scalar(f, f); // ||f||

eps = sqrt(rPrevScalar / normF);

size\_t iter;

for (iter = 1; iter <= maxIter && eps > minEps; iter++)

{

lu.USlauSolve(z, tmp);

A.MultToVec(tmp, t);

lu.LSlauSolve(t, tmp);

lu.LTranspSlauSolve(tmp, t);

A.TranspMultToVec(t, tmp);

lu.UTranspSlauSolve(tmp, t);

a = rPrevScalar / Vec::Scalar(t, z); // a\_k = (r\_k-1, r\_k-1) / (t\_k-1, z\_k-1)

for (uint16\_t i = 0; i < size; i++)

{

local\_x[i] += a \* z[i]; // local\_x\_k = local\_x\_k-1 + a \* z\_k-1

r[i] -= a \* t[i]; // r\_k = r\_k-1 - a \* t\_k-1

}

rScalar = Vec::Scalar(r, r);

b = rScalar / rPrevScalar; // b = (r\_k, r\_k) / (r\_k-1, r\_k-1)

for (uint16\_t i = 0; i < size; i++)

{

z[i] = r[i] + b \* z[i]; // z\_k = r\_k + b \* z\_k-1

}

rPrevScalar = rScalar;

eps = sqrt(rPrevScalar / normF);

// Выводим на то же место, что и раньше (со сдвигом каретки)

if (debugOutput)

{

cout << format("\rИтерация: {0:<10} относительная невязка: {1:<15.3e}", iter, eps);

}

if (isinf(eps))

{

break;

}

}

lu.USlauSolve(local\_x, x); // x = U^-1 \* local\_x

if (debugOutput)

{

cout << endl;

if (isinf(eps))

{

cout << "Выход по переполнению метода" << endl << endl;

}

else if (iter > maxIter)

{

cout << "Выход по числу итераций" << endl << endl;

}

else

{

cout << "Выход по относительной невязке" << endl << endl;

}

}

return iter - 1;

}

}

namespace LOS {

size\_t resetIter = 10;

inline void Init\_Default(size\_t size) {

VecInit(\_tmp1, size); // Массив для вектора r метода

VecInit(\_tmp2, size); // Массив для вектора z

VecInit(\_tmp3, size); // Массив для вектора p

VecInit(\_tmp4, size); // Массив для вектора Ar

}

size\_t Default(const Matrix& A, const vector<double>& f, vector<double>& x, double& eps, bool debugOutput) {

uint16\_t size = x.size();

Init\_Default(size);

vector<double>& r = \*\_tmp1;

A.MultToVec(x, r);

for (uint16\_t i = 0; i < size; i++) r[i] = f[i] - r[i]; // r0 = f - A \* x

vector<double>& z = \*\_tmp2; // z0

z = r;

vector<double>& p = \*\_tmp3; // p0 = A \* z0

A.MultToVec(z, p);

vector<double>& Ar = \*\_tmp4; // A \* r

double ppScalar;

double nev = Vec::Scalar(r, r);

double ffScalar = Vec::Scalar(f, f);

eps = nev / ffScalar;

double a; // alpha

double b; // beta

size\_t iter;

for (iter = 1; iter <= maxIter && eps > minEps; iter++)

{

ppScalar = Vec::Scalar(p, p); // (p\_k-1, p\_k-1)

a = Vec::Scalar(p, r) / ppScalar; // (p\_k-1, r\_k-1) / (p\_k-1, p\_k-1)

for (uint16\_t i = 0; i < size; i++)

{

x[i] += a \* z[i]; // [x\_k] = [x\_k-1] + a\*z\_k-1

r[i] -= a \* p[i]; // [r\_k] = [r\_k-1] - a\*p\_k-1

}

A.MultToVec(r, Ar); // A \* r\_k

b = -Vec::Scalar(p, Ar) / ppScalar; // b = - (p\_k-1, A \* r\_k) / (p\_k-1, p\_k-1)

for (uint16\_t i = 0; i < size; i++)

{

z[i] = r[i] + b \* z[i]; // [z\_k] = r\_k + b \* [z\_k-1]

p[i] = Ar[i] + b \* p[i]; // [p\_k] = A \* r\_k + b \* [p\_k-1]

