**Создаю примеры**

Я работаю над тестовыми примерами, создавая матрицы размера 10x10 с различной обусловленностью от 1 до 10^8, включая симметричные и положительно-определенные для определенных методов.

**Проверка точности и анализа**

Умножаю матрицу на известное решение и оцениваю точность бесконечной нормой вектора. Графики рассчитываю на C, а выводы делаю на Python.

**Создание тестовых примеров**

Я выделяю создание матриц размером не менее 10x10, с различными числами условий, от 1 до 10^8. Использую симметричные и положительно определенные матрицы, а также выбор известного точного решения. Определяю вектор правой части и проверяю точность решения с помощью нормы бесконечности.

**Изучаю зависимости**

Добавляю возмущения к вектору и элементам матрицы, вычисляю ошибки и проверяю результаты. Визуализирую графики и делаю выводы.

**Создание кода и анализа**

Основное внимание уделяется генерации тестовых матриц, реализации LU-разложения в C, расчету характеристик матриц, выполнению экспериментов и визуализации данных на Python.

**Создание испытательных матриц**

Сначала я генерирую матрицы размером не менее 10x10 с различными условиями числа. Затем выбираю известный вектор решения и рассчитываю вектор правой части. Решаю систему с использованием LU-разложения. Важно интересно, как это соотносится с точностью.

**Постановка задачи**

Необходимо провести исследования решения систем линейных алгебраических уравнений (СЛАУ) с использованием метода LU-разложения. Требуется:

1. Создать набор матриц размерности не менее 10x10 с различными числами обусловленности от 1 до 108108.
2. Выбрать точное решение (например, вектор из единиц).
3. Получить вектор правой части путем умножения матрицы на известное решение.
4. Проверить точность полученного решения с помощью бесконечной нормы вектора.
5. Провести исследования:
   * Зависимость точности от числа обусловленности.
   * Зависимость времени выполнения от числа обусловленности.
   * Зависимость относительной погрешности от возмущения вектора правой части.
   * Зависимость относительной погрешности от возмущения элементов матрицы.

Все вычисления выполнить на языке C, а для построения графиков использовать Python.

**1. Реализация на языке C**

**1.1 Структура проекта**

* **main.c** — основная программа.
* **matrix.h / matrix.c** — функции для работы с матрицами.
* **lu.h / lu.c** — функции для LU-разложения и решения СЛАУ.
* **utils.h / utils.c** — вспомогательные функции (например, вычисление нормы, числа обусловленности и т.д.).

**3. Выводы по результатам исследований**

**3.1 Зависимость точности от числа обусловленности**

График показывает, что с увеличением числа обусловленности матрицы погрешность решения увеличивается. Это ожидаемо, так как плохо обусловленные матрицы усиливают ошибки, возникающие из-за ограниченной точности вычислений.

**3.2 Зависимость времени выполнения от числа обусловленности**

Время выполнения алгоритма LU-разложения незначительно зависит от числа обусловленности матрицы. Это связано с тем, что сложность алгоритма O(n3)*O*(*n*3) и определяется размером матрицы, а не ее числом обусловленности.

**3.3 Зависимость относительной погрешности от возмущения вектора правой части**

График демонстрирует, что относительная погрешность решения увеличивается с возрастанием возмущения вектора правой части. Это показывает чувствительность решения к изменениям вектора b*b*, особенно для плохо обусловленных матриц.

**3.4 Зависимость относительной погрешности от возмущения элемента матрицы**

Возмущение элементов матрицы приводит к увеличению относительной погрешности решения. Возмущение наибольшего элемента матрицы оказывает более значительное влияние на точность решения по сравнению с возмущением наименьшего элемента.

**Заключение**

В ходе выполнения работы были проведены исследования решения СЛАУ методом LU-разложения. Были изучены влияния числа обусловленности матрицы и возмущений на точность решения. Результаты показали, что:

* Плохо обусловленные матрицы приводят к увеличению погрешности решения.
* Возмущения в векторе правой части и элементах матрицы негативно сказываются на точности решения.
* Время выполнения LU-разложения в основном зависит от размера матрицы и незначительно зависит от числа обусловленности.

Представленные программы на языке C и скрипты на Python позволяют воспроизвести результаты и могут быть использованы для дальнейших исследований.

**Примечание:** Представленный код является примером и может потребовать доработки для учета всех нюансов работы с матрицами большой размерности и особенностей численных методов.

