Project3: A Libray for Matrix Operations in C

- 作者：罗嘉诚（Jiacheng Luo）
- 学号：12112910

1 需求分析

结合 `Project3` 给出的说明文档 `Project3.pdf` 我们分析本次项目需要实现的矩阵运算库的功能。

1.1 编程语言与项目结构

在说明文档中，`Requirement 1` 如下：

> The programming language can only be C, not C++. Please save your source code into `*.c` files, and compile them using a C compiler such as gcc (not g++). Try to use Makefile or CMake to manage your source code.

从项目关于编程语言与项目结构的要求中，我们归纳出如下几点：

- 编程语言只能是 C 语言，不能是C++语言，将代码保存为`*.c`格式，只能使用 C 编译器进行编译。
- 使用 `Makefile` 或者 `CMake` 来组织项目。

1.2 矩阵存储

在说明文档中，`Requirement 2` 如下：

> Design a `struct` for matrices, and the `struct` should contain the data of a matrix, the number of columns, the number of rows, etc.

在存储矩阵时，应当采用`struct` 来进行存储，在结构体中描述这个矩阵的所有信息，如行数、列数、元素。

1.3 矩阵元素

在说明文档中，`Requirement 3` 如下：

> Only `float` elements in a matrix are supported. You do not need to implement some other types.

矩阵中的各个元素只需要支持 `float` 即可，不需要支持其他类型。

1.4 支持函数

在说明文档中，`Requirement 4` 如下：

Implement some functions to

- create a matrix, `createMatrix()`.
- delete a matrix, `deleteMatrix()`.
- copy a matrix (copy the data from a matrix to another), `copyMatrix()`.
- add two matrices, `addMatrix()`.
- subtraction of two matrices, `subtractMatrix()`
- add a scalar to a matrix.
- subtract a scalar from a matrix.
- multiply a matrix with a scalar.
- multiply two matrices.
- find the minimal and maximal values of a matrix.
- some other functions needed

支持矩阵的一系列运算，使之能成为一个方便使用的矩阵库，至少需要支持：

- 创建矩阵：使用多种方法创建矩阵，便于用户选择最适合使用情景的矩阵。
- 删除矩阵：安全、高效、高鲁棒性地删除矩阵，不会由于用户操作使得库崩溃。
- 拷贝矩阵：使用地址拷贝、值拷贝两种方式，便于用户选择最合适使用情景的拷贝方式。
- 矩阵加减矩阵：参考广播机制，使得矩阵加减操作更加灵活。
- 矩阵加减标量：矩阵每个位置都加上或者减去一个标量。
- 矩阵乘法：正确、安全地计算矩阵乘法。
- 矩阵中最小数和最大数：正确、安全地计算矩阵中最小数和最大数。
- 其他需要的函数：多样化的其他函数支持，提高用户的使用体验。

在本矩阵库的实现中，支持丰富的函数，重点都在于优化用户的使用体验。

- 创建矩阵，函数命名都采用`createMatrix...`进行命名，统一格式，便于用户寻找和使用。

支持生成数值填充值矩阵 矩阵（包括全 0 矩阵、全 1 矩阵、全空值矩阵、全特定值矩阵、全随机值矩阵）、单位对角 矩阵（包括任意形状、方阵情况两种矩阵）、特殊数据类型填充矩阵 矩阵（可以采用数组填充、字符串填充、步长填充、文件填充）。

- 删除矩阵，采用一些办法保证其安全和高效。
- 拷贝矩阵，支持 值拷贝、地址拷贝 、子矩阵拷贝 三种模式。
- 矩阵加减矩阵，同时还支持乘、除、取模运算（以及自定义二元函数计算），都采用元素对元素 进行操作，并且采用广播机制。
- 矩阵加减标量，同时还支持乘、除、取模运算（以及对自定义的一元函数计算）。
- 矩阵乘法，采用一些办法保证其安全和高效。
- 矩阵中最小数和最大，采用一些办法正确、安全地计算矩阵中最小数和最大数。
- 其他所需要的函数：矩阵打印 矩阵求大小 矩阵求对应位置元素值 矩阵修改对应元素值 矩阵求元素和 矩阵迭代器（包括开始迭代器，结束迭代器，所有元素迭代器） 矩阵形状改变 矩阵求转置 矩阵快速幂 矩阵水平拼接 矩阵竖直拼接 矩阵求秩 矩阵求逆元 等函数。

1.5 用户体验

在说明文档中，`Requirement 5` 如下：

> The designed functions should be safe and easy to use. Suppose you are designing a library for others to use. You do not need to focus on the optimization in this project, ease of use is more important.

