Федеральное государственное бюджетное образовательное учреждение высшего образования

«Сибирский государственный университет телекоммуникаций и информатики»

(СибГУТИ)

Факультет информатики и вычислительной техники

09.03.01 "Информатика и вычислительная техника"

профиль "Программное обеспечение средств вычислительной техники и автоматизированных систем"

Кафедра прикладной математики и кибернетики

**Лабораторная работа №2 по дисциплине  
 Алгоритмы и вычислительные методы оптимизации**

Выполнил:

студент гр. ИП-712 Алексеев Степан Владимирович

ФИО студента

«\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_ 2020 г.

Проверил

Галкина Марина Юрьевна

ФИО преподавателя

«\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_ 2020 г. Оценка\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Новосибирск 2020 г.

Оглавление

[Задание: 2](#_Toc38483869)

[Source.cpp 2](#_Toc38483870)

[Fractions.h 7](#_Toc38483871)

[Fractions.cpp 8](#_Toc38483872)

[Результаты тестирования 11](#_Toc38483873)

Задание:

Занятие 4 (лабораторная работа №2) Нахождение базисных и опорных решений систем линейных уравнений Написать программу, находящую все базисные и опорные решения системы линейных уравнений методом Жордана-Гаусса. Программа должна выводить промежуточные матрицы после каждого шага исключений и все базисные и опорные решения. Все вычисления должны выполняться в простых дробях, для этого использовать класс простых дробей, реализованный в лабораторной. Например номер 1:

4 -11 13 -6 8 8

7 12 5 -3 9 54

-6 9 -17 13 -7 -16

-17 -7 -30 30 -14 -86

# Source.cpp

#include "fractions.h"

#include "stdc++.h"

using namespace std;

long long fact(long long n)

{

if (n <= 1)

return 1;

return n \* fact(n - 1);

}

long long combinationNum(long long a, long long b)

{

if (a == b)

return 1;

return fact(b) / (fact(b - a) \* fact(a));

}

bool NextSet(int\* a, int n, int m)

{

int k = m;

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

if (a[i] < n - k + i + 1)

{

++a[i];

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

a[j] = a[j - 1] + 1;

return true;

}

return false;

}

void read(Fraction\*\* arr, int n, int m)

{

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

{

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

{

cin >> arr[i][j];

}

}

}

void print(Fraction\*\* arr, int n, int m)

{

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

{

for (int j = 0; j < n - 1; j++)

{

cout << setw(7) << arr[i][j] << " ";

}

cout << "| " << arr[i][n - 1] << endl;

}

cout << endl;

}

void swap(Fraction\*\* arr, int n, int from, int to)

{

Fraction c;

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

{

c = arr[from][i];

arr[from][i] = arr[to][i];

arr[to][i] = c;

}

}

void copy(Fraction\*\* from, Fraction\*\* to, int n, int m)

{

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

{

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

to[i][j] = from[i][j];

}

}

void JordanGauss(Fraction\*\* arr, int n, int m)

{

int i, j, y = 0, mx\_ind = 0, zz = 0;

bool check;

Fraction mx;

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

{

if (y > m - 1 || i == n - 1)

break;

check = false;

mx\_ind = y;

mx = Fraction(0, 1);

for (j = y; j < m; j++)

{

if (Fraction::Abs(arr[j][i]) > mx)

{

mx = Fraction::Abs(arr[j][i]);

mx\_ind = j;

check = true;

}

}

swap(arr, n, y, mx\_ind);

if (!check)

continue;

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

{

for (int l = 0; l < m; l++)

{

if (l != y)

{

arr[l][j] = arr[l][j] - (arr[y][j] \* arr[l][i]) / arr[y][i];

}

}

}

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

{

arr[y][j] = arr[y][j] / arr[y][i];

}

arr[y][i] = arr[y][i] / arr[y][i];

for (j = 0; j < m; j++)

{

if (j != y)

arr[j][i] = Fraction(0, 1);

}

y++;

}

int rang = 0;

int cnt = 0;

for (i = 0; i < m; i++)

{

cnt = 0;

for (j = 0; j < n - 1; j++)

{

if (arr[i][j] == Fraction(1, 1))

cnt++;

}

if (cnt == 1)

rang++;

}

m = rang;

print(arr, n, m);

int combNum = combinationNum(rang, n - 1);

int comb\_ind = rang;

int\* tmp\_comb = new int[rang];

int\* comb = new int[rang \* combNum];

for (i = 0; i < rang; i++)

