

课 程 设 计 报 告

课程名称 C++面向对象编程

课题名称 矩阵处理类库

班 级 无 42

学 号 2014011050

姓 名 陈佳榕

2015 年 7 月 28 日

任务书：

基本要求（必须满足）：（1）至少包括三个类，每个类要求具有复数个属性与方法。同时，还必须满足以下四个要求中的3个：（1）运算符重载与函数重载；（2）类的继承与抽象基类。（3）动态内存分配与文件操作；（4）使用链表/栈/队列等数据结构；

程序设计的基本要求：

（1）要求利用面向对象的方法以及 C++ 的编程思想来完成系统的设计；

（2）要求在设计的过程中，建立清晰的类层次；

（3）根据课题完成以下主要工作：①完成系统需求分析：包括系统设计目的与意义；系统功能需求（系统流程图）；输入输出的要求。②完成系统总体设计：包括系统功能分析；系统功能模块划分与设计（系统功能模块图）。③完成系统详细设计：数据文件；类层次图；界面设计与各功能模块实现。④系统调试：调试出现的主要问题，编译语法错误及修改，重点是运行逻辑问题修改和调整。⑤使用说明书及编程体会：说明如何使用你编写的程序，详细列出每一步的操作步骤。⑥关键源程序（带注释）

（4）自己设计测试数据，将测试数据存在文件中，通过文件来进行数据读写来测试。

（5）按规定格式完成课程设计报告，并在网络学堂上按时提交。

（6）不得抄袭他人程序、课程设计报告，每个人应体现自己的个性设计。

创新要求：

在基本要求达到后，可进行创新设计，如根据查找结果进行修改的功能。

矩阵处理类库

目 录

1.	系统需求分析	4
2.	系统总体设计	5
3.	系统详细设计	7
4.	系统调试	11
5.	测试结果、使用说明书及编程体会	12
6.	总结	17
附录：源程序清单及评分表		

1. 系统需求分析

对矩阵的处理包括矩阵的相加、相减、相乘、转置，方阵的求行列式、求逆以及求解实系数线性代数方程组。编写矩阵处理类库，使之能提供以下功能：

- (1) 矩阵相加：实现对矩阵的相加。
- (2) 矩阵相减：实现对矩阵的相减。
- (3) 矩阵相乘：实现对矩阵的相乘。
- (4) 矩阵转置：实现对矩阵的转置。
- (5) 求方阵的行列式：返回方阵的行列式。
- (6) 方阵求逆：实现对方阵的求逆，若方阵不存在逆则作出相应提示。
- (7) 求解向量：根据给出的方阵及常数向量，求出解向量，若方阵不可逆则作出相应提示。
- (8) 保存矩阵：保存矩阵的所有元素。

2. 系统总体设计

矩阵处理类库包含八个大的功能，分别是：矩阵相加、矩阵相减、矩阵相乘、矩阵转置、求方阵的行列式、方阵求逆、求解向量，保存矩阵。

在对矩阵进行相加时，用户只要用加号连接两个矩阵的名称（如 $A+B$ ）即可。

在对矩阵进行相减时，用户只要用减号连接两个矩阵的名称（如 $A-B$ ）即可。

在对矩阵进行相乘时，用户只要用乘号连接两个矩阵的名称（如 $A*B$ ）即可。

在对矩阵进行转置时，用户只要调用转置函数 `transposition()`（如 `A.transposition()`）即可。

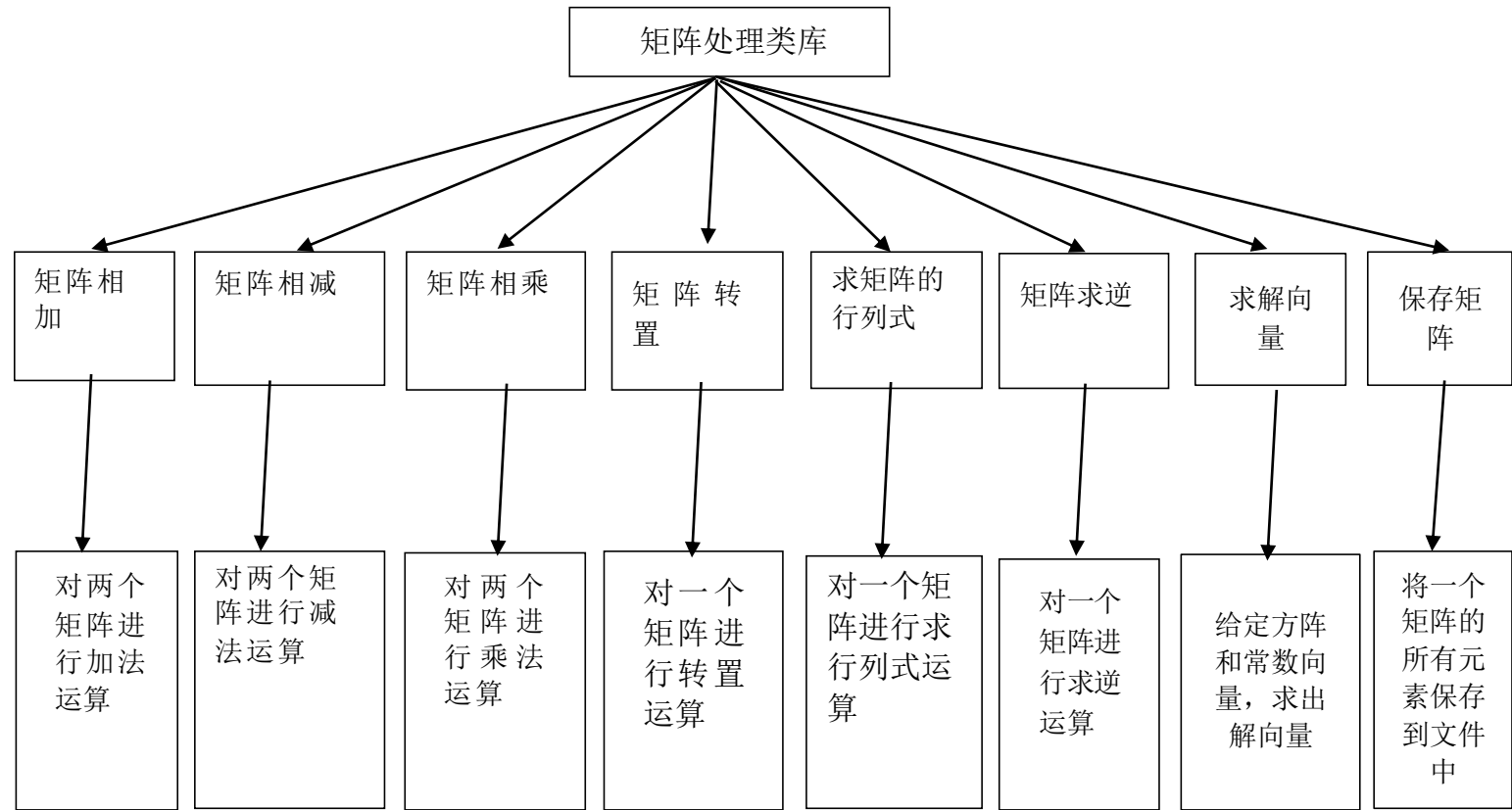
在求方阵的行列式时，用户只要调用求行列式函数 `det()`，以方阵名称为实参（如 `det(A)`）即可。若用户对不是方阵的矩阵求行列式则会引发错误，用户可以通过错误提示找到原因所在。

