# Задание

1. Операции над матрицами

а) Сортировка одномерного массива (реализовать 2 алгоритма сортировки и исследовать возможности их оптимизации).

б) Вычисление детерминанта, поиск максимума по строкам, минимума по столбцам.

в) Нахождение обратной матрицы, выполнение умножения матриц А и В.

Листинг кода

#include "Header.hpp"

using namespace std;

int main(){

setlocale(LC\_ALL, "Russian");

cout << "\tМеню:" << endl;

cout << "\t0 Минимальный и максимальный элементы матрицы" << endl;

cout << "\t1 Умножение матриц" << endl;

cout << "\t2 Определитель квадратной матрицы" << endl;

cout << "\t3 Обратная матрица к квадратной матрице" << endl;

cout << "\t4 Алгоритм быстрой сортировки" << endl;

cout << "\t5 Алгоритм битонической сортировки" << endl;

int menu; cin >> menu;

switch(menu){

case 0:

{

// Минимальный и максимальный элементы матрицы

int size1;

int size2;

cout << "\tРазмеры матрицы [2]:" << endl; cin >> size1; cin >> size2;

double \*\*a = new double\*[size1];

for(int i = 0; i < size1; i++)

a[i] = new double[size2];

cout << "\tМатрица A["<<size1<<","<<size2<<"]:" << endl;

for(int i = 0; i < size1; i++)

for(int j = 0; j < size2; j++)

cin >> a[i][j];

clock\_t t = clock();

double min, max;

findMinMax(a, size1, size2, min, max);

t = clock()-t;

double seconds = ((double)t)/CLOCKS\_PER\_SEC;

cout << "\tВремя выполнения: " << seconds << "sec" << endl;

cout << "Минимальный элемент матрицы: " << min << endl

<< "Максимальный элемент матрицы: " << max << endl;

for(int i = 0; i < size1; i++) delete a[i];

delete a;

}

break;

case 1:

{

// Умножение матриц

int size1;

int size2;

int size3;

cout << "\tРазмеры матриц [3]:" << endl; cin >> size1 >> size2 >> size3;

double \*\*a = new double\*[size1];

double \*\*b = new double\*[size2];

for(int i = 0; i < size1; i++)

a[i] = new double[size2];

for(int i = 0; i < size2; i++)

b[i] = new double[size3];

cout << "\tМатрица A["<<size1<<","<<size2<<"]:" << endl;

for(int i = 0; i < size1; i++)

for(int j = 0; j < size2; j++)

cin >> a[i][j];

cout << "\tМатрица B["<<size2<<","<<size3<<"]:" << endl;

for(int i = 0; i < size2; i++)

for(int j = 0; j < size3; j++)

cin >> b[i][j];

clock\_t t = clock();

double \*\*c = mult(a, b, size1, size2, size3);

t = clock()-t;

double seconds = ((double)t)/CLOCKS\_PER\_SEC;

cout << "\tВремя выполнения: " << seconds << "sec" << endl;

cout << "\tРезультат произведения" << endl;

for(int i = 0; i < size1; i++)

{

for(int j = 0; j < size3; j++)

cout << c[i][j] << '\t';

cout << endl;

}

for(int i = 0; i < size1; i++) delete a[i];

for(int i = 0; i < size2; i++) delete b[i];

for(int i = 0; i < size1; i++) delete c[i];

delete a;

delete b;

delete c;

}

break;

case 2:

{

// Определитель квадратной матрицы

int size;

cout << "\tРазмеры матрицы:" << endl; cin >> size;

double \*\*a = new double\*[size];

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

a[i] = new double[size];

cout << "\tМатрица A["<<size<<","<<size<<"]:" << endl;

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

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

cin >> a[i][j];

clock\_t t = clock();

cout << "\tОпределитель матрицы равен " << det(size, a) << endl;

t = clock()-t;

double seconds = ((double)t)/CLOCKS\_PER\_SEC;

cout << "\tВремя выполнения: " << seconds << "sec" << endl;

for(int i = 0; i < size; i++) delete a[i];

delete a;

}

break;

case 3:

{

// Обратная матрица к квадратной матрице

int size;

cout << "\tРазмеры матрицы:" << endl; cin >> size;

double \*\*a = new double\*[size];

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

a[i] = new double[size];

cout << "\tМатрица A["<<size<<","<<size<<"]:" << endl;

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

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

cin >> a[i][j];

clock\_t t = clock();

double \*\*b = inverse(a, size);

t = clock()-t;

double seconds = ((double)t)/CLOCKS\_PER\_SEC;

cout << "\tВремя выполнения: " << seconds << "sec" << endl;

if(b == NULL)

{

cout << "Ошибка: определитель матрицы равен 0. Обратной матрицы не существует." << endl;

getchar();

getchar();

return -1;

}

cout << "\tОбратная матрица: " << endl;

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

{

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

cout << b[i][j] << '\t';

cout << endl;

}

for(int i = 0; i < size; i++) delete a[i];

for(int i = 0; i < size; i++) delete b[i];

delete a;

delete b;

}

break;

case 4:

{

// Алгоритм быстрой сортировки

int size;

cout << "\tРазмер массива:" << endl; cin >> size;

double \*a = new double[size];

cout << "\tИсходный массив["<<size<<"]:" << endl;

for(int i = 0; i < 10; i++) cin >> a[i];

clock\_t t = clock();

quickSort(a, 0, size-1);

t = clock()-t;

double seconds = ((double)t)/CLOCKS\_PER\_SEC;

cout << "\tВремя выполнения: " << seconds << "sec" << endl;

cout << "\tСортированный массив["<<size<<"]:" << endl;

for(int i = 0; i < 10; i++) cout << a[i] << '\t'; cout << endl;

delete a;

}

break;

case 5:

{

// Алгоритм битонической сортировки

int size;

cout << "\tРазмер массива:" << endl; cin >> size;

double \*a = new double[size];

cout << "\tмассив["<<size<<"]:" << endl;

for(int i = 0; i < 10; i++) cin >> a[i];

clock\_t t = clock();

host\_bitonic\_sort(a, size, 1);

t = clock()-t;

double seconds = ((double)t)/CLOCKS\_PER\_SEC;

cout << "\tВремя выполнения: " << seconds << "sec" << endl;

cout << "\tСортированный массив["<<size<<"]:" << endl;

for(int i = 0; i < 10; i++) cout << a[i] << '\t'; cout << endl;

delete a;

}

break;

}

getchar();

getchar();

return 0;

}

Сортировка одномерного массива

В программе реализованы два алгоритма сортировки массива: и быстрая сортировка Хоара и битоническая сортировка.