}

if (iter % resetIter == 0)

{

A.MultToVec(x, r);

for (uint16\_t i = 0; i < size; i++) r[i] = f[i] - r[i];

z = r;

A.MultToVec(z, p);

}

nev = Vec::Scalar(r, r);

eps = sqrt(nev / ffScalar);

// Выводим на то же место, что и раньше (со сдвигом каретки)

if (debugOutput)

{

//cout << format("Итерация: {0:<10} относительная невязка: {1:<15.3e}\n", iter, eps);

cout << format("\rИтерация: {0:<10} относительная невязка: {1:<15.3e}", iter, eps);

}

if (isinf(eps))

{

break;

}

}

if (debugOutput)

{

cout << endl;

if (isinf(eps))

{

cout << "Выход по переполнению метода" << endl << endl;

}

else if (iter > maxIter)

{

cout << "Выход по числу итераций" << endl << endl;

}

else

{

cout << "Выход по относительной невязке" << endl << endl;

}

}

return iter - 1;

}

inline void Init\_DiagPrecond(size\_t size) {

VecInit(\_tmp1, size); // Массив для вектора r метода

VecInit(\_tmp2, size); // Массив для вектора z

VecInit(\_tmp3, size); // Массив для вектора p

VecInit(\_tmp4, size); // Массив для вектора Ar

VecInit(\_tmp5, size); // Массив для вектора D

VecInit(\_tmp6, size); // Массив для вектора tmp

}

size\_t DiagPrecond(const Matrix& A, const vector<double>& f, vector<double>& x, double& eps, bool debugOutput) {

uint16\_t size = x.size();

Init\_DiagPrecond(size);

vector<double>& D = \*\_tmp5; // обратный корень от диагонали матрицы

for (uint16\_t i = 0; i < size; i++) D[i] = 1 / sqrt(A.di[i]);

vector<double>& r = \*\_tmp1; // r0 = L^-1 \* (f - A \* x)

A.MultToVec(x, r);

for (uint16\_t i = 0; i < size; i++) r[i] = f[i] - r[i];

Vec::Mult(D, r, r);

vector<double>& z = \*\_tmp2; // z0 = U^-1 \* r

Vec::Mult(D, r, z);

vector<double>& p = \*\_tmp3; // p0 = L^-1 \* A \* z0

A.MultToVec(z, p);

Vec::Mult(D, p, p);

vector<double>& Ar = \*\_tmp4; // Ar = L^-1 \* A \* U^-1 \* r

vector<double>& tmp = \*\_tmp6;

double ppScalar;

double nev = Vec::Scalar(r, r);

double ffScalar = Vec::Scalar(f, f);

eps = nev / ffScalar;

double a; // alpha

double b; // beta

size\_t iter;

for (iter = 1; iter <= maxIter && eps > minEps; iter++)

{

ppScalar = Vec::Scalar(p, p); // (p\_k-1, p\_k-1)

a = Vec::Scalar(p, r) / ppScalar; // (p\_k-1, r\_k-1) / (p\_k-1, p\_k-1)

for (uint16\_t i = 0; i < size; i++)

{

x[i] += a \* z[i]; // [x\_k] = [x\_k-1] + a\*z\_k-1

r[i] -= a \* p[i]; // [r\_k] = [r\_k-1] - a\*p\_k-1

}

Vec::Mult(D, r, tmp);

A.MultToVec(tmp, Ar);

Vec::Mult(D, Ar, Ar); // Ar = L^-1 \* A \* U^-1 \* r

b = -Vec::Scalar(p, Ar) / ppScalar; // b = - (p\_k-1, L^-1 \* A \* U^-1 \* r\_k) / (p\_k-1, p\_k-1)

Vec::Mult(D, r, tmp); // tmp = U^-1 \* r\_k

for (uint16\_t i = 0; i < size; i++)

{

z[i] = tmp[i] + b \* z[i]; // [z\_k] = U^-1 \* r\_k + b \* [z\_k-1]

p[i] = Ar[i] + b \* p[i]; // [p\_k] = A \* r\_k + b \* [p\_k-1]