在对方阵求逆时，对于小矩阵，用户可以调用求逆函数 `inv_small_matrix()`，以方阵名称为实参（如 `inv_small_matrix(A)`）即可；对于较大规模的方阵，用户可以调用另一个求逆函数 `inv()`，以方阵名称为实参（如 `inv(A)`）即可。若用户对不是方阵的矩阵求逆，则会提示不能求逆。

在求解向量时，用户只要调用求解向量函数 `gauss_solve()`，以方阵名及常数向量名为实参（如 `gauss_solve(A,b)`）即可。若用户对不是方阵的矩阵求解向量则会引发错误，用户可以通过错误提示找到原因所在。

在保存矩阵时，用户只要调用保存函数 `Save_Matrix()`，以矩阵名为实参（`Save_Matrix(A)`）即可。若以后想查看，可在对应目录下找到“Matrix.dat”文件查看。

矩阵处理类库中功能模块图：



3. 系统详细设计

矩阵处理类库中三个类的类层次图为：

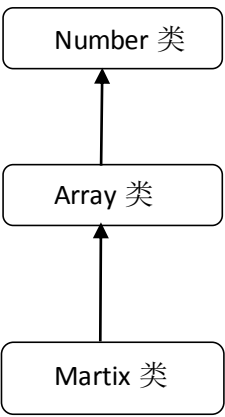
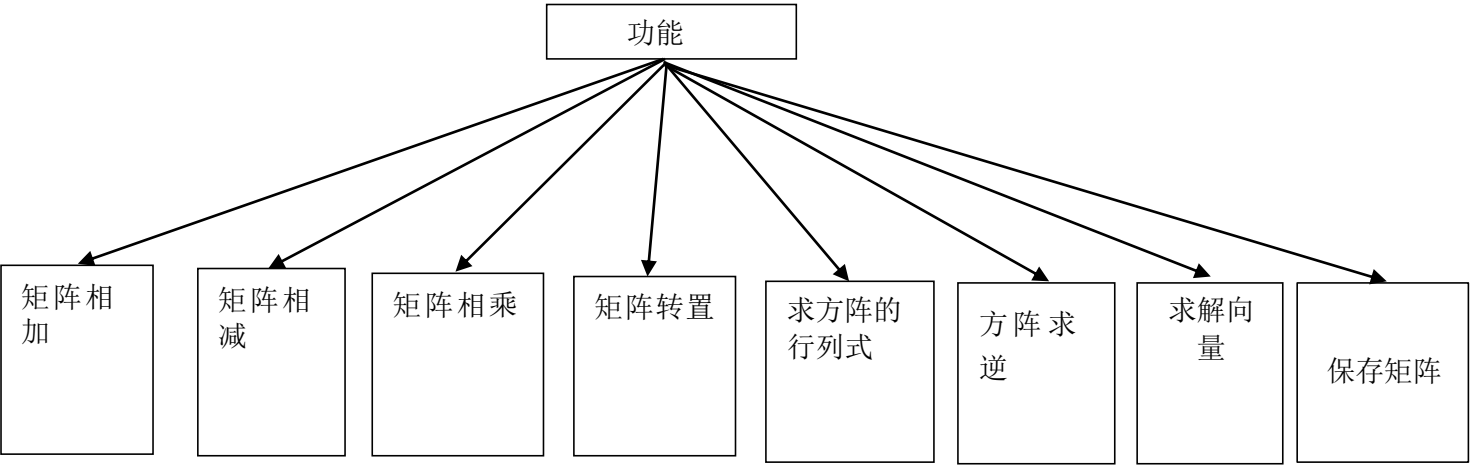