# Быстрая сортировка

Общая идея алгоритма состоит в следующем:

* Выбрать из массива элемент, называемый опорным. Это может быть любой из элементов массива или же число, вычисленное на основе значений элементов. На практике обычно выбирается средний элемент массива.
* Сравнить все остальные элементы с опорным и переставить их в массиве так, чтобы разбить массив на три непрерывных отрезка, следующие друг за другом: «меньшие опорного», «равные» и «большие».
* Для отрезков «меньших» и «больших» значений выполнить рекурсивно ту же последовательность операций, если длина отрезка больше единицы.

Листинг кода

#include "Header.hpp"

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Быстрая сортировка Хоара

\*data - массив элементов

left - номер первого элемента сортируемого массива

right - номер последнего элемента сортируемого массива

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void quickSort(double \*data, int left, int right)

{

int left1 = left;

int right1 = right;

double p;

double ref = data[(left+right)/2];

do{

while(data[left1] < ref)left1++;

while(data[right1] > ref)right1--;

if(left1<=right1){

if(data[left1] > data[right1])

data[left1] = (p = data[right1], data[right1] = data[left1], p);

left1++;

right1--;

}

}while (left1 <= right1);

if(left1 < right)

quickSort(data, left1, right);

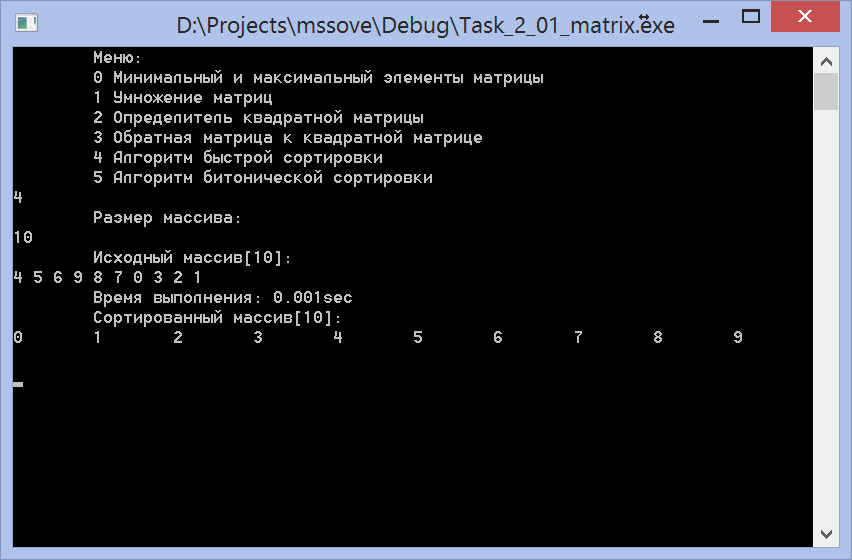
if(left < right1)

quickSort(data, left, right1);

}

# Контрольные примеры работы программы (быстрая сортировка)

Пример 1.



# Битоническая сортировка

* В основе этой сортировки лежит операция Bn (полуочиститель, half - cleaner) над массивом, параллельно упорядочивающая элементы пар xi и xi + n / 2
* Сортировка основана на понятии битонической последовательности и утверждении :

***Если набор полуочистителей правильно сортирует произвольную последовательность нулей и единиц, то он корректно сортирует произвольную последовательность.***

* Последовательность [a0, a1, …, an – 1] называется битонической, если она или состоит из двух монотонных частей (т.е. либо сначала возрастает, а потом убывает, либо наоборот), или получена путем циклического сдвига из такой последовательности.
  + Так, последовательность 5, 7, 6, 4, 2, 1, 3 битоническая, поскольку получена из 1, 3, 5, 7, 6, 4, 2 путем циклического сдвига влево на два элемента.
* Доказано, что если применить полуочиститель Bn к битонической последовательности [a0, a1, …, an–1], то получившаяся последовательность обладает следующими свойствами :

1. обе ее половины также будут битоническими.
2. любой элемент первой половины будет не больше любого элемента второй половины.
3. хотя бы одна из половин является монотонной.

* Применив к битонической последовательности [a0, a1, …, an–1] полуочиститель Bn, получим две последовательности длиной n/2, каждая из которых будет битонической, а каждый элемент первой не превысит каждый элемент второй.
* Далее применим к каждой из получившихся половин полуочиститель Bn/2, получим уже четыре битонические последовательности длины n/4.
* Применим к каждой из них полуочиститель Bn/2 и продолжим этот процесс до тех пор, пока не придем к n/2 последовательностей из двух элементов.
* Применив к каждой из них полуочиститель B2, поскольку все последовательности уже упорядочены, получим отсортированную последовательность.
* Итак, последовательное применение полуочистителей Bn, Bn/2, …, B2 сортирует произвольную битоническую последовательность.
* Эту операцию называют битоническим слиянием и обозначают Mn.
* Пусть задан одномерный неотсортированный массив [a0, a1, …, an–1]
* Очевидно, что данный массив может быть преобразован к битонической последовательности с помощью разделения массива на два подмассива и сортировки одной половины по-возрастанию, а другой половины по убыванию
* После того, как массив будет преобразован к битонической последовательности к нему можно будет применить операцию битонического слияния Mn и полученный массив станет отсортированным
* Для сортировки подмассивов может быть применён алгоритм битонической сортировки, либо любой другой алгоритм сортировки
* В данной реализации для сортировки подмассивов применён алгоритм битонической сортировки.
* Поскольку в основе алгоритма лежит последовательное применение полуочистителей Bn, Bn/2, …, B2 для массива [a0, a1, …, an–1], то очевидно, что это накладывает ограничение на размер массива к которому может быть применён алгоритм битонической сортировки – размер массива должен быть равен степени двойки, то есть