}

if (iter % resetIter == 0)

{

A.MultToVec(x, r);

for (uint16\_t i = 0; i < size; i++) r[i] = f[i] - r[i];

Vec::Mult(D, r, r);

Vec::Mult(D, r, z);

A.MultToVec(z, p);

Vec::Mult(D, p, p);

}

nev = Vec::Scalar(r, r);

eps = sqrt(nev / ffScalar);

// Выводим на то же место, что и раньше (со сдвигом каретки)

if (debugOutput)

{

//cout << format("Итерация: {0:<10} относительная невязка: {1:<15.3e}\n", iter, eps);

cout << format("\rИтерация: {0:<10} относительная невязка: {1:<15.3e}", iter, eps);

}

if (isinf(eps))

{

break;

}

}

if (debugOutput)

{

cout << endl;

if (isinf(eps))

{

cout << "Выход по переполнению метода" << endl << endl;

}

else if (iter > maxIter)

{

cout << "Выход по числу итераций" << endl << endl;

}

else

{

cout << "Выход по относительной невязке" << endl << endl;

}

}

return iter - 1;

}

inline void Init\_LuPrecond(size\_t diSize, size\_t luSize) {

VecInit(\_tmp1, diSize); // Массив для вектора r метода

VecInit(\_tmp2, diSize); // Массив для вектора z

VecInit(\_tmp3, diSize); // Массив для вектора p

VecInit(\_tmp4, diSize); // Массив для вектора Ar

VecInit(\_tmp5, diSize); // Массив для вектора tmp

if (\_lu\_mat == nullptr)

{

\_lu\_mat = new LU(diSize, luSize);

}

else if (\_lu\_mat->ggl.size() != luSize || \_lu\_mat->di.size() != diSize)

{

\_lu\_mat->Resize(diSize, luSize);

}

}

size\_t LuPrecond(const Matrix& A, const vector<double>& f, vector<double>& x, double& eps, bool debugOutput) {

uint16\_t size = x.size();

Init\_LuPrecond(size, A.ggl.size());

LU& lu = \*\_lu\_mat;

lu.MakeLuFor(A);

vector<double>& tmp = \*\_tmp5;

vector<double>& r = \*\_tmp1; // r0 = L^-1 \* (f - A \* x)

A.MultToVec(x, tmp);

for (uint16\_t i = 0; i < size; i++) tmp[i] = f[i] - tmp[i];

lu.LSlauSolve(tmp, r);

vector<double>& z = \*\_tmp2; // z0 = U^-1 \* r

lu.USlauSolve(r, z);

vector<double>& p = \*\_tmp3; // p0 = L^-1 \* A \* z0

A.MultToVec(z, tmp);

lu.LSlauSolve(tmp, p);

vector<double>& Ar = \*\_tmp4; // Ar = L^-1 \* A \* U^-1 \* r

double ppScalar;

double nev = Vec::Scalar(r, r);

double ffScalar = Vec::Scalar(f, f);

eps = nev / ffScalar;

double a; // alpha

double b; // beta

size\_t iter;

for (iter = 1; iter <= maxIter && eps > minEps; iter++)

{

ppScalar = Vec::Scalar(p, p); // (p\_k-1, p\_k-1)

a = Vec::Scalar(p, r) / ppScalar; // (p\_k-1, r\_k-1) / (p\_k-1, p\_k-1)

for (uint16\_t i = 0; i < size; i++)

{

x[i] += a \* z[i]; // [x\_k] = [x\_k-1] + a\*z\_k-1

r[i] -= a \* p[i]; // [r\_k] = [r\_k-1] - a\*p\_k-1

}

lu.USlauSolve(r, Ar);

A.MultToVec(Ar, tmp);

lu.LSlauSolve(tmp, Ar); // Ar = L^-1 \* A \* U^-1 \* r

//Vec::Mult(D, r, tmp);

//A.MultToVec(tmp, Ar);

//Vec::Mult(D, Ar, Ar);

b = -Vec::Scalar(p, Ar) / ppScalar; // b = - (p\_k-1, L^-1 \* A \* U^-1 \* r\_k) / (p\_k-1, p\_k-1)

lu.USlauSolve(r, tmp); // tmp = U^-1 \* r\_k

for (uint16\_t i = 0; i < size; i++)