图 2 矩阵处理类库中三个类的类层次图

矩阵处理类库中各功能的实现：

图 3 矩阵处理类库中的功能图



1、矩阵相加功能：

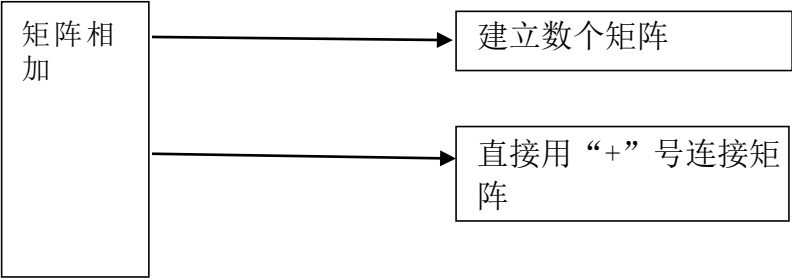


图 4 矩阵相加的功能图

2、矩阵相减的功能

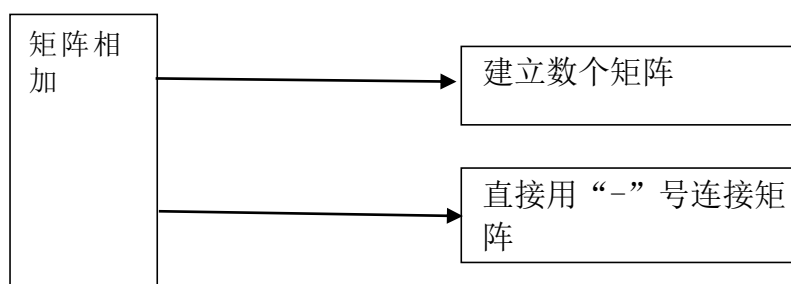


图 5 矩阵相减的功能图

3 矩阵相乘的功能：

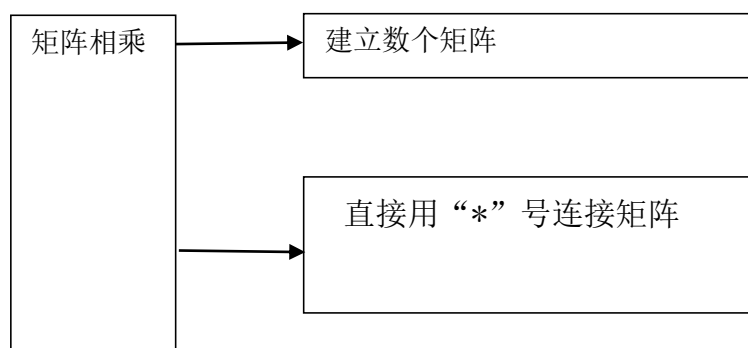


图 6 矩阵相乘的功能图

4、矩阵转置的功能：

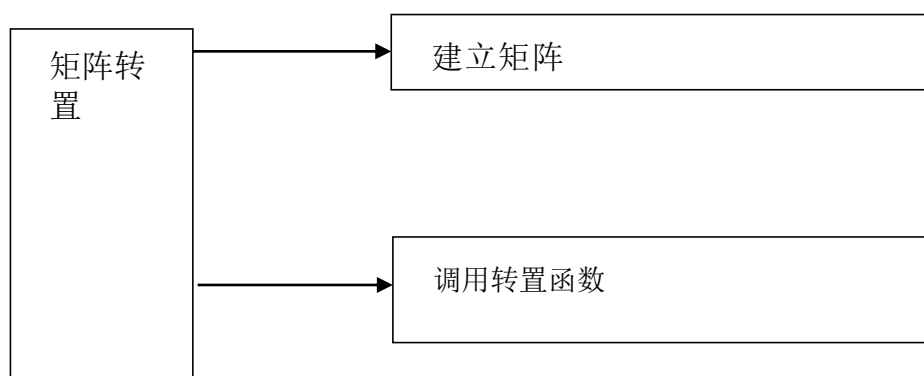


图 7 矩阵转置的功能图

5. 求方阵的行列式的功能：

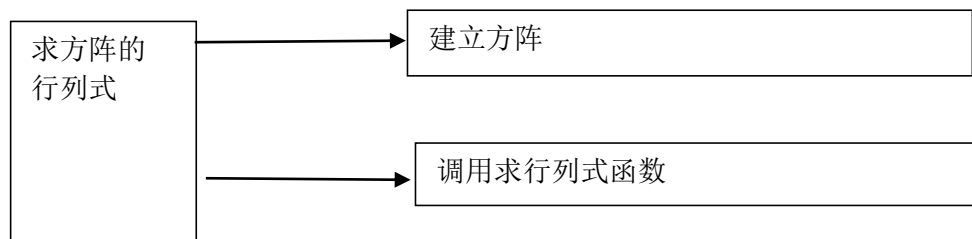


图 8 求方阵的行列式的功能图

6、方阵求逆的功能：

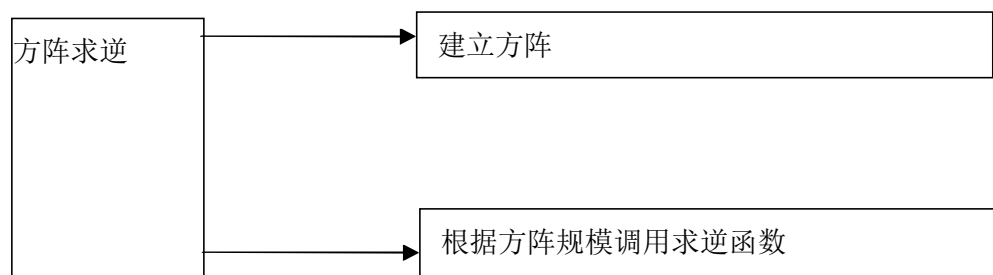


图 9 方阵求逆功能图

7、求解向量的功能：

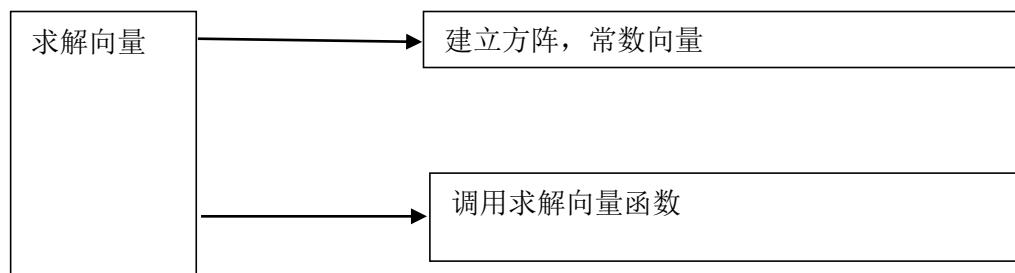


图 10 求解向量功能图

8、保存矩阵的功能

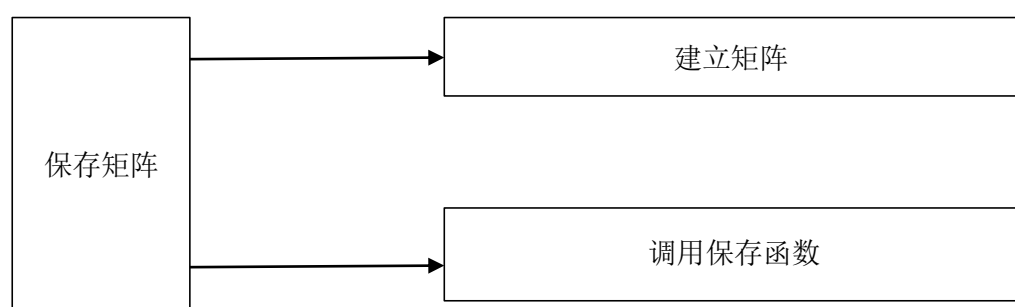


图 11 保存矩阵功能图

矩阵处理类库三个类的 UML 类图为：

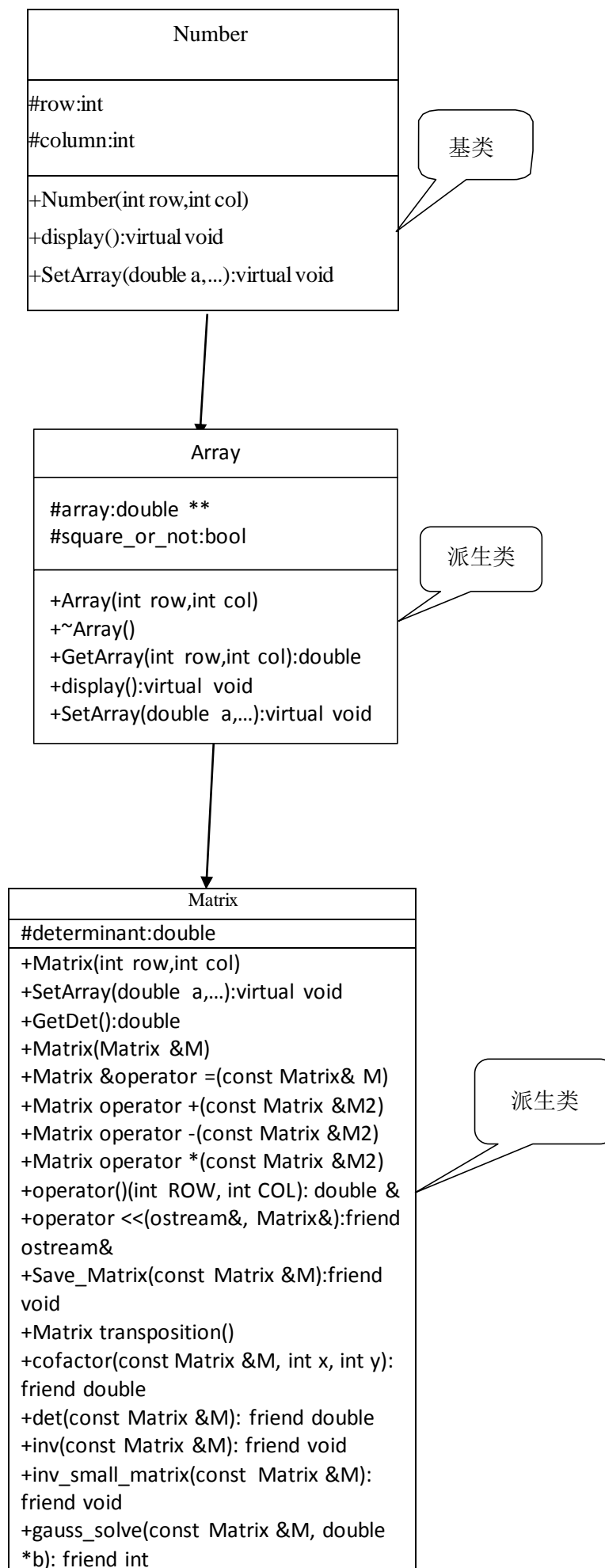


图 9

矩阵处理类库中三个类的UML 类图

4. 系统调试

程序调试过程中出现了以下问题：

①在写可变参数函数 `SetArray(double a, ...)` 的时候，一开始我用的不是 `double` 类型而是 `float` 类型，然后就总是有 bug。后来我试着换成 `double` 类型，然后就成功了。上网一查，原来是函数中用到的一个函数 `va_arg(ap, type)` 中的 `type` 不能为 `float` 类型。

