Отчет по дисциплине: «Численные методы»

Лабораторная работа №4

«**Решение систем с симметричными матрицами методом сопряженных градиентов и используя матрицы преобуславливания**»

Подготовил студент 3 курса 4 группы

Кондратович Артём

Постановка задачи:

Необходимый для выполнения работы теоретический материал имеется в файлах «Предобусловливание», «Лаб\_Вспомогательные материалы».

Цель работы. Изучить основы понятия предобусловливания. Проверить (для рассматриваемого примера), насколько для итерационного процесса эффективны самые простые предобусловливатели – диагональные.

Задание 1. Исследовать влияние диагональных предобусловливателей на скорость сходимости метода сопряженных градиентов и метода Якоби.

Выполнить Случай 1 лабораторной работы 5 и две модификации.

Случай 1 лабораторной работы 5 повторить.

Случай 1Я: ещё раз выполнить случай 1, но перед началом итерационного процесса произвести предобусловливание Якоби.

Случай 1М: ещё раз выполнить случай 1, но перед началом итерационного процесса произвести масштабирование.

Система уравнений во всех трёх случаях изначально одна и та же.

Входные и выходные данные – как в лабораторной работе 5.

Листинг программы:

#include <iostream>

#include <random>

#include <cmath>

#include <chrono>

#include <iomanip>

#include "Operators.h"

const int k = 4;

const double m = 4.;

const int l\_it = 50;

double const epsilon = 0.000001;

double realRand() /\* generate float/double value in range [-1000, 0] \*/

{

static std::random\_device rd;

static std::mt19937 gen(rd());

static std::uniform\_real\_distribution<double> dis(-1000, 0);

return dis(gen);

}

int intRand() /\* generate n value in range [1500, 2000]\*/

{

static std::random\_device rd; // Will be used to obtain a seed for the random number engine

static std::mt19937 gen(rd());

static std::uniform\_int\_distribution<int> dis(1500, 2000);

return dis(gen);

}

std::vector<std::vector<double>> GenerateMatrixA(int n)

{

std::vector<std::vector<double>> A(n, std::vector<double>(n));

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

{

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

{

A[i][j] = realRand();

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

}

}

A[0][0] = std::pow(10, 2 - k);

for (int i = 1; i < n; ++i) {

A[0][0] += -A[0][i];

}

for (int i = 1; i < n; ++i) {

A[i][i] = -A[i][0];

for (int j = 0; j < n; ++j) {

if (i != j)

A[i][i] += -A[i][j];

}

}

return A;

}

std::vector<double> GenerateVectorX(int n)

{

std::vector<double> x(n);

x[0] = m;

for (int i = 1; i < n; i++)

{

x[i] = 1.0 + x[i - 1];

}

return x;

}

std::vector<double> GenerateVectorF(const std::vector<std::vector<double>>& matrix, const std::vector<double>& x)

{

size\_t n = matrix.size();

std::vector<double> f(n);

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

{

double sum = 0.;

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

{

sum += matrix[i][j] \* x[j];

}

f[i] = sum;

}

return f;

}

double RelativeError(const std::vector<double>& x, const std::vector<double>& x1)

{

double normNumerator = 0.;

double normDenominator = 0.;

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

{

normNumerator += std::pow(x[i] - x1[i], 2);

normDenominator += std::pow(x[i], 2);

}

normNumerator = std::sqrt(normNumerator);

normDenominator = std::sqrt(normDenominator);

return normNumerator / normDenominator;

}

void GetMatrixDecomposition(double\*\* matrix, int size)

{

auto t = new double[size];

// forward Gaussian Method

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

{

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

{

t[j] = matrix[j][i];

matrix[j][i] /= matrix[i][i];

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

{

matrix[j][k] -= t[k] \* matrix[j][i];

}

}

}

delete[] t;

}

void CopyVectorToMatrix(const std::vector<std::vector<double>>& vec, double\*\*& matrix, size\_t rows, size\_t cols)

{

matrix = new double\* [rows];