项目要求制作的库有良好的用户体验，包括但不限于：

- 要求所有函数的安全，不发生内存泄漏，有良好的错误反馈机制。
- 不要求任何优化，但要求易于使用。

在本项目中，所有函数都是安全的，不会有额外的内存开销，任何处理时被检测出的错误都会被库所捕获，并且进行反馈。项目所实现的函数接口个数和名称都很合适，也便于使用。

2 矩阵存储与打印

2.1 存储

本项目中，矩阵使用 `_Matrix` 类（`struct _Matrix`）来存储，其中有关键属性值：

- `int rows` 矩阵的行数。
- `int cols` 矩阵的列数。
- `float *head_pointer` 矩阵元素的首指针。

这意味着类中各元素的值，都是被连续地存放在 `head_pointer[0] - head_pointer[rows * cols - 1]` 中。

这样的设计非常自然，既能高效地组织数据，又有一定地扩展性。

如果使用 `Matrix` 用来 `typedef` 代表 `struct _Matrix` 那么可以在后续地编程中更方便一些。

```
typedef struct _Matrix {
  int rows, cols;
  float *head_pointer;
} Matrix;
```

对于矩阵：$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$，存储为矩阵 `matrix`

存储时，`Matrix` 中各变量的值分别为：

- `rows = 2` `cols = 3`
- `head_pointer = 0x55a2bcbe02c0 (地址值)`

若我们输出 `head_pointer[0] ... head_pointer[5]` 各值，将会在控制台输出：

```
1.000 2.000 3.000 4.000 5.000 6.000
```

2.2 打印

矩阵的打印采用函数 `void showMatrix(const Matrix *matrix);` 来实现。

用户只需要输入一个矩阵的首地址`matrix`，将会在终端打印：`矩阵的长宽`、`矩阵各元素的值`。

需要注意的是，此处我们定义了元素输出格式变量：`const char *kPrintFloatFormat = "%.3f\t";`

您如果需要更多的小数位数或者不同的格式，可以在矩阵元素打印格式中修改。

在默认情况下，将采用`保留 3 位小数` 以及 `\t` （制表符）分隔、`每行结束换行`的格式输出。

如果我们在默认情况下，输出上述矩阵，调用`showMatrix(matrix);` 在控制台将输出：

```
MatrixSize = [2, 3]
1.000    2.000    3.000
4.000    5.000    6.000
```

2.3 输出到文件

矩阵除了可以进行程序运行过程中的软保存，还可以通过生成`文本文件`的方式进行硬保存。

使用函数 `void printMatrixToFile(const Matrix *matrix, const char *file_name);` 其中 `matrix` 表示需要存储的矩阵，`file_name` 表示需要保存的文件地址。

test.c U

project03 > src > test.c

```c
#include "matrix.h"
int main() {
  Matrix *a = createMatrixString("1 2 3;4 5 6");
  printMatrixToFile(a, "a.txt");
  return 0;
}
```

a.txt U

project03 > src > a.txt

```
1.000    2.000    3.000
4.000    5.000    6.000
```

3 创建新矩阵

库中采用`createMatrix...`类似的函数名称实现了创建新矩阵的若干方法，便于用户针对自身实际情况选用。

注意，创建新矩阵的元素总个数不得超过设置值`kMatrixMaxNum` （默认为 `1e7`）。

3.1 数值填充值矩阵

其中用数值填充矩阵的方式产生新矩阵的函数有：

- `Matrix *createMatrix(const int rows, const int cols, const double value);`

  其中 `rows` 表示需要构建矩阵的行数，`cols` 表示需要构建矩阵的列数，`value` 表示填充的值。

```c
#include "matrix.h"
int main() {
    Matrix *a = createMatrix(3, 4, -3.14);
    showMatrix(a);
    return 0;
}
```

问题 输出 调试控制台 终端 JUPYTER

```
MatrixSize = [3, 4]
-3.140  -3.140  -3.140  -3.140
-3.140  -3.140  -3.140  -3.140
-3.140  -3.140  -3.140  -3.140
```

- `Matrix *createMatrixZeros(const int rows, const int cols);`

  填充全 0 矩阵，其中 `rows` 表示需要构建矩阵的行数，`cols` 表示需要构建矩阵的列数。

```c
#include "matrix.h"
int main() {
    Matrix *a = createMatrixZeros(3, 4);
    showMatrix(a);
    return 0;
}
```

问题 输出 调试控制台 终端 JUPYTER

```
MatrixSize = [3, 4]
0.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000
```

- `Matrix *createMatrixOnes(const int rows, const int cols);`

  填充全 1 矩阵，其中 `rows` 表示需要构建矩阵的行数，`cols` 表示需要构建矩阵的列数。

```c
#include "matrix.h"
int main() {
    Matrix *a = createMatrixOnes(3, 4);
    showMatrix(a);
    return 0;
}
```

问题 输出 调试控制台 终端 JUPYTER

```
MatrixSize = [3, 4]
1.000   1.000   1.000   1.000
1.000   1.000   1.000   1.000
1.000   1.000   1.000   1.000
```

- `Matrix *createMatrixEmpty(const int rows, const int cols);`

  填充默认值（操作系统不同而不同），`rows` 表示需要构建矩阵行数，`cols` 表示需要构建矩阵列数。

```c
#include "matrix.h"
int main() {
    Matrix *a = createMatrixEmpty(3, 4);
    showMatrix(a);
    return 0;
}
```

问题 输出 调试控制台 终端 JUPYTER

```
MatrixSize = [3, 4]
0.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000
```

- `Matrix *createMatrixRandom(const int rows, const int cols);`

  矩阵各个元素填充$[0,1)$的随机数，`rows` 表示需要构建矩阵行数，`cols` 表示需要构建矩阵列数。

```c
#include "matrix.h"
int main() {
    Matrix *a = createMatrixRandom(3, 4);
    showMatrix(a);
    return 0;
}
```