②构建深拷贝构造函数时总是出错，后来得到重载 “=” 运算符函数的启示，在复制数组元素之前先释放原有的内存空间，然后再重新申请空间，这下问题顺利解决了。

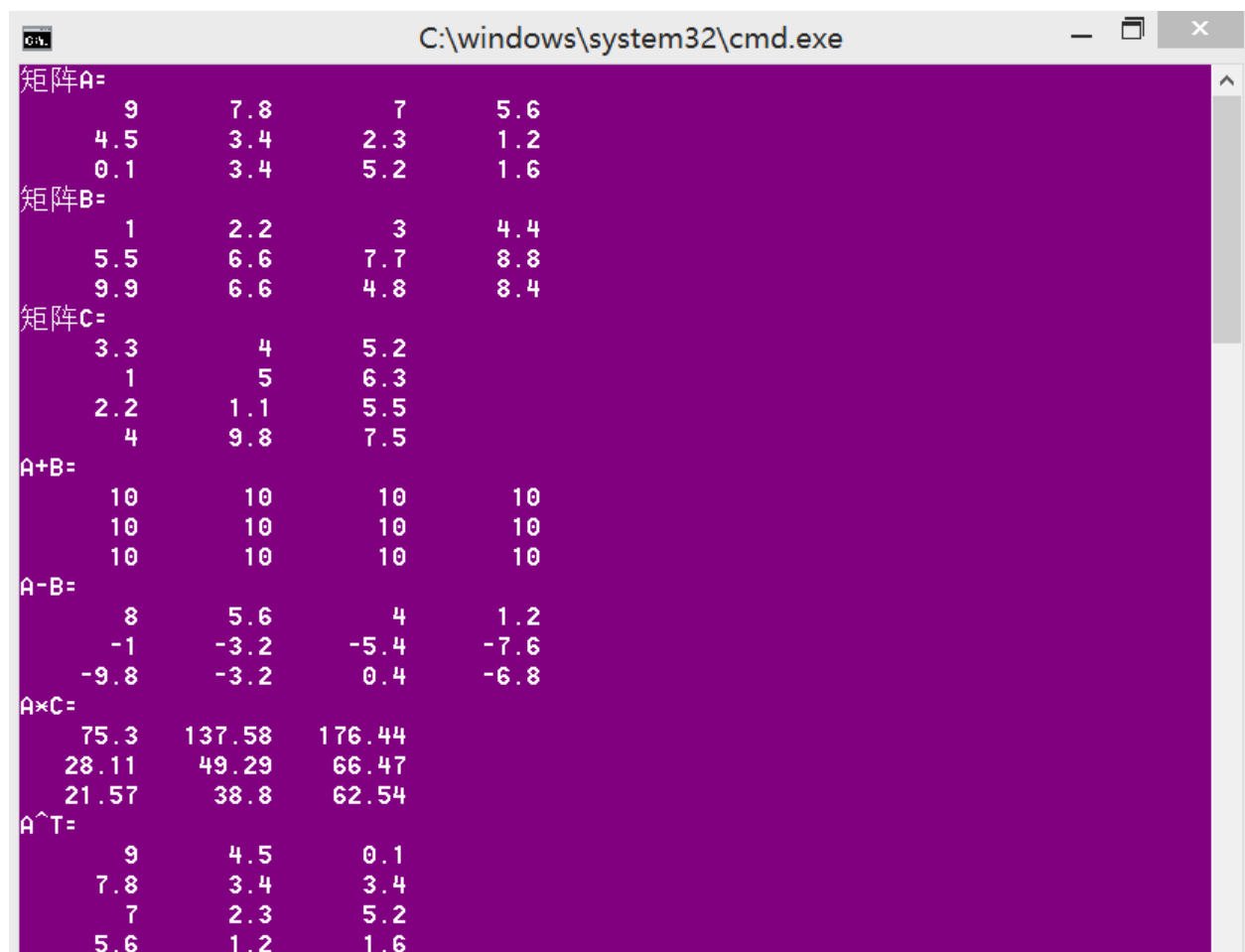
③一开始动态申请了一个对象后，删除这个对象的时候我用的是 `delete[] p`，可是却总是出 bug，后来我仔细想想，`delete[] p` 这种删除动态空间的方法应该是用于删除动态申请的数组的，要删除动态申请的对象应该用 `delete p`，改过来之后果然就正常了。

④一开始求逆函数我用的是克莱姆法则，可是运行的时候发现运行速度很慢。通过查找资料，发现是因为用克莱姆法则的话，当矩阵的规模较大时工作量大的难以容忍。所以我作了一点变动，只有当矩阵规模较小时才利用克莱姆法则进行求逆。然后又写了另一个求逆函数，利用的是全选主元高斯-若尔当消去法。

⑤测试结果出来后，我通过自己的计算，发现对于矩阵的求逆，结果出错了。重新思考之后，发现是在恢复矩阵时，原本应当将求逆过程中的所有行变换变成对逆矩阵的列变换，也应该将求逆过程中的所有列变换变成对逆矩阵的行变换，可是我却将求逆过程中的所有行变换变成对逆矩阵的行变换，所有列变换变成对逆矩阵的列变换，导致了结果错误。改正过来以后，结果通过了检验。

5.测试结果、使用说明书及编程体会

本程序的利用主程序文件“Main.cpp”作为测试文件，测试结果如下：



```
C:\windows\system32\cmd.exe
矩阵A=
  9      7.8      7      5.6
  4.5    3.4    2.3    1.2
  0.1    3.4    5.2    1.6
矩阵B=
  1      2.2      3      4.4
  5.5    6.6    7.7    8.8
  9.9    6.6    4.8    8.4
矩阵C=
  3.3      4      5.2
  1      5      6.3
  2.2    1.1    5.5
  4      9.8    7.5
A+B=
  10      10      10      10
  10      10      10      10
  10      10      10      10
A-B=
  8      5.6      4      1.2
 -1     -3.2     -5.4     -7.6
 -9.8    -3.2      0.4     -6.8
AxC=
  75.3  137.58  176.44
  28.11  49.29  66.47
  21.57  38.8   62.54
A^T=
  9      4.5      0.1
  7.8    3.4      3.4
  7      2.3      5.2
  5.6    1.2      1.6
```

```

矩阵F=
    1     2     3
    4     5     6
    7     8     9
对小矩阵F求逆，结果为：
矩阵不可逆！
矩阵G=
    2     5     3
    5     1     3
    8     9     7
对小矩阵G求逆，结果为：
矩阵的逆为：
   -1.25   -0.5    0.75
  -0.6875 -0.625   0.5625
   2.3125   1.375  -1.4375
矩阵M=
    1     2     3     4     5
    0     2     3     4     5
    0     0     4     5     6
    0     0     0     5     6
    0     0     0     0     2
矩阵M的行列式为： 80
对矩阵M求逆，结果为：
方阵的逆为：
    1     -1     -0     -0     -0
    0     0.5   -0.375 -0.025 -0.05
    0     0     0.25  -0.25    0
    0     -0    -0     0.2   -0.6
   -0     0     0     0     0.5
常数向量b=(3,5,4,5.5,7.3)
Mx=b的解向量为：x=(-2,0.4975,-0.375,-3.28,3.65)
请按任意键继续. . .

