|  |  |  |
| --- | --- | --- |
|  | | |
| 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 8 |  |
|  |  |
| 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:

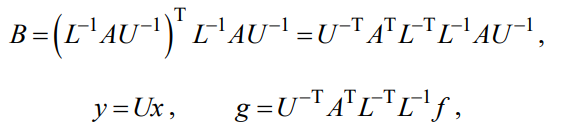
Далее для k = 1, 2, … проводятся следующие вычисления:

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

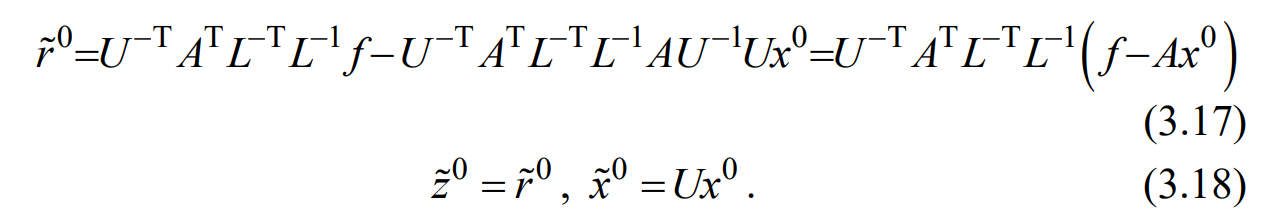
Для ускорения сходимости будем использовать предобуславливание матрицы А неполной факторизацией:

И в таком случае мы будем решать следующую систему:

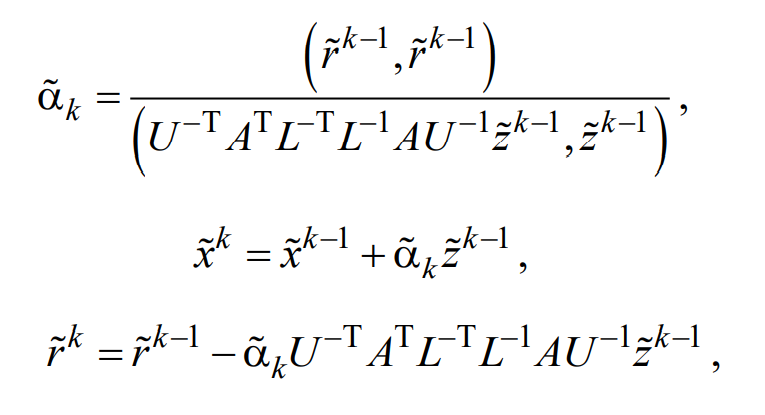
В которой:

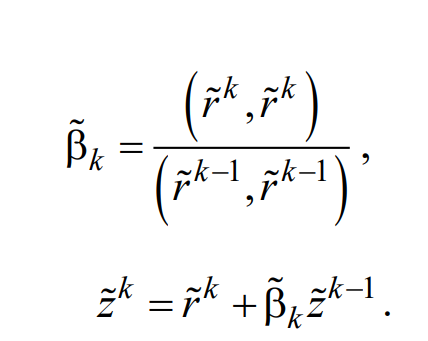


Аналогично выбирается начальное приближение x0 и полагается:

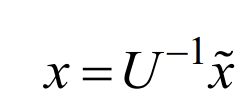


Далее для k = 1, 2, … проводятся следующие вычисления:





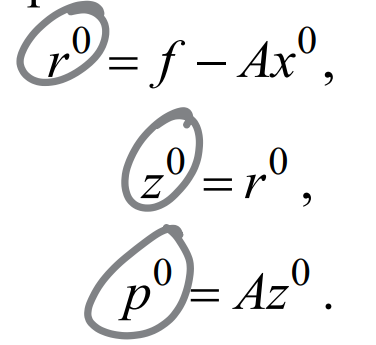
По окончании итерационного процесса вектор решения вычисляется следующим образом:



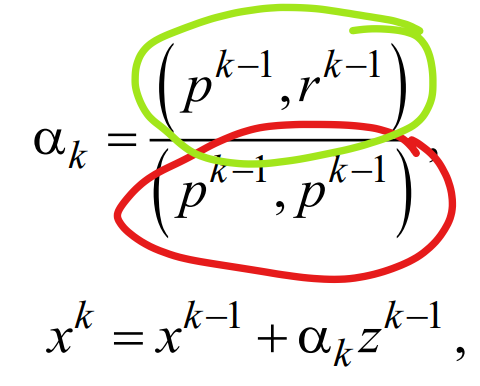
Для неполной факторизации можно принять (диагональ матрицы А).

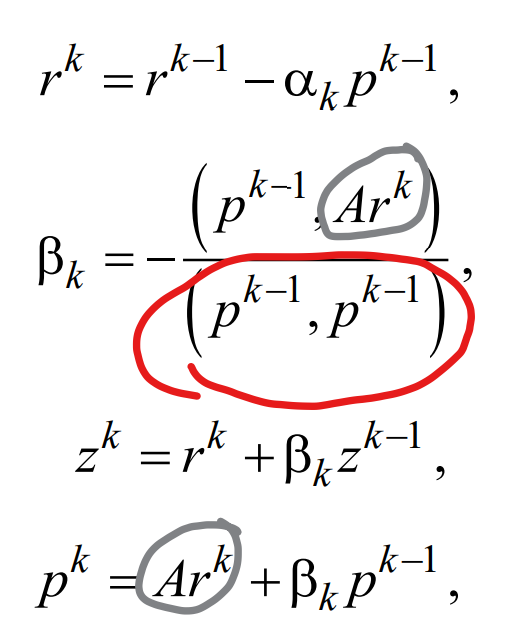
Локально-оптимальная схема (ЛОС):

Выбирается начальное приближение x0 и полагается:



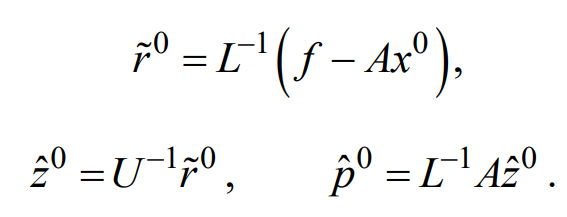
Далее для k = 1, 2, … проводятся следующие вычисления:



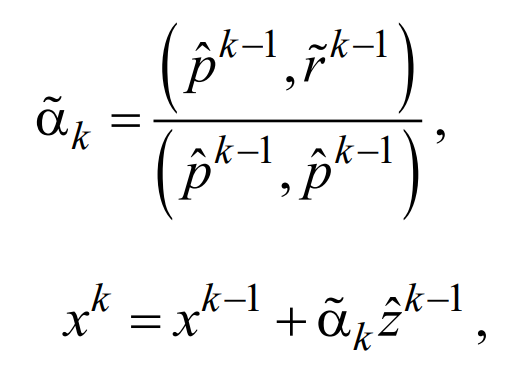


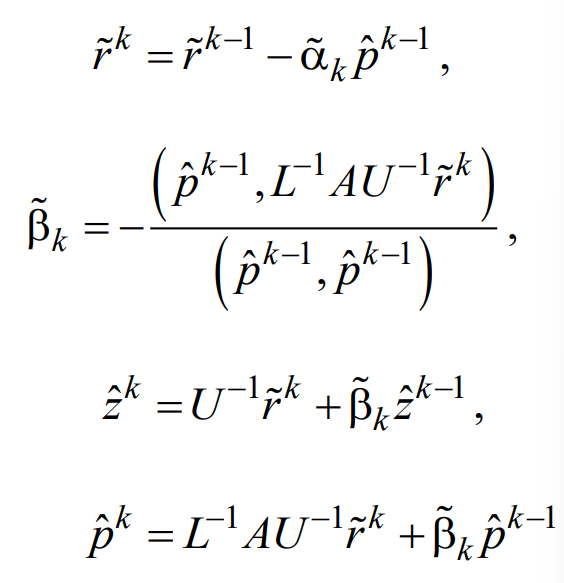
Выход из итерационного процесса происходит либо по превышению максимально допустимого числа итераций, либо по условию малости относительной невязки:

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:

# Текст программы

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 {

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

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

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;

}

/// <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 {

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

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

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;

}

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;

{

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 (mat.ig[0] == 1)

{

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];

}

//uint16\_t var;

//while (!jgS.eof())

//{

// jgS >> var;

// mat.jg.push\_back(var);

//}

}

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 matrix.h:

#pragma once

#include <vector>

#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 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 main.cpp:

#include <iostream>

#include <fstream>

#include <vector>

#include <format>

#include "Chrono\_Timer.h"

#include "Matrix.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;

namespace MSG\_Assimetric {

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

size\_t size = x.size();

vector<double> r = A \* x;

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

r = A.TranspMultToVec(r); // r0 = A^t \* (f - A \* x)

vector<double> z = r; // z0

vector<double> t(size);

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 = DBL\_MAX;

size\_t iter;

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

{

A.TranspMultToVec(A \* z, 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(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;

}

}

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;

}

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

size\_t size = x.size();

vector<double> D(size); // D = обратный корень от диагонали матрицы

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

vector<double> local\_x(size);

Vec::Mult(D, x, local\_x);

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

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

vector<double> tmp = A \* x;

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 = r;

vector<double> t(size); // 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++)

{

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;

}

}

Vec::Mult(D, 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;

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

vector<double> r = A \* x;

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

vector<double> z = r; // z0

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

vector<double> Ar(x.size()); // 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

uint16\_t size = x.size();

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;

}

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

uint16\_t size = x.size();

vector<double> D(size); // обратный корень от диагонали матрицы

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

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

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

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

vector<double> z = Vec::Mult(D, r); // z0 = U^-1 \* r

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

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

vector<double> Ar(size); // Ar = L^-1 \* A \* U^-1 \* r

vector<double> tmp(size);

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)

{

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

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

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

vector<double> z = Vec::Mult(D, r); // z0 = U^-1 \* r

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

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;

}

}

};

int main() {

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

uint16\_t matrixSize;

Matrix mat;

vector<double> f;

vector<double> x;

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;

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

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

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

cout << " 3) ЛОС (без предобуславливания)" << endl;

cout << " 4) ЛОС (диагональное предобуславливание)" << endl;

int userCase;

cin >> userCase;

switch (userCase)

{

case 1:

{

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

Timer timer;

double eps = 0;

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

timer.elapsed();

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;

Timer timer;

double eps = 0;

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

timer.elapsed();

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

Timer timer;

double eps = 0;

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

timer.elapsed();

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 (диагональное предобуславливание)" << endl << endl;

Timer timer;

double eps = 0;

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

timer.elapsed();

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;

}

}

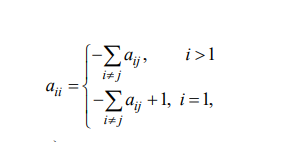
outFile.close();

return 0;

