CS205 C/ C++ Programming - Project2

name: 廖铭骞

SID: 12012919

Part 1 - Analysis

文件的读取与写入

首先通过查阅书籍，了解到流是进出程序的字节流，而缓冲区是内存中的临时存储区域，信息在缓冲区和文件之间传输的时候将使用设备处理效率最高的尺寸以大块数据的方式进行传输，而C++通过将一个被缓冲流通程序以及其输入源相连来处理输入，通过将一个被缓冲流与程序及其输出目标相连来处理输出。

文件的读取分析

首先要创建一个ifstream对象来管理输入流，之后将这个对象与指定的文件关联起来，之后便可以使用该对象读取文件当中的信息。
使用ifstream对象打开文件的时候有可能发生文件不存在指定目录当中或者文件与流关联失败，即没能成功打开文件的情况，所以在使用该对象关联文件之后要通过使用方法is_open()确认文件是否与流关联成功，如果没有关联成功便要输出错误信息并通过exit(EXIT_FAILURE)与操作系统通信来终止程序，如果关联成功也要提示用户文件已经关联成功。
由于文件提供的是一个方阵矩阵，所以在文件读取的过程中也涉及到如何读取一个方阵矩阵的问题，在程序当中我使用了除检查文件是否被打开之外的另一种流状态检查的方法，即通过检查EOF来检查文件当前读取的位置是否已经到达了文件尾部，如果已经到达了尾部就退出读取文件的循环。
在文件读取完成之后要使用close()方法及时关闭文件，将缓存当中的数据刷新出来。在使用了close方法之后，原先的流对象就可以被用来打开其他的文件了。

文件的写入分析

在矩阵乘法运算完成之后还需要将结果矩阵重新输出到一个文本文件当中，与读取文件相似，写入文件需要在程序当中包含头文件fstream，并且创建一个ofstream对象将一个文件关联起来，该文件如果不存在，将在指定目录下创建该文件，如果该文件已经存在，那么在默认情况下，该文件在接收程序输出的时候将会被截短，即删除之前的所有内容。
在创建了流对象之后就可以使用该对象来进行文件的写入了，在写入完成之后使用close()函数关闭该文件。

矩阵乘法分析

根据矩阵乘法定义：
假设矩阵A的大小为 m × n，矩阵B的大小为n × q则，两个矩阵相乘结果矩阵C的大小为 m × q，两个矩阵相乘的公式为：

$$C_{iq} = \sum_{k=1}^{n} A_{ik} B_{kq}$$

计时的方式分析

由于本程序是在Linux环境下运行，所以使用Linux环境下的计时函数gettimeofday()进行程序的计时，该计时函数精度较高，可以达到微秒级别。
通过查看计时函数的结构可以发现，该结构体的定义为

```
struct timeval
{
  __time_t tv_sec;      /* Seconds.  */
  __suseconds_t tv_usec;    /* Microseconds.  */
};
```

这个函数获取从1970年1月1日到现在经过的时间和时区（UTC时间），但是按照Linux的官方文档，该时区已经不再使用，所以在使用的时候传入NULL即可。

在需要计时的函数开头以及结尾分别获取一次当前时间，它们之间的差值即为程序运行的时间。操作代码如下

```
    gettimeofday(&read_t11, NULL);
   //function
    gettimeofday(&read_t12, NULL);
    read_time_use += (read_t12.tv_sec - read_t11.tv_sec) + (double)(read_t12.tv_usec -
read_t11.tv_usec) / 1000000.0;
```

为了后续试验比较不同输入规模对文件读取，写入以及矩阵乘法运算效率，在本程序当中使用了三段计数函数，分别计算将文件数据读取的时间、将结果矩阵数据写入文本的时间以及矩阵乘法运算的时间。

命令行处理技术分析

文件处理程序通常需要使用命令行参数来指定输入输出文件。在本程序中，命令行的第一个参数是程序的名称，后三个参数分别是矩阵A， 矩阵B， 以及结果矩阵C输出的文件名称。
如果命令行参数小于4，代表着输入输出文件名称不完整，那么此时将提示用户需要输入三个文件名，之后使用`exit(EXIT_FAILURE)`与操作系统通信，退出程序。
在命令行参数符合要求时，在后续操作会逐步判断输入的源文件是否存在，以及源文件是否可以正常打开等，并执行相关后续操作。
通过命令行输出可以更有效地控制输入的规范性，更便于控制程序运行的稳定性。

矩阵乘法的速度优化及分析

如果按照普通的实现，即嵌套三层循环的矩阵乘法的实现，如下图所示：

```
//size是方阵的行数，res是结果矩阵，mat1，mat2是两个相乘的矩阵
for(int i = 0; i < size; ++i){
        for(int j = 0; j < size; ++j){
            for(int k = 0; k < size; ++k){
                res[i][j] += mat1[i][k] * mat2[k][j];
            }
        }
    }
```

借鉴在归并排序算法中的思想，矩阵的乘法还可以分而治之，通过对矩阵进行拆解，分别运算之后合并，这样计算从计算机层面思考一方面可以使得CPU更加快速地将矩阵从内存中搬运出来进行运算，同时在另一方面，运算更小规模的矩阵的时间也会显著减少，两方面结合，对矩阵进行分块运算可以加快运算的速度以及提高内存读取的效率。

代码实现如下所示，主要思想是把一个大矩阵分成block_size份，分别对每一部分进行运算之后进行合并：

```
int block_size = 5;
    for (int block_i = 0; block_i < size1; block_i += block_size) {
        for (int block_j = 0; block_j < size1; block_j += block_size) {
            for (int block_k = 0; block_k < size1; block_k += block_size) {
                for (int i = block_i; i < min(block_i + block_size, size1); i++) {
                    for (int j = block_j; j < min(block_j + block_size, size1); j++) {
                        for (int k = block_k; k < min(block_k + block_size, size1);
k++) {
                            res[i][j] += mat1[i][k] * mat2[k][j];
                        }
                    }
                }
            }
        }
    }
```

但是速度并没有因此显著加快，于是进行对分块矩阵乘法内部运算顺序的颠倒，使得其访问内存的效率提高进一步，进行实验，代码如下所示：

```
int block_size = 5;
    for (int block_k = 0; block_k < size1; block_k += block_size) {
        for (int block_i = 0; block_i < size1; block_i += block_size) {
            for (int block_j = 0; block_j < size1; block_j += block_size) {
                for (int k = block_k; k < min(block_k + block_size, size1); k++) {
                    for (int i = block_i; i < min(block_i + block_size, size1); i++) {
                        for (int j = block_j; j < min(block_j + block_size, size1);
j++) {
                            res[i][j] += mat1[i][k] * mat2[k][j];
                        }
                    }
                }
            }
        }
    }
```

但是代码速度依然没有比一般的矩阵乘法要快，于是推测矩阵分块的数量对矩阵乘法速度也会有影响，因为分块之后的矩阵规模对单块矩阵的乘法运算会有影响，如果分块过大，则不能很快地进行单块矩阵的运算，另一方面，如果分块过小，虽然运算单个矩阵的速度变快，但是进行矩阵分块以及合并的过程会消耗大量的时间，两个方面都会不利于矩阵的运算，所以要选择合适的矩阵分块值以达到最好的分块矩阵乘法的运算效率。

