**Реализую итерации**

Предлагаю начать с анализа клеохонов и решения систем линеарных уравнений используя численные методы. Для этого нужны тестовые примеры и визуализация.

**Проверяю симметрию**

Я анализирую симметрию и положительную определенность матрицы, выбирая простое решение и вычисляя вектор правой части для проверки точности.

**Исследую вычисления**

Окей, я смотрю на вычисления определителей матрицы с близкими к нулю значениями, используя C и Python для анализа и визуализации.

**Идет разработка**

Вроде бы разрабатываю метод простых итераций на языке С и создаю тестовые матрицы. Интересно, как точность зависит от числа условности.

**Изучение зависимости**

Я исследую зависимость времени выполнения и ошибок от числа обусловленности матриц, а также влияние точности на их погрешности и количество итераций.

**Создание и анализ**

Я предлагаю написать код на C для создания нескольк**Решение СЛАУ итерационными методами (Методом простых итераций)**

**Постановка задачи**

Необходимо исследовать решение систем линейных алгебраических уравнений (СЛАУ) с использованием **метода простых итераций** (также известного как метод Якоби). Требуется:

1. Создать набор матриц размерности не менее 10x10 с различными числами обусловленности от 11 до 108108.
2. Матрицы должны быть симметричными и положительно определенными только для методов, где это необходимо. В остальных случаях эти свойства должны нарушаться.
3. Выбрать точное решение (например, вектор из единиц или вектор чисел от 1 до 10).
4. Получить вектор правой части путем умножения матрицы на известное решение.
5. Проверить точность полученного решения с помощью бесконечной нормы вектора.
6. Рассмотреть заданную точность: 10−15,10−14,…,10−210−15,10−14,…,10−2.

**Исследования:**

1. Зависимость точности от числа обусловленности.
2. Зависимость времени выполнения от числа обусловленности.
3. Исследовать зависимость погрешности от заданной точности для матриц плохой и хорошей обусловленности.
4. Исследовать требуемое число итераций от заданной точности для матриц плохой и хорошей обусловленности.
5. Исследовать уменьшение погрешности с ходом итераций для матриц плохой и хорошей обусловленности.
6. Рассмотреть случаи, когда определитель матрицы близок к нулю (10−15,10−14,…,10−210−15,10−14,…,10−2).

**1. Реализация на языке C**

**1.1 Структура проекта**

* **main.c** — основная программа.
* **jacobi.h / jacobi.c** — функции для метода простых итераций (метода Якоби).
* **matrix.h / matrix.c** — функции для работы с матрицами.
* **utils.h / utils.c** — вспомогательные функции (нормы, генерация матриц и т.д.).

их файлов и объяснения его работы, а затем использовать Python для построения графиков и их анализа.

**Создаю код**

Прописываю задачу и код на C, включая функции для вычислений, а также генерирую матрицы с различными условными числами для метода простых итераций.

**3. Выводы по результатам исследований**

**3.1 Зависимость точности от числа обусловленности**

График показывает, что с увеличением числа обусловленности матрицы погрешность решения методом простых итераций увеличивается. Это связано с тем, что плохо обусловленные матрицы приводят к медленной сходимости итерационного процесса и усилению ошибок округления.

**3.2 Зависимость времени выполнения от числа обусловленности**

Время выполнения метода простых итераций увеличивается с ростом числа обусловленности матрицы. Это обусловлено увеличением числа итераций, необходимых для достижения заданной точности.

**3.3 Зависимость погрешности от заданной точности**

Для матриц с хорошей обусловленностью погрешность решения соответствует заданной точности. Для плохо обусловленных матриц погрешность может быть значительно выше, что указывает на ограниченную эффективность метода в таких случаях.

**3.4 Требуемое число итераций от заданной точности**

Число итераций увеличивается с повышением требуемой точности. Для матриц с плохой обусловленностью рост числа итераций более значителен, что свидетельствует о медленной сходимости метода.

**3.5 Уменьшение погрешности с ходом итераций**

Для матриц с хорошей обусловленностью погрешность быстро уменьшается с каждым шагом итерации. В случае плохой обусловленности снижение погрешности происходит значительно медленнее, требуя большего числа итераций для достижения приемлемой точности.

**3.6 Зависимость погрешности от малого определителя**

При уменьшении определителя матрицы до малых значений погрешность решения увеличивается. Это связано с тем, что матрица становится близкой к вырожденной, и решение системы становится неустойчивым.