}

# 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

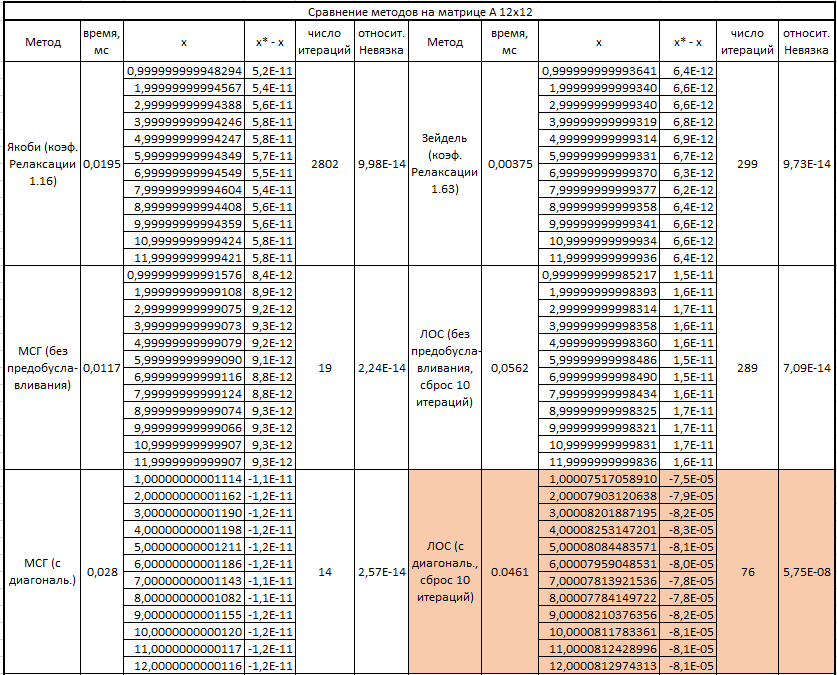
Параметры тестирования для матриц:

Максимальное число итераций: 2000

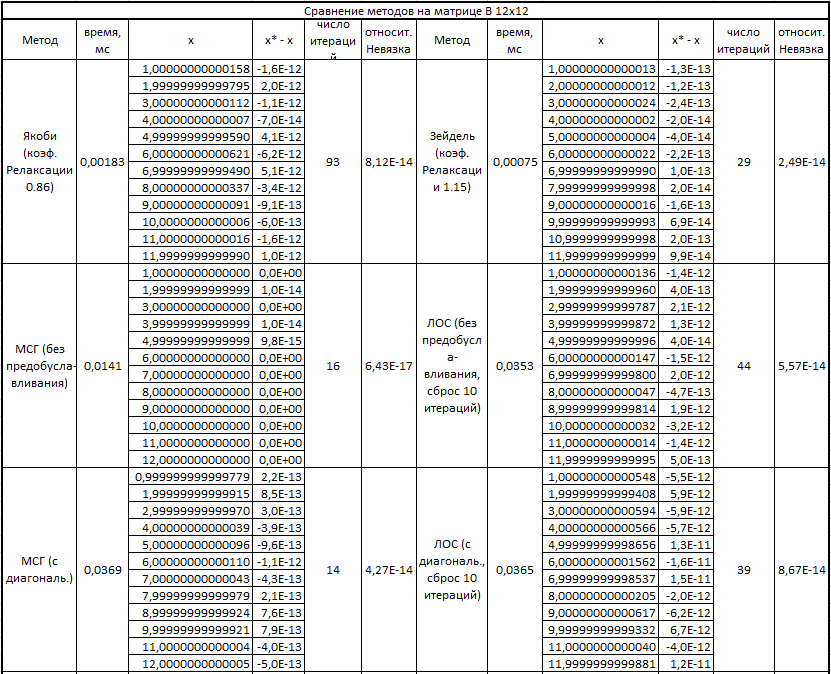
Начальное приближение: нулевой вектор

Минимальная относительная невязка: 10-13

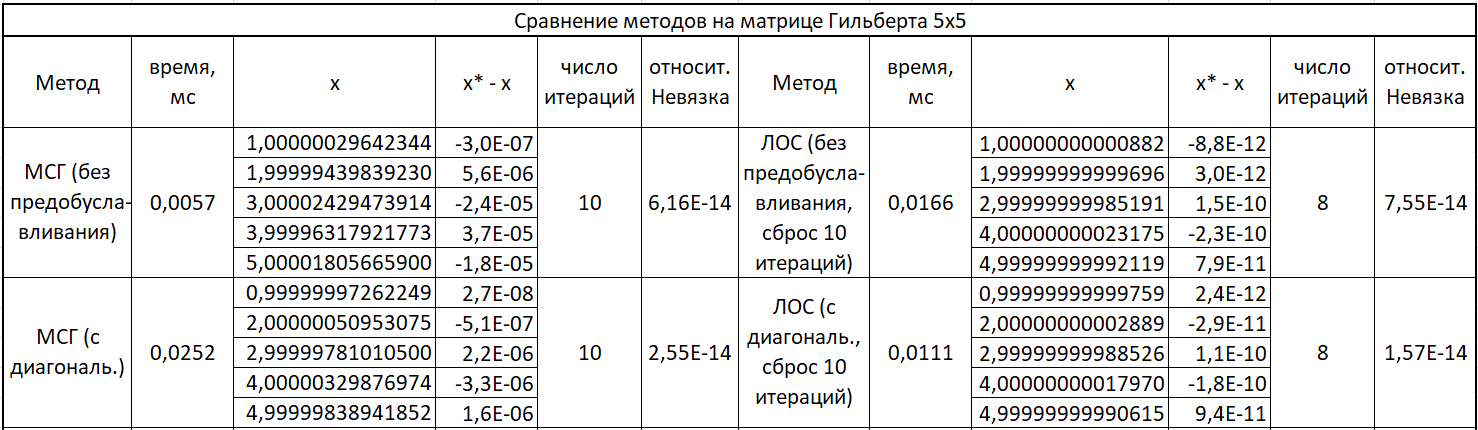
**Матрица А:**

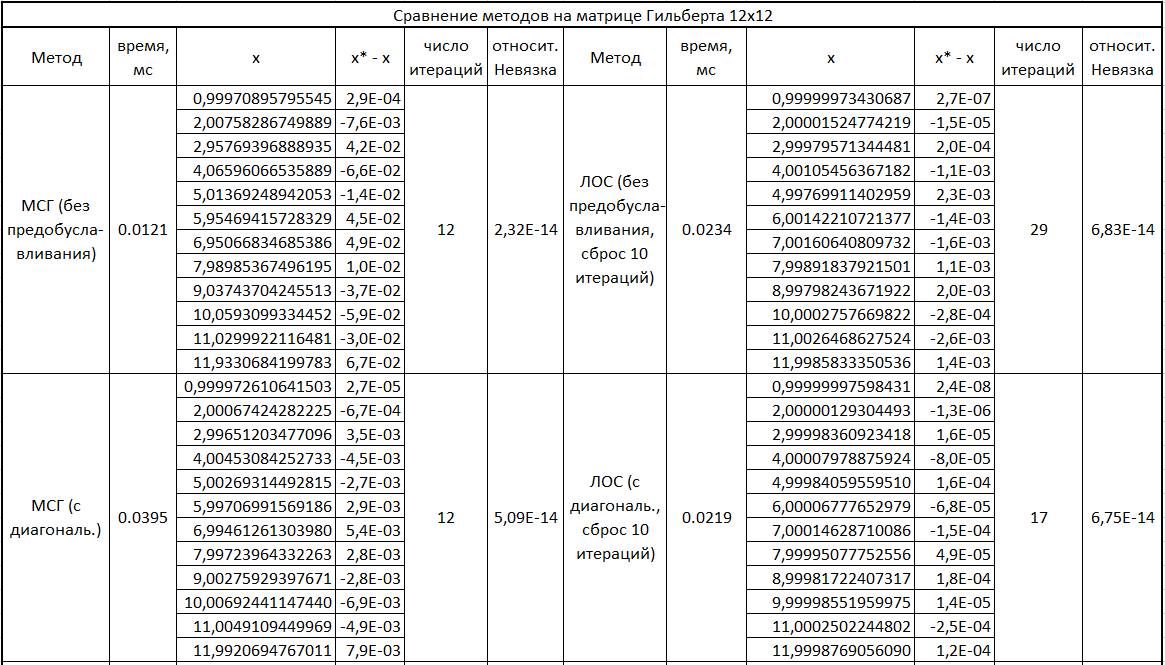


**Матрица В:**

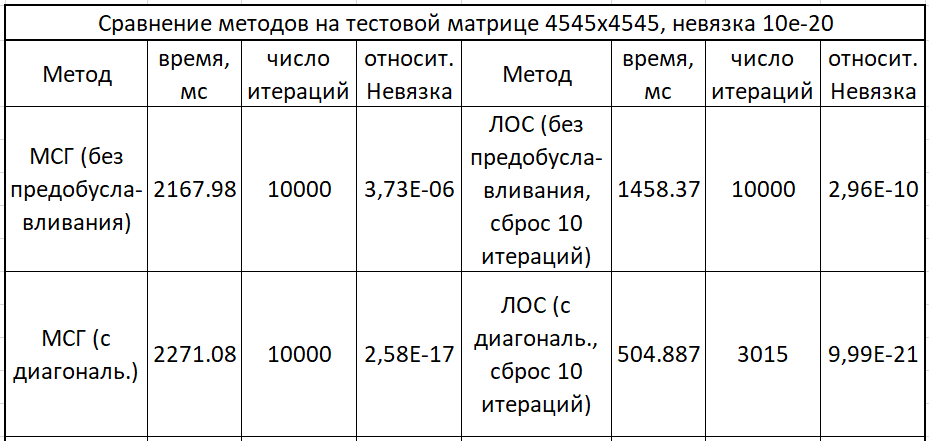


Сравнение на матрицах Гильберта:





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



# Выводы:

1. В ходе исследования мы заметили, что от значения параметра релаксации зависит то, за сколько итераций метод сойдётся к приблизительно верному ответу. Причём для каждой матрицы и для каждого метода решения есть своё идеальное значение параметра релаксации, при котором решение будет достигнуто быстрее всего.
2. Для обоих рассмотренных методов, при отдалении в любую сторону от идеального значения параметра релаксации увеличивается число итераций, необходимых для достижения приблизительно верного ответа, причём зависимость эта не линейна.
3. Для метода Гаусса-Зейделя, есть такие значения параметра релаксации, при котором метод начинает расходиться и, соответственно, решение получить невозможно. Для метода Якоби на матрице А, оптимальное значение параметра релаксации оказалось больше единицы. Скорее всего это связано с тем, что мы работаем с нессиметричными положительно определёнными матрицами, тогда как эти границы справедливы для симметричных матриц.
4. При оценке числа обусловленности матрицы, для оптимального решения разных методов на одной матрице были получены сильно отличающиеся значения, причём для метода Якоби они всегда больше. Связано это с тем, что метод Зейделя получил более точные решения (и следственно меньшую относительную невязку), чем метод Якоби.
5. При оценке числа обусловленности матрицы для оптимального решения метода Зейделя получилось значение, меньшее единицы. Связано это с тем, что сама оценка числа обусловленности - это ограничение числа обусловленности снизу, поэтому она может быть меньше 1.
6. Для матрицы А с большим числом числа обусловленности потребовалось больше шагов, чтобы достичь правильного ответа, нежели для матрицы B. Также для матрицы A для эффективной сходимости методов потребовались значения параметра релаксации большие, чем для эффективной сходимости методов для матрицы B с меньшим числом обусловленности.
7. Метод Гаусса-Зейделя оказался эффективнее на обоих матрицах по сравнению с методом Якоби: даже на одинаковых значениях параметра релаксации метод Зейделя сходился быстрее, чем метод Якоби. Можно полагать, что метод Зейделя оказался эффективнее за счёт того, что при вычислениях каждого нового элемента вектора Х он использует уже вычисленные до этого новые значения вектора Х, более приближенных к верному ответу.