进一步分析，根据课上学习的内容，数组是一段连续的内存，可以发现在运算的过程当中mat2[k][j]在读取内存数据的时候是不连续的，在二维数组当中，mat2[k][j]是按照列读取的顺序来读取的，这样会使得其在内存中不断地"跳跃"，引起访问效率的降低，这种效率的降低当输入的矩阵规模非常大的时候是非常显著的，而且在数据量很大的情况下会消耗巨大的读写内存，这种显著的性能降低情况会在之后的报告中给出具体的说明。

要同时实现顺序访问mat1以及mat2，又要满足下列公式

$$C_{iq} = \sum_{k=1}^{n} A_{ik} B_{kq}$$

所以可以通过改变循环的顺序来减少内存"跳跃"的次数，实现初步的优化：

首先思考能否把mat2当中对 j 的循环放在最底层，把 k 放在第二层，把 i 放在最外层进行循环，这样相当于先对结果矩阵的每一行元素进行循环计算，在每一行计算完成之后计算下一行直到所有的行都计算完毕为止，但是针对结果矩阵每一行元素的循环计算是不正确的，举一个例子，当对结果矩阵第一个元素即C[0][0]进行运算的时候，反而是计算了mat1的行元素和mat2的行元素，所以这种方法经过验证是不正确的。

第一种矩阵相乘的优化没有成功，思考能否将k的循环放在最外层，将i放在第二层进行循环，依旧将 j的循环放在最内层呢？这样的循环代码如下所示：

```
for(int k = 0; k < size; k++){
        for(int i = 0; i < size; i++){
            float r = mat1[i][k];
            for(int j = 0; j < size; j++){
                res[i][j] += r * mat2[k][j];
            }
        }
    }
```

经过与标准三层循环程序运行三种矩阵规模结果的比对，可以得到两种算法的正确性一致的结论。
并可以明显地感觉到运算速度的加快，详细的对比结果将在下文给出。

Part 2 - Code

```
#include<iostream>
#include<sstream>
#include<string>
#include<fstream>
#include<sys/time.h>
#include<vector>
#include<iomanip>
using namespace std;
int main(int argc, char * argv[]){
    struct timeval total_t1, total_t2, read_t11, read_t12, read_t21, read_t22, t1, t2,
write_t1, write_t2;
    gettimeofday(&total_t1, NULL);
    if(argc < 4){//在没有文件名输入的情况下报错并退出程序
```

```
        cout << "Three files needed." << endl;
        cerr << "Usage: " <<  argv[0] << " filename[s]\n";
        exit(EXIT_FAILURE);
    }

    //进行计时变量的初始化
    double time_use = 0.0, read_time_use = 0.0, total_time_use = 0.0;
    ifstream fin1;
    string str1 = argv[1];
    istringstream is(str1);
    string str_size;
    //从文件名当中获取矩阵的规模
    for(int i = 0; i < 2; i++){
        getline(is, str_size, '-');
    }
    getline(is, str_size, '.');

    int size = stoi(str_size);

    fin1.open(str1);
    //检测文件1是否能正常打开
    if(!fin1.is_open()){
        cerr << "Could not open " << str1 << endl;
        fin1.clear();
        exit(EXIT_FAILURE);
    }else{
        cout << "Access file1 successfully" << endl;
    }
    //创建并初始化定长数组
    double ** mat1 = new double*[size];
    double ** mat2 = new double*[size];
    double ** res = new double*[size];
    for(int i = 0; i < size; i++)
    {
        mat1[i] = new double[size];
        mat2[i] = new double[size];
        res[i] = new double[size];
    }
    gettimeofday(&read_t11, NULL);//在文件流创建后，在文件开始读取前，记录起始时间
    int line = 0;
    while(!fin1.eof()){
        for(int i = 0; i < size; i++){
            fin1 >> mat1[line][i];
        }
        line++;
    }
    gettimeofday(&read_t12, NULL);//在文件流关闭之前，在文件读取结束之后，记录结束时间
    fin1.close();

    //将读取文件一所用的时间加入文件读取的总时间当中
    read_time_use += (read_t12.tv_sec - read_t11.tv_sec) + (double)(read_t12.tv_usec -
read_t11.tv_usec) / 1000000.0;

    ifstream fin2;

    string str2 = argv[2];
```

```
    fin2.open(str2);

    if(!fin2.is_open()){
        cerr << "Could not open " << str2 << endl;
        fin2.clear();
        exit(EXIT_FAILURE);
    }else{
        cout << "Access file2 successfully" << endl;
    }


    gettimeofday(&read_t21, NULL);
    int line2 = 0;
    while(!fin2.eof()){
        for(int i = 0; i < size; i++){
            fin2 >> mat2[line2][i];
        }
        line2++;
    }
    gettimeofday(&read_t22, NULL);
    fin2.close();

    read_time_use += (read_t22.tv_sec - read_t21.tv_sec) + (double)(read_t22.tv_usec -
read_t21.tv_usec) / 1000000.0;


    gettimeofday(&t1, NULL);
    //进行矩阵的运算
    for(int k = 0; k < size; k++){
        for(int i = 0; i < size; i++){
            double r = mat1[i][k];
            for(int j = 0; j < size; j++){
                res[i][j] += r * mat2[k][j];
            }
        }
    }gettimeofday(&t2, NULL);


    time_use = (t2.tv_sec - t1.tv_sec) + (double)(t2.tv_usec - t1.tv_usec) / 1000000.0;

    cout << "The reading process used: " << read_time_use << "s" << endl;
    cout << "The computation used: " << time_use << "s" << endl;

    ofstream fout(argv[3]);

    //设置文件格式为科学计数法
    fout.setf(ios_base::scientific, ios_base::floatfield);
    gettimeofday(&write_t1, NULL);
    for(int i = 0; i < size; i++){
        for(int j = 0; j < size; j++){
            fout << res[i][j] << " ";
        }
        fout << endl;
    }
    gettimeofday(&write_t2, NULL);
    fout.close();
    //在完成了数组的使用之后释放内存
```

```
    delete[] mat1;
    delete[] mat2;
    delete[] res;
    //计算写入文件的时间
    double write_time_use = (write_t2.tv_sec - write_t1.tv_sec) + (double)
(write_t2.tv_usec - write_t1.tv_usec) / 1000000.0;
    cout << "The writing process used: " << write_time_use << "s" << endl;

    gettimeofday(&total_t2, NULL);
    //程序的总用时
    total_time_use = (total_t2.tv_sec - total_t1.tv_sec) + (double)(total_t2.tv_usec -
total_t1.tv_usec) / 1000000.0;
    cout << "The total time used in the program is: " << total_time_use << "s" << endl;
    //计算出文件IO时间占总时间的比例
    double rate = (read_time_use + write_time_use) / total_time_use * 100.0;
    cout << "The IO time take up " << rate << "%" << "of the total time" << endl;
    return 0;

}
```