**Заключение**

В ходе выполнения работы были исследованы особенности решения СЛАУ методом простых итераций. Результаты показали:

* Метод простых итераций эффективен для матриц с хорошей обусловленностью.
* Плохо обусловленные матрицы приводят к увеличению погрешности и времени вычислений.
* Заданная точность существенно влияет на число итераций и точность решения.
* Малые значения определителя матрицы негативно сказываются на устойчивости и точности решения.

Представленные программы на языке C и скрипты на Python позволяют воспроизвести результаты и могут быть использованы для дальнейших исследований и анализа итерационных методов решения СЛАУ.

**Примечание:** Код является демонстрационным и может требовать дополнительных проверок и оптимизаций для использования в производственных условиях. Рекомендуется проводить тестирование и валидацию перед применением.

**1.2 Файлы заголовков и реализаций**

**matrix.h**

#ifndef MATRIX\_H

#define MATRIX\_H

#include <stddef.h>

void generate\_matrix(double \*\*A, size\_t n, double cond\_number, int is\_spd, int determinant\_case);

void multiply\_matrix\_vector(double \*\*A, double \*x, double \*b, size\_t n);

double calculate\_condition\_number(double \*\*A, size\_t n);

double calculate\_determinant(double \*\*A, size\_t n);

#endif // MATRIX\_H

**matrix.c**

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include "matrix.h"

#include "utils.h"

void generate\_matrix(double \*\*A, size\_t n, double cond\_number, int is\_spd, int determinant\_case) {

// Генерация матрицы с заданным числом обусловленности

// Метод: SVD или специальная конструкция

// Начальная матрица

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

for (size\_t j = 0; j < n; j++) {

if (i == j) {

A[i][j] = 1.0;

} else {

A[i][j] = ((double)rand() / RAND\_MAX) \* 0.01;

}

}

}

// Изменение собственных значений для достижения заданного числа обусловленности

A[0][0] = 1.0;

A[n - 1][n - 1] = 1.0 / cond\_number;

// Если необходимо, сделать матрицу симметричной положительно определенной

if (is\_spd) {

// A = A^T \* A

double \*\*At = allocate\_matrix(n);

transpose\_matrix(A, At, n);

double \*\*AtA = allocate\_matrix(n);

multiply\_matrices(At, A, AtA, n);

copy\_matrix(AtA, A, n);

free\_matrix(At, n);

free\_matrix(AtA, n);

}

// Вариант для случая с малым определителем

if (determinant\_case) {

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

A[i][i] = determinant\_case;

}

}

}

void multiply\_matrix\_vector(double \*\*A, double \*x, double \*b, size\_t n) {

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

b[i] = 0.0;

for (size\_t j = 0; j < n; j++) {

b[i] += A[i][j] \* x[j];

}

}

}

double calculate\_condition\_number(double \*\*A, size\_t n) {

// Приближенное вычисление числа обусловленности

// Используем отношение норм

double \*\*A\_inv = allocate\_matrix(n);

invert\_matrix(A, A\_inv, n);

double norm\_A = matrix\_norm(A, n);

double norm\_A\_inv = matrix\_norm(A\_inv, n);

free\_matrix(A\_inv, n);

return norm\_A \* norm\_A\_inv;

}

double calculate\_determinant(double \*\*A, size\_t n) {

// Используем метод LU-разложения без выбора главного элемента

double det = 1.0;

double \*\*LU = allocate\_matrix(n);

copy\_matrix(A, LU, n);

for (size\_t k = 0; k < n; k++) {

if (LU[k][k] == 0) return 0.0;

for (size\_t i = k + 1; i < n; i++) {

LU[i][k] /= LU[k][k];

for (size\_t j = k + 1; j < n; j++) {

LU[i][j] -= LU[i][k] \* LU[k][j];

}

}

det \*= LU[k][k];

}

free\_matrix(LU, n);

return det;

}

**jacobi.h**

#ifndef JACOBI\_H

#define JACOBI\_H

#include <stddef.h>

int jacobi\_method(double \*\*A, double \*b, double \*x, size\_t n, double tol, int max\_iter, double \*errors);

#endif // JACOBI\_H

**jacobi.c**

#include <stdlib.h>

#include <math.h>

#include "jacobi.h"

#include "utils.h"

int jacobi\_method(double \*\*A, double \*b, double \*x, size\_t n, double tol, int max\_iter, double \*errors) {

double \*x\_old = (double \*)malloc(n \* sizeof(double));

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

x[i] = 0.0; // Начальное приближение

}

for (int iter = 0; iter < max\_iter; iter++) {

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

x\_old[i] = x[i];

}

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

double sum = 0.0;

for (size\_t j = 0; j < n; j++) {

if (i != j) {

sum += A[i][j] \* x\_old[j];