问题 输出 调试控制台 终端 JUPYTER

```
MatrixSize = [3, 4]
0.815   0.293   0.234   0.936
0.586   0.550   0.139   0.447
0.372   0.108   0.249   0.278
```

3.2 单位对角矩阵

我们将大小为 $m \times n$ 的矩阵 A 叫做单位对角矩阵，这个矩阵满足：

$$A[i][j] = \begin{cases} 0 & (i \neq j) \\ 1 & (i = j) \end{cases} \quad \forall i,j \quad \text{s.t.} \quad 1 \leq i \leq m, 1 \leq j \leq n$$

该矩阵在线性代数中，有重要作用，采用两个函数来生成这个矩阵：

- `Matrix *createMatrixEye(const int rows, const int cols);`

  其中 `rows` 表示需要构建矩阵的行数，`cols` 表示需要构建矩阵的列数。

```c
#include "matrix.h"
int main() {
    Matrix *a = createMatrixEye(3, 4);
    showMatrix(a);
    return 0;
}
```

问题 输出 调试控制台 终端 JUPYTER

```
MatrixSize = [3, 4]
1.000   0.000   0.000   0.000
0.000   1.000   0.000   0.000
0.000   0.000   1.000   0.000
```

- `Matrix *createMatrixIdentity(const int orders);`

生成单位矩阵，`orders` 表示需要构建单位矩阵的阶数。

project03 > src > C test.c

```c
#include "matrix.h"
int main() {
    Matrix *a = createMatrixIdentity(3);
    showMatrix(a);
    return 0;
}
```

问题 输出 调试控制台 终端 JUPYTER

```
MatrixSize = [3, 4]
1.000   0.000   0.000   0.000
0.000   1.000   0.000   0.000
0.000   0.000   1.000   0.000
```

3.3 特殊数据类型填充矩阵

为了快速方便创建规模较的小矩阵和规模较大的矩阵，为有此需求用户的提供如下四种特殊类型生成矩阵：

- `Matrix *createMatrixArray(const int rows, const int cols, const float *arr);`

使用数组 `arr` （数组首指针） 前 `rows * cols` 个数按行优先填充到新建的矩阵中，其中 `rows` 表示需要构建矩阵的行数，`cols` 表示需要构建矩阵的列数。

```c
#include "matrix.h"
int main() {
    float arr[100];
    for (int i = 0; i < 100; i++) arr[i] = i;
    Matrix *a = createMatrixArray(10, 10, arr);
    showMatrix(a);
    return 0;
}
```

问题 输出 调试控制台 终端 JUPYTER

```
MatrixSize = [10, 10]
0.000   1.000   2.000   3.000   4.000   5.000   6.000   7.000   8.000   9.000
10.000  11.000  12.000  13.000  14.000  15.000  16.000  17.000  18.000  19.000
20.000  21.000  22.000  23.000  24.000  25.000  26.000  27.000  28.000  29.000
30.000  31.000  32.000  33.000  34.000  35.000  36.000  37.000  38.000  39.000
40.000  41.000  42.000  43.000  44.000  45.000  46.000  47.000  48.000  49.000
50.000  51.000  52.000  53.000  54.000  55.000  56.000  57.000  58.000  59.000
60.000  61.000  62.000  63.000  64.000  65.000  66.000  67.000  68.000  69.000
70.000  71.000  72.000  73.000  74.000  75.000  76.000  77.000  78.000  79.000
80.000  81.000  82.000  83.000  84.000  85.000  86.000  87.000  88.000  89.000
90.000  91.000  92.000  93.000  94.000  95.000  96.000  97.000  98.000  99.000
```

- `Matrix *createMatrixArange(const float start, const float stop, float step);`

生成$1 \times k$ 的矩阵，矩阵各元素依次是`start` （含） 到 `stop` （含）依次间隔 `step` 的实数。

```c
#include "matrix.h"
int main() {
    Matrix *a = createMatrixArange(11, -10, -2);
    showMatrix(a);
    return 0;
}
```

问题 输出 调试控制台 终端 JUPYTER

```
MatrixSize = [1, 11]
11.000  9.000   7.000   5.000   3.000   1.000   -1.000  -3.000  -5.000  -7.000  -9.000
```

- `Matrix *createMatrixString(const char *str);`

仿照`Matlab`矩阵的构建方法，使用空格或者`,`表示同一行的不同列数字，使用`;`表示不同行数字。

将这样的字符串数据，转化成矩阵，便于小数据下的输入。

```c
#include "matrix.h"
int main() {
    Matrix *a = createMatrixString("-1.1 +1.2 -3.14;     1 2,,3,; 1 2 3;");
    showMatrix(a);
    return 0;
}
```

问题 输出 调试控制台 终端 JUPYTER

```
MatrixSize = [4, 5]
-1.100  1.200   -3.140  0.000   0.000
1.000   2.000   0.000   3.000   0.000
1.000   2.000   3.000   0.000   0.000
0.000   0.000   0.000   0.000   0.000
```