Part 3 - Result and Verification

两种不同的浮点数据类型对矩阵乘法精确度的影响及分析

在测试精确度之前，根据所学的知识以及在本地运行`sizeof(float)`和`sizeof(double)`得到的结果

```
    cout << "sizeof(float) = " << sizeof(float) << endl;
    cout << "sizeof(double) = " << sizeof(double) << endl;
```

在本地中，`float`数据类型的有效位数是32位，即4个字节，`double`数据类型为64位，即8个字节，所以使用`double`数据类型会使得运算结果更接近精确值，但是由于在计算机当中，底部数据的存储形式是二进制形式，浮点数在计算机当中的存储形式由三部分组成，以双精度浮点数（64位）为例，第一部分，即最高位是符号位，存储浮点数对应的符号；第二部分，即中间的11位，存储的是指数位；剩余的52位存储尾数。如此，在计算机当中无法存储所有的浮点数，例如十进制小数0.20，我们无法通过二进制小数来准确表示，只能不断增加二进制长度来提高精度。在进行浮点数运算的时候会发生精度损失，浮点数运算本身就存在着误差。但是`float`类型默认情况下只能保存6位的有效数字，所以当有效位数大于6的时候便会损失精度，系统确保`double`类型可以确保15位的有效位，使用`double`运算时，只要有效位数小于15便不会损失精度，但是在有效位数大于15的时候，也会存在精度的损失。但是由于`double`存储的位数比`float`多，所以可以认为double的运算结果会比`float`更接近精确值。

下面的程序将使用官方的三种规模的矩阵以及自己生成的三种规模的矩阵分别进行六组实验，每次将同一种规模的两个矩阵使用float数据类型运算的结果与double数据类型运算的结果进行比较，以double进行运算的结果作为标准值，定义N是方阵的大小，平均误差比率 θ 表示每一种矩阵相乘的平均误差，误差比率θ_{ij} 表示在结果矩阵位置为第 i 行和第 j 列使用两种类型计算的相对误差，使用 $ d_{ij} $表示在结果矩阵位置为第 i 行和第 j 列使用double运算的结果，使用f_{ij}表示在结果矩阵位置为第 i 行和第 j 列使用float`运算的结果，计算的结果计算公式如下所示：

$$\theta_{ij} = \frac{|d_{ij} - b_{ij}|}{d_{ij}} \times 100\%$$

$$\theta = \frac{\sum_{i=1}^{N}\sum_{j=1}^{N}\theta_{ij}}{N \times N}$$

计算误差比率的程序中需要同时对两个结果文件的每一个数字进行比较分析，最终得到每一种矩阵规模使用两种数值类型相乘得到的平均误差比率。

实现的程序如下所示：

```
#include<iostream>
#include<fstream>
#include<cmath>
using namespace std;
int main(int argc, char * argv[]){
    string file1 = argv[1];
    string file2 = argv[2];
    string str = argv[3];
    int size = stoi(str);
    double dif = 0;
    ifstream fin1(file1);
    ifstream fin2(file2);
    for(int i = 0; i < size; i++){
        for(int j = 0; j < size; j++){
            double num1, num2;
            fin1 >> num1;
            fin2 >> num2;
            dif += fabs(num1 - num2) / num1;
        }
    }
    double avg = dif / (size * size);
    double rate = avg * 100;
    cout << "The average rate is: " << avg  << "%"<< endl;

}
```

输出结果如下所示：

将结果制作成图表如下所示：

矩阵规模：			32 * 32	256 * 256	512 * 512	1024 * 1024	2048 * 2048	4096 * 4096
float与double计算的平均误差率\%：			4.52E-09	1.91E-09	1.97E-06	1.04E-06	7.16E-07	2.40E-06

可以发现`float`计算的结果与`double`计算的结果精确度上相差不大，但是当矩阵的计算规模增大之后，`float`和`double`值计算的结果相差大体上在逐渐增加，经过分析，这是因为在计算大矩阵的乘法的时候涉及的位数在增加，使用`float`很容易丢失大部分的精度，只能保存前六位的有效数字，在存储比较大的浮点数的时候精度损失的就比`double`类型的要多，所以在进行较大规模的矩阵乘法的时候，对精度要求比较高的时候，相较于使用`float`类型，使用double进行运算是更优的。

两种不同的浮点数据类型对矩阵乘法速度的影响及分析

前面进行了两种不同的浮点数据类型对矩阵乘法精确度的比较以及分析，得到了如果对大规模的矩阵乘法精度要求比较高的时候应该使用`double`类型进行矩阵的存储以及运算，但是不难发现，使用两种不同的数据类型产生结果的误差是几乎可以忽略不计的，误差结果如下表所示：

矩阵规模：			32 * 32	256 * 256	512 * 512	1024 * 1024	2048 * 2048	4096 * 4096
float与double计算的平均误差率\%：			4.52E-09	1.91E-09	1.97E-06	1.04E-06	7.16E-07	2.40E-06

但是，如果对精度要求不是极高的情况下，更追求程序运行的速度的时候，使用`float`会比使用`double`进行运算更加快速？针对这个问题，进行了实验探究，在实验当中，分别使用`float`和`double`对六种规模的矩阵进行运算，每种规模的矩阵计算三遍求取平均值，比较矩阵乘法运算所用的时间，制作成图进行对比，如下所示：

float类型运算耗时/s	矩阵规模	double类型运算耗时/s
0.000143	32 * 32	0.000129
0.059127	256 * 256	0.060022
0.425145	512 * 512	0.427325
3.38588	1024 * 1024	3.84498
27.4333	2048*2048	28.7979
224.494	4096 * 4096	232.822

可以发现在同等规模的矩阵乘法运算当中，使用double类型平均下来会比使用float类型慢2%左右，且随着矩阵规模的增大，使用double类型进行运算会比使用float慢越来越多。

从两种数据类型所占的字节数以及精度层面进行分析，float是单精度浮点数，而double是双精度浮点数，考虑到CPU的指令集，假设在一个CPU周期里面可以同时执行2条256bit 加法和2条256bit乘法，那么在单个GPU周期当中对单精度浮点数的计算次数就会是双精度浮点数计算次数的两倍。所以使用单精度运算更快的原因是float所占字节数更少，消耗内存比double类型也更加少，所以在运算的时候速度便会比double更快，另一方面，在float类型进行计算的时候，系统所要保证的精度只有六位，而在double类型进行计算的时候，系统所要保证的精度是15位，相对来说，在float进行计算的时候，在保证精度的层面，对float的精度要求并没有double精度要求那么苛刻，在第六位之后就可以进行四舍五入截断了，但是使用double类型每一次的运算精度都要保证在15位，所以随着矩阵运算规模的增大，运算的精度上面也会掣肘double类型的运算速度，这也上使得当运算矩阵规模越大的时候，使用float类型进行计算的速度比使用double进行计算要更快。