{

z[i] = tmp[i] + b \* z[i]; // [z\_k] = U^-1 \* r\_k + b \* [z\_k-1]

p[i] = Ar[i] + b \* p[i]; // [p\_k] = A \* r\_k + b \* [p\_k-1]

}

if (iter % resetIter == 0)

{

A.MultToVec(x, tmp);

for (uint16\_t i = 0; i < size; i++) tmp[i] = f[i] - tmp[i];

lu.LSlauSolve(tmp, r);

lu.USlauSolve(r, z);

A.MultToVec(z, tmp);

lu.LSlauSolve(tmp, p);

}

nev = Vec::Scalar(r, r);

eps = sqrt(nev / ffScalar);

// Выводим на то же место, что и раньше (со сдвигом каретки)

if (debugOutput)

{

//cout << format("Итерация: {0:<10} относительная невязка: {1:<15.3e}\n", iter, eps);

cout << format("\rИтерация: {0:<10} относительная невязка: {1:<15.3e}", iter, eps);

}

if (isinf(eps))

{

break;

}

}

if (debugOutput)

{

cout << endl;

if (isinf(eps))

{

cout << "Выход по переполнению метода" << endl << endl;

}

else if (iter > maxIter)

{

cout << "Выход по числу итераций" << endl << endl;

}

else

{

cout << "Выход по относительной невязке" << endl << endl;

}

}

return iter - 1;

}

}

void Destruct() {

delete \_tmp1, \_tmp2, \_tmp3, \_tmp4, \_tmp5, \_tmp6;

\_tmp1 = \_tmp2 = \_tmp3 = \_tmp4 = \_tmp5 = \_tmp6 = nullptr;

delete \_lu\_mat;

\_lu\_mat = nullptr;

}

};

Файл main.cpp:

#include <iostream>

#include <fstream>

#include "Chrono\_Timer.h"

#include "IterSolvers.h"

using namespace std;

int main() {

setlocale(LC\_ALL, "ru-RU");

uint16\_t matrixSize;

Matrix mat;

vector<double> f;

vector<double> x;

cout << "\*\*\* Программа для вычисления СЛАУ трёхшаговыми методами \*\*\*" << endl;

cout << "Начинается считывание данных из файла..." << endl;

auto kuslauF = ifstream("./iofiles/kuslau.txt");

if (!kuslauF.is\_open())

{

cout << "Файл ./iofiles/kuslau.txt отсутствует в директории" << endl;

return 1;

}

kuslauF >> matrixSize >> IterSolvers::maxIter >> IterSolvers::minEps;

kuslauF.close();

try

{

mat = Matrix::ReadFromFiles(matrixSize, "./iofiles/ig.txt", "./iofiles/jg.txt", "./iofiles/ggl.txt", "./iofiles/ggu.txt", "./iofiles/di.txt");

f = ReadVecFromFile(matrixSize, "./iofiles/pr.txt");

x = ReadVecFromFile(matrixSize, "./iofiles/initX.txt");

}

catch (exception& e)

{

cout << e.what() << endl;

return 1;

}

IterSolvers::globalDebugOutput = false;

cout << "Все данные успешно считанны из файлов." << endl << endl;

cout << "Выберите метод для решения СЛАУ: " << endl;

cout << " 1) МСГ для несимметричных матриц (без предобуславливания)" << endl;

cout << " 2) МСГ для несиметричных матриц (диагональное предобуславливание)" << endl;

cout << " 3) МСГ для несиметричных матриц (неполное LU(sq)-предобуславливание)" << endl;

cout << " 4) ЛОС (без предобуславливания)" << endl;

cout << " 5) ЛОС (диагональное предобуславливание)" << endl;

cout << " 6) ЛОС (неполное LU(sq)-предобуславливание)" << endl;

int userCase;

cin >> userCase;

switch (userCase)

{

case 1:

{

cout << "Начало вычислений для метода МСГ для несимметричных матриц (без предобуславливания)" << endl << endl;

double eps = 0;