}

}

x[i] = (b[i] - sum) / A[i][i];

}

double error = vector\_norm\_diff(x, x\_old, n);

errors[iter] = error;

if (error < tol) {

free(x\_old);

return iter + 1; // Успешная сходимость

}

}

free(x\_old);

return max\_iter; // Достигнуто максимальное число итераций

}

**utils.h**

#ifndef UTILS\_H

#define UTILS\_H

#include <stddef.h>

double \*\*allocate\_matrix(size\_t n);

void free\_matrix(double \*\*A, size\_t n);

void copy\_matrix(double \*\*src, double \*\*dest, size\_t n);

void transpose\_matrix(double \*\*A, double \*\*At, size\_t n);

void multiply\_matrices(double \*\*A, double \*\*B, double \*\*C, size\_t n);

double matrix\_norm(double \*\*A, size\_t n);

double vector\_norm(double \*v, size\_t n);

double vector\_norm\_diff(double \*v1, double \*v2, size\_t n);

void invert\_matrix(double \*\*A, double \*\*A\_inv, size\_t n);

#endif // UTILS\_H

**utils.c**

#include <stdlib.h>

#include <math.h>

#include "utils.h"

double \*\*allocate\_matrix(size\_t n) {

double \*\*A = (double \*\*)malloc(n \* sizeof(double \*));

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

A[i] = (double \*)malloc(n \* sizeof(double));

}

return A;

}

void free\_matrix(double \*\*A, size\_t n) {

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

free(A[i]);

}

free(A);

}

void copy\_matrix(double \*\*src, double \*\*dest, size\_t n) {

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

for (size\_t j = 0; j < n; j++) {

dest[i][j] = src[i][j];

}

}

}

void transpose\_matrix(double \*\*A, double \*\*At, size\_t n) {

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

for (size\_t j = 0; j < n; j++) {

At[j][i] = A[i][j];

}

}

}

void multiply\_matrices(double \*\*A, double \*\*B, double \*\*C, size\_t n) {

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

for (size\_t j = 0; j < n; j++) {

C[i][j] = 0.0;

for (size\_t k = 0; k < n; k++) {

C[i][j] += A[i][k] \* B[k][j];

}

}

}

}

double matrix\_norm(double \*\*A, size\_t n) {

// Норма матрицы (норма Фробениуса)

double norm = 0.0;

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

for (size\_t j = 0; j < n; j++) {

norm += A[i][j] \* A[i][j];

}

}

return sqrt(norm);

}

double vector\_norm(double \*v, size\_t n) {

// Бесконечная норма вектора

double max = fabs(v[0]);

for (size\_t i = 1; i < n; i++) {

if (fabs(v[i]) > max) {

max = fabs(v[i]);

}

}

return max;

}

double vector\_norm\_diff(double \*v1, double \*v2, size\_t n) {

double max = fabs(v1[0] - v2[0]);

for (size\_t i = 1; i < n; i++) {

double diff = fabs(v1[i] - v2[i]);

if (diff > max) {

max = diff;

}

}

return max;

}

void invert\_matrix(double \*\*A, double \*\*A\_inv, size\_t n) {

// Используем метод Гаусса-Жордана для инверсии матрицы

// Создаем расширенную матрицу [A | I]

double \*\*aug = (double \*\*)malloc(n \* sizeof(double \*));

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

aug[i] = (double \*)malloc(2 \* n \* sizeof(double));

for (size\_t j = 0; j < n; j++) {

aug[i][j] = A[i][j];

aug[i][j + n] = (i == j) ? 1.0 : 0.0;

}

}

// Приводим к верхнетреугольному виду

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

double pivot = aug[i][i];

if (fabs(pivot) < 1e-12) {

// Система вырожденная

for (size\_t k = 0; k < n; k++) {

free(aug[k]);

}

free(aug);

return;

}

for (size\_t j = 0; j < 2 \* n; j++) {

aug[i][j] /= pivot;

}

for (size\_t k = 0; k < n; k++) {

if (k != i) {

double factor = aug[k][i];

for (size\_t j = 0; j < 2 \* n; j++) {

aug[k][j] -= factor \* aug[i][j];

}

}

}

}

// Извлечение обратной матрицы

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

for (size\_t j = 0; j < n; j++) {

A\_inv[i][j] = aug[i][j + n];

}

free(aug[i]);

}

free(aug);

}

**1.3 Основная программа**

**main.c**

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <time.h>

#include "jacobi.h"

#include "matrix.h"

#include "utils.h"