**n=2k**

* Для сортировки массива произвольного размера N, исходный массив должен быть разделён на подмассивы размера степени двойки. Каждый подмассив сортируется битонической сортировкой, а затем производится **слияние** уже отсортированных подмассивов в итоговый отсортированный массив

Листинг кода

#include "Header.hpp"

/\*

Битоническая сортировка (bitonic sort)

В основе этой сортировки лежит операция Bn(полуочиститель, half - cleaner) над массивом, параллельно

упорядочивающая элементы пар xi и xi + n / 2.На рис. 1 полуочиститель может упорядочивать элементы пар как по

возрастанию, так и по убыванию.Сортировка основана на понятии битонической последовательности и

утверждении : если набор полуочистителей правильно сортирует произвольную последовательность нулей и

единиц, то он корректно сортирует произвольную последовательность.

Последовательность a0, a1, …, an - 1 называется битонической, если она или состоит из двух монотонных

частей(т.е.либо сначала возрастает, а потом убывает, либо наоборот), или получена путем циклического

сдвига из такой последовательности.Так, последовательность 5, 7, 6, 4, 2, 1, 3 битоническая, поскольку

получена из 1, 3, 5, 7, 6, 4, 2 путем циклического сдвига влево на два элемента.

Доказано, что если применить полуочиститель Bn к битонической последовательности a0, a1, …, an - 1,

то получившаяся последовательность обладает следующими свойствами :

• обе ее половины также будут битоническими.

• любой элемент первой половины будет не больше любого элемента второй половины.

• хотя бы одна из половин является монотонной.

Применив к битонической последовательности a0, a1, …, an - 1 полуочиститель Bn, получим две

последовательности длиной n / 2, каждая из которых будет битонической, а каждый элемент первой не превысит

каждый элемент второй.Далее применим к каждой из получившихся половин полуочиститель Bn / 2.Получим

уже четыре битонические последовательности длины n / 4.Применим к каждой из них полуочиститель Bn / 2 и

продолжим этот процесс до тех пор, пока не придем к n / 2 последовательностей из двух элементов.Применив к

каждой из них полуочиститель B2, отсортируем эти последовательности.Поскольку все последовательности

уже упорядочены, то, объединив их, получим отсортированную последовательность.

Итак, последовательное применение полуочистителей Bn, Bn / 2, …, B2 сортирует произвольную

битоническую последовательность.Эту операцию называют битоническим слиянием и обозначают Mn.

Например, к последовательности из 8 элементов a 0, a1, …, a7 применим полуочиститель B2, чтобы на

соседних парах порядок сортировки был противоположен.На рис. 2 видно, что первые четыре элемента

получившейся последовательности образуют битоническую последовательность.Аналогично последние

четыре элемента также образуют битоническую последовательность.Поэтому каждую из этих половин можно

отсортировать битоническим слиянием, однако проведем слияние таким образом, чтобы направление

сортировки в половинах было противоположным.В результате обе половины образуют вместе битоническую

Битоническая сортировка последовательности из n элементов разбивается пополам и каждая из

половин сортируется в своем направлении.После этого полученная битоническая последовательность

сортируется битоническим слиянием.

\*/

void device\_exchange(double \*x, double \*y, int count);

void device\_copy(double \*x, double \*y, int count);

int device\_comparer(double \*x, double \*y);

void global\_bitonic\_worker(double \* data, int n, int i, int j, int blocks\_threads, int loops, int direction);

void global\_bitonic\_merger(double \* data, double \* data2, int \* sizes, int n, int direction);

void host\_bitonic\_sort(double \*data, int n, int direction)

{

// data - массив данных

// n - количество элементов в исходном массиве для сортировки

// direction - способ сортировки

// -1 означает сортировку по убыванию,

// 1 означает сортировку по возрастанию

double \*device\_data;

double \*device\_data2;

int \*device\_size;

// Всего надо выполнить k\*(k-1)/2\*2^(k-1) операций сравнения, где k = log2 n

// За одну итерацию запуска процессов будет выполнено n/2 = 2^(k-1) операций

// Шаг первый - копируем исходный массив в память GPU

device\_data = new double[n];

memcpy(device\_data, data, n\*sizeof(double));

// Число n представимо в виде суммы степеней двойки,

// Поэтому, разбиваем исходные данные на подмассивы с длинами равными слагаемым этой суммы

// и сортируем каждый подмассив битоническим алгоритмом

// В разультате получим равное числу слагаеммых отсортированных массивов длинами равным степеням двойки

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

if ( n & (1<<k) ) {

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

for( int j = i; j >= 0 ; j-- ) {

// Определим оптимальное разбиения на процессы, нити и циклы

// одна нить в просессе будет будет выполнять цикл с указанным количеством итераций

int blocks = 1 << (int)k/3;

int threads = 1 << (int)k/3;

int loops = ((1 << (k - 1)) + (blocks\*threads - 1))/(blocks\*threads);

assert((1<<(k - 1)) == blocks\*threads\*loops);

// одинаковый шаг в каждом блоке гарантирует отсутствие коллизий (одновременного доступа к одним и тем же данным)

global\_bitonic\_worker (&device\_data[n&((1<<k)-1)], n&(1<<k), i, j, blocks\*threads, loops, direction);

}

}

}

}

// Теперь надо произвести слияние уже отсортированных массивов

// Для этого выделяет массив такого же размера как и первый

// и массив размеров очередей

device\_data2 = new double[n];

device\_size = new int[sizeof(int)\*8];

global\_bitonic\_merger (device\_data, device\_data2, device\_size, n , direction);

// Возвращаем результаты в исходный массив

memcpy(data, device\_data2, n\*sizeof(double));

// Освобождаем память на устройстве

delete device\_data;

delete device\_data2;

delete device\_size;