**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);

void perturb\_matrix(double \*\*A, size\_t n, double perturbation, int perturb\_max);

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

double calculate\_condition\_number(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) {

// Генерация матрицы с заданным числом обусловленности

// Метод: используя сингулярное разложение

double \*U[n], \*V[n], \*S;

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

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

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

}

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

// Заполнение U и V случайными ортогональными матрицами

random\_orthogonal\_matrix(U, n);

random\_orthogonal\_matrix(V, n);

// Заполнение сингулярных значений

S[0] = 1.0;

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

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

S[i] = S[n - 1] + (S[0] - S[n - 1]) \* ((double)rand() / RAND\_MAX);

}

// Сборка матрицы A = U \* S \* V^T

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

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

A[i][j] = 0.0;

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

A[i][j] += U[i][k] \* S[k] \* V[j][k];

}

}

}

// Если требуется симметричная положительно определенная матрица

if (is\_spd) {

// A = A^T \* A

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

transpose\_and\_multiply(A, AtA, n);

copy\_matrix(AtA, A, n);

free\_matrix(AtA, n);

}

// Очистка памяти

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

free(U[i]);

free(V[i]);

}

free(S);

}

void perturb\_matrix(double \*\*A, size\_t n, double perturbation, int perturb\_max) {

// Внесение возмущения в матрицу

size\_t index\_i = 0, index\_j = 0;

double target\_value = A[0][0];

// Поиск наибольшего или наименьшего элемента

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

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

if ((perturb\_max && fabs(A[i][j]) > fabs(target\_value)) ||

(!perturb\_max && fabs(A[i][j]) < fabs(target\_value))) {

target\_value = A[i][j];

index\_i = i;

index\_j = j;

}

}

}

// Внесение возмущения

A[index\_i][index\_j] \*= (1.0 + perturbation);

}

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];

}

}

}

// Функция для вычисления числа обусловленности (приближенно)

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

// Используем отношение норм

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

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

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

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

free\_matrix(A\_inv, n);

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

}

**lu.h**

#ifndef LU\_H

#define LU\_H

#include <stddef.h>

void lu\_decomposition(double \*\*A, double \*\*L, double \*\*U, size\_t n);

void lu\_solve(double \*\*L, double \*\*U, double \*b, double \*x, size\_t n);

#endif // LU\_H

**lu.c**

#include <stdlib.h>

#include "lu.h"

void lu\_decomposition(double \*\*A, double \*\*L, double \*\*U, size\_t n) {

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

// Верхний треугольник

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

double sum = 0.0;

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

sum += L[i][j] \* U[j][k];

}

U[i][k] = A[i][k] - sum;

}

// Нижний треугольник

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

if (i == k)

L[i][i] = 1.0;

else {

double sum = 0.0;

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

sum += L[k][j] \* U[j][i];

}

L[k][i] = (A[k][i] - sum) / U[i][i];

}

}

}

}

void lu\_solve(double \*\*L, double \*\*U, double \*b, double \*x, size\_t n) {

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

// Прямая подстановка L \* y = b

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

y[i] = b[i];

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

y[i] -= L[i][j] \* y[j];

}

y[i] /= L[i][i];

}

// Обратная подстановка U \* x = y

for (int i = n - 1; i >= 0; i--) {

x[i] = y[i];

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

x[i] -= U[i][j] \* x[j];

}

x[i] /= U[i][i];

}

free(y);

}

**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 random\_orthogonal\_matrix(double \*\*A, size\_t n);

void transpose\_and\_multiply(double \*\*A, double \*\*AtA, size\_t n);

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

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

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

double vector\_infinity\_norm(double \*v, 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 random\_orthogonal\_matrix(double \*\*A, size\_t n) {

// Используем метод Грама-Шмидта для генерации ортогональной матрицы

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

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

// Заполняем случайными значениями

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

Q[i][j] = (double)rand() / RAND\_MAX;

}

}

// Ортогонализация

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

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

double dot = 0.0;

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

dot += Q[k][i] \* Q[k][j];

}

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

Q[k][i] -= dot \* Q[k][j];

}

}

// Нормализация

double norm = 0.0;

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

norm += Q[k][i] \* Q[k][i];

}

norm = sqrt(norm);

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

Q[k][i] /= norm;

}

}

// Транспонируем для удобства

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

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

A[i][j] = Q[j][i];

}

}

free\_matrix(Q, n);

}

void transpose\_and\_multiply(double \*\*A, double \*\*AtA, size\_t n) {

// AtA = A^T \* A

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

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

AtA[i][j] = 0.0;

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

AtA[i][j] += A[k][i] \* A[k][j];

}

}

}

}

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

// Используем метод Гаусса-Жордана для инверсии матрицы

// Создаем расширенную матрицу [A | I]

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

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++) {

// Поиск ведущего элемента

double max = fabs(aug[i][i]);

size\_t max\_row = i;