- `Matrix *createMatrixFile(const char *file_name);`

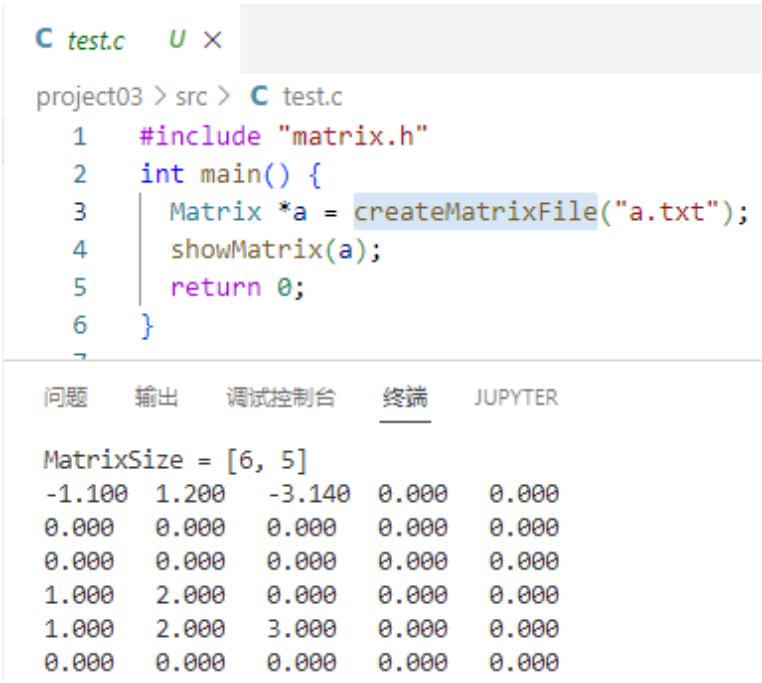
从文件`file_name`中读入矩阵信息，除了`;`看作矩阵换行以外，`\n`也被视作矩阵换行。

从文件中读取矩阵信息，是这个库可以处理较大规模的特定矩阵，也更方便交互。

a.txt U ×

project03 > src > a.txt

```
1   -1.100  1.200   -3.140  ,,;;
2   1.000   2.000   0.000
3   1.000   +2.000   3.000
4   0.000   0.000   0.000
```



注意到，这可以和前面提到的 输出矩阵到文件 函数 `void printMatrixToFile(const Matrix *matrix, const char *file_name);` 配合使用，做到矩阵的硬保存，方便用户的使用。

4 删除矩阵

为了防止使用者误操作（将删除的矩阵对应的地址进行计算）造成不可控的错误，库中使用 二级指针 进行矩阵的删除。

函数接口为：`void deleteMatrix(Matrix **matrix);`

若先前已经使用 `Matrix *matrix = createMatrix(rows, cols, value)` 生成矩阵，则删除时，需要进行的操作是将 `matrix` 的地址传入 `deleteMatrix` 函数接口中。

即：`deleteMatrix(&matrix);`

```c
void deleteMatrix(Matrix **matrix) {
  if (matrix == NULL) return;
  if (*matrix == NULL) return;
  free((*matrix)->head_pointer);
  free(*matrix);
  (*matrix) = NULL;
```

为了防止用户误操作，可以看到在函数具体实现时，在释放内存后，将 `matrix` 的值（即原矩阵地址）置空。

5 矩阵信息获得

如何在不改变矩阵结构，不增加额外的内存开销的前提下，通过遍历矩阵获得矩阵的一些信息，是本部分需要解决的重点问题，为此设计的库中实现了一些函数。

5.1 矩阵元素值信息

- `int MatrixSize(const Matrix *matrix);`

求矩阵中元素的个数，`matrix` 即为传入的矩阵地址，返回矩阵中元素的个数。

```c
#include "matrix.h"
#include "stdio.h"
int main() {
    Matrix *a = createMatrixString("1 2 3 4; 5 6 7 8; 9 10 11 12");
    showMatrix(a);
    printf("Size of matrix \"a\" is %d\n", MatrixSize(a));
    return 0;
}
```

问题 输出 调试控制台 终端 JUPYTER

```
MatrixSize = [3, 4]
1.000   2.000   3.000   4.000
5.000   6.000   7.000   8.000
9.000   10.000  11.000  12.000
Size of matrix "a" is 12
```

- `float MatrixElement(const Matrix *matrix, const int row_id, const int col_id);`

求矩阵中给定行列位置的值，`matrix` 即为传入的矩阵地址，`row_id` 表示行号，`col_id` 表示列号。

```c
#include "matrix.h"
#include "stdio.h"
int main() {
    Matrix *a = createMatrixString("1 2 3 4; 5 6 7 8; 9 10 11 12");
    showMatrix(a);
    printf("a[2][3] = %.3f\n", MatrixElement(a, 2, 3));
    return 0;
}
```

问题 输出 调试控制台 终端 JUPYTER

```
MatrixSize = [3, 4]
1.000   2.000   3.000   4.000
5.000   6.000   7.000   8.000
9.000   10.000  11.000  12.000
a[2][3] = 7.000
```