}

void global\_bitonic\_merger(

double \* data,

double \* data2,

int \* size,

int n,

int direction)

{

for(int k=0; k<8\*sizeof(int) ; k++ ) size[k] = n & (1<<k);

int total = n;

while(total > 0) {

int k = 8\*sizeof(int); while( (k-->0) && (size[k] == 0) ) ;

for (int i=k; i-- ; ) {

if (size[i] > 0 &&

direction\*device\_comparer(

&data[(n&((1<<k)-1))+size[k]-1],

&data[(n&((1<<i)-1))+size[i]-1]) < 0)

{

k = i;

}

}

total--;

size[k]--;

device\_copy(&data2[total],&data[(n&((1<<k)-1))+size[k]],1);

}

}

void global\_bitonic\_worker(

double \* data,

int n, int i, int j,

int blocks\_threads, int loops,

int direction)

{

for(int block = 0; block<blocks\_threads; block++){

int step = 1<<j;

for(int y=0; y<loops; y++) {

// Получаем идентификатор шага цикла

int id = block\*loops+y;

int offset = ((id>>j)<<(j+1))+(id&((1<<j)-1));

int parity = (id >> i);

while(parity>1) parity = (parity>>1) ^ (parity&1);

parity = 1-(parity<<1); // теперь переменная parity может иметь только 2 значения 1 и -1

if ((offset+step) < n) {

int value = parity\*direction\*device\_comparer(&data[offset],&data[offset+step]);

if (value > 0) device\_exchange(&data[offset],&data[offset+step],1);

}

}

}

}

// Перестановка двух блоков в памяти устройства

void device\_exchange(double \*x, double \*y, int count)

{

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

double ch = x[i] ; x[i] = y[i] ; y[i] = ch;

}

}

// Копирование одного участка памяти в другой

void device\_copy(double \*x, double \*y, int count)

{

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

x[i] = y[i] ;

}

}

// Функция сравнения данных xранимых в памяти как целых чисел типа double

// comparison function which returns ​a negative integer value if the first argument is less than the second,

// a positive integer value if the first argument is greater than the second and zero if the arguments are equal.

int device\_comparer(double \*x, double \*y)

{

if ((\*x)<(\*y)) return -1;

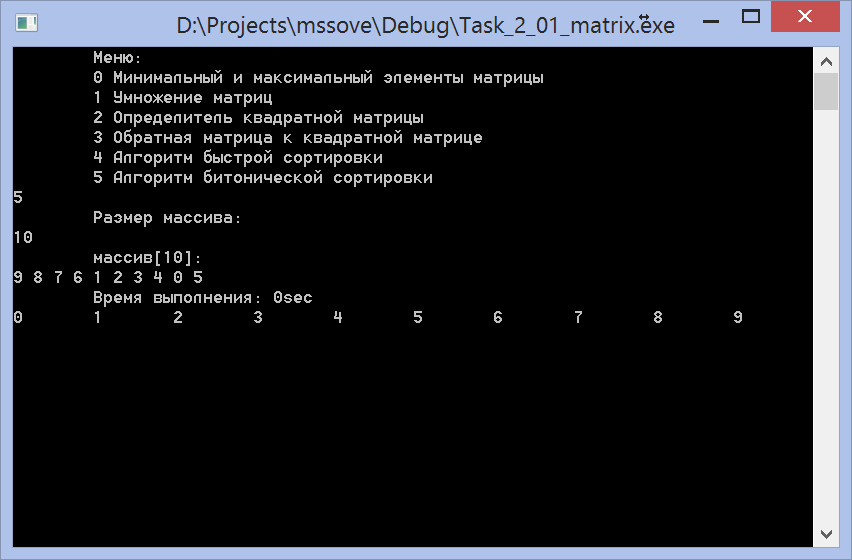
else if ((\*x)>(\*y)) return 1;

else return 0;

}

# Контрольные примеры работы программы

Пример 1.



# Матрицы. Операции над матрицами.

Будем представлять матрицу в виде двумерного массива чисел двойной точности.

# Вычисление определителя.

Определитель будем вычислять, используя формулу разложения по строке:

https://upload.wikimedia.org/math/1/2/f/12f60bbd0049cc8c8b58f162c80c9a44.png , где \bar M_j^1 — [дополнительный минор](https://ru.wikipedia.org/wiki/%D0%94%D0%BE%D0%BF%D0%BE%D0%BB%D0%BD%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D1%8B%D0%B9_%D0%BC%D0%B8%D0%BD%D0%BE%D1%80) к элементу a_{1j}.

Листинг кода

#include "Header.hpp"

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Определитель квадратной матрицы

size - сторона матрицы

\*\*data - матрица

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

double det(int size, double \*\*data){

int str, col, currCol;

double result = 0;//результат выполнения (детерминант)

double \*temp = new double[size+1]; //+1 для нулевого размера

if(size == 1)

return data[0][0];

for(col = 0; col < size; col++){

double \*\*newData = new double\*[size-1];

for(str = 0; str < size-1; str++)

newData[str] = new double[size-1];

//Создание новой матрицы из старой, выкидыванием столбца col и строки str

for(currCol = 0; currCol < size; currCol++){

if(currCol != col)

for(str = 1; str < size; str++){

newData[str-1][currCol-(currCol>col)] = data[str][currCol];

}

}

if(col%2)

temp[col] =- data[0][col]\*det(size-1, newData);

else

temp[col] = data[0][col]\*det(size-1, newData);

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

delete []newData[i];

delete []newData;

}

for(col = 0; col < size; col++) result+=temp[col];

delete temp;

return result;

}

# Контрольные примеры работы программы (вычисление определителя квадратной матрицы)

Пример 1.

Вычисление определителя треугольной матрицы 5х5 (рис. 9). Как известно, он равен произведению элементов главной диагонали.