IterSolvers::MSG\_Assimetric::Init\_Default(mat.Size());

Timer timer;

size\_t it = IterSolvers::MSG\_Assimetric::Default(mat, f, x, eps);

timer.elapsed();

IterSolvers::Destruct();

cout << "Метод закончил работу за " << timer.elapsedValue \* 1000 << " мс" << endl << endl;

cout << "Количество итераций: " << it << endl;

cout << "Относительная невязка: " << eps << endl;

break;

}

case 2:

{

cout << "Начало вычислений для метода МСГ для несимметричных матриц (диагональное предобуславливание)" << endl << endl;

double eps = 0;

IterSolvers::MSG\_Assimetric::Init\_DiagPrecond(mat.Size());

Timer timer;

size\_t it = IterSolvers::MSG\_Assimetric::DiagPrecond(mat, f, x, eps);

timer.elapsed();

IterSolvers::Destruct();

cout << "Метод закончил работу за " << timer.elapsedValue \* 1000 << " мс" << endl << endl;

cout << "Количество итераций: " << it << endl;

cout << "Относительная невязка: " << eps << endl;

break;

}

case 3:

{

cout << "Начало вычислений для метода МСГ для несимметричных матриц (неполное LU(sq)-предобуславливание)" << endl << endl;

double eps = 0;

IterSolvers::MSG\_Assimetric::Init\_LuPrecond(mat.Size(), mat.ggl.size());

Timer timer;

size\_t it = IterSolvers::MSG\_Assimetric::LuPrecond(mat, f, x, eps);

timer.elapsed();

IterSolvers::Destruct();

cout << "Метод закончил работу за " << timer.elapsedValue \* 1000 << " мс" << endl << endl;

cout << "Количество итераций: " << it << endl;

cout << "Относительная невязка: " << eps << endl;

break;

}

case 4:

{

cout << "Начало вычислений для метода ЛОС (без предобуславливания)" << endl << endl;

double eps = 0;

IterSolvers::LOS::Init\_Default(mat.Size());

Timer timer;

size\_t it = IterSolvers::LOS::Default(mat, f, x, eps);

timer.elapsed();

IterSolvers::Destruct();

cout << "Метод закончил работу за " << timer.elapsedValue \* 1000 << " мс" << endl << endl;

cout << "Количество итераций: " << it << endl;

cout << "Относительная невязка: " << eps << endl;

break;

}

case 5:

{

cout << "Начало вычислений для метода ЛОС (диагональное предобуславливание)" << endl << endl;

double eps = 0;

IterSolvers::LOS::Init\_DiagPrecond(mat.Size());

Timer timer;

size\_t it = IterSolvers::LOS::DiagPrecond(mat, f, x, eps);

timer.elapsed();

IterSolvers::Destruct();

cout << "Метод закончил работу за " << timer.elapsedValue \* 1000 << " мс" << endl << endl;

cout << "Количество итераций: " << it << endl;

cout << "Относительная невязка: " << eps << endl;

break;

}

case 6:

{

cout << "Начало вычислений для метода ЛОС (неполное LU(sq)-предобуславливание)" << endl << endl;

double eps = 0;

IterSolvers::LOS::Init\_LuPrecond(mat.Size(), mat.ggl.size());

Timer timer;

size\_t it = IterSolvers::LOS::LuPrecond(mat, f, x, eps);

timer.elapsed();

IterSolvers::Destruct();

cout << "Метод закончил работу за " << timer.elapsedValue \* 1000 << " мс" << endl << endl;

cout << "Количество итераций: " << it << endl;

cout << "Относительная невязка: " << eps << endl;

break;

}

default:

break;

}

if (IterSolvers::globalDebugOutput)

{

cout << "Полученное решение: " << endl;

cout.precision(15);

cout.setf(std::ios\_base::fixed);

}

auto outFile = ofstream("./iofiles/resultX.txt");

outFile.precision(15);

outFile.setf(std::ios\_base::fixed);

for (auto& el : x)

{

outFile << el << endl;

if (IterSolvers::globalDebugOutput)

{

cout << el << endl;

}

}

outFile.close();

return 0;

}