- `float MatrixMaxElement(const Matrix *matrix);`

求矩阵中最大的元素值，`matrix` 即为传入的矩阵地址。

- `float MatrixMinElement(const Matrix *matrix);`

求矩阵中最小的元素值，`matrix` 即为传入的矩阵地址。

- `float MatrixElementSum(const Matrix *matrix);`

求矩阵中元素值之和，`matrix` 即为传入的矩阵地址。

```c
#include "matrix.h"
#include "stdio.h"
int main() {
    Matrix *a = createMatrixString("1 2 3 4; 5 6 7 8; 9 10 11 12");
    showMatrix(a);
    printf("Maximum Element of matrix a is %.3f\n", MatrixMaxElement(a));
    printf("Minimum Element of matrix a is %.3f\n", MatrixMinElement(a));
    printf("Sum of matrix a's Elements is %.3f\n", MatrixElementSum(a));
    return 0;
}
```

问题 输出 调试控制台 终端 JUPYTER

```
MatrixSize = [3, 4]
1.000   2.000   3.000   4.000
5.000   6.000   7.000   8.000
9.000   10.000  11.000  12.000
Maximum Element of matrix a is 12.000
Minimum Element of matrix a is 1.000
Sum of matrix a's Elements is 78.000
```

5.2 矩阵遍历信息

为了实现按行遍历矩阵内所有元素，库中实现了矩阵的迭代器。

`float *MatrixBegin(const Matrix *matrix);` 表示获得矩阵元素的首地址。

`float *MatrixEnd(const Matrix *matrix);` 表示获得矩阵元素的末地址。

依据上述两个信息，事实上可以直接对矩阵进行按行优先进行遍历。

此外，函数还实现 `float **MatrixIterators(const Matrix *matrix);` 获得所有地址。也可以实现矩阵元素的按行遍历。

```c
#include "matrix.h"
#include "stdio.h"
int main() {
  Matrix *a = createMatrixString("1 2 3 4; 5 6 7 8; 9 10 11 12");
  showMatrix(a);
  for (float *iter = MatrixBegin(a); iter <= MatrixEnd(a); iter++) {
    printf("index: %p -> value: %.3f\n", iter, *iter);
  }
  printf("\n");
  float **iters = MatrixIterators(a);
  for (int i = 0; i < MatrixSize(a); i++)
    printf("index: %p -> value: %.3f\n", iters[i], *iters[i]);
  return 0;
}
```

问题　输出　调试控制台　终端　JUPYTER

```
MatrixSize = [3, 4]
1.000   2.000   3.000   4.000
5.000   6.000   7.000   8.000
9.000   10.000  11.000  12.000
index: 0x55e15fffa2c0 -> value: 1.000
index: 0x55e15fffa2c4 -> value: 2.000
index: 0x55e15fffa2c8 -> value: 3.000
index: 0x55e15fffa2cc -> value: 4.000
index: 0x55e15fffa2d0 -> value: 5.000
index: 0x55e15fffa2d4 -> value: 6.000
index: 0x55e15fffa2d8 -> value: 7.000
index: 0x55e15fffa2dc -> value: 8.000
index: 0x55e15fffa2e0 -> value: 9.000
index: 0x55e15fffa2e4 -> value: 10.000
index: 0x55e15fffa2e8 -> value: 11.000
index: 0x55e15fffa2ec -> value: 12.000

index: 0x55e15fffa2c0 -> value: 1.000
index: 0x55e15fffa2c4 -> value: 2.000
index: 0x55e15fffa2c8 -> value: 3.000
index: 0x55e15fffa2cc -> value: 4.000
index: 0x55e15fffa2d0 -> value: 5.000
index: 0x55e15fffa2d4 -> value: 6.000
index: 0x55e15fffa2d8 -> value: 7.000
index: 0x55e15fffa2dc -> value: 8.000
index: 0x55e15fffa2e0 -> value: 9.000
index: 0x55e15fffa2e4 -> value: 10.000
index: 0x55e15fffa2e8 -> value: 11.000
index: 0x55e15fffa2ec -> value: 12.000
```

可以看到这里，使用两种遍历方式都成功获得了按行优先进行遍历的效果，其地址和值都是相同的。

5.3 子矩阵信息

可以使用函数 `Matrix *getSubMatrix(const Matrix *matrix, const int row_id1, const int col_id1, const int row_id2, const int col_id2);` 求出特定函数的子矩阵。

其中参数 `matrix` 表示给出矩阵的首地址，`row_id1` 和 `col_id1` 分别表示子矩阵左上元素的行号和列号，`row_id2` 和 `col_id2` 表示子矩阵右下元素的行号和列号，返回值是一个新创建的矩阵，保存该子矩阵。

```c
#include "matrix.h"
#include "stdio.h"
int main() {
    Matrix *a = createMatrixString("1 2 3 4; 5 6 7 8; 9 10 11 12");
    showMatrix(a);
    Matrix *sub_matrix = getSubMatrix(a, 1, 2, 2, 3);
    showMatrix(sub_matrix);
    return 0;
}
```

问题 输出 调试控制台 终端 JUPYTER

```
MatrixSize = [3, 4]
1.000   2.000   3.000   4.000
5.000   6.000   7.000   8.000
9.000   10.000  11.000  12.000
MatrixSize = [2, 2]
2.000   3.000
6.000   7.000
```

6 矩阵运算

本部分内容中，将以从微观到宏观的顺序介绍库中关于矩阵运算的一些工作。

6.1 矩阵元素计算

在矩阵元素层面上，库可以很方便的对矩阵中所有元素进行批量计算处理。

- `void modifyMatrixElement(const Matrix *matrix, const int row_id, const int col_id, const float value);`