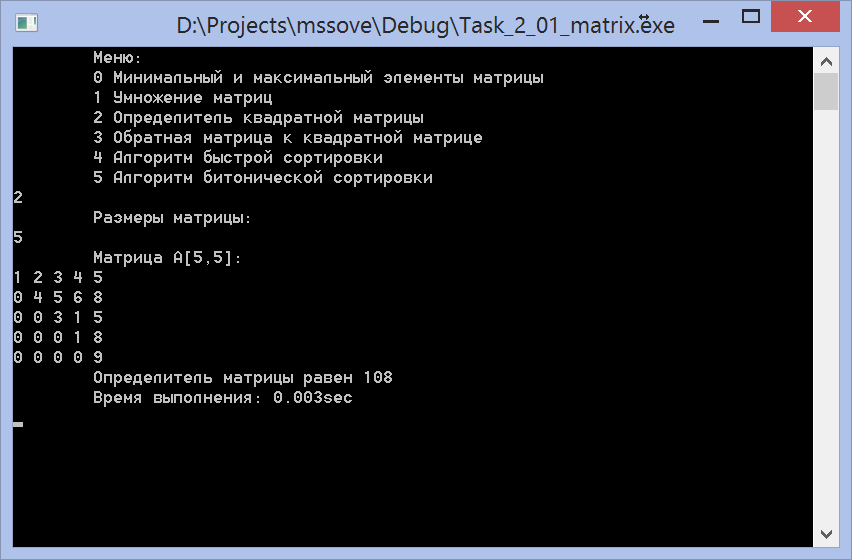
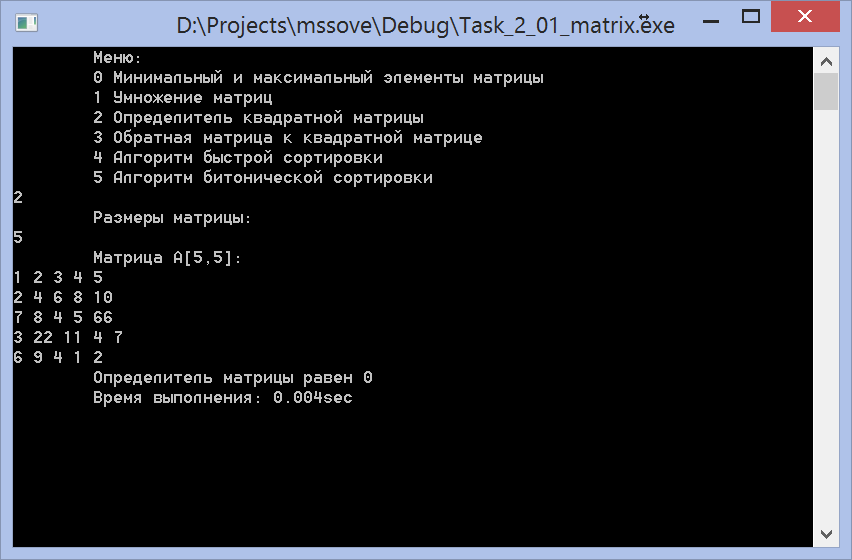


Рис.9. Вычисление определителя треугольной матрицы.

Пример 2.

Вычисление определителя с двумя линейно зависимыми строками (рис. 10). Как известно, он равен нулю.

 Рис.10. Вычисление определителя матрицы с линейно зависимыми строками (1-й и 2-й).

Пример 3.

Вычисление определителя обычной матрицы 2х2 (рис. 11).

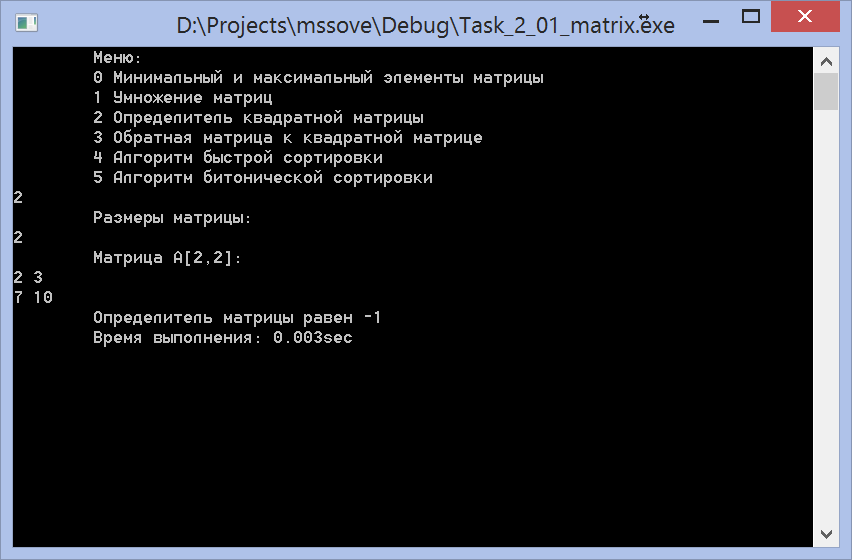
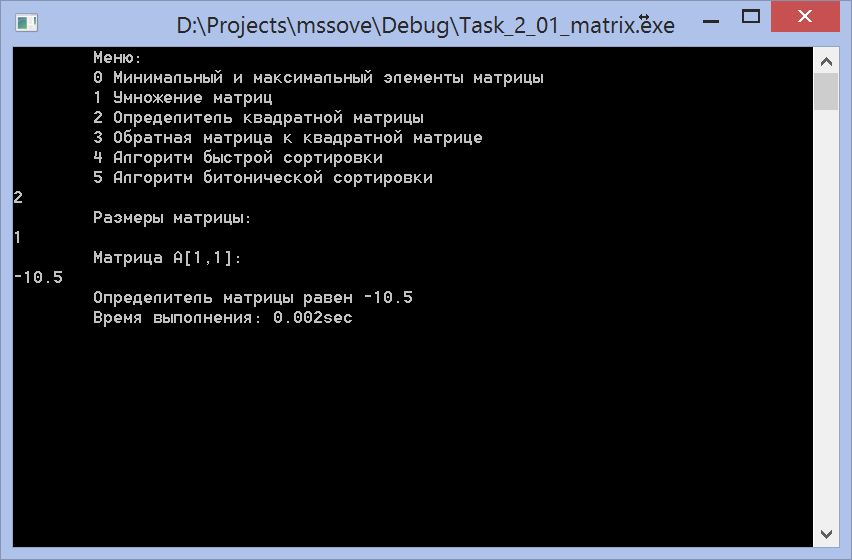


Рис.11. Вычисление определителя матрицы 2х2.

