Отчет по дисциплине: «Численные методы»

Лабораторная работа №2

«Решение систем на основе разложения симметричных матриц»

Подготовил студент 3 курса 4 группы

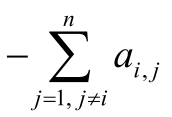
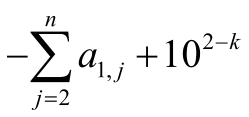
Кондратович Артём

Постановка задачи:

Программно реализовать (в качестве языка программирования выбрать C или C++) вычисления для рассматриваемого примера. Использовать алгоритм (5) файла «LDLt\_RtR разложения», требующий хранения только нижнего треугольника матрицы. В процессе факторизации матрицы *A* (*A=LDLT*) нижняя треугольная матрица *L* (за исключением единиц на главной диагонали) хранится на месте нижнего треугольника матрицы *A,* диагональная матрица *D* хранится на месте главной диагонали матрицы *A*.

Входные данные:

Матрицу (порядка *n*) системы сформировать следующим образом:

* недиагональные элементы *ai,j*, *i<j*, выбираются случайным образом из диапазона от 0 до *–*1000; если *i>j*, то полагается *ai,j*=*aj,i*.
* *ai,i=*,  2≤*i*≤*n*;  *a*11*=*.

Правую часть *b* задать умножением матрицы *A* на вектор *x=*(*m*, *m*+1, ... , *n*+*m*–1):  *b=Ax*. Для вычислений выбрать параметры:

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

Реализация

Листинг всей программы:

#include <iostream>

#include <random>

#include <cmath>

#include <chrono>

#include <iomanip>

// m = 4, k = 4

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);

}

double\*\* GenerateMatrixA(int size, int k)

{

auto A = new double \* [size];

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

{

A[i] = new double[size];

}

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

{

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

{

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

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

}

}

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

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

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

}

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

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

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

if (i != j)

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

}

}

return A;

}

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 TransposeMatrix(double\*\* matrix, int size)

{

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

{

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

{

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

}

}

}

double\* SolveLyb(double\*\* L, const double\* b, int size)

{

auto y = new double[size];

for (int i = 0; i < 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;

}

double\* SolveDzy(double\*\* D, const double\* y, int size)

{

auto z = new double[size];

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

{

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

}

return z;

}

double\* SolveLtxz(double\*\* Lt, const double\* z, int n)

{

auto solution = new double[n];

solution[n - 1] = z[n - 1];

for (int i = n - 2; i >= 0; i--)

{

double sum = 0.;

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

{

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

}

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

}

return solution;

}

double\* backwardStroke(double\*\* A, const double\* b, int n)

{

auto solution = new double[n];

solution[n - 1] = b[n - 1] / A[n - 1][n - 1];

for (int i = n - 2; i >= 0; i--)

{

double sum = 0;

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

{

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

}

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

}

return solution;

}

double\* GaussianMethodWithoutElement(double\*\* A, double\* b, int n)

{

//forward

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

{

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

{

double c = A[j][i] / A[i][i];

A[j][i] = 0;

b[j] -= b[i] \* c;

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

{

A[j][k] -= A[i][k] \* c;

}

}

}

return backwardStroke(A, b, n);

}

double\* GenerateVectorX(int size, double m)

{

auto vector = new double[size];

vector[0] = m;

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

{

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

}

return vector;

}

double\* GenerateVectorB(double\*\* matrix, const double\* vector, int size)

{

auto b = new double[size];

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

{

double sum = 0.;

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

{

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

}

b[i] = sum;

}

return b;

}

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

{

std::cout << std::endl;

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

{

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

{

std::cout << matrix[i][j] << " ";

}

std::cout << std::endl;

}

}

double RelativeError(const double\* x,const double\* x1, int size)

{

double normNumerator = 0.;

double normDenominator = 0.;

for (size\_t i = 0; i < 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;

}

double\*\* makeCopyMatrix(double\*\* A, int n)

{

auto m = new double\* [n];

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

{

m[i] = new double[n];

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

{

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

}

}

return m;

}

void PrintReport(const double\* x, const double\* x1, const double\* x11, double t1, double t2, int size)

{

std::cout << std::setprecision(15);

std::cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Solution based on symmetric matrix decomposition\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << std::endl;

std::cout << "1) Generated n = " << size << " m = " << 4 << " k = " << 4 << std::endl;

std::cout << "2) Accurate solution x = ";

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

std::cout << x[i] << " ";

std::cout << std::endl;

std::cout << "3) Approximate solution x\* = ";

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

std::cout << x1[i] << " ";

std::cout << std::endl;

std::cout << "4) Relative error = " << RelativeError(x, x1, size) << std::endl;

std::cout << "5) Time = " << t1 << " s" << std::endl << std::endl;

std::cout << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Solution based on the Gaussian method\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << std::endl;

std::cout << "1) Generated n = " << size << " m = " << 4 << " k = " << 4 << std::endl;

std::cout << "2) Accurate solution x = ";

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

std::cout << x[i] << " ";

std::cout << std::endl;

std::cout << "3) Approximate solution x\* = ";

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

std::cout << x11[i] << " ";

std::cout << std::endl;

std::cout << "4) Relative error = " << RelativeError(x, x11, size) << std::endl;

std::cout << "5) Time = " << t2 << " s" << std::endl << std::endl;