```

搜狗拼音输入法 全：

使用说明书:

首先,用户要在项目中加入头文件“Matrix.h”和cpp文件“Matrix.cpp”。

在利用矩阵处理类库进行矩阵处理时,需要遵循以下原则:

①使用一个矩阵之前都要先建立矩阵类对象,并给出所要建立矩阵的行数和列数,例如:“Matrix A(3,3);”,其中第一个3就是行数,第二个3就是列数。

②建立一个矩阵类对象之后,如果要给出矩阵各元素,需要调用成员函数 SetArray(double a, ...), 用户给出的实参个数要与矩阵所有元素个数一致,而且都必须是小数的形式,否则会出错。例子如下:

“A.SetArray(1.0,2.0,3.3,4.4,5.5,6.0,7.0,8.8,9.3);”。

③输出矩阵的时候,用户可以选择调用成员函数display(),如:“A.display();”,也可以直接输出,如:“cout<<A;”。

④进行矩阵加法时,用户要保证进行加法的两个矩阵的行数和列数是一致的。在此基础之上,用户只需将两个矩阵直接相加,然后将得到的结果赋给另一个同样大小的矩阵,或者直接输出。例子如:“C=A+B;”或者“cout<<A+B;”。

⑤进行矩阵减法法时,用户要保证进行减法的两个矩阵的行数和列数是一致的。在此基础之上,用户只需将两个矩阵直接相减,然后将得到的结果赋给另一个同样大小的矩阵,或者直接输出。例子如:“C=A-B;”或者“cout<<A-B;”。

⑥进行矩阵乘法时,用户要保证进行乘法的两个矩阵的大小是匹配的,即前一个矩阵的列数要和后一个矩阵的行数相等。在此基础之上,用户只需将两个矩阵直接相乘,然后将得到的结果赋给另一个大小匹配(即这个矩阵的行数等于第一个矩阵的行数,列数等于第二个矩阵的列数)的矩阵,或者直接输出。例子如:“C=A*B;”或者“cout<<A*B;”。

⑦进行矩阵转置时,用户只要调用成员函数 transposition(), 如:“A.transposition();”,可以将转置后的矩阵赋值给一个大小匹配的矩阵(指这个矩阵的行数和列数要分别和被转置的矩阵的列数和行数相等),也可以直接输出(如:“cout<<A.transposition();”。

⑧对矩阵求逆时,要保证矩阵为方阵,否则无法求逆,只能得到“矩阵不可逆”的提示。对较小规模的矩阵,可以调用函数 inv_small_matrix(), 如:“inv_small_matrix(A);”,则结果会自动显示,若矩阵不可逆则会显示“矩阵不可逆”,否则输出逆矩阵。虽然结果是精确的,但是由于这个函数的计算量较大,所以一般只用于较小规模的矩阵。当矩阵规模较大时,一般调用另外一个求逆函数 inv(), 如:“inv(A);”,结果同上一个求逆函数。

⑨求解向量时，用户要先定义常数向量，且要保证常数向量的长度与方阵匹配（即常数向量的长度应该与方阵的阶数相等），然后调用求解向量函数 `gauss_solve()`，若函数返回值不等于 0 则有解，如：“`gauss_solve(A,b)!=0;`”，此时常数向量实际上已经成为解向量了，通过查看向量 `b` 中各元素即可得到解向量。

⑩保存矩阵时，用户只需调用保存函数 `Save_Matrix()`，如“`Save_Matrix(A);`”，便可以将矩阵的所有元素保存到文件“`Matrix.dat`”中去，以后便可以在文件里查看矩阵的元素，不同矩阵的元素直接用“`#`”分开。

编程体会：

这是第二个程设大作业，是我第一次将如此多在数学课上学到的东西运用于编程之中，深深体会到编程的强大，原来编程也可以用来解决数学题，可谓解放脑力，我们完全可以将繁琐的数学计算交给计算机，而将更多的精力用于解决数学难题上。

我做得比较好的我想应该是以下几点：

①输入矩阵的元素，由于我用了可变参数函数，使得可以按照矩阵的大小去赋值，这个使得用起来方便了许多。

②对于程序需要的内存和时间开销考虑得比较多，尽量使内存和时间开销尽可能的少。

③还有做得比较好的是注释，关键的地方都作了注释。。

④此外，我用了比较多的动态申请空间，进一步让我熟悉了这些操作，消除畏惧感。至于不足之处，我想有以下几点：

①在求逆函数与求解向量之中出现了相似度较高的代码段，可能也是因为这两个函数用到的方法本来就有一定的关系。但因为中间有一些代码不一样，包含了比较特殊的部分，所以比较难分割出一个函数来。不过也许是可以完成的，只是受限于我的能力，这也对我以后的学习生涯中提出了要求，但这也是激励，激励着我去不断接收新的知识，强化自己的编程能力。

② 像对矩阵求逆，我只实现了对方阵求逆，虽然数学课上学过了不是方阵的矩阵也可以有左逆和右逆，可是因为求解过程比较难，我没能写出代码来。还有求解向量也是，而且也没有实现对不可逆矩阵的解向量的求解。不过困难总是催人上进，这让我想通过认真学习下学期的数据与算法这门课去找到解决的方法，相信问题总会有解决的一天的。

6. 总结

小学期的程设给了我一个综合运用所学知识的机会，也给我一个查漏补缺的机会。个人感觉相比上学期，程序设计的能力是有进步的。

矩阵处理类库要求至少要有三个类，一番思考后我确定了三个类，并确定了彼此之间的关系。不过在分配属性和方法上的能力还是不够。通过两个大作业，我更加意识到熟能生巧的道理，多多练习编程才能一步步提高自己的编程能力。而第二个作业不仅给了我们练习的机会，也给了我们充分创新的机会，可以按照自己所想充分利用所学到的知识。

