Министерство образования и науки Российской Федерации

НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Кафедра прикладной математики

Лабораторная работа №1

по дисциплине

«Численные методы решения систем уравнений»

Факультет ПМИ

Группа ПМ-85

Студенты Жукова Д.И.

Пучинина А.С

Преподаватель Патрушев И.И.

Вариант 12

Новосибирск

2020

1. Цель

Разработать программу решения СЛАУ методом с LU\*-разложения и хранением матрицы в профильном формате. Исследовать накопление погрешности и ее зависимость от числа обусловленности. Сравнить реализованный метод по точности получаемого решения и количеству действий с методом Гаусса.

1. Анализ

Пусть необходимо решить СЛАУ вида:

 (1.1)

Тогда матрица А может быть представлена в виде:

 (1.2)

Подставляя (1.2) в (1.1), получаем:

 (1.3)

Обозначим:

 (1.4)

тогда подставляя (1.4) в (1.3), получим:

 (1.5)

Таким образом, решение системы (1.1) сводится к трем основным этапам:

1) из элементов матрицы  найти элементы матриц  и \*;

2) решить систему (1.5) с нижнетреугольной матрицей  (прямой ход);

3) решить систему (1.4) с верхнетреугольной матрицей \* (обратный ход).

Рассмотрим алгоритм получения \* – разложения. Матрицы  и \* будем искать в следующем виде:

, (1.6)

Учитывая равенство (1.2) и умножая последовательно строки матрицы  на столбцы матрицы , получаем систему, состоящую из  уравнений с  неизвестными  и  ( – размерность СЛАУ)

Решая систему данную систему, можно получить общие формулы для нахождения элементов матриц  и :

,  (1.8)

1. Тексты программ

Matrix.h

#pragma once

#include <stdlib.h>

#include <stdio.h>

#include <conio.h>

#include <math.h>

#include <vector>

#include <fstream>

#include <locale>

#include <iostream>

using namespace std;

class MatrixProf;

typedef double real;

typedef double dubl;

class Matrix

{

private:

int n;

int col;

vector< vector<real> > matrix;

vector<real> F;

public:

Matrix(int x, ifstream &vect);

Matrix(int x);

Matrix(void);

~Matrix(void);

void getCol();

void setMatrix(vector< vector<real> > A, int x, vector<real> B);

void ToProf(MatrixProf \*A);

vector<real> Gauss();

vector<real> GoodGauss();

void Gilbert();

void GilbertVect();

void Plus(real x);

};

class MatrixProf

{

private:

int n;

int col;

vector<real> di;

vector<int> ia;

vector<real> al;

vector<real> au;

vector<real> F;

public:

MatrixProf(void);

MatrixProf(int x, int c);

~MatrixProf(void);

void load (ifstream &matrix, ifstream &vect);

void save (ofstream &output);

void LUDec();

void Direct();

void Reverse();

void Plus(real x);

void ToTight(Matrix \*A);

vector<real> SLAU();

void setProf(int bn, int bcol, vector<real> bdi, vector<int> bia, vector<real> bal, vector<real> bau, vector<real> bF);

};

Matrix.cpp

#include "Matrix.h"

Matrix::Matrix(int x, ifstream &vect)

{

this->n = x;

col = 0;

vector< vector< real > > B(n);

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

{

vector<real> buf(n);

B[i] = buf;

}

matrix = B;

F.resize(n);

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

{

vect >> F[i];

}

}

Matrix::Matrix(int x)

{

this->n = x;

col = 0;

vector< vector< real > > B(n);

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

{

vector<real> buf(n);

B[i] = buf;

}

matrix = B;

F.resize(n);

}

void Matrix::setMatrix(vector< vector<real> > A, int x, vector<real> B)

{

n = x;

matrix = A;

F = B;

}

Matrix::Matrix(void)

{

}

Matrix::~Matrix(void)

{

}

vector<real> Matrix::Gauss() // метод гауса обычный

{

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

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

{

real m = matrix[j][i-1]/matrix[i-1][i-1];

for (int k = 0; k < n; k++)

matrix[j][k]=matrix[j][k]-m\*matrix[i-1][k];

F[j] = F[j] - m \* F[i-1];

}

for (int k = n-1; k >= 0; k--)

{

dubl buf = 0;

for (int j = k+1; j < n; j++)

{

buf += matrix[k][j]\*F[j];

}

F[k] = F[k] - buf;

F[k] = F[k]/matrix[k][k];

}

return F;

}

vector<real> Matrix::GoodGauss() // метод гаусса с ведущим элементом не работает

{

real b = 0;