综上所述，如果计算程序对浮点数矩阵运算的精确度要求没有达到1E-06%的时候，同时对矩阵乘法的速度更为看重的时候，使用float类型进行矩阵的运算会在运算效率以及成本方面优于使用double类型进行运算。

矩阵规模变化对矩阵乘法速度的影响及分析

以float数据类型为例，通过对不同数量规模矩阵进行乘法运算并计时，通过矩阵乘法运算的时间可以显著地发现，随着矩阵规模的增大，进行矩阵乘法运算的速度会显著地变慢，为了探讨矩阵规模变化对矩阵乘法运算速度的影响呈现的数学规律，在指定的矩阵文件之外，可以额外生成随机的浮点数矩阵，本次实验中，为了尽可能使得找出的数学规律正确，我额外生成三个矩阵文件，三个矩阵的规模分别为512 * 512， 1024 * 1024， 4096 * 4096的矩阵

随机浮点矩阵生成代码如下：

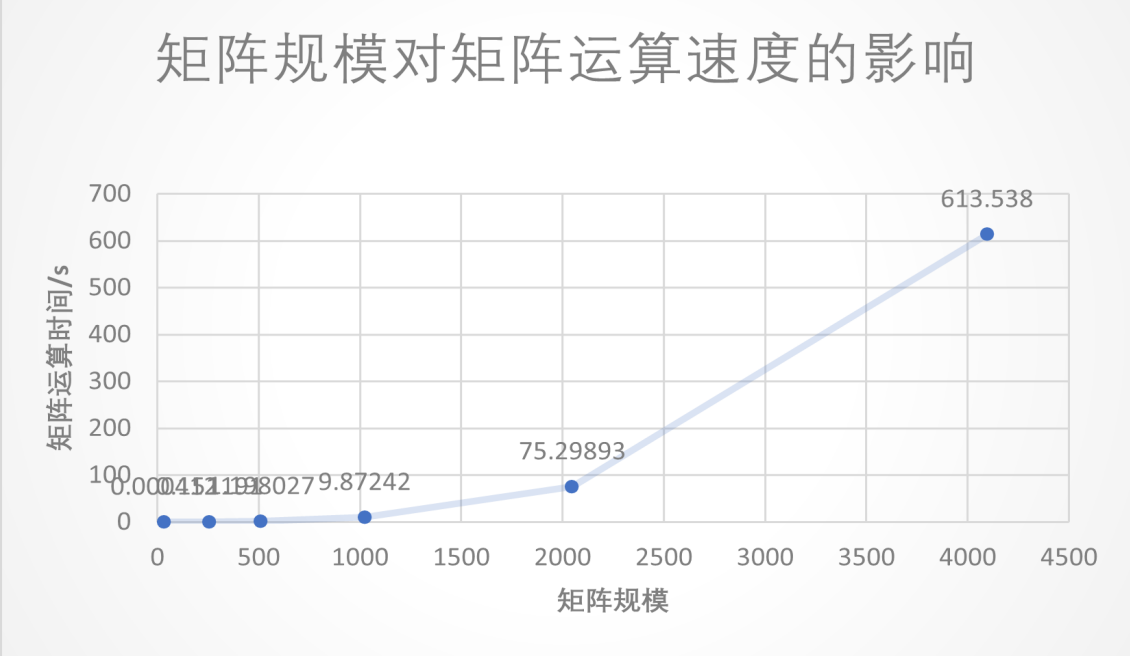
```
#include <iostream>
#include <stdlib.h>
#include <time.h>
#include<fstream>
using namespace std;
int main(){
    //将输出流与将要生成的文件绑定起来
    ofstream fout("mat-A-512.txt");
    //srand()用来设置rand()产生随机数时的随机数种子
    srand((unsigned)time(NULL));
```

```
    //生成规模为512 * 512 的矩阵
    for(int i = 0; i < 512; i++){
        for(int j = 0; j < 512; j++){
            //用于产生0 到 100 之间的随机浮点数
            float value = (rand() % (100 * 10 - 1)) / 10.0;
            fout << value;
            if(j != 511){
                fout << " ";
            }
        }
        fout << "\n";
    }
}
```

并分别对它们进行矩阵的乘法运算，由于4096 * 4096运行一次的时间过长，所以除了这个规模的矩阵乘法之外，每个程序运行三次，计算平均值之后对结果进行建立折线图进行可视化，得到的图表如下所示：

矩阵规模	32	256	512	1024	2048	4096
平均值	0.000412	0.151191	1.198027	9.87242	75.29893	613.538

根据这一表中的结果做出相应折线图如下所示



通过使用数学工具对数据进行分析可以得到矩阵乘法的运算时间与矩阵规模有95%的置信系数呈指数增长的关系，即随着矩阵规模倍数的增加，矩阵乘法的运算时间与矩阵的规模大致呈三次方的关系。

分析：由于优化了矩阵乘法之后依然需要使用三层循环，每一层循环都必须遍历N个元素，所以尽管使用了优化，但是矩阵乘法的时间复杂度依然是$O(N^3)$，所以根据理论分析以及做出的相关图像，可以得出在这种算法下，矩阵乘法的运算时间与矩阵规模大致呈三次方的关系。

不同数据类型对文件读取以及写入速度的影响及分析

要测量文件读取的时间，在本程序中依旧使用的是Linux环境下的计时函数`gettimeofday()`进行程序的计时，因为该计时函数精度较高，可以达到微妙级别。

首先是对文件一，也就是对第一个矩阵文件的读取，读取前后代码如下所示：

```
gettimeofday(&read_t11, NULL);//在文件流创建后，在文件开始读取前，记录起始时间
    int line = 0;
    while(!fin1.eof()){
        for(int i = 0; i < size; i++){
            fin1 >> mat1[line][i];
        }
        line++;
    }
    gettimeofday(&read_t12, NULL);//在文件流关闭之前，在文件读取结束之后，记录结束时间
    fin1.close();

    read_time_use += (read_t12.tv_sec - read_t11.tv_sec) + (double)(read_t12.tv_usec -
read_t11.tv_usec) / 1000000.0; //将读取文件一所用的时间加入文件读取的总时间当中
```

因为所要记录的只是读取文件的时间，而并不包括创建以及关闭输入流的时间，所以需要在文件流创建后，在文件开始读取前，记录起始时间，在在文件流关闭之前，在文件读取结束之后，记录结束时间，之后将两部分时间之差加入到读取文件的总时间当中。

文件的写入时间的记录也是类似，因为当文件不存在的时候会先在输出流创建的时候创建文件，这会对文件写入的时间计算造成影响，所以应当在输出流创建之后记录写入文件的起始时间，在输出流关闭之前记录文件写入的结束时间，并且计算出两者的差值，其中在秒级的差值可以直接记录到总的文件写入的时间当中，微秒级的差值需要除以10^6再计入到总的时间当中。

经过对文件读取以及写入速度的时间的计算，得到以下对比图：

下图为两种不同数据类型 `float` 和 `double` 在相同矩阵的规模之下对文件读取时间的比较

float类型文件读取时间/	矩阵规模	double类型文件读取时间/s
0.000669	32 * 32	0.000654
0.031931	256 * 256	0.031217
0.123961	512 * 512	0.108472
0.423184	1024 * 1024	0.409658
1.62976	2048*2048	1.71456
6.06042	4096 * 4096	6.53086