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

if (fabs(aug[k][i]) > max) {

max = fabs(aug[k][i]);

max\_row = k;

}

}

// Перестановка строк

if (max\_row != i) {

double \*temp = aug[i];

aug[i] = aug[max\_row];

aug[max\_row] = temp;

}

// Нормализация ведущего элемента

double diag = aug[i][i];

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

aug[i][j] /= diag;

}

// Обнуление остальных элементов в столбце

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);

}

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

double max = 0.0;

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

double sum = 0.0;

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

sum += fabs(A[i][j]);

}

if (sum > max) {

max = sum;

}

}

return max;

}

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

double max = fabs(v[0]);

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

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

max = fabs(v[i]);

}

}

return max;

}

**1.3 Основная программа**

**main.c**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <math.h>

#include "matrix.h"

#include "lu.h"

#include "utils.h"

#define N 10 // Размер матрицы

#define NUM\_COND 8 // Количество различных чисел обусловленности

int main() {

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

double cond\_numbers[NUM\_COND] = {1e0, 1e1, 1e2, 1e3, 1e4, 1e5, 1e6, 1e8};

double perturbations[5] = {0.01, 0.02, 0.03, 0.04, 0.05};

// Выделение памяти

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

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

double \*\*U = 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));

// Точное решение (вектор из единиц)

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

x\_exact[i] = 1.0;

}

// Файлы для записи результатов

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

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

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

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

// Исследование зависимости 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 - не SPD

// Вычисление реального числа обусловленности (приближенно)

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();

lu\_decomposition(A, L, U, N);

lu\_solve(L, U, b, x, N);

clock\_t end\_time = clock();

// Вычисление ошибки

double error = 0.0;

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

error += fabs(x[i] - x\_exact[i]);

}

// 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);

}

// Возмущение вектора правой части

double cond\_number = 1e5;

generate\_matrix(A, N, cond\_number, 0);

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

lu\_decomposition(A, L, U, N);

lu\_solve(L, U, b, x, N);

double error\_base = 0.0;

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

error\_base += fabs(x[i] - x\_exact[i]);

}

for (size\_t idx = 0; idx < 5; idx++) {

double perturbation = perturbations[idx];

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

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

b\_perturbed[i] = b[i] \* (1.0 + ((double)rand() / RAND\_MAX - 0.5) \* 2 \* perturbation);

}

lu\_solve(L, U, b\_perturbed, x, N);

double error = 0.0;

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

error += fabs(x[i] - x\_exact[i]);

}

double relative\_error = error / error\_base;

fprintf(file\_perturb\_b, "%f %e\n", perturbation \* 100, relative\_error);

free(b\_perturbed);

}

// Возмущение элемента матрицы

for (size\_t idx = 0; idx < 5; idx++) {

double perturbation = perturbations[idx];

generate\_matrix(A, N, cond\_number, 0);

perturb\_matrix(A, N, perturbation, 1); // 1 - возмущаем наибольший элемент

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

lu\_decomposition(A, L, U, N);

lu\_solve(L, U, b, x, N);

double error = 0.0;

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

error += fabs(x[i] - x\_exact[i]);

}

double relative\_error = error / error\_base;

fprintf(file\_perturb\_A, "%f %e\n", perturbation \* 100, relative\_error);

}

// Очистка памяти и закрытие файлов

free\_matrix(A, N);

free\_matrix(L, N);

free\_matrix(U, N);

free(x\_exact);

free(b);

free(x);

fclose(file\_accuracy);

fclose(file\_time);

fclose(file\_perturb\_b);

fclose(file\_perturb\_A);

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 относительной погрешности от возмущения вектора правой части**

**plot\_error\_vs\_perturb\_b.py**

import numpy as np

import matplotlib.pyplot as plt

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

perturbations = data[:, 0]

relative\_errors = data[:, 1]

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

plt.plot(perturbations, relative\_errors, marker='o')

plt.xlabel('Возмущение вектора правой части (%)')

plt.ylabel('Относительная погрешность')

plt.title('Зависимость относительной погрешности от возмущения вектора правой части')

plt.grid(True)

plt.show()

**2.4 Зависимость относительной погрешности от возмущения элемента матрицы**

**plot\_error\_vs\_perturb\_A.py**

import numpy as np

import matplotlib.pyplot as plt

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

perturbations = data[:, 0]

relative\_errors = data[:, 1]

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

plt.plot(perturbations, relative\_errors, marker='o')

plt.xlabel('Возмущение элемента матрицы (%)')

plt.ylabel('Относительная погрешность')

plt.title('Зависимость относительной погрешности от возмущения элемента матрицы')

plt.grid(True)

plt.show()