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

{

dubl max = matrix[i][i];

int k = i;

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

{

if (matrix[c][i] > max)

{

max = matrix[c][i];

k = c;

}

}

matrix[i].swap(matrix[k]);

b = F[i];

F[i] = F[k];

F[k] = b;

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

{

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

for (int k = 0; k < n; k++)

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

F[j] = F[j] - m \* F[i];

}

}

for (int k = n-1; k >= 0; k--)

{

dubl buf = 0;

for (int j = k+1; j < n; j++)

{

buf += matrix[k][j]\*F[j];

}

F[k] = F[k] - buf;

F[k] = F[k]/matrix[k][k];

}

return F;

}

void Matrix::Gilbert() // построение матрицы гильберта

{

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

{

F[i] = i + 1;

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

matrix[i][j] = (double)1.0/(i+j+1);

}

GilbertVect();

}

void Matrix::ToProf(MatrixProf \*A) // перевод матрицы в профильный формат

{

getCol();

vector <real> bdi(n);

vector <int> bia(n+1);

vector <real> bau(col);

vector <real> bal(col);

int s = 0;

int flag;

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

{

bdi[i] = matrix[i][i];

bia[i] = s;

flag = 0;

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

{

if (matrix[i][j] != 0 || matrix[j][i] != 0)

{

flag = 1;

}

if (flag == 1)

{

bau[s] = matrix[i][j];

bal[s] = matrix[j][i];

s++;

}

}

}

bia[n] = s;

A->setProf(n, col, bdi, bia, bal, bau, F);

}

void Matrix::getCol() // подсчет значащих элементов в нижнем и верхнем треугольнике матрицы

{

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

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

{

if (matrix[i][j] != 0 || matrix[j][i] != 0)

col++;

}

}

void Matrix::GilbertVect()

{

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

{

dubl sum = 0;

for (int k = 0 ; k < n; k++)

{

sum += matrix[i][k]\*(k+1);

}

F[i] = sum;

}

}

void Matrix::Plus(real x)

{

matrix[0][0] += x;

F[0] += x;

}

Profmatrix.cpp

#include "Matrix.h"

MatrixProf::MatrixProf(void)

{

n = 0;

col = 0;

}

MatrixProf::MatrixProf(int x, int c)

{

n = x;

col = c;

di.resize(n);

ia.resize(n+1);

al.resize(col);

au.resize(col);

F.resize(n);

}

MatrixProf::~MatrixProf(void)

{

}

void MatrixProf::load(ifstream &matrix, ifstream &vect)

{

for( int i = 0; i < n+1; i++)

{

matrix >> ia[i];

}

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

{

matrix >> di[i];

}

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

{

matrix >> al[i];

}

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

{

matrix >> au[i];

}

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

{

vect >> F[i];

}

}

void MatrixProf::LUDec() // LU разложение

{

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

int i0 = ia[i];

int i1 = ia[i+1];

int j = i - (ia[i+1]-ia[i]);

dubl bdi = 0;

for(int k = ia[i]; k < ia[i+1]; k++,j++)

{

int ki = ia[i];

int kj = ia[j];

int dif = k - ia[i] - ia[j+1] + ia[j];

if(dif < 0)

kj += abs(dif);

else

ki += dif;

dubl bal = 0;

dubl bau = 0;

for(ki; ki<k; ki++,kj++)

{

bal += al[ki]\*au[kj];

bau += au[ki]\*al[kj];

}

al[k] = al[k] - bal;

au[k] = au[k] - bau;

au[k] = au[k] / di[j];

bdi += al[k]\*au[k];

}

di[i] -= bdi;

}

}

void MatrixProf::ToTight(Matrix \*A) // перевод матрицы в плотный вид

{

vector< vector< real > > B(n);

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

{

vector<real> buf(n);

B[i] = buf;

}

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

{

int j = i - (ia[i+1] - ia[i]);

for (int k = ia[i]; k < ia[i+1]; k++, j++)

{

B[i][j] = al[k];

B[j][i] = au[k];

}

B[i][i] = di[i];

}

A->setMatrix(B, n, F);

}

void MatrixProf::Direct() // прямой ход Ly=F

{

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

{

int j = i - (ia[i+1]-ia[i]);

dubl sum = 0;

for (int k = ia[i]; k < ia[i+1]; k++, j++)

{

sum += F[j] \* al[k];

}

F[i] = (F[i] - sum) / di[i];

}

}

void MatrixProf::Reverse() // обратный ход Ux=y

{

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

{

int j = i - (ia[i+1]-ia[i]);

for (int k = ia[i]; k < ia[i+1]; k++, j++)