使用上面的函数，可以修改特定矩阵特定位置的元素值。

其中参数`matrix`表示给出矩阵的首地址，`row_id` 和 `col_id` 分别表示待修改元素的行号和列号，`value` 表示修改后的值。

使用该函数。可以帮助用户减少繁琐的计算对应元素存储的内存，提高使用效率。

```c
#include "matrix.h"
#include "stdio.h"
#include "math.h"
int main() {
    Matrix *a = createMatrixString("1 2 3; 4 5 6; 7 8 9");
    showMatrix(a);
    modifyMatrixElement(a, 2, 2, -1);
    showMatrix(a);
    return 0;
}
```

问题 输出 调试控制台 终端 JUPYTER

```
MatrixSize = [3, 3]
1.000   2.000   3.000
4.000   5.000   6.000
7.000   8.000   9.000
MatrixSize = [3, 3]
1.000   2.000   3.000
4.000   -1.000  6.000
7.000   8.000   9.000
```

- `Matrix *addScalarToMatrix(const Matrix *matrix, const float value);`
- `Matrix *subtractScalarToMatrix(const Matrix *matrix, const float value);`
- `Matrix *multiplyScalarToMatrix(const Matrix *matrix, const float value);`
- `Matrix *devideScalarToMatrix(const Matrix *matrix, const float value);`
- `Matrix *modScalarToMatrix(const Matrix *matrix, const float value);`

上述五个函数，都是对矩阵每个元素都加上、减去、乘以、除以、取模一个给定值`value`。

```c
#include "matrix.h"
#include "stdio.h"
int main() {
    Matrix *a = createMatrixString("1 2 3; 4 5 6; 7 8 9");
    showMatrix(a);
    showMatrix(addScalarToMatrix(a, 10));
    showMatrix(subtractScalarToMatrix(a, 10));
    showMatrix(multiplyScalarToMatrix(a, 10));
    showMatrix(devideScalarToMatrix(a, 10));
    showMatrix(modScalarToMatrix(a, 3));
    return 0;
}
```

问题 输出 调试控制台 终端 JUPYTER

```
MatrixSize = [3, 3]
1.000   2.000   3.000
4.000   5.000   6.000
7.000   8.000   9.000
MatrixSize = [3, 3]
11.000  12.000  13.000
14.000  15.000  16.000
17.000  18.000  19.000
MatrixSize = [3, 3]
-9.000  -8.000  -7.000
-6.000  -5.000  -4.000
-3.000  -2.000  -1.000
MatrixSize = [3, 3]
10.000  20.000  30.000
40.000  50.000  60.000
70.000  80.000  90.000
MatrixSize = [3, 3]
0.100   0.200   0.300
0.400   0.500   0.600
0.700   0.800   0.900
MatrixSize = [3, 3]
1.000   2.000   0.000
1.000   2.000   0.000
1.000   2.000   0.000
```

上述五个操作可以看作，对于矩阵每个元素，都对其进行一个函数运算，因此通过C - style 函数指针的方式，我们能通过自定义函数方式，对矩阵中每个元素进行特定函数运算。

这通过函数接口：`Matrix *elementProcessMatrix(Matrix *matrix, const double (*fun)(double));` 来实现。

```c
#include "matrix.h"
#include "stdio.h"
#include "math.h"
int main() {
    Matrix *a = createMatrixString("1 2 3; 4 5 6; 7 8 9");
    showMatrix(a);
    showMatrix(elementProcessMatrix(a, sin));
    return 0;
}
```

问题 输出 调试控制台 终端 JUPYTER

```
MatrixSize = [3, 3]
1.000   2.000   3.000
4.000   5.000   6.000
7.000   8.000   9.000
MatrixSize = [3, 3]
0.841   0.909   0.141
-0.757  -0.959  -0.279
0.657   0.989   0.412
```

在上面的例子中，我们将 `math.h` 库中 `sin` 函数使用函数指针的方法将矩阵中所有元素都进行相关运算。这体现了库良好的可扩展性，在这个问题上，库可以扩展任何一元的函数操作。

值得注意的是，如果使用在`math.h`中的函数，需要在连接时增加 `-lm` 指令，寻找数学函数具体的实现库。

如编译执行上面的`C` 文件的时候，可以简单采用编译命令：`gcc matrix.c test.c -o -lm run && ./run`

使用 `cmake` 或 `makefile` 时也需要尤为注意。

6.2 矩阵计算

6.2.1 矩阵整体计算

- `Matrix *addMatrix(const Matrix *matrix1, const Matrix *matrix2);`
- `Matrix *subtractMatrix(const Matrix *matrix1, const Matrix *matrix2);`
- `Matrix *dotMatrix(const Matrix *matrix1, const Matrix *matrix2);`
- `Matrix *devideMatrix(const Matrix *matrix1, const Matrix *matrix2);`
- `Matrix *modMatrix(const Matrix *matrix1, const Matrix *matrix2);`