下图为两种不同数据类型 float 和 double 在相同矩阵规模之下对文件写入时间的比较

float类型文件写入时间/	矩阵规模	double 类型文件写入时间/s
0.001783	32 * 32	0.002115
0.064539	256 * 256	0.06347
0.222061	512 * 512	0.239367
0.97134	1024 * 1024	0.938943
4.39722	2048*2048	4.43006
18.1522	4096 * 4096	19.0932

在一开始我只是单想探究两种不同的数据类型对文件从磁盘中读取时间的影响，但是考虑到读取方面的计时还包括了数组的赋值过程，这一部分可能对计算文件从磁盘中读取出来时间会有所影响，所以又加入了对文件写入过程时间的测量，因为从指针数组中获取元素的时间不会对写入文件的时间产生太多影响，所以同时进行了对文件读取以及写入过程时间的测量以保证结果的正确性。

经过对两张对比图的分析以及结合float所占的字节数是4，以及double所占的字节数是8，float类型占用的位数更少，所以从文件当中读取到缓冲区的速度会比double更快，同理，使用float的时候从程序中将字节输出到缓冲区的时候也会更加快。

矩阵乘法的不同实现对矩阵乘法速度的效率影响及分析

首先在未优化的情况下，采用普通的三层循环进行矩阵的乘法运算，普通的三层循环代码如下所示：

```
for(int i = 0; i < size; ++i){
        for(int j = 0; j < size; ++j){
            for(int k = 0; k < size; ++k){
                res[i][j] += mat1[i][k] * mat2[k][j];
            }
        }
    }
```

改进后的循环代码如下所示：

```
for(int k = 0; k < size; k++){
        for(int i = 0; i < size; i++){
            double r = mat1[i][k];
            for(int j = 0; j < size; j++){
                res[i][j] += r * mat2[k][j];
            }
        }
    }
```

分别使用这两种程序对四种不同规模的矩阵进行运算，运行得到对比图如下所示：

改进算法耗时/s	矩阵规模	普通算法耗时/s
0.000146	32	0.000405
0.059	256	0.112195
0.417	512	0.953651
3.640	1024	15.9128
28.500	2048	139.193

可以发现当数据量越大的时候，改进后的算法与未改进的算法所用的时间相差的就越显著。

其实经过之前的分析不难得出，两者的时间复杂度都是$O(N^3)$，改进后的算法通过改变内存的访问顺序在数据规模极大的情况下显著提升了矩阵乘法运算的速度。
改进算法代码如下：

```
for(int k = 0; k < size; k++){
        for(int i = 0; i < size; i++){
            for(int j = 0; j < size; j++){
                res[i][j] += mat1[i][k]; * mat2[k][j];
            }
        }
    }
```

在最底层循环将原本访问mat2数组元素的过程从依次跳过N个元素进行访问，变成了连续对mat2元素进行访问，尽管并不是连续地对mat1进行访问但是在矩阵规模很大的情况下，对mat1的访问远远没有对mat2的访问那么频繁，所以通过改变使用最频繁的矩阵内存的访问顺序可以在很大程度上提高矩阵运算的效率。

可变长数组和定长数组对矩阵乘法程序速度的对比及分析

根据课上所学内容得知使用可变长数组进行矩阵的存储以及运算会比使用定长数组要慢，现借矩阵乘法为例探究这一性质。

以数据类型为double为例，首先将代码中所有涉及到可变长数组的部分全部改成指针定长数组来实现，将文件读取写入时间与矩阵运算时间进行再次计算，对每一种矩阵的规模进行矩阵乘法运算过程的计时，由于规模为4096 * 4096的矩阵规模很大，运算时间比较慢，所以除了这一组数据之外每一组矩阵规模数据的计时都采用了计算三次算取平均值的方法，确保结果的稳定性。
并将同样运行产生的结果与可变长数组运行的时间进行对比分析，得到的定长数组的耗时表格如下所示：

矩阵规模		32	256	512	1024	2048	4096

矩阵运算时间		1.64E-04	0.063704	0.420103	3.65727	28.7119	
		0.000145	0.057955	0.44084	3.62064	27.9789	
		0.000129	0.056598	0.391128	3.64168	28.7905	228.326
平均值		1.46E-04	5.94E-02	4.17E-01	3.64E+00	2.85E+01	2.28E+02

动态数组的耗时表格如下所示：

矩阵规模		32 * 32	256 * 256	512 * 512	1024 * 1024	2048 * 2048	4096 * 4096
矩阵运算时间/s		0.000728	0.52401	3.87264	9.81819	74.8076	
		0.000849	0.404488	2.32141	9.79537	75.6394	
		0.001589	0.27074	2.8464	9.55155	78.8308	
平均值		0.001055333	0.399746	3.013483333	9.721703333	76.42593333	617.701

根据得到的两组数据进行对比分析，做出对比蝴蝶图如下图所示：

定长数组耗时/s	矩阵规模	动态数组耗时/s
0.000146	32	0.001055
0.059	256	0.399746
0.417	512	3.013483
3.640	1024	9.721703
28.500	2048	76.42593
228.000	4096	617.701

从图中不难发现，对于每一组矩阵规模的乘法运算而言，使用可变长数组vector所消耗的时间是大于使用定长数组的，且随着矩阵规模的增大，动态变长数组耗时与指针定长数组的耗时差距会越来越大。

分析其中原因，不难从内存分配的角度去分析，可变长数组相对于定长数组最大的不同就是可变长数组的长度并不是固定的，在本程序当中是随着数据的输入一直在动态改变的，而在矩阵乘法运算的过程当中，涉及的仅仅是数据的访问与运算，可变长数组与定长数组的差距竟然也会如此之大，为了更加体现出可变长数组访问效率与定长数组的区别，下面进行一个访问的实验，事先将定长数组以及可变长数组赋值为递增的若干个元素，之后对这些元素进行访问，并进行计时，计算它们的运行时间。
程序代码如下：

```
#include<iostream>
#include<vector>
#include<sys/time.h>
using namespace std;
int main(){
    vector<double> nums;
    double * ds = new double[200000000];

    for(long long i = 0; i < 200000000; i++){
        nums.push_back((double)i * 1.0);
        ds[i] = i * 1.0;
    }

    timeval t1, t2, t3, t4;
    gettimeofday(&t1, NULL);
    for(long long i = 0; i < 200000000; i++){
        double j = nums[i];
    }

    gettimeofday(&t2, NULL);

    gettimeofday(&t3, NULL);
    for(long long i = 0; i < 200000000; i++){
        double j = ds[i];
    }
    gettimeofday(&t4, NULL);

    double time1 = (t2.tv_sec - t1.tv_sec) + (double)(t2.tv_usec - t1.tv_usec) /
1000000.0;
    double time2 = (t4.tv_sec - t3.tv_sec) + (double)(t4.tv_usec - t3.tv_usec) /
1000000.0;
    cout << "The dynamic vector used: " << time1 << "s" << endl;
    cout << "The static array used: " << time2 << "s" << endl;


}
```