}

void solution()

{

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

int n = intRand();

double m = 4.;

int k = 4;

auto A = GenerateMatrixA(n, k);

auto Ac = makeCopyMatrix(A, n);

auto x = GenerateVectorX(n, m);

auto b = GenerateVectorB(A, x, n);

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

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

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

GetMatrixDecomposition(A, n);

auto y = SolveLyb(A, b,n);

auto z = SolveDzy(A, y,n);

TransposeMatrix(A, n);

auto x1 = SolveLtxz(A, z, n);

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

auto t5 = std::chrono::steady\_clock::now();

auto x11 = GaussianMethodWithoutElement(Ac, b, n);

auto t6 = std::chrono::steady\_clock::now();

PrintReport(x, x1,x11, std::chrono::duration<double>(t4 - t3).count(), std::chrono::duration<double>(t6 - t5).count(), n);

}

int main()

{

solution();

}

Пояснения

Для создания матрицы A:

double\*\* GenerateMatrixA(int size, int k)

{

auto A = new double \* [size];

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

{

A[i] = new double[size];

}

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

{

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

{

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

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

}

}

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

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

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

}

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

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

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

if (i != j)

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

}

}

return A;

}

Генерация случайного n [1500, 2000]:

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);

}

Подсчёт вектора b

double\* GenerateVectorB(double\*\* matrix, const double\* vector, int size)

{

auto b = new double[size];

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

{

double sum = 0.;

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

{

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

}

b[i] = sum;

}

return b;

}

Инициализация вектора x = (4, 5, 6, ...)

double\* GenerateVectorX(int size, double m)

{

auto vector = new double[size];

vector[0] = m;

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

{

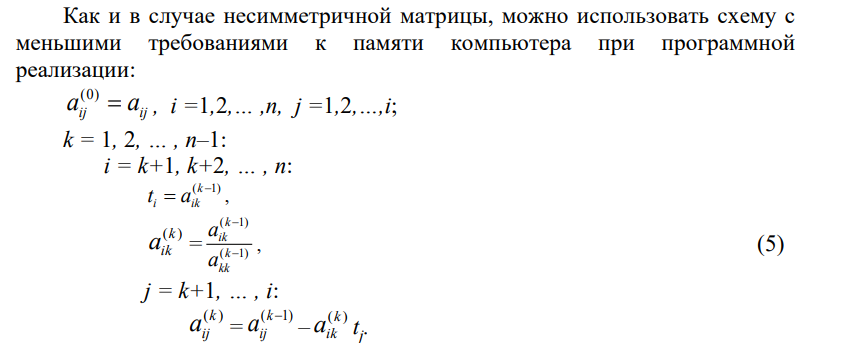
vector[i] = 1.0 + vector[i - 1];

}

return vector;

}

Реализация алгоритма разложение матрицы А:



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] ;

}

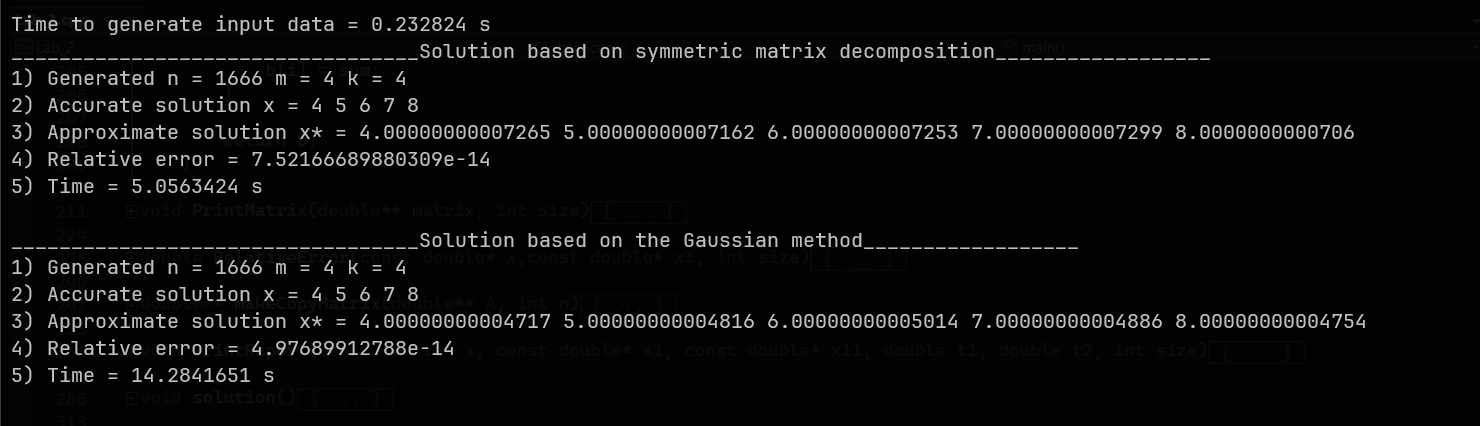
}

}

delete[] t;

}

Отчет по данным:



Выводы:

Как мы можем заметить решение разложением матрицы на A = LtDL значительно быстрее, чем стандартный метод Гаусса, но мы видим, что погрешность решения выше, это связано с тем что при решении уравнений

Ly=b, Dz=y, Ltx=z увеличивает погрешность.