上述五个函数，都是对一个矩阵每个元素都加上、减去、乘以、除以、取模另一矩阵对应位置上的数值。

```c
#include "matrix.h"
#include "stdio.h"
#include "math.h"
int main() {
    Matrix *a = createMatrixString("1 2 3; 4 5 6; 7 8 9");
    Matrix *b = createMatrixString("9 8 7; 6 5 4; 3 2 1");
    showMatrix(a);
    showMatrix(b);
    showMatrix(addMatrix(a, b));
    showMatrix(subtractMatrix(a, b));
    showMatrix(dotMatrix(a, b));
    showMatrix(devideMatrix(a, b));
    showMatrix(modMatrix(a, b));
    return 0;
}
```

问题 输出 调试控制台 终端 JUPYTER

```
MatrixSize = [3, 3]
1.000   2.000   3.000
4.000   5.000   6.000
7.000   8.000   9.000
MatrixSize = [3, 3]
9.000   8.000   7.000
6.000   5.000   4.000
3.000   2.000   1.000
MatrixSize = [3, 3]
10.000  10.000  10.000
10.000  10.000  10.000
10.000  10.000  10.000
MatrixSize = [3, 3]
-8.000  -6.000  -4.000
-2.000  0.000   2.000
4.000   6.000   8.000
MatrixSize = [3, 3]
9.000   16.000  21.000
24.000  25.000  24.000
21.000  16.000  9.000
MatrixSize = [3, 3]
0.111   0.250   0.429
0.667   1.000   1.500
2.333   4.000   9.000
MatrixSize = [3, 3]
1.000   2.000   3.000
4.000   0.000   2.000
1.000   0.000   0.000
```

上述五个操作可以看作，对于两个矩阵每个对应元素，都对其进行一个二元函数运算，因此通过C - style 函数指针的方式，我们能通过自定义函数方式，对矩阵中每个元素进行特定函数运算。

这通过函数接口：`Matrix *operatorMatrix(const Matrix *matrix1, const Matrix *matrix2, const double (*fun)(double, double));` 来实现。

```c
#include "matrix.h"
#include "stdio.h"
#include "math.h"
int main() {
    Matrix *a = createMatrixString("1 2 3; 4 5 6; 7 8 9");
    Matrix *b = createMatrixString("1 2 3; 1 2 3; 1 2 3");
    showMatrix(a);
    showMatrix(b);
    showMatrix(operatorMatrix(a, b, pow));
    return 0;
}
```

问题　输出　调试控制台　终端　JUPYTER

```
MatrixSize = [3, 3]
1.000   2.000   3.000
4.000   5.000   6.000
7.000   8.000   9.000
MatrixSize = [3, 3]
1.000   2.000   3.000
1.000   2.000   3.000
1.000   2.000   3.000
MatrixSize = [3, 3]
1.000   4.000   27.000
4.000   25.000  216.000
7.000   64.000  729.000
```

在上面的例子中，我们将 `math.h` 库中 `pow` 函数使用函数指针的方法将两个矩阵中所有对应元素都进行相关运算。这体现了库良好的可扩展性，在这个问题上，库可以扩展任何二元的函数操作。

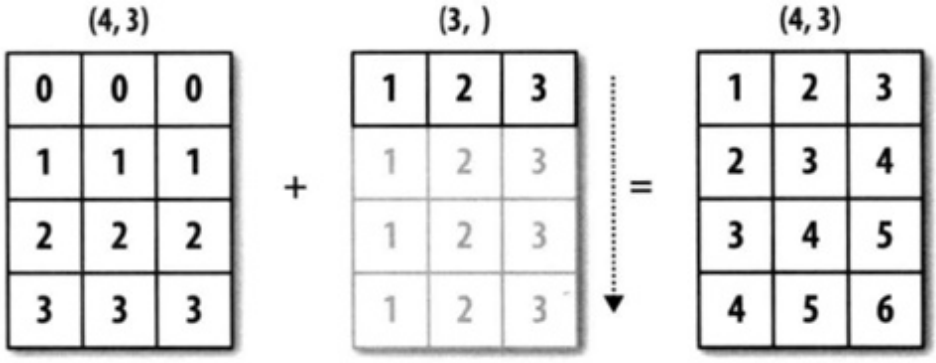
值得注意的是，如果使用在`math.h`中的函数，需要在连接时增加 `-lm` 指令，寻找数学函数具体的实现库。

如编译执行上面的`C` 文件的时候，可以简单采用编译命令：`gcc matrix.c test.c -o  run -lm && ./run`

使用 `cmake` 或 `makefile` 时也需要尤为注意。

此外，在人工智能的矩阵库 *Numpy* 中，矩阵与矩阵间的对应元素进行运算，还存在一种广播机制，这可以大大简化用户使用库编写代码的复杂性，在本库中也引入的相应的机制。

如果两个数组的后缘维度（trailing dimension，即从末尾开始算起的维度）的轴长度相符，或其中的一方的长度为1，则认为它们是广播兼容的。广播会在缺失和（或）长度为1的维度上进行。



```c
#include "matrix.h"
#include "stdio.h"
#include "math.h"
int main() {
    Matrix *a = createMatrixString("0 0 0; 1 1 1; 2 2 2; 3 3 3");
    Matrix *b = createMatrixString("1 2 3");
    showMatrix(a);
    showMatrix(b);
    showMatrix(addMatrix(a, b));
    return 0;
}
```

