Отчет по дисциплине: «Численные методы»

Лабораторная работа №4

«**Решение систем с симметричными матрицами методом сопряженных градиентов**»

Подготовил студент 3 курса 4 группы

Кондратович Артём

Задание. Разработать программу численного решения СЛАУ методом сопряженных градиентов. Результаты вычислительных экспериментов сравнить с аналогичными результатами, полученными с помощью разработанной ранее программы численного решения СЛАУ на основе LDLT-разложения (лабораторная работа 2).

Пусть матрица (порядка *n*) системы *Ax=f* симметричная: *ai,j=aj,i*. СЛАУ формируется как в лабораторной работе 2; но теперь правая часть обозначается не *b*, а *f*.

Для вычислений выбрать параметры:

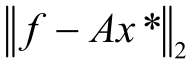
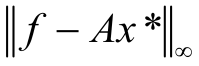
* *n* – одно из чисел в пределах от 1500до 2000;
* *m* – номер в списке студенческой группы;
* *k* – номер студенческой группы.

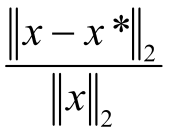
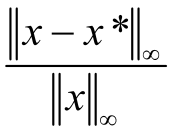
Случай 1. Программно реализовать псевдокод алгоритма CG для рассматриваемого примера (в качестве языка программирования выбрать C или C++). Взять *x*0*=*0; *lit=*50.

Случай 2. Получить приближение *x\** к точному решению с помощью разработанной ранее программы численного решения СЛАУ на основе LDLT-разложения (лабораторная работа 2).

Для вычислений использовать тип double. Система уравнений в обоих случаях одна и та же. Для обоих случаев в выходных данных отчета должны быть представлены:

1. Первые 5 координат вектора приближённого решения *x\**.

2. Норма вектора невязки  (или ).

3. Относительная погрешность  (или ), где https://lh7-us.googleusercontent.com/YprGKiRxB2kR6RkQKLy_Ew_TgAdRGBCpJ50JH5y5ZRJQSTSiKxfoeXu85fSYOWmRn96Y6w3jyJgwB5C1e60AC9v2tnV6mlfvGlWH7ZQWvyfde8aTTab2n-2M1QlyFDdyFz0JTEGVOjsZ9-dUHtFLhg – точное решение (*m*, *m*+1, ... , *n*+*m*–1).

4. Время выполнения (можно приблизительно).

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;

const double bound = std::pow(10, -14);

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];

}

}

}

void TransposeMatrix(std::vector<std::vector<double>>& matrix)

{

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

{

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

{

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

}

}

}

std::vector<double> SolveLyb(const std::vector<std::vector<double>>& L, const std::vector<double>& b)

{

std::vector<double> y(L.size());

for (int i = 0; i < L.size(); i++)

{

double sum = 0.;

for (int j = 0; j < i; j++)

{

sum += y[j] \* L[i][j];

}

y[i] = b[i] - sum;

}

return y;

}

std::vector<double> SolveDzy(const std::vector<std::vector<double>>& D, const std::vector<double>& y)

{

std::vector<double> z(D.size());

for (int i = 0; i < D.size(); i++)

{

z[i] = y[i] / D[i][i];

}

return z;

}

std::vector<double> SolveLtxz(const std::vector<std::vector<double>>& Lt, const std::vector<double>& z)

{

std::vector<double> solution(Lt.size());

solution[Lt.size()- 1] = z[Lt.size() - 1];

for (int i = Lt.size() - 2; i >= 0; i--)

{

double sum = 0.;

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

{

sum += Lt[i][j] \* solution[j];

}

solution[i] = z[i] - sum;

}

return solution;

}

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> 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 (i < l\_it && r \* r > bound)

{

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

auto x\_new = 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;

x = x\_new;

p = p\_new;

r = r\_new;

i++;

}

return x;

}

void SolveByMatrixDecomposition(std::vector<std::vector<double>> A, int n)

{

std::cout << std::endl << "Solve by Matrix Decomposition" << std::endl;

double\*\* matrix;

CopyVectorToMatrix(A, matrix, n, n);

auto x = GenerateVectorX(n);

auto f = GenerateVectorF(A, x);

auto t3 = std::chrono::steady\_clock::now();

GetMatrixDecomposition(matrix, n);

CopyMatrixToVector(matrix, A, n, n);

auto y = SolveLyb(A, f);

auto z = SolveDzy(A, y);

TransposeMatrix(A);

auto x1 = SolveLtxz(A, z);

auto t4 = std::chrono::steady\_clock::now();

std::cout << "Solution = " << x1;

std::cout << "Time = " << std::chrono::duration<double>(t4 - t3).count() << std::endl;

}

void SolveByConjugateGradientMethod(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 << "Iterations = " << iterations << std::endl;

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

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

}

int main()

{

auto n = intRand();

auto A = GenerateMatrixA(n);

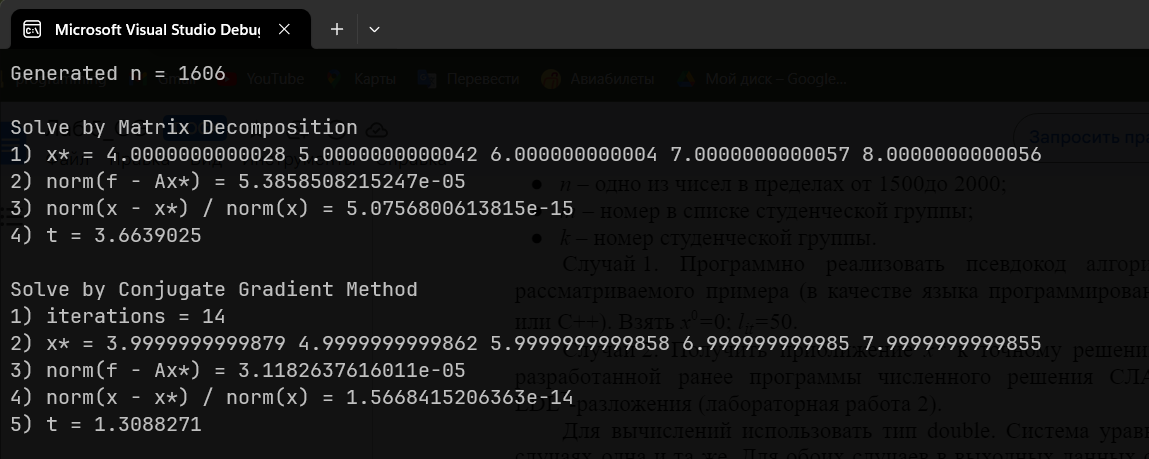
std::cout << "Generated n = " << n << std::endl;

SolveByMatrixDecomposition(A, n);

SolveByConjugateGradientMethod(A, n);

}

Результаты:



Выводы:

По результатам выполнения мы видим значительный прирост скорости нахождения решения СЛАУ с симметричной матрицей (в среднем быстрее ≈ 2 раза).