对这段代码运行了两组数据，分别是2_0000_0000和20_0000_0000，分别进行访问数组元素的测试，得到两组数据

```
The dynamic vector used: 0.052697s
The static array used: 0.046492s
```

```
The dynamic vector used: 0.513479s
The static array used: 0.53454s
```

```
The dynamic vector used: 69.1339s
The static array used: 74.2336s
```

由这三组数据可以发现，当矩阵规模越来越大的时候，使用可变长数组的访问元素的速度会比使用定长数组更慢，所以当矩阵规模很大的时候，使用这两种不同的数组带来的访问速度上的差异对最终矩阵的运算速度也会有很大的影响。

除此之外，可变长数组在将数据从文件读取进入二维数组的时候速度也会比定长数组慢很多。以这次矩阵乘法为例，进行六种规模的矩阵乘法并测量它们的文件IO总时间，进行可变长数组和定长数组的用时比较，比较如下表所示：

定长数组	矩阵规模	可变长数组
0.002846667	32 * 32	0.010940667
0.096945333	256 * 256	0.288477667
0.356910667	512 * 512	1.087019667
1.381736	1024 * 1024	1.457028
6.338206667	2048*2048	6.340516667
24.05074	4096 * 4096	27.45377

经过观察表中数据以及进行分析，可变长数组的文件IO时间总体上大于使用定长数组的文件IO时间，原因可能是使用可变长数组在增添元素的时候会重新申请一遍空间，而定长数组的申请空间过程在一开始就已经完成，不会再更改，所以在文件输入数据到二维数组的时候，申请空间的时间使得可变长数组的效率低于定长的数组，而这种效率的变低在矩阵规模非常大，需要从文件当中读取数据的量非常大的时候，这种频繁申请空间的操作会使得可变长数组的效率越来越低。从文件读取的一方面考虑，也使得使用可变长数组进行计算在整个程序层面效率变得比定长数组要低。

文件IO 占总程序运行时间占比的分析

上一个实验分析了可变长数组会在文件IO上面比定长数组耗费更多的时间，也分析了可变长数组在访问数据并进行运算的时候会比定长数组要慢不少，为了再次验证这一点，我设计了一个实验，可以通过计算相同矩阵规模之下，使用可变长数组的文件IO占总程序运行的时间和定长数组的文件IO占总程序运行的时间来进一步说明，具体运行结果见表格所示：

定长数组的文件IO占比		矩阵规模	可变长数组的文件IO占比

8.83%		32 * 32	16.31%
48.01%		256 * 256	38.87%
43.08%		512 * 512	24.77%
27.76%		1024 * 1024	13.89%
17.62%		2048*2048	7.72%
9.77%		4096 * 4096	3.93%

从表格当中可以发现，使用可变长数组进行矩阵的运算反而在文件IO占程序运行总时间的比例总体上比定长数组还要低，由前面的数据可知，在可变长数组进行文件IO会比在定长数组中的文件IO要慢，所以这次实验可以从另一个角度说明使用可变长数组进行数组元素的访问以及运算的效率是比使用定长数组进行元素访问以及运算的效率要低的。

结果正确性的保证

首先本程序先是读取了矩阵的规模，之后严格按照该规模以行为单位进行数据的读取，直到运行到文件的结尾为止，这样的操作可以有效地避免数组越界的危险问题，从而保证程序运算的准确性。