问题 输出 调试控制台 终端 JUPYTER

```
MatrixSize = [4, 3]
0.000   0.000   0.000
1.000   1.000   1.000
2.000   2.000   2.000
3.000   3.000   3.000
MatrixSize = [1, 3]
1.000   2.000   3.000
warning: [Broadcast Matrix Operation] The broadcast mechanism is triggered due to the different sizes of the two matrices.
MatrixSize = [4, 3]
1.000   2.000   3.000
2.000   3.000   4.000
3.000   4.000   5.000
4.000   5.000   6.000
```

同样的，所有两个矩阵对应元素进行二元函数计算，都能进行广播，并使用 `warning` 进行警告。

6.2.2 矩阵数值计算

求出方阵的逆、方阵的幂、方阵的行列式、矩阵乘法是线性代数中矩阵数值计算中非常重要的函数，对于问题的解决起到非常关键的作用，在库中我们也实现了相关的函数接口：

- `float MatrixDet(Matrix *matrix);` 求方阵的行列式。

```c
#include "matrix.h"
#include "stdio.h"
int main() {
    Matrix *a = createMatrixString("5 3 -1 2 0; 1 7 2 5 2; 0 -2 3 1 0;0 -4 -1 4 0; 0 2 3 5 0");
    printf("%.3f\n", MatrixDet(a));
    return 0;
}
```

问题 输出 调试控制台 终端 JUPYTER

```
-1080.000
```

- `Matrix *MatrixInv(Matrix *matrix);` 求方阵的逆元。

```c
#include "matrix.h"
#include "stdio.h"
int main() {
    Matrix *a = createMatrixString("0 1 2; 1 1 4; 2 -1 0");
    showMatrix(a);
    showMatrix(MatrixInv(a));
    return 0;
}
```

问题 输出 调试控制台 终端 JUPYTER

```
MatrixSize = [3, 3]
0.000   1.000   2.000
1.000   1.000   4.000
2.000   -1.000  0.000
MatrixSize = [3, 3]
2.000   -1.000  1.000
4.000   -2.000  1.000
-1.500  1.000   -0.500
```

- `Matrix *MatrixPower(Matrix *Matrix, int power);` 求方阵的幂

```c
#include "matrix.h"
#include "stdio.h"
int main() {
    Matrix *a = createMatrixString("0 1 2; 1 1 4; 2 -1 0");
    showMatrix(a);
    showMatrix(MatrixPower(a, 10));
    showMatrix(MatrixPower(a, -10));
    return 0;
}
```

问题 输出 调试控制台 终端 JUPYTER

```
MatrixSize = [3, 3]
0.000   1.000   2.000
1.000   1.000   4.000
2.000   -1.000  0.000
MatrixSize = [3, 3]
876.000 1.000   878.000
1461.000        1.000   1464.000
148.000 -1.000  146.000
MatrixSize = [3, 3]
1.572   -1.000  0.572
3.287   -2.000  0.287
-1.571  1.000   -0.571
```

- `Matrix *multiplyMatrix(const Matrix *matrix1, const Matrix *matrix2);` 计算矩阵乘法

```c
#include "matrix.h"
#include "stdio.h"
#include "math.h"
int main() {
    Matrix *a = createMatrixString("2 0 -1;1 3 2");
    Matrix *b = createMatrixString("1 7 -1; 4 2 3; 2 0 1");
    showMatrix(a);
    showMatrix(b);
    showMatrix(multiplyMatrix(a, b));
    return 0;
}
```

问题 输出 调试控制台 终端 JUPYTER

```
MatrixSize = [2, 3]
2.000   0.000   -1.000
1.000   3.000   2.000
MatrixSize = [3, 3]
1.000   7.000   -1.000
4.000   2.000   3.000
2.000   0.000   1.000
MatrixSize = [2, 3]
0.000   14.000  -3.000
17.000  13.000  10.000
```

6.2.3 矩阵形状重排

若两个矩阵元素个数相等，那么这两个矩阵可以相互转换，可以互相改变形状。

`void reshapeMatrix(Matrix *matrix, const int rows, const int cols);`

其中`matrix`表示传入待改变的矩阵，`rows` 和 `cols` 表示该矩阵改变形状后的长度和宽度。

```c
#include "matrix.h"
#include "stdio.h"
#include "math.h"
int main() {
    Matrix *a = createMatrixString("1 2 3 4; 5 6 7 8; 9 10 11 12");
    showMatrix(a);
    reshapeMatrix(a, 4, 3);
    showMatrix(a);
    return 0;
}

```

问题 输出 调试控制台 终端 JUPYTER

```
MatrixSize = [3, 4]
1.000   2.000   3.000   4.000
5.000   6.000   7.000   8.000
9.000   10.000  11.000  12.000
MatrixSize = [4, 3]
1.000   2.000   3.000
4.000   5.000   6.000
7.000   8.000   9.000
10.000  11.000  12.000
```

矩阵的转置同样也是一个重要的形状重排，我们使用`Matrix *transposeMatrix(const Matrix *matrix);`接口来实现。

```c
#include "matrix.h"
#include "stdio.h"
#include "math.h"
int main() {
    Matrix *a = createMatrixString("1 2 3 4; 5 6 7 8; 9 10 11 12");
    showMatrix(a);
    showMatrix(transposeMatrix(a));
    return 0;
}
```

问题　输出　调试控制台　终端　JUPYTER

```
MatrixSize = [3, 4]
1.000