Понимание и создание матриц для решения систем линейных уравнений.

**Проверяю точность решения**

Я смотрю на точность, используя бесконечную норму вектора. Это заставляет меня задуматься о зависимости точности от числа обусловленности и времени выполнения.

**Использую методы**

Для построения графиков в Python, применяю данные из LU-разложения. Открываю новые способы решения уравнений, включая вращательные матрицы.

**Создание матриц и решений**

Я занимаюсь генерацией C кода для создания матриц, решением SLAE с помощью ортогонализаций и вращений (например, вращения Гивенса), исследую и генерирую данные для графиков, пишу Python код для их построения и делаю выводы.

оказываю помощь

**Сравнение методов**

Я исследую зависимость относительной погрешности от возмущений наименьшего и наибольшего элемента матрицы, используя код на C для расчетов и Python для графиков.

**Решение СЛАУ методами ортогонализации с использованием матриц вращения**

**Постановка задачи**

Необходимо исследовать решение систем линейных алгебраических уравнений (СЛАУ) с использованием метода ортогонализации с помощью матриц вращения (метод Гивенса). Требуется:

1. Создать набор матриц размерности не менее 10x10 с различными числами обусловленности от 1 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. 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.
* **givens.h / givens.c** — функции для ортогонализации методом Гивенса.
* **utils.h / utils.c** — вспомогательные функции (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 возрастает. Это связано с тем, что плохо обусловленные матрицы усиливают ошибки округления и вычислений, что особенно заметно при использовании методов ортогонализации.

**3.2 Зависимость времени выполнения от числа обусловленности**

Время выполнения метода ортогонализации с помощью матриц вращения незначительно зависит от числа обусловленности. Основное влияние на время оказывает размер матрицы, а не ее число 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 указывает на чувствительность метода к изменениям в данных и необходимость контроля точности входных данных.

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

Возмущение элементов матрицы приводит к увеличению относительной погрешности решения. Особенно заметно влияние возмущения наибольшего элемента матрицы, что может быть связано с его доминирующим вкладом в вычисления.

**Заключение**

В ходе выполнения работы были проведены исследования решения СЛАУ методом ортогонализации с использованием матриц вращения. Результаты показали:

* С увеличением числа обусловленности матрицы точность решения снижается.
* Метод ортогонализации устойчив к возмущениям, однако точность все же зависит от качества входных данных.
* Возмущения в векторе правой части и элементах матрицы отрицательно сказываются на точности решения.
* Время выполнения метода в основном зависит от размера матрицы и мало зависит от числа обусловленности.

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

**Примечание:** Код является примером и может потребовать дополнительных проверок и оптимизаций для конкретных случаев. Рекомендуется провести тестирование и валидацию кода перед использованием в критических приложениях.

**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);

void copy\_matrix(double \*\*src, double \*\*dest, 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 (SVD)

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);

}

// Сортировка сингулярных значений для уменьшения

for (size\_t i = 0; i < n - 1; i++) {

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

if (S[i] < S[j]) {

double temp = S[i];

S[i] = S[j];

S[j] = temp;

}

}

}

// Сборка матрицы A = U \* S \* V^T

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

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

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

temp[i][j] = U[i][j] \* S[j];

}

}

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] += temp[i][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);

free\_matrix(temp, n);

}

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 \* ((double)rand() / RAND\_MAX));

}

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 с использованием 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;

}

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];

}

}

}

**givens.h**

#ifndef GIVENS\_H

#define GIVENS\_H

#include <stddef.h>

void givens\_rotation(double \*\*A, double \*b, size\_t n);

void back\_substitution(double \*\*R, double \*b, double \*x, size\_t n);

#endif // GIVENS\_H

**givens.c**

#include <stdlib.h>

#include <math.h>

#include "givens.h"

void givens\_rotation(double \*\*A, double \*b, size\_t n) {

// Преобразование A к верхнетреугольному виду с помощью матриц Гивенса

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

for (size\_t i = n - 1; i > j; i--) {

double a = A[i - 1][j];

double b\_elem = A[i][j];

double r = sqrt(a \* a + b\_elem \* b\_elem);

double c = a / r;

double s = -b\_elem / r;

// Применение вращения к строкам A

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

double temp = c \* A[i - 1][k] - s \* A[i][k];

A[i][k] = s \* A[i - 1][k] + c \* A[i][k];

A[i - 1][k] = temp;

}

// Применение вращения к вектору b

double temp\_b = c \* b[i - 1] - s \* b[i];

b[i] = s \* b[i - 1] + c \* b[i];

b[i - 1] = temp\_b;

}

}

}

void back\_substitution(double \*\*R, double \*b, double \*x, size\_t n) {

// Обратная подстановка для верхнетреугольной матрицы R

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

x[i] = b[i];

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

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

}

x[i] /= R[i][i];

}

}

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

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

double vector\_infinity\_norm(double \*v1, double \*v2, 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) - 0.5;

}

}

// Ортогонализация

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

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

double dot = 0.0;

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

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

}

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

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

}

}

// Нормализация

double norm = 0.0;

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

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

}

norm = sqrt(norm);

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

Q[j][k] /= norm;

}

}

// Копируем результат

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

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

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

}

}

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

// Используем метод Гаусса-Жордана для инверсии матрицы

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 \*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;

}

**1.3 Основная программа**

**main.c**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <math.h>

#include "matrix.h"

#include "givens.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 \*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 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\_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

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

copy\_matrix(A, A\_copy, N);

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

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

b\_copy[i] = b[i];

}

clock\_t start\_time = clock();

givens\_rotation(A\_copy, b\_copy, N);

back\_substitution(A\_copy, b\_copy, x, N);

clock\_t end\_time = clock();

// Вычисление ошибки

double error = vector\_infinity\_norm(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);

free\_matrix(A\_copy, N);

free(b\_copy);

}

// Возмущение вектора правой части

double cond\_number = 1e5;

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

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

// Решение без возмущения

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

copy\_matrix(A, A\_copy, N);

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

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

b\_copy[i] = b[i];

}

givens\_rotation(A\_copy, b\_copy, N);

back\_substitution(A\_copy, b\_copy, x, N);

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

free\_matrix(A\_copy, N);

free(b\_copy);

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);

}

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

copy\_matrix(A, A\_copy, N);

givens\_rotation(A\_copy, b\_perturbed, N);

back\_substitution(A\_copy, b\_perturbed, x, N);

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

double relative\_error = error / error\_base;

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

free\_matrix(A\_copy, N);

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);

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

copy\_matrix(A, A\_copy, N);

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

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

b\_copy[i] = b[i];

}

givens\_rotation(A\_copy, b\_copy, N);

back\_substitution(A\_copy, b\_copy, x, N);

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

double relative\_error = error / error\_base;

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

free\_matrix(A\_copy, N);

free(b\_copy);

}

// Очистка памяти и закрытие файлов

free\_matrix(A, 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()