另一方面，通过将程序进行矩阵乘法运算的结果与MATLAB工具进行矩阵乘法运算的结果进行比对，比对截图如下所示，图中以32 * 32 的矩阵运算为例，比对了程序运算的结果与matlab运算的结果进行比对，发现结果几乎一样，没有大的区别，区别可能是在进行double浮点数运算的时候带来的不可避免的精度误差。

```
1   6.027915e+04 6.733141e+04 5.663333e+04 6.081126e+04 7.337449e+04 6.670471e+04 7.197914e+04 7.142771e+04 5.736819e
2   6.278781e+04 7.780466e+04 7.642764e+04 7.371781e+04 9.096316e+04 7.412579e+04 9.606066e+04 7.347415e+04 6.672042e
3   5.834378e+04 6.766249e+04 5.732566e+04 5.746363e+04 7.832203e+04 5.799606e+04 7.632863e+04 6.230781e+04 6.334588e
4   6.511606e+04 7.344154e+04 7.033309e+04 6.117042e+04 8.106548e+04 7.229754e+04 7.713734e+04 6.735816e+04 6.949110e
5   7.329750e+04 8.832774e+04 7.679511e+04 8.111910e+04 9.537184e+04 8.573006e+04 9.695482e+04 8.324732e+04 7.636423e
6   7.062764e+04 7.483881e+04 6.782594e+04 7.183101e+04 8.941845e+04 7.755192e+04 8.657121e+04 7.798891e+04 6.460144e
7   7.004144e+04 8.649973e+04 7.413191e+04 7.202921e+04 1.017892e+05 8.103009e+04 9.079456e+04 8.254069e+04 7.724665e
8   7.192127e+04 8.066366e+04 7.584162e+04 6.432426e+04 9.049937e+04 8.023633e+04 8.582382e+04 7.194200e+04 7.494009e
9   5.635195e+04 5.871494e+04 6.012312e+04 5.830336e+04 7.424932e+04 6.082802e+04 7.274847e+04 6.669544e+04 5.926383e
10  7.664699e+04 8.539224e+04 7.411217e+04 7.030413e+04 8.472899e+04 7.796485e+04 8.449738e+04 8.048066e+04 7.037392e
11  6.265836e+04 6.814818e+04 7.290931e+04 5.831881e+04 7.831458e+04 7.680091e+04 8.108149e+04 7.867275e+04 6.012272e
12  5.745684e+04 6.645359e+04 5.835081e+04 5.358140e+04 7.032175e+04 6.714323e+04 6.379397e+04 7.318084e+04 5.703518e
13  6.384671e+04 7.767532e+04 6.977245e+04 5.682827e+04 8.222042e+04 8.182258e+04 7.656428e+04 7.716225e+04 6.706182e
14  5.269816e+04 6.366520e+04 6.349970e+04 4.678028e+04 7.328243e+04 6.972063e+04 6.537416e+04 5.393634e+04 6.099224e
15  7.418509e+04 9.575478e+04 7.093368e+04 8.056383e+04 9.898713e+04 8.914417e+04 9.488534e+04 8.464447e+04 7.300346e
16  7.217975e+04 8.133153e+04 8.156760e+04 6.449258e+04 9.264148e+04 8.779310e+04 9.261817e+04 8.230371e+04 7.383137e
17  7.382315e+04 7.552202e+04 7.649203e+04 6.236193e+04 8.569093e+04 7.802088e+04 8.440828e+04 7.359697e+04 7.542833e
18  8.323174e+04 7.538252e+04 8.916689e+04 6.483667e+04 9.378344e+04 8.890951e+04 9.481572e+04 8.435512e+04 7.952081e
19  6.833908e+04 8.112868e+04 7.174779e+04 5.757929e+04 8.052354e+04 7.843654e+04 7.438646e+04 7.210703e+04 7.009730e
20  6.303053e+04 7.515794e+04 7.069782e+04 5.675626e+04 8.301827e+04 8.142635e+04 6.853669e+04 7.920140e+04 6.404285e
21  7.823854e+04 9.584322e+04 8.661256e+04 8.157708e+04 9.895610e+04 9.390564e+04 9.459214e+04 8.843090e+04 8.737776e
22  7.903626e+04 9.080858e+04 7.926665e+04 7.944322e+04 9.536218e+04 8.314596e+04 8.919400e+04 8.812634e+04 8.150112e
23  7.338336e+04 8.393971e+04 8.295745e+04 7.081791e+04 9.365054e+04 8.741024e+04 8.712773e+04 8.105416e+04 7.704919e
```

```
1.0e+05 *

列 1 至 21

 0.6028   0.6733   0.5663   0.6081   0.7337   0.6670   0.7198   0.7143   0.5737   0.7711   0.7895   0.7674   0.8260   0.8128   0.6969   0.6321   0.6408   0.6722   0.8167   0.5944   0.6611
 0.6279   0.7780   0.7643   0.7372   0.9096   0.7413   0.9606   0.7347   0.6672   0.9285   0.8563   0.8448   0.9364   0.9294   0.8014   0.6839   0.6994   0.7873   0.8562   0.7171   0.7273
 0.5834   0.6766   0.5733   0.5746   0.7832   0.5800   0.7633   0.6231   0.6335   0.7910   0.7550   0.7124   0.7795   0.7463   0.5910   0.6581   0.5760   0.6302   0.7840   0.5905   0.6239
 0.6512   0.7344   0.7033   0.6117   0.8107   0.7230   0.7714   0.6736   0.6949   0.9190   0.8123   0.7558   0.9220   0.8439   0.6660   0.7676   0.6069   0.7472   0.8470   0.6293   0.7190
 0.7330   0.8833   0.7680   0.8112   0.9537   0.8573   0.9695   0.8325   0.7636   0.9625   0.9657   0.8787   0.9489   0.9684   0.8268   0.8119   0.7500   0.9753   0.9745   0.8098   0.7235
 0.7063   0.7484   0.6783   0.7183   0.8942   0.7755   0.8657   0.7799   0.6460   0.8396   0.8024   0.8466   0.8567   0.8613   0.8491   0.8030   0.7157   0.7963   0.8566   0.7627   0.6996
 0.7004   0.8650   0.7413   0.7203   1.0179   0.8103   0.9079   0.8254   0.7725   0.9995   0.9274   0.8322   0.9603   0.8770   0.8160   0.7354   0.8255   0.7903   0.9167   0.7833   0.7564
 0.7192   0.8066   0.7584   0.6432   0.9050   0.8024   0.8582   0.7194   0.7494   1.0093   0.8248   0.7361   0.9446   0.8963   0.7554   0.7430   0.6726   0.6786   0.8820   0.7080   0.6825
 0.5635   0.5871   0.6012   0.5830   0.7425   0.6083   0.7275   0.6670   0.5926   0.6966   0.7070   0.6867   0.7045   0.7166   0.6679   0.6324   0.5793   0.6774   0.7261   0.6638   0.6214
 0.7665   0.8539   0.7411   0.7030   0.8473   0.7796   0.8450   0.8048   0.7037   0.8970   0.9229   0.8499   0.9156   0.9305   0.8112   0.8606   0.6924   0.8947   0.9369   0.7197   0.7463
 0.6266   0.6815   0.7291   0.5832   0.7831   0.7680   0.8108   0.7867   0.6012   0.8132   0.7515   0.8004   0.7523   0.8289   0.8079   0.6700   0.6927   0.6949   0.7600   0.6213   0.7169
 0.5746   0.6645   0.5835   0.5358   0.7032   0.6714   0.6379   0.7318   0.5704   0.7650   0.7404   0.7112   0.7841   0.7850   0.7326   0.5661   0.6195   0.5714   0.7492   0.5664   0.6920
 0.6385   0.7768   0.6977   0.5683   0.8222   0.8182   0.7656   0.7716   0.6706   0.8415   0.8189   0.8279   0.8630   0.8962   0.8020   0.6736   0.8155   0.7376   0.8677   0.6981   0.7055
 0.5270   0.6367   0.6350   0.4678   0.7328   0.6972   0.6537   0.5394   0.6099   0.8226   0.6537   0.7638   0.7574   0.7212   0.6139   0.6238   0.5697   0.6503   0.7429   0.5052   0.5906
 0.7419   0.9575   0.7093   0.8056   0.9899   0.8914   0.9489   0.8464   0.7300   0.9420   0.9187   0.9295   1.0293   0.9675   0.8809   0.9299   0.8423   0.8825   0.9394   0.8094   0.7740
 0.7218   0.8133   0.8157   0.6449   0.9264   0.8779   0.9262   0.8230   0.7383   0.9579   0.8098   0.9064   0.9088   0.9009   0.8251   0.7601   0.7535   0.8874   0.8793   0.7091   0.8103
 0.7382   0.7552   0.7649   0.6236   0.8569   0.7802   0.8441   0.7360   0.7543   0.8885   0.8662   0.8223   0.8883   0.8644   0.8050   0.7821   0.6551   0.7435   0.8747   0.6721   0.7150
 0.8323   0.7538   0.8917   0.6484   0.9378   0.8891   0.9482   0.8436   0.7952   1.0416   0.8692   0.8367   0.9347   0.9564   0.8484   0.7885   0.7611   0.9164   0.9205   0.7382   0.8255
 0.6834   0.8113   0.7175   0.5758   0.8052   0.7844   0.7439   0.7211   0.7010   0.8921   0.8734   0.7959   0.8914   0.9210   0.7835   0.8053   0.7219   0.7710   0.9255   0.7412   0.7001
 0.6303   0.7516   0.7070   0.5676   0.8302   0.8143   0.6854   0.7920   0.6404   0.8198   0.7951   0.7025   0.8445   0.7859   0.8031   0.6926   0.7085   0.6495   0.7315   0.6241   0.6823
 0.7824   0.9584   0.8661   0.8158   0.9896   0.9391   0.9459   0.8843   0.8738   1.1023   1.0689   1.0163   1.0031   0.9898   0.8706   0.8754   0.7411   0.8936   0.9801   0.7294   0.8596
 0.7904   0.9081   0.7927   0.7944   0.9536   0.8315   0.8919   0.8813   0.8150   1.0062   1.0764   0.8695   1.0319   1.0121   0.9009   0.8188   0.8168   0.8151   1.0095   0.7894   0.7735
 0.7338   0.8394   0.8296   0.7082   0.9365   0.8741   0.8713   0.8105   0.7705   1.1040   0.9308   0.8802   0.9756   0.9378   0.8434   0.7277   0.7452   0.7533   0.9344   0.6995   0.7813
 0.6856   0.8632   0.6605   0.6546   0.8148   0.7881   0.8149   0.7610   0.7032   0.8922   0.8924   0.8897   0.8858   0.9292   0.7510   0.7296   0.6970   0.8080   0.9418   0.7138   0.7676
 0.8275   0.9422   0.9017   0.7904   0.9148   0.9343   0.9954   0.9200   0.7179   1.0455   0.9675   0.9274   1.0212   1.0828   0.9656   0.8176   0.7691   0.9138   0.9934   0.7825   0.7709
 0.7149   0.8354   0.6686   0.7958   0.8476   0.7090   0.9298   0.9269   0.6875   0.8824   0.9047   0.8308   0.8999   0.9851   0.9243   0.7285   0.7275   0.7541   0.9268   0.7959   0.7466
 0.7108   0.7858   0.7076   0.6405   0.8481   0.6755   0.8192   0.6678   0.6938   0.8374   0.8928   0.7590   0.8891   0.8583   0.7717   0.8031   0.6226   0.6987   0.8623   0.6751   0.6787
 0.6238   0.7000   0.6806   0.5593   0.7903   0.7330   0.8674   0.7684   0.6564   0.8563   0.8389   0.7346   0.7424   0.8718   0.7513   0.7317   0.6613   0.7000   0.8423   0.6415   0.7105
 0.7533   0.8181   0.8389   0.6120   0.8710   0.8297   0.8700   0.7436   0.6915   1.0003   0.9032   0.8336   0.9778   1.0377   0.8046   0.7216   0.6766   0.8138   0.9396   0.7031   0.7307
 0.7530   1.0285   0.8062   0.7775   1.0174   1.0022   0.9231   0.9518   0.8663   1.1016   1.0183   0.9824   1.0130   1.0452   0.8603   0.8889   0.8786   0.8648   1.0289   0.8584   0.8824
 0.8398   1.0643   0.8463   0.9497   1.1335   0.9802   1.0728   1.0237   0.9246   1.0791   1.1061   1.0178   1.1232   1.0930   0.9834   0.9672   0.9423   0.9142   1.0635   0.9226   0.9441
```