Пример 4.

Вычисление определителя матрицы 1х1 (рис. 12).

 Рис.12. Вычисление определителя матрицы 1х1.

# Поиск максимума по строкам, минимума по столбцам.

Искать максимальный элемент матрицы по строкам и минимальный по столбцам будем с помощью перебора всех элементов.

Листинг кода

#include "Header.hpp"

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Поиск максимального и минимального элемента

\*\*data - матрица

height - высота матрицы (кол-во строк)

width - ширина матрицы (кол-во столбцов)

&min - минимальный элемент

&max - максимальный элемент

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void findMinMax(double \*\*data, int height, int width, double &min, double &max){

min = max = data[0][0];

double \*temp = new double[height+1]; //+1 для нулевого размера

//Поиск максимального элемента по строкам

for(int str = 0; str < height; str++) {

temp[str]=data[str][0];

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

if(temp[str] < data[str][col])

temp[str] = data[str][col];

}

//Поиск максимального элемента среди максимальных элементов строк

//непараллельные вычисления

for(int str = 0; str < height; str++) {

if(max < temp[str])

max = temp[str];

}

//Поиск минимального элемента по столбцам

for(int str = 0; str < height; str++) {

temp[str]=data[str][0];

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

if(temp[str] > data[str][col])

temp[str] = data[str][col];

}

//Поиск минимального элемента среди минимальных элементов строк

//непараллельные вычисления

for(int str = 0; str < height; str++) {

if(min > temp[str])

min = temp[str];

}

delete temp;

}

Пример (рис 13).

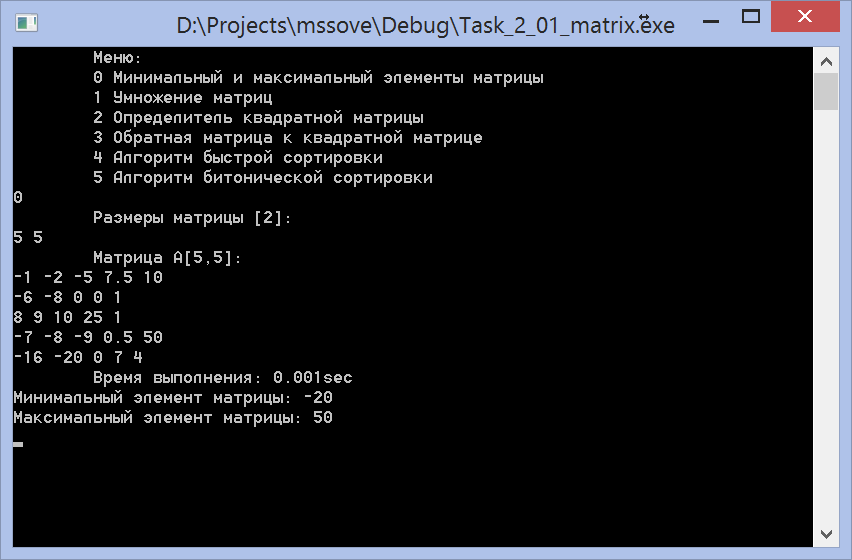


Рис.13. Нахождение минимального и максимального элементов.

# Умножение матриц.

Произведение матриц A \* B = C будем находить по формуле:

 c_{ij} = \sum_{r=1}^n a_{ir}b_{rj} \;\;\; \left(i=1, 2, \ldots m;\; j=1, 2, \ldots q \right).

При этом, матрицы А и B имеют размеры m x n и n x q соответственно.

Листинг кода

#include "Header.hpp"

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Произведение матриц

\*\*data1 - первый множитель

\*\*data2 - второй множитель

size1 - кол-во строк в первой матрице

size2 - колво столбцов в первой матрице и кол-во строк во второй

size3 - кол-во столбцов во второй матрице

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

double \*\*mult(double \*\*data1, double \*\*data2, const int size1, const int size2, const int size3){

double \*\* result;

result = new double \*[size1];//результат произведения

for(int i = 0; i < size1; i++)

result[i] = new double[size3];

for(int str = 0; str < size1; str++)//строки первой матрицы

{

for(int col = 0; col < size3; col++)//столбцы второй матрицы

{

result[str][col] = 0;

for(int i = 0; i < size2; i++)

result[str][col] += data1[str][i]\*data2[i][col];

}

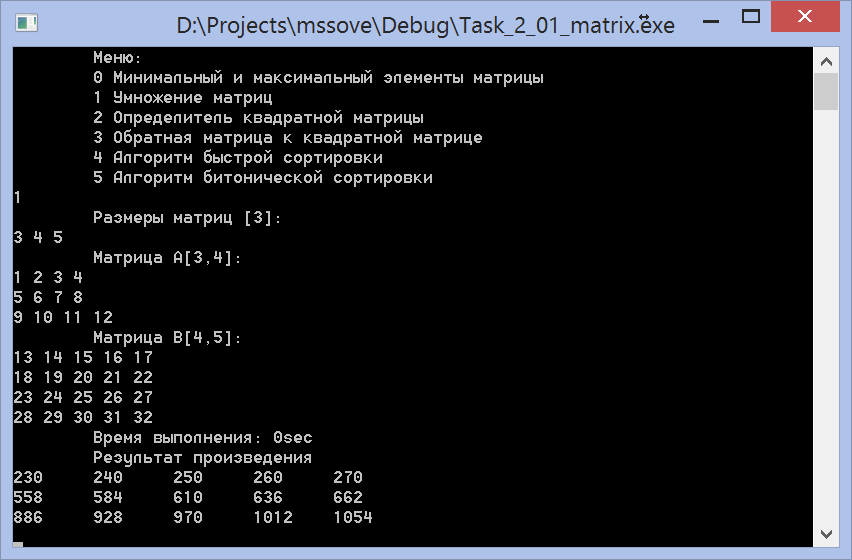
}

return result;

}

# Контрольные примеры работы программы (умножение матриц)

Пример 1.