{

tmp\_comb[i] = i + 1;

comb[i] = i + 1;

}

while (NextSet(tmp\_comb, n - 1, rang))

{

for (i = 0; i < rang; i++)

{

comb[comb\_ind] = tmp\_comb[i];

comb\_ind++;

}

}

cout << "Rang :" << rang << endl << "N: " << n - 1 << endl << "Number of combinations: " << combNum << endl;

Fraction\*\* temp = new Fraction \* [m];

Fraction result;

bool flag1, flag2, flagO;

int counter = 0;

// int \*ch0 = new int[rang];

int\*\* ed = new int\* [n];

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

ed[i] = new int[n]; // добавил из инета

}

//int \*\*ed = new int[rang][n];

int\* ch0 = new int[rang];

for (i = 0; i < m; i++)

temp[i] = new Fraction[n];

copy(arr, temp, n, m);

print(temp, n, m);

cout << endl;

for (i = 0; i < rang \* combNum; i++) cout << comb[i] << " ";

cout << endl;

for (i = 0; i < rang; i++) ch0[i] = 0;

for (int k = 0; k < rang \* combNum; k++)

{

int index = comb[k];

flag1 = 0;

flag2 = 0;

flagO = 1;

counter = 0;

for (i = 0; i < rang; i++)

{

//cout << temp[i][index-1] <<"+++"<< ch0[i] << endl;

if (!(temp[i][index - 1] == Fraction(0, 1)) && ch0[i] == 0)

{

// cout << "\_\_" << i << "-" << index-1 << " " << endl;

ch0[i] = 1;

flag1 = 1;

for (j = 0; j < rang; j++)

{

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

{

if (i == j || index - 1 == l)

continue;

temp[j][l] = temp[j][l] - (temp[j][index - 1] \* temp[i][l]) / temp[i][index - 1];

}

}

for (j = 0; j < rang; j++)

{

if (j != i)

temp[j][index - 1] = Fraction(0, 1);

}

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

{

if (j != index - 1)

temp[i][j] = temp[i][j] / temp[i][index - 1];

}

temp[i][index - 1] = Fraction(1, 1);

ed[i][index - 1] = 1;

counter++;

cout << endl;

print(temp, n, m);

}

if (flag1) break;

if (i == rang - 1 && counter != rang) flag2 = 1;

}

if ((k + 1) % rang == 0)

{

for (i = 0; i < rang; i++) ch0[i] = 0;

cout << endl << endl << "Basic Resolving #" << ((k - 1) / rang) + 1 << " ";

cout << "[ ";

for (i = 0; i < rang; i++) cout << comb[k - rang + i + 1] << ", ";

cout << "]";

cout << endl;

if (flag2 == 1) {

cout << "No basic resolve";

copy(arr, temp, n, m);

cout << endl << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << endl;

continue;

}

for (j = 0; j < n - 1; j++) {

result = Fraction(0, 1);

for (i = 0; i < rang; i++) {

if (ed[i][j] == 1) result = temp[i][n - 1];

if (Fraction(0, 1) > result) flagO = 0;

}

if (!(result == Fraction(1, 1))) cout << setw(3) << result << "; ";

else cout << 0 << " ";

}

if (!flagO) cout << "NE ";

cout << "opornoe!";

cout << endl << "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_" << endl;

for (i = 0; i < rang; i++) {

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

ed[i][j] = 0;

}

}

copy(arr, temp, n, m);

}

}

}

int main()

{

int n, m;

cin >> n >> m;

Fraction\*\* arr = new Fraction \* [m];

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

{

arr[i] = new Fraction[n];

}

read(arr, n, m);

print(arr, n, m);

JordanGauss(arr, n, m);

return 0;

}

# Fractions.h

#pragma once

#ifndef FRACTIONS\_H

#define FRACTIONS\_H

#include <iostream>

class Fraction

{

private:

long long numerator;

long long denominator;

public:

Fraction() :numerator(0), denominator(1) {};

Fraction(long long num, long long denom) :numerator(num), denominator(denom) {};

Fraction(long long num) :numerator(num), denominator(1) {};

static long long gcd(long long a, long long b);

static long long lcm(long long a, long long b);

long long getNumerator();

long long getDenominator();

void setNumerator(long long num);

void setDenominator(long long denom);

friend std::istream& operator>>(std::istream& input, Fraction& data);

friend std::ostream& operator<<(std::ostream& output, Fraction& data);