同时，将自己程序分别使用变长数组和定长数组进行实现，发现运算的结果都是完全一样的，这个比对和与MATLAB的比对结果进一步地说明了代码的正确性。

Part 4 - Difficulties & Solutions

文件读取失败及其处理

在进行文件读取的时候有时候会遇到文件不存在或者其他问题导致文件不能正常打开读取，这时候就需要进行流的状态检测，在本程序当中，我使用了`ifstream`对象的判断流状态的方法`is_open()`，如果文件读取失败就会发出错误信息，并与操作系统通信退出程序，如果文件读取成功，则会提示用户"Access file1 successfully"，进一步保证了用户能够知道文件是否正常打开的信息，以便做出下一步的决策。

数据样本点不够，以及随机数生成数据相同的问题及其处理

要客观地反映与刻画矩阵乘法运行时间与矩阵规模的关系以及文件读取与写入速度与矩阵规模之间的关系，仅凭提供的三个样本数据点是不够的，于是就面临着生成多个不同规模矩阵的问题，在解决这个问题的时候我就想到要使用上课提到的随机数算法生成三组不同规模的矩阵来增加样本点以及增强关系的说服力。

于是一开始我使用了`srand()`函数以及`rand()`函数来生成随机数，并且为了防止生成的随机数一致，还在`srand()`函数当中使用了`time_t`类型的变量`time`，以当前的时间作为生成随机数的"种子"来生成随机数，但是反而出现了生成所有数据完全一样的问题，经过分析，发现是在每一次使用`rand()`函数之前都紧跟着使用`srand()`函数，由于两次调用`rand()`的时间在一次循环里面是几乎一样的，所以便会在生成数据很大的范围之内产生的"随机数"的值都是一样的。

进过理解和分析，我将`srand()`函数放在了循环产生随机数的循环的外面，这样保证了每次生成随机数的时间都不一样，进而保证了生成矩阵数据的随机性，解决了数据样本点不够以及随机数生成数据相同的问题。

输出文件数字表示方式不一致的问题以及处理

在进行数据验证的过程当中发现个别结果矩阵的文件当中结果并不为浮点数的显示形式，即显示的数字并没有使用科学表示法进行表示，结合所学知识，根据cout对象可以使用cout.setf(ios_base::scientific, ios_base::floatfield)进行科学表示法的切换，推测ofstream对象fout具有同样的性质，也可以通过相同方式设置科学表示法，使用方法如下：

```
ofstream fout(argv[3]);
fout.setf(ios_base::scientific, ios_base::floatfield);
```

于是成功设置文件表示形式为科学表示法，表示方法成功统一为科学计数法。

电脑插电与不插电性能对程序运行的影响问题以及解决

在运行程序的时候，发现电脑在插上充电器时的矩阵乘法运算速度以及文件的读取效率都会提高不少，为了保持实验的单一变量原则，每次运行程序的时候，都会保证电脑自身的状态基本一致，尽量排除除了程序本身之外的因素对程序运行效率的影响。

指针定长二维数组的释放空间问题以及解决

发现如果创建了指针定长数组之后，使用完成不进行释放内存的话，在运行大规模的矩阵乘法的时候程序会占用非常大的内存空间，如果不释放，就有可能会造成内存泄漏。于是我在使用完成了数组之后就会及时地进行释放内存的操作，即进行delete[]的方法。

Part 5 - Thinking and Summarize

通过这次project，我比上一次project更加了解了float和double数值类型的实质以及使用，更加理解了它们在矩阵乘法当中对精确度以及运算速度的影响，也熟悉了C++当中对文件的读取以及写入的相关操作，懂得了如何从文件当中提取出目标的数据并进行处理运算。通过二维数组对矩阵的表示，我也进行了对可变长数组与定长数组在文件IO以及矩阵运算方面效率的探讨，通过上网以及查阅书籍更加了解了两种数组在读取文件以及CPU处理这两种数组方面的差异。在使用定长数组的过程中，也了解了c++的指针存储机制，并且懂得了在创建对象之后要及时地释放内存，不然可能会导致内存泄漏等等问题。通过研究矩阵乘法本身，我也进行了两次优化，第一次是对矩阵进行简单地分块并进行运算，但是这样并不能显著提高矩阵运算的效率，第二次优化我考虑到了三层循环当中最内层循环对第二个矩阵内存访问效率的影响，并交换了循环顺序，通过改变矩阵对内存的访问顺序，在时间复杂度不变的情况下，显著地提升了矩阵的运算效率。为了探究不同规模矩阵对文件读取写入速度以及矩阵乘法算法优化在不同规模矩阵下的体现，我还使用了c++的随机数生成器，产生了新的三组具有代表性的不同规模的矩阵，加上原来的三组矩阵，六组矩阵为所有的实验提供了更高的可信度以及普遍性。

以上是我报告的全部内容，感谢老师的阅读！