# Нахождение обратной матрицы

Вычисление обратной матрицы будем выполнять с помощью алгебраических дополнений:

Пусть задана матрица An×n. Для того, чтобы найти элементы матрицы A−1, требуется осуществить три шага:

1. Найти определитель матрицы A и убедиться, что он не равен нулю 0, т.е. что матрица А – невырожденная.
2. Составить алгебраические дополнения Aij каждого элемента матрицы A и записать матрицу A∗n×n=(Aij) из найденных алгебраических дополнений.
3. Записать обратную матрицу с учетом формулы A−1= A∗)T, где Т – транспонирование.

Листинг кода

#include "Header.hpp"

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Вычисление обратной матрицы (только для квадратных)

\*\*data - матрица

size - сторона матрицы

Функция возвращает NULL, если обратной матрицы не существует.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

double \*\*inverse(double \*\*data, int size){

double \*\*additionMatrix;//Матрица алгебраических дополнений

double detData;

if((detData = det(size, data)) == 0)//вычисляем детерминант, если он равен нулю, то обратной матрицы не существует

return NULL;

additionMatrix = new double\*[size];

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

additionMatrix[str] = new double[size];

//Отдельный случай для матрицы 1х1

if(size == 1)

{

additionMatrix[0][0] = 1/data[0][0];

return additionMatrix;

}

//Составляем матрицу алгебраических дополнений

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

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

if(((str+col) % 2) == 0)

additionMatrix[str][col] = minor(data, size, str, col);

else

additionMatrix[str][col] = -minor(data, size, str, col);

transposeSQR(additionMatrix, size);//транспонируем матрицу

// используется параллельная обработка элементов массива

for(int str = 0; str < size; str++)//делим элементы на детерминант

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

additionMatrix[str][col] /= detData;

return additionMatrix;

}

Для реализации этой функции нам потребовались ещё две: вычисление минора матрицы для элемента Aij и транспонирование матрицы.

Листинг кода

#include "Header.hpp"

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Вычисление минора для квадратной матрицы

\*\*data - матрица

size - сторона матрицы

strin - строка, где располагается минор

colon - столбец, где располагается минор

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

double minor(double \*\*data, int size, int strin, int colon){

double resultMinor;

double \*\*result = new double\*[size-1];

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

result[i] = new double[size-1];

//Создание новой матрицы из старой, выкидыванием столбца colon и строки strin

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

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

if(col != colon && str != strin)

result[str-(str>strin)][col-(col>colon)] = data[str][col];

}

}

resultMinor = det(size-1, result);//Вычисляем минор

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

delete []result[i];

delete []result;

return resultMinor;

}

#include "Header.hpp"

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Транспонирование квадратной матрицы

\*\*data - матрица

size - сторона матрицы

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void transposeSQR(double \*\*data, int size){

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

for(int col = str+1; col < size; col++) {

double temp;

data[str][col] = (temp = data[col][str], data[col][str] = data[str][col], temp);

}

}

# Контрольные примеры работы программы (вычисление обратной матрицы)

Пример 1.

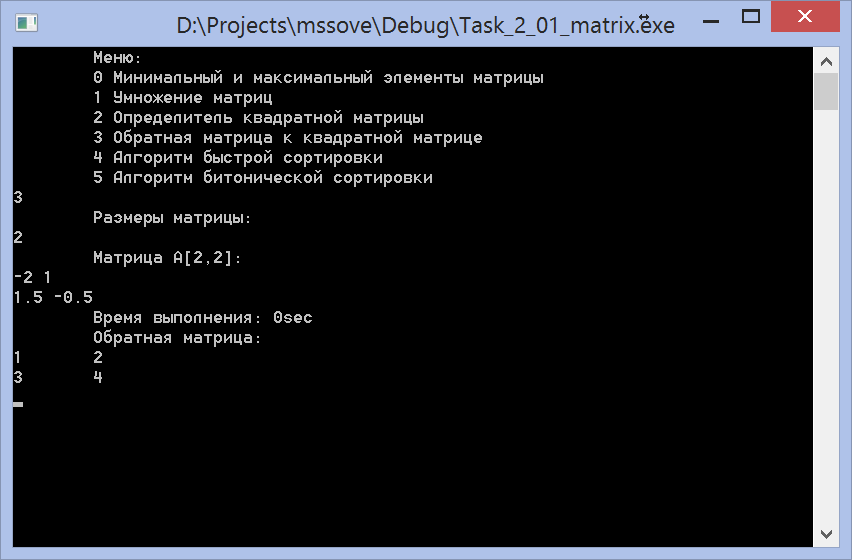


Рис.16. Вычисление обратной матрицы размера 2х2.

Пример 2.

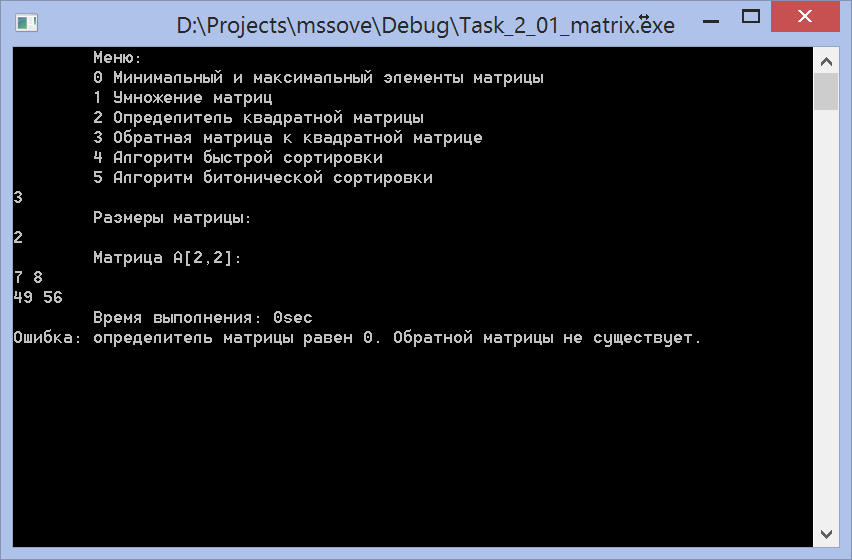


Рис.17. Вычисление обратной матрицы размера 2х2. Программа отреагировала на ошибку.