#define N 10 // Размер матрицы

#define MAX\_ITER 10000

int main() {

srand((unsigned int)time(NULL));

double cond\_numbers[] = {1e0, 1e2, 1e4, 1e6, 1e8};

int num\_cond = sizeof(cond\_numbers) / sizeof(cond\_numbers[0]);

double tolerances[] = {1e-15, 1e-14, 1e-13, 1e-12, 1e-11,

1e-10, 1e-9, 1e-8, 1e-7, 1e-6,

1e-5, 1e-4, 1e-3, 1e-2};

int num\_tol = sizeof(tolerances) / sizeof(tolerances[0]);

double \*\*A = allocate\_matrix(N);

double \*x\_exact = (double \*)malloc(N \* sizeof(double));

double \*b = (double \*)malloc(N \* sizeof(double));

double \*x = (double \*)malloc(N \* sizeof(double));

double \*errors = (double \*)malloc(MAX\_ITER \* sizeof(double));

// Точное решение (вектор из 1 до 10)

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

x\_exact[i] = (double)(i + 1);

}

// Файлы для записи результатов

FILE \*file\_accuracy = fopen("accuracy\_vs\_cond.txt", "w");

FILE \*file\_time = fopen("time\_vs\_cond.txt", "w");

FILE \*file\_iterations = fopen("iterations\_vs\_tol.txt", "w");

FILE \*file\_error\_progress\_good = fopen("error\_progress\_good.txt", "w");

FILE \*file\_error\_progress\_bad = fopen("error\_progress\_bad.txt", "w");

// 1) Зависимость точности и времени от числа обусловленности

for (int idx = 0; idx < num\_cond; idx++) {

double cond\_number = cond\_numbers[idx];

// Генерация матрицы

generate\_matrix(A, N, cond\_number, 0, 0); // 0 - не SPD, 0 - обычный случай

// Вычисление реального числа обусловленности

double real\_cond = calculate\_condition\_number(A, N);

// Получение вектора правой части

multiply\_matrix\_vector(A, x\_exact, b, N);

// Решение системы

clock\_t start\_time = clock();

int iterations = jacobi\_method(A, b, x, N, 1e-10, MAX\_ITER, errors);

clock\_t end\_time = clock();

// Вычисление ошибки

double error = vector\_norm\_diff(x, x\_exact, N);

// Запись результатов

fprintf(file\_accuracy, "%e %e\n", real\_cond, error);

fprintf(file\_time, "%e %lf\n", real\_cond, (double)(end\_time - start\_time) / CLOCKS\_PER\_SEC);

}

// 3) Зависимость погрешности от заданной точности

// Для матрицы с хорошей обусловленностью

generate\_matrix(A, N, 1e2, 0, 0);

multiply\_matrix\_vector(A, x\_exact, b, N);

for (int i = 0; i < num\_tol; i++) {

double tol = tolerances[i];

int iterations = jacobi\_method(A, b, x, N, tol, MAX\_ITER, errors);

double error = vector\_norm\_diff(x, x\_exact, N);

fprintf(file\_iterations, "%e %d %e\n", tol, iterations, error);

}

// 5) Уменьшение погрешности с ходом итераций

// Для хорошей обусловленности

generate\_matrix(A, N, 1e2, 0, 0);

multiply\_matrix\_vector(A, x\_exact, b, N);

int iterations = jacobi\_method(A, b, x, N, 1e-15, MAX\_ITER, errors);

FILE \*file\_error\_good = fopen("error\_progress\_good.txt", "w");

for (int i = 0; i < iterations; i++) {

fprintf(file\_error\_good, "%d %e\n", i + 1, errors[i]);

}

fclose(file\_error\_good);

// Для плохой обусловленности

generate\_matrix(A, N, 1e8, 0, 0);

multiply\_matrix\_vector(A, x\_exact, b, N);

iterations = jacobi\_method(A, b, x, N, 1e-15, MAX\_ITER, errors);

FILE \*file\_error\_bad = fopen("error\_progress\_bad.txt", "w");

for (int i = 0; i < iterations; i++) {

fprintf(file\_error\_bad, "%d %e\n", i + 1, errors[i]);

}

fclose(file\_error\_bad);

// 6) Рассмотрение случаев с малым определителем

double determinants[] = {1e-15, 1e-14, 1e-13, 1e-12, 1e-11,

1e-10, 1e-9, 1e-8, 1e-7, 1e-6,

1e-5, 1e-4, 1e-3, 1e-2};

int num\_det = sizeof(determinants) / sizeof(determinants[0]);