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

{

matrix[i] = new double[cols];

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

{

matrix[i][j] = vec[i][j];

}

}

}

void CopyMatrixToVector(double\*\* matrix, std::vector<std::vector<double>>& vec, size\_t rows, size\_t cols)

{

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

{

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

{

vec[i][j] = matrix[i][j];

}

}

}

double Norm(const std::vector<double>& v)

{

double sum = 0.;

for (auto const& item : v)

{

sum += std::pow(item, 2);

}

return std::sqrt(sum);

}

std::vector<double> GetDiagonal(const std::vector<std::vector<double>>& A)

{

std::vector<double> diagonal;

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

{

diagonal.push\_back(A[i][i]);

}

return diagonal;

}

std::vector<double> GetNormalizedDiagonal(const std::vector<std::vector<double>>& A)

{

std::vector<double> diagonal;

for (auto const& line : A)

{

diagonal.push\_back(Norm(line));

}

return diagonal;

}

std::vector<double> ConjugateGradientMethod(const std::vector<std::vector<double>>& A, const std::vector<double>& f, int& i)

{

std::vector<double> x(A.size(), 0.);

auto r = f;

auto p = r;

while (Norm(r) > epsilon)

{

auto alpha = (r \* r) / ((A \* p) \* p);

x = x + alpha \* p;

auto r\_new = r - alpha \* (A \* p);

auto beta = (r\_new \* r\_new) / (r \* r);

auto p\_new = r\_new + beta \* p;

p = p\_new;

r = r\_new;

i++;

}

return x;

}

std::vector<double> ConjugateGradientMethod(const std::vector<std::vector<double>>& A, const std::vector<double>& f,const std::vector<double>& d, int& i)

{

std::vector<double> x(A.size(), 0.);

auto r = f;

auto p = r / d;

while (Norm(r) > epsilon)

{

auto alpha = ((r / d) \* r) / ((A \* p) \* p);

x = x + alpha \* p;

auto r\_new = r - alpha \* (A \* p);

auto beta = ((r\_new / d) \* r\_new) / ((r / d) \* r);

auto p\_new = r\_new / d + beta \* p;

p = p\_new;

r = r\_new;

i++;

}

return x;

}

void WithoutDiagonal(const std::vector<std::vector<double>>& A, int n)

{

auto x = GenerateVectorX(n);

auto f = GenerateVectorF(A, x);

std::cout << std::endl << "Solve by Conjugate Gradient Method" << std::endl;

auto t1 = std::chrono::steady\_clock::now();

int iterations = 0;

auto solution = ConjugateGradientMethod(A, f, iterations);

auto t2 = std::chrono::steady\_clock::now();

std::cout << "1) iterations = " << iterations << std::endl;

std::cout << "2) x\* = " << solution;

std::cout << "3) norm(f - Ax\*) = " << Norm(f - A \* solution) << std::endl;

std::cout << "4) norm(x - x\*) / norm(x) = " << Norm(x - solution) / Norm(x) << std::endl;

std::cout << "5) t = " << std::chrono::duration<double>(t2 - t1).count() << std::endl;

}

void Diagonal(const std::vector<std::vector<double>>& A, int n, const std::vector<double>& diagonal)

{

auto x = GenerateVectorX(n);

auto f = GenerateVectorF(A, x);

std::cout << std::endl << "Solve by Conjugate Gradient Method with diagonal a[i,i]" << std::endl;

auto t1 = std::chrono::steady\_clock::now();

int iterations = 0;

auto solution = ConjugateGradientMethod(A, f, diagonal, iterations);

auto t2 = std::chrono::steady\_clock::now();

std::cout << "1) iterations = " << iterations << std::endl;

std::cout << "2) x\* = " << solution;

std::cout << "3) norm(f - Ax\*) = " << Norm(f - A \* solution) << std::endl;

std::cout << "4) norm(x - x\*) / norm(x) = " << Norm(x - solution) / Norm(x) << std::endl;

std::cout << "5) t = " << std::chrono::duration<double>(t2 - t1).count() << std::endl;

}

void NormalizedDiagonal(const std::vector<std::vector<double>>& A, int n, const std::vector<double>& normalized)

{

auto x = GenerateVectorX(n);

auto f = GenerateVectorF(A, x);

std::wcout << std::endl << "Solve by Conjugate Gradient Method with diagonal alpha[i] = |A[i]|" << std::endl;

auto t1 = std::chrono::steady\_clock::now();

int iterations = 0;

auto solution = ConjugateGradientMethod(A, f, normalized, iterations);

auto t2 = std::chrono::steady\_clock::now();

std::cout << "1) iterations = " << iterations << std::endl;

std::cout << "2) x\* = " << solution;

std::cout << "3) norm(f - Ax\*) = " << Norm(f - A \* solution) << std::endl;

std::cout << "4) norm(x - x\*) / norm(x) = " << Norm(x - solution) / Norm(x) << std::endl;

std::cout << "5) t = " << std::chrono::duration<double>(t2 - t1).count() << std::endl;

}

int main()

{

auto n = intRand();

auto A = GenerateMatrixA(n);

std::cout << "Generated n = " << n << std::endl;

WithoutDiagonal(A, n);

auto d = GetDiagonal(A);

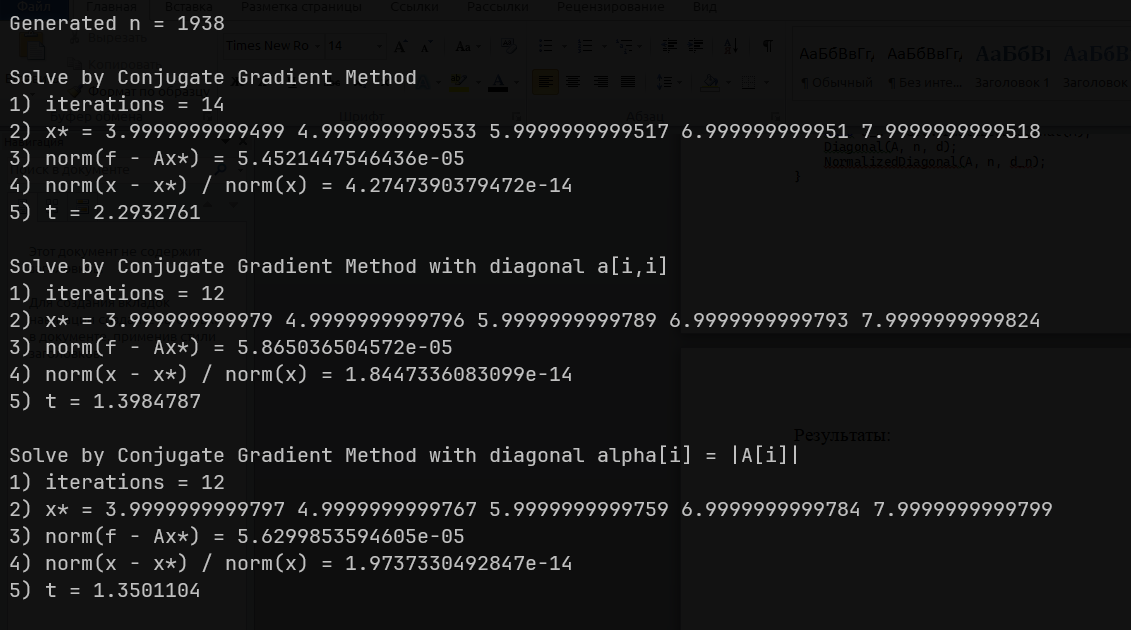
auto d\_n = GetNormalizedDiagonal(A);

Diagonal(A, n, d);

NormalizedDiagonal(A, n, d\_n);

}

Результаты:



Выводы:

Заметим что используя предобусловливание, количество итераций уменьшается, почти в 2 раза быстрее работает алгоритм, и погрешность в 3 раза меньше, так что можно сказать, что преобусловливание улучшает метод сопряжённых градиентов.