第二个大作业，我是按照功能一个一个来写的，写完一个功能就会测试一遍，确保无误才会进行下一个功能的编写，所以可以及时解决问题，避免问题成山。同写第一个大作业的时候一样，一旦遇到问题，我会先看看是不是低级错误，如果是的话很容易解决，不是的话我会先自己思考问题所在，尽量自己解决，如果实在解决不了我会将错误原因通过百度查找，这确实很有帮助，一般都可以发现问题所在，而且很多其实是我们所不了解的，这是一箭双雕，不仅解决了自己的问题还学习到了新知识。此外，有时候想到要实现自己所要的功能，但凭借自己所掌握的知识却不能够解决时，比如我想要实现的可变参数函数，就需要在网上搜索自己想要的函数，然后根据网上对函数的具体介绍，稍作理解然后为己所用，这不失为获取新知识的好办法。

写这个作业的时候，有时候因为自己的粗心，造成某个地方写错了，可能仅仅是写错了一个变量名，虽然可以运行，可是结果却是错误的。这让我看到了检验结果的重要性，更感受到细心的重要，细心才能避免一些不必要的错误。

第二个大作业也让我看到了编程在解决数学，特别是与线性代数相关的问题上的巨大潜力，计算机凭借其巨大的计算能力，完全可以解决许多的数学问题，将我们从繁琐的计算中解救出来。而我也因此对于下学期将要学习的数据与算法这门课感到莫大的兴趣，相信这门课一定可以很有收获，也许学完再回头看今天的大作业，会有很不一样的感受。

总而言之，第二个大作业给我的收获就是编程的潜力是无比的巨大的，它绝对有潜力解决你想都不敢想的难题，这激发了我新一步学习的动力，燃起我编程的热情。学海无涯，努力去探索，终会发现编程之美。

附录：源程序清单及评分表

附录 1：源程序清单

```
//头文件Matrix.h存放各类的声明
#pragma once
#include <iostream>
using namespace std;
//抽象基类Number类
class Number
{
public:
    Number(int row = 0, int col = 0);
    virtual void display() const = 0;           //纯虚函数
    virtual void SetArray(double a, ...) = 0;
protected:
    int row;
    int column;
};

//派生类Array类
class Array :public Number
{
public:
    Array(int row = 1, int col = 1);
    ~Array();
    double GetArray(int row, int col) const;    //读取数组元素函数
    virtual void display() const;
    virtual void SetArray(double a, ...);       //可变参数函数
protected:
    double **array;
    bool square_or_not;
};

//派生类Matrix类
class Matrix :public Array
{
public:
    Matrix(int row = 1, int col = 1);
    virtual void SetArray(double a, ...);
    double GetDet();
    Matrix(Matrix &M);                          //拷贝构造函数
    Matrix &operator =(const Matrix& M);         //重载运算符函数声明
    Matrix operator +(const Matrix &M2);
    Matrix operator -(const Matrix &M2);
    Matrix operator *(const Matrix &M2);
```

```

        double & operator()(int ROW, int COL);
        friend ostream& operator <<(ostream&, Matrix&);
        friend void Save_Matrix(const Matrix &M);          //保存矩阵函数
        Matrix transposition();                             //声明转置函数
//声明求代数余子式函数
        friend double cofactor(const Matrix &M, int x, int y);
        friend double det(const Matrix &M);                //声明求行列式函数
        friend void inv(const Matrix &M);                  //声明求逆函数
//声明小方阵的求逆函数
        friend void inv_small_matrix(const Matrix &M);
//声明求解向量函数
        friend int gauss_solve(const Matrix &M, double *b);
protected:
        double determinant;
};

```

```

//Matrix.cpp存放各类方法的实现及各种函数的定义
#include <iostream>
#include <iomanip>
#include <cmath>
#include <fstream>
#include <stdarg.h>
#include <assert.h>
#include "Matrix.h"
using namespace std;

//Number类成员函数定义
Number::Number(int row, int col) :row(row), column(col) {}

//Array类成员函数定义
Array::Array(int row, int col) : Number(row, col)
{
    if (row == col)                                //判断是否方阵
        square_or_not = true;
    else
        square_or_not = false;
    array = new double*[row];                      //动态申请二维数组
    for (int i = 0; i < row; i++)
        array[i] = new double[col];
}
Array::~~Array()
{
    for (int i = 0; i < row; i++)                  //删除动态申请的二维数组
    {
        delete[] array[i];
    }
    delete[] array;
}
//读取二维数组第i+1行第j+1列的元素
double Array::GetArray(int i, int j) const
{
    assert(i >= 0 && i < row && j >= 0 && j < column);

    return array[i][j];
}
//打印二维数组所有元素
void Array::display() const
{

```

```

        for (int i = 0; i < row; i++)
        {
            for (int j = 0; j < column; j++)
            {
                cout << setw(8) << GetArray(i, j) << " ";
            }
            cout << endl;
        }
    }
//设置二维数组元素,采用可变参数
void Array::SetArray(double a, ...)
{
    array[0][0] = a;
    va_list argptr;
    va_start(argptr, a); //初始化变元指针
    for (int i = 0; i < row; i++)
        for (int j = 0; j < column; j++)
        {
            if (!(i == 0 && j == 0))
                array[i][j] = va_arg(argptr, double);
//返回下一个参数
        }
    va_end(argptr); //释放argptr
}

//Matrix类成员函数定义
Matrix::Matrix(int row, int col) :Array(row, col) {}
//设置矩阵元素,采用可变参数
void Matrix::SetArray(double a, ...)
{
    array[0][0] = a;
    va_list argptr;
    va_start(argptr, a); //初始化变元指针
    for (int i = 0; i < row; i++)
        for (int j = 0; j < column; j++)
        {
            if (!(i == 0 && j == 0))
                array[i][j] = va_arg(argptr, double);
//返回下一个参数
        }
    va_end(argptr); //释放argptr
    if (row == column)
        determinant = det(*this); //计算行列式
}
//返回行列式
double Matrix::GetDet()

```

```

{
    return determinant;
}
//深拷贝构造函数
Matrix::Matrix(Matrix &M)
{
    int former_row = row;
    row = M.row;                                //拷贝常规成员
    column = M.column;
    for (int i = 0; i < former_row; i++)        //释放原有的内存资源
    {
        delete[] array[i];
    }
    delete[] array;
    array = new double*[M.row];                , //分配新的内存资源
    for (int i = 0; i < M.row; i++)
        array[i] = new double[M.column];
    for (int i = 0; i < M.row; i++)            //复制数组元素
        for (int j = 0; j < M.column; j++)
            array[i][j] = M.array[i][j];
}
//重载运算符 “=”
Matrix& Matrix::operator =(const Matrix& M)
{
    assert(row == M.row);                      //检验两矩阵大小相同与否
    assert(column == M.column);

    if (this == &M)                            //检查自复制
        return *this;
    for (int i = 0; i < row; i++)                //释放原有的内存资源
    {
        delete[] array[i];
    }
    delete[] array;
    array = new double*[M.row];                //分配新的内存资源
    for (int i = 0; i < row; i++)
        array[i] = new double[M.column];
    for (int i = 0; i < M.row; i++)            //复制数组元素
        for (int j = 0; j < M.column; j++)
            array[i][j] = M.array[i][j];
    return *this;
}
//重载运算符 “+”
Matrix Matrix::operator +(const Matrix &M2)
{
    assert(row == M2.row);                    //检验两矩阵大小相同与否