Fraction operator+(Fraction other);

Fraction operator-(Fraction other);

Fraction operator\*(Fraction other);

Fraction operator/(Fraction other);

bool operator==(Fraction other);

bool operator>(Fraction other);

operator double() { return ((double)numerator / denominator); };

static Fraction Abs(Fraction fr);

};

#endif

# Fractions.cpp

#include "fractions.h"

//#include <string>

#include "stdc++.h"

long long Fraction::gcd(long long a, long long b)

{

return b ? gcd(b, a % b) : a;

}

long long Fraction::lcm(long long a, long long b)

{

return std::abs(a \* b) / gcd(a, b);

}

std::istream& operator>>(std::istream& input, Fraction& data)

{

std::string str, str1;

bool check = false;

long long num = 0, denom = 1;

input >> str;

for (unsigned long long i = 0; i < str.length(); i++)

{

if (str[i] == '/')

{

str1 = str.substr(0, i);

num = std::stoi(str1);

str1 = str.substr(i + 1, str.length() - 1 - i);

denom = std::stoi(str1);

check = true;

break;

}

}

data.setNumerator(num);

data.setDenominator(denom);

if (!check)

data.setNumerator(std::stoi(str));

return input;

}

std::ostream& operator<<(std::ostream& output, Fraction& data)

{

if (data.getDenominator() == 1)

output << data.getNumerator();

else

output << data.getNumerator() << "/" << data.getDenominator();

return output;

}

long long Fraction::getNumerator()

{

return numerator;

}

long long Fraction::getDenominator()

{

return denominator;

}

void Fraction::setNumerator(long long num)

{

numerator = num;

}

void Fraction::setDenominator(long long denom)

{

denominator = denom;

}

Fraction Fraction::operator+(Fraction other)

{

long long g = Fraction::lcm(this->getDenominator(), other.getDenominator());

Fraction result((g / this->getDenominator()) \* this->getNumerator() + (g / other.getDenominator()) \* other.getNumerator(), g);

long long gc = Fraction::gcd(result.getNumerator(), result.getDenominator());

if (gc < 0)

gc \*= -1;

result.setNumerator(result.getNumerator() / gc);

result.setDenominator(result.getDenominator() / gc);

return result;

}

Fraction Fraction::operator-(Fraction other)

{

long long g = Fraction::lcm(this->getDenominator(), other.getDenominator());

Fraction result((g / this->getDenominator()) \* this->getNumerator() - (g / other.getDenominator()) \* other.getNumerator(), g);

long long gc = Fraction::gcd(result.getNumerator(), result.getDenominator());

if (gc < 0)

gc \*= -1;

result.setNumerator(result.getNumerator() / gc);

result.setDenominator(result.getDenominator() / gc);

return result;

}

Fraction Fraction::operator\*(Fraction other)

{

long long num = this->getNumerator() \* other.getNumerator();

long long denom = this->getDenominator() \* other.getDenominator();

long long g = Fraction::gcd(num, denom);

Fraction result(num / g, denom / g);

if (result.getDenominator() < 0)

{

result.setNumerator(result.getNumerator() \* -1);

result.setDenominator(result.getDenominator() \* -1);

}

return result;

}

Fraction Fraction::operator/(Fraction other)

{

long long num = this->getNumerator() \* other.getDenominator();

long long denom = this->getDenominator() \* other.getNumerator();

long long g = Fraction::gcd(num, denom);

Fraction result(num / g, denom / g);

if (result.getDenominator() < 0)

{

result.setNumerator(result.getNumerator() \* -1);

result.setDenominator(result.getDenominator() \* -1);

}

return result;

}

bool Fraction::operator==(Fraction other)

{

if (this->getNumerator() \* other.getDenominator() == this->getDenominator() \* other.getNumerator())

return true;

return false;

}

bool Fraction::operator>(Fraction other)

{

if (this->getNumerator() \* other.getDenominator() > this->getDenominator()\* other.getNumerator())

return true;

return false;

}

Fraction Fraction::Abs(Fraction fr)

{

Fraction c = fr;

if (c.getNumerator() < 0)

c.setNumerator(c.getNumerator() \* -1);

return c;

}

# Результаты тестирования