FILE \*file\_error\_determinant = fopen("error\_vs\_determinant.txt", "w");

for (int idx = 0; idx < num\_det; idx++) {

double det\_value = determinants[idx];

generate\_matrix(A, N, 1e2, 0, det\_value);

multiply\_matrix\_vector(A, x\_exact, b, N);

int iterations = jacobi\_method(A, b, x, N, 1e-10, MAX\_ITER, errors);

double error = vector\_norm\_diff(x, x\_exact, N);

fprintf(file\_error\_determinant, "%e %e\n", det\_value, error);

}

fclose(file\_error\_determinant);

// Очистка памяти и закрытие файлов

free\_matrix(A, N);

free(x\_exact);

free(b);

free(x);

free(errors);

fclose(file\_accuracy);

fclose(file\_time);

fclose(file\_iterations);

return 0;

}

**2. Построение графиков с помощью Python**

**2.1 Зависимость точности от числа обусловленности**

**plot\_accuracy\_vs\_cond.py**

import numpy as np

import matplotlib.pyplot as plt

data = np.loadtxt('accuracy\_vs\_cond.txt')

cond\_numbers = data[:, 0]

errors = data[:, 1]

plt.figure(figsize=(8, 6))

plt.loglog(cond\_numbers, errors, marker='o')

plt.xlabel('Число обусловленности')

plt.ylabel('Погрешность решения')

plt.title('Зависимость точности от числа обусловленности')

plt.grid(True, which='both')

plt.show()

**2.2 Зависимость времени выполнения от числа обусловленности**

**plot\_time\_vs\_cond.py**

import numpy as np

import matplotlib.pyplot as plt

data = np.loadtxt('time\_vs\_cond.txt')

cond\_numbers = data[:, 0]

times = data[:, 1]

plt.figure(figsize=(8, 6))

plt.loglog(cond\_numbers, times, marker='o')

plt.xlabel('Число обусловленности')

plt.ylabel('Время выполнения (сек)')

plt.title('Зависимость времени выполнения от числа обусловленности')

plt.grid(True, which='both')

plt.show()

**2.3 Зависимость погрешности от заданной точности**

**plot\_error\_vs\_tol.py**

import numpy as np

import matplotlib.pyplot as plt

data = np.loadtxt('iterations\_vs\_tol.txt')

tolerances = data[:, 0]

errors = data[:, 2]

plt.figure(figsize=(8, 6))

plt.loglog(tolerances, errors, marker='o')

plt.xlabel('Заданная точность')

plt.ylabel('Погрешность решения')

plt.title('Зависимость погрешности от заданной точности')

plt.grid(True, which='both')

plt.show()

**2.4 Требуемое число итераций от заданной точности**

**plot\_iterations\_vs\_tol.py**

import numpy as np

import matplotlib.pyplot as plt

data = np.loadtxt('iterations\_vs\_tol.txt')

tolerances = data[:, 0]

iterations = data[:, 1]

plt.figure(figsize=(8, 6))

plt.loglog(tolerances, iterations, marker='o')

plt.xlabel('Заданная точность')

plt.ylabel('Число итераций')

plt.title('Зависимость числа итераций от заданной точности')

plt.grid(True, which='both')

plt.show()

**2.5 Уменьшение погрешности с ходом итераций**

**plot\_error\_progress.py**

import numpy as np

import matplotlib.pyplot as plt

# Для хорошей обусловленности

data\_good = np.loadtxt('error\_progress\_good.txt')

iterations\_good = data\_good[:, 0]

errors\_good = data\_good[:, 1]

# Для плохой обусловленности

data\_bad = np.loadtxt('error\_progress\_bad.txt')

iterations\_bad = data\_bad[:, 0]

errors\_bad = data\_bad[:, 1]

plt.figure(figsize=(8, 6))

plt.semilogy(iterations\_good, errors\_good, label='Хорошая обусловленность')

plt.semilogy(iterations\_bad, errors\_bad, label='Плохая обусловленность')

plt.xlabel('Номер итерации')

plt.ylabel('Погрешность')

plt.title('Уменьшение погрешности с ходом итераций')

plt.legend()

plt.grid(True, which='both')

plt.show()

**2.6 Зависимость погрешности от малого определителя**

**plot\_error\_vs\_determinant.py**

import numpy as np

import matplotlib.pyplot as plt

data = np.loadtxt('error\_vs\_determinant.txt')

determinants = data[:, 0]

errors = data[:, 1]

plt.figure(figsize=(8, 6))

plt.loglog(determinants, errors, marker='o')

plt.xlabel('Определитель матрицы')

plt.ylabel('Погрешность решения')

plt.title('Зависимость погрешности от малого определителя')

plt.grid(True, which='both')

plt.show()