```

```

    assert(column == M2.column);

    Matrix M(M2.row, M2.column);           //建立局部对象
    for (int i = 0; i < M2.row; i++)
        for (int j = 0; j < M2.column; j++)
            M.array[i][j] = array[i][j] + M2.array[i][j];
    return M;
}
//重载运算符“-”
Matrix Matrix::operator -(const Matrix &M2)
{
    assert(row == M2.row);                 //检验两矩阵大小相同与否
    assert(column == M2.column);

    Matrix M(M2.row, M2.column);           //建立局部对象
    for (int i = 0; i < M2.row; i++)
        for (int j = 0; j < M2.column; j++)
            M.array[i][j] = array[i][j] - M2.array[i][j];
    return M;
}
//重载运算符“*”
Matrix Matrix::operator *(const Matrix &M2)
{
    assert(column == M2.row);               //检验两矩阵能否相乘

    Matrix M(row, M2.column);               //建立局部对象
    for (int i = 0; i < row; i++)             //所有元素初始化为0
        for (int j = 0; j < M2.column; j++)
            M.array[i][j] = 0;
    for (int i = 0; i < row; i++)             //一行一行地计算出所有元素
        for (int k = 0; k < column; k++)
            if (array[i][k]) //array[i][j]为0时不用进行相乘操作
                for (int j = 0; j < M2.column; j++)
                    M.array[i][j] += array[i][k] *
M2.array[k][j];
    return M;
}
//重载运算符“()”，实现矩阵类多重下标访问
double & Matrix::operator()(int ROW, int COL)
{
    assert(ROW >= 0 && ROW < row);           //检验下标是否越界
    assert(COL >= 0 && COL < column);

    return array[ROW][COL];
}
//重载流插入运算符“<<”

```

```

ostream& operator <<(ostream& output, Matrix& M)
{
    for (int i = 0; i < M.row; i++)
    {
        for (int j = 0; j < M.column; j++)
        {
            cout << setw(8) << M.GetArray(i, j) << " ";
        }
        cout << endl;
    }
    return output;
}

void Save_Matrix(const Matrix &M) //保存矩阵函数
{
    ofstream outfile("Matrix.dat", ios::out | ios::app); //打开文件
    if (!outfile)
    {
        cerr << "保存矩阵时打开文件失败！" << endl;
        exit(1);
    }
    for (int i = 0; i < M.row; i++) //保存矩阵元素
        for (int j = 0; j < M.column; j++)
            outfile << M.GetArray(i, j) << " ";
    outfile << "#<<" "; //不同矩阵之间的分隔符
    outfile.close();
}

//定义转置函数
Matrix Matrix::transposition()
{
    Matrix M(column, row);
    for (int i = 0; i < column; i++)
        for (int j = 0; j < row; j++)
            M.array[i][j] = array[j][i];
    return M;
}

//求代数余子式函数
double cofactor(const Matrix &M, int x, int y)
{
    assert(M.square_or_not); //只有矩阵为方阵时进行计算才有意义

    int order = M.row; //矩阵的阶数
    int i = 0, j = 0;
    int ROW, COL;
    Matrix *p = new Matrix(order - 1, order - 1);
    //新建一个矩阵，存放除去M.array[x][y]元素所在的行与列后剩下的元素
    for (i = 0, ROW = 0; i < order; i++)

```



```

        if (i != x)
        {
            for (j = 0, COL = 0; j < order; j++)
                if (j != y)
                    p->array[ROW][COL++] = M.array[i][j];
            ROW++;
        }
        double cofactor = pow(-1, x + y)*det(*p);    //计算代数余子式
        delete p;
        return cofactor;
    }
//定义求行列式函数
double det(const Matrix &M)
{
    assert(M.square_or_not);           //方阵才有行列式

    int order = M.row;                 //方阵阶数
    switch (order)
    {
        case 1: return M.array[0][0]; break;
        case 2: return (M.array[0][0] * M.array[1][1] - M.array[0][1] *
M.array[1][0]); break;
        default:
        {
            double determinant = 0;
            for (int i = 0; i < order; i++)
                if (M.array[0][i])
                    determinant += M.array[0][i] * cofactor(M, 0,
i);           //与求行列式函数相互递归调用
            return determinant;
        }
    }
}
//方阵求逆函数, 选用全选主元高斯-若尔当消去法
void inv(const Matrix &M)
{
    assert(M.square_or_not);           //只适合于方阵的求逆

    if (M.determinant == 0)
    {
        cerr << "矩阵不可逆!" << endl;
    }
    else
    {
        int order = M.row;
        Matrix *A = new Matrix(order, order);

```

```

*A = M;
int *change_col = new int[order]; //记忆列交换信息
int *change_row = new int[order]; //记忆行交换信息
int i, j, k;
double pivot, tmp;
for (k = 0; k < order; k++)
{
    pivot = 0.0;
    for (i = k; i < order; i++) //全选主元
        for (j = k; j < order; j++)
        {
            tmp = fabs((*A)(i, j));
            if (tmp > pivot)
            {
                pivot = tmp;
                change_row[k] = i; //记忆行交换信息
                change_col[k] = j; //记忆列交换信息
            }
        }
    if (change_row[k] != k)
        for (j = 0; j < order; j++) //进行行交换
        {
            tmp = (*A)(k, j);
            A->array[k][j] = (*A)(change_row[k], j);
            A->array[change_row[k]][j] = tmp;
        }
    if (change_col[k] != k)
        for (i = 0; i < order; i++) //进行列交换
        {
            tmp = (*A)(i, k);
            A->array[i][k] = (*A)(i, change_col[k]);
            A->array[i][change_col[k]] = tmp;
        }
    A->array[k][k] = 1.0 / (*A)(k, k); //归一化计算
    for (j = 0; j < order; j++) //归一化计算
        if (j != k)
            A->array[k][j] = ((*A)(k, j)) / ((*A)(k, k));
    for (i = 0; i < order; i++) //消元计算
        if (i != k)
            for (j = 0; j < order; j++)
                if (j != k)
                    A->array[i][j] = (*A)(i, j) -
                        ((*A)(i, k)) * ((*A)(k, j));
    for (i = 0; i < order; i++) //消元计算
        if (i != k)

```

```

        A->array[i][k] = -((*A)(i, k))*((*A)(k,
k));
    }
    for (k = order - 1; k >= 0; k--)          //恢复所得的逆阵
    {
        if (change_col[k] != k)                //恢复行，进行行交换
            for (j = 0; j < order; j++)
            {
                tmp = (*A)(k, j);
                A->array[k][j] = (*A)(change_col[k], j);
                A->array[change_col[k]][j] = tmp;
            }
        if (change_row[k] != k)                //恢复列，进行列交换
            for (i = 0; i < order; i++)
            {
                tmp = (*A)(i, k);
                A->array[i][k] = (*A)(i, change_row[k]);
                A->array[i][change_row[k]] = tmp;
            }
    }
    cout << "方阵的逆为：" << endl;
    cout << *A;
    delete A;
    delete[] change_col;
    delete[] change_row;
}
}
//定义小矩阵的求逆函数
void inv_small_matrix(const Matrix &M)
{
    assert(M.square_or_not);                  //只适合于方阵的求逆

    if (M.determinant == 0)
    {
        cerr << "矩阵不可逆！" << endl;
    }
    else
    {
        int order = M.row;
        Matrix *inv_M = new Matrix(order, order);
        for (int i = 0; i < order; i++)        //采用伴随矩阵的方法求逆
            for (int j = 0; j < order; j++)
                inv_M->array[i][j] = cofactor(M, j, i) /
M.determinant;
        cout << "矩阵的逆为：" << endl;
        cout<<*inv_M;
    }
}