Пример 3.

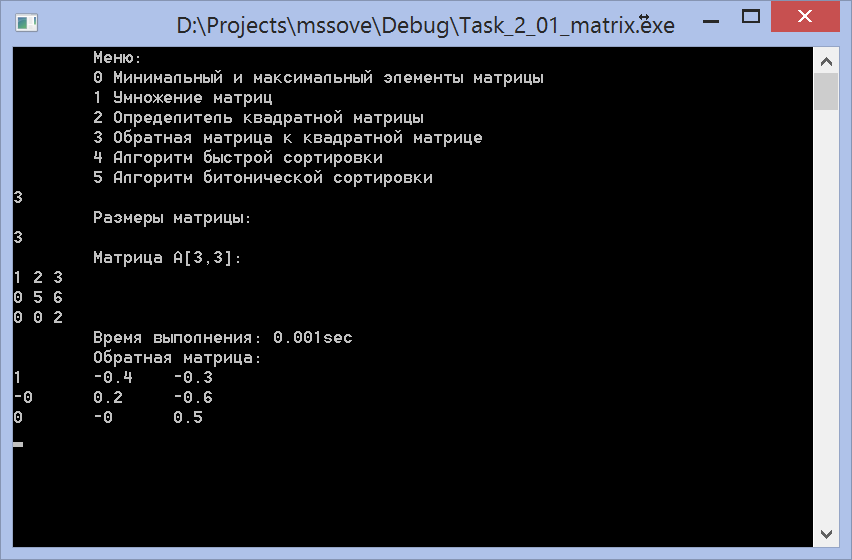


Рис.18. Вычисление обратной матрицы размера 3х3.

Пример 4.

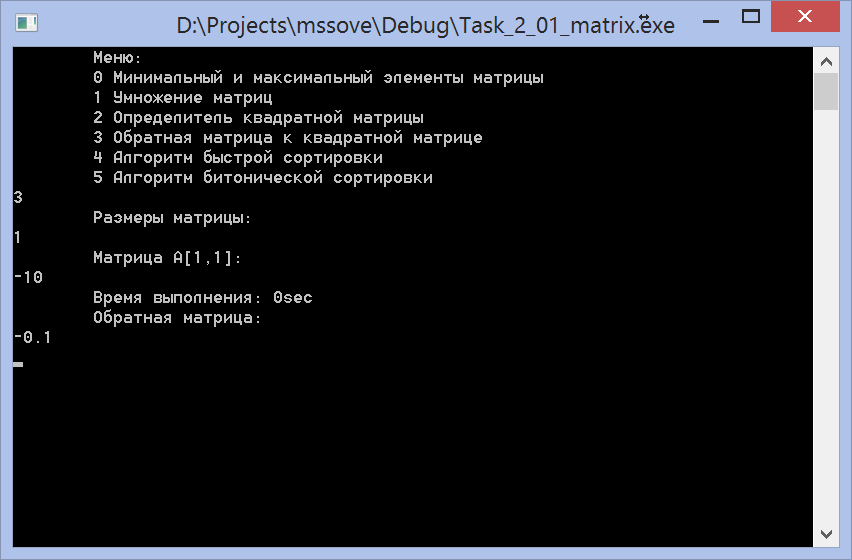


Рис.19. Вычисление обратной матрицы размера 1x1.

Пример 5.

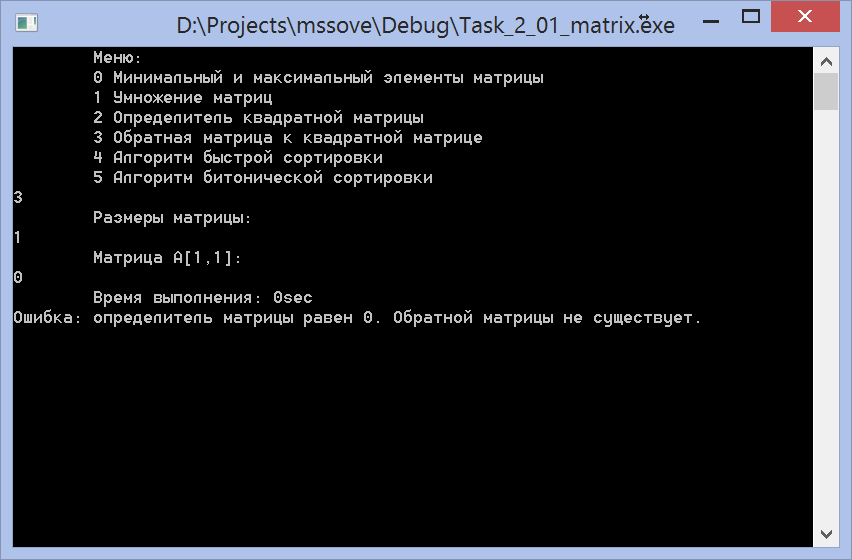


Рис.20. Вычисление обратной матрицы размера 1x1. Программа отреагировала на ошибку.

# Заключение

В ходе данной лабораторной был изучены алгоритмы сортировки (битоническая и быстрая сортировка Хоара). Была сделана программная реализация данных алгоритмов. Для данной программной реализации были проведены ряд тестов, показывающие правильность работы алгоритмов сортировки.

Была проделана реализация работы с матрицами, а именно: вычисление детерминанта, произведения матриц и нахождение обратной матрицы. Были представлены тестовые примеры, подтверждающие правильность работы программы, а так же разобран ряд критических случаев.