{

F[j] -= F[i]\*au[k];

}

}

}

vector <real> MatrixProf::SLAU() // функция, которая решает

{

LUDec();

Direct();

Reverse();

return F;

}

void MatrixProf::Plus(real x)

{

di[0] += x;

F[0] += x;

}

void MatrixProf::setProf(int bn, int bcol, vector<real> bdi, vector<int> bia, vector<real> bal, vector<real> bau, vector<real> bF) // сет функция. мне стыдно

{

di.resize(bn);

ia.resize(bn+1);

al.resize(bcol);

au.resize(bcol);

F.resize(bn);

n = bn;

col = bcol;

di = bdi;

ia = bia;

al = bal;

au = bau;

F = bF;

}

Main.cpp

#include "Matrix.h"

void main()

{

setlocale (LC\_CTYPE, "Russian");

ifstream size ("size.txt");

ifstream matr("matrix.txt");

ifstream vect("vector.txt");

ofstream proffile("prof.txt");

ofstream tightfile("tight.txt");

int n, col;

size >> n >> col;

MatrixProf myMat(n, col);

vector<real> X(n);

vector<real> Y(n);

Matrix tightMat(n);

int flag = 0;

try

{

cout << "1 - LU из файла. 2 - гильберт в плотной и в LU. 3" << endl;

cin >> flag;

switch(flag)

{

case 1:

{

real x = 0;

for (int i = 0; i < 20; i++)

{

ifstream matr("matrix.txt");

ifstream vect("vector.txt");

myMat.load(matr,vect);

x = pow(10.0,-i);

//myMat.Plus(x);

myMat.ToTight(&tightMat);

X = myMat.SLAU();

//Y = tightMat.Gauss();

tightfile.precision(7);

proffile.precision(7);

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

{

//tightfile << Y[i] << endl;

proffile << X[i] << endl;

}

tightfile << endl;

proffile << endl;

}

break;

}

case 2:

{

for (int i = 10; i < 11; i++)

{

Matrix tightMat(i);

tightMat.Gilbert();

MatrixProf myMat(1, 1);

tightMat.ToProf(&myMat);

X = myMat.SLAU();

Y = tightMat.Gauss();

tightfile.precision(15);

proffile.precision(15);

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

{

tightfile << Y[j] << endl;

proffile << X[j] << endl;

}

tightfile << endl;

proffile << endl;

}

break;

}

default:

{

cout << "другого не дано" << endl;

break;

}

}

}

catch(int error)

{

switch (error)

{

case 1:

{

cout << "что-то пошло не так";

system("pause");

break;

}

}

}

}

1. Влияние увеличения числа обусловленности на точность решения

Пусть дана вырожденная матрица А:



И вектор FT:



Прибавим к первому элементу матрицы и вектора 10-k, тогда мы получим невырожденную матрицу, которая будет иметь «правильное» решение. Постепенно будем увеличивать значение k, для увеличения числа обусловленности. Проведем исследования с разными типами данных (таблицы см. далее).

Выводы: С увеличением k падает точность вычислений. Это связано с тем, что при этом уменьшается число обусловленностей. Также при k=7, для 4-байтового вещественного типа, и при k=15 для 8-байтового, СЛАУ не решается, ибо в этом случае A + 10-k = A. Это связано со свойствами машинной арифметики и размером мантиссы. Использование разных типов данных, для всех переменных и для переменных накопителей не дает практически никакого выигрыша в точности на данной матрице. Использование чисел двойной точности дает нам значительное уменьшение погрешности вычислений, но при этом мы получаем удвоенные затраты по памяти.

1. Исследование на матрице Гилберта различной размерности:

Пусть дана матрица Гилберта. Она строится по формуле:



Данная матрица имеет большое число обусловленностей. При увеличении размера этой матрицы увеличивается погрешность решения. Также при k=7, для 4-байтового вещественного типа, и при k=15 для 8-байтового, решение становится неправильным. Таблицы приведены далее.













 

1. Сравнение реализованного метода с методом Гаусса

По сложности алгоритма:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Метод |  |  | прямой ход | обратный ход | всего |
|  |  |  |  |  |  |
| Метод Гаусса | - | - |  |  |  |
| Метод Гаусса с ведущим главным элементом | - | - |  |  |  |

По значению погрешности:



Вывод:

Метод Гаусса с ведущим главным элементом имеет небольшое уменьшение погрешности. При этом имеет бо’льшие вычислительные затраты.

1. Дополнительное задание разложение матрицы LDLT

Сложность решения СЛАУ данным методом: 