```

```

        delete inv_M;
    }
}
//利用全选主元高斯消去法求解实系数线性代数方程组
int gauss_solve(const Matrix &M, double *b)
{
    if (M.square_or_not == false)
    {
        cerr << "求解失败，此矩阵不是方阵。" << endl;
        return 0;
    }
    else
    {
        int order = M.row;
        Matrix *A = new Matrix(order, order);
        *A = M;
        int *change_col = new int[order];
//动态申请空间以存放列交换信息
        int change_row; //记忆行交换
        int i, j, k;
        int judge_singular = 1; //方阵奇异与否标志
        double pivot, tmp;
        for (k = 0; k < order - 1; k++)
        {
            pivot = 0.0;
            for (i = k; i < order; i++) //全选主元
                for (j = k; j < order; j++)
                {
                    tmp = fabs((*A)(i, j));
                    if (tmp > pivot)
                    {
                        pivot = tmp;
                        change_col[k] = j; //记忆列交换信息
                        change_row = i; //记忆行交换信息
                    }
                }
            if (pivot == 0.0) //方阵奇异
                judge_singular = 0;
            else
            {
                if (change_col[k] != k)
                    for (i = 0; i < order; i++) //进行列交换
                    {
                        tmp = (*A)(i, k);
                        A->array[i][k] = (*A)(i,
change_col[k]);

```

```

        A->array[i][change_col[k]] = tmp;
    }
    if (change_row != k)
    {
        for (j = k; j < order; j++) //进行行交换
        {
            tmp = (*A)(k, j);
            A->array[k][j] = (*A)(change_row,
j);
            A->array[change_row][j] = tmp;
        }
        tmp = b[k];
        b[k] = b[change_row];
        b[change_row] = tmp;
    }
}
if (judge_singular == 0) //方阵奇异
{
    delete A;
    delete[] change_col;
    cerr << "失败，方阵不可逆。" << endl;
    return 0;
}
else
{
    pivot = (*A)(k, k);
    for (j = k + 1; j < order; j++) //归一化操作
        A->array[k][j] = (*A)(k, j) / pivot;
    b[k] = b[k] / pivot;
    for (i = k + 1; i < order; i++) //消元操作
    {
        for (j = k + 1; j < order; j++)
            A->array[i][j] = (*A)(i, j) -
(((*A)(i, k))*((*A)(k, j)));
        b[i] = b[i] - ((*A)(i, k))*b[k];
    }
}
}
pivot = (*A)(order - 1, order - 1);
if (fabs(pivot) == 0.0) //方阵奇异
{
    delete A;
    delete[] change_col;
    cerr << "失败，方阵不可逆。" << endl;
    return 0;
}
}

```

```

        b[order - 1] = b[order - 1] / pivot; //解向量的最后一个元素
        for (i = order - 2; i >= 0; i--) //回代操作
        {
            tmp = 0.0;
            for (j = i + 1; j < order; j++)
                tmp += (*A)(i, j)*b[j];
            b[i] = b[i] - tmp;
        }
        change_col[order - 1] = order - 1;
        for (k = order - 1; k >= 0; k--) //恢复解向量
            if (change_col[k] != k) //进行行交换
            {
                tmp = b[k];
                b[k] = b[change_col[k]];
                b[change_col[k]] = tmp;
            }
        delete A;
        delete[] change_col;
        return 1; //正常返回
    }
}

```

//主程序文件Main.cpp, 进行测试

```

#include <iostream>
#include <stdio.h>
#include "Matrix.h"
using namespace std;
int main()
{
    system("color 5F");
    Matrix A(3, 4);
    A.SetArray(9.0, 7.8, 7.0, 5.6, 4.5, 3.4, 2.3, 1.2, 0.1, 3.4, 5.2,

```

```

1.6);
    cout << "矩阵A=" << endl;
    A.display();
    Matrix B(3, 4);
    B.SetArray(1.0, 2.2, 3.0, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9, 6.6, 4.8,
8.4);
    cout << "矩阵B=" << endl;
    B.display();
    Matrix C(4, 3);
    C.SetArray(3.3, 4.0, 5.2, 1.0, 5.0, 6.3, 2.2, 1.1, 5.5, 4.0, 9.8,
7.5);
    cout << "矩阵C=" << endl;
    C.display();
    Matrix D(3, 4);
    D = A + B;
    cout << "A+B=" << endl << D;
    D = A - B;
    cout << "A-B=" << endl << D;
    Matrix E(3, 3);
    E = A*C;
    cout << "A*C=" << endl << E;
    Matrix AT(4, 3);
    AT = A.transposition();
    cout << "A^T=" << endl << AT;
    Matrix F(3, 3);
    F.SetArray(1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0);
    cout << "矩阵F=" << endl << F;
    cout << "对小矩阵F求逆, 结果为: " << endl;
    inv_small_matrix(F);
    Matrix G(3, 3);
    G.SetArray(2.0, 5.0, 3.0, 5.0, 1.0, 3.0, 8.0, 9.0, 7.0);
    cout << "矩阵G=" << endl << G;
    cout << "对小矩阵G求逆, 结果为: " << endl;
    inv_small_matrix(G);
    Matrix M(5, 5);
    M.SetArray(1., 2., 3., 4., 5.,
        0., 2., 3., 4., 5.,
        0., 0., 4., 5., 6.,
        0., 0., 0., 5., 6.,
        0., 0., 0., 0., 2.);
    cout << "矩阵M=" << endl << M;
    cout << "矩阵M的行列式为: " << det(M) << endl;
    cout << "对矩阵M求逆, 结果为: " << endl;
    inv(M);
    Save_Matrix(M);
    double b[5] = { 3., 5., 4., 5.5, 7.3 };

```

```

cout << "常数向量b=";
for (int i = 0; i < sizeof(b) / sizeof(b[0]) - 1; i++)
    cout << b[i] << ", ";
cout << b[sizeof(b) / sizeof(b[0]) - 1] << ")" << endl;
if (gauss_solve(M, b) != 0)
{
    cout << "Mx=b的解向量为:x=";
    for (int i = 0; i < sizeof(b) / sizeof(b[0]) - 1; i++)
        cout << b[i] << ", ";
    cout << b[sizeof(b) / sizeof(b[0])-1] << ")" << endl;
}
system("pause 0");
return 0;
}

```

附录 2：评分表

第 2 题目评分标准：

项 目	评 价	
选题报告与设计说明书	1	包括选题报告与 结题报告两方面
程序基本要求涵盖情况	4	包括基本要求与 基本工作量 (500 行左右)
程序扩展要求与创新	3	包括扩展要求与 同学自己附加的 工作
程序代码编写素养情况	2	代码结构与风格
设计与运行结果	4	运行情况与鲁棒 性
综合成绩	15	总分

教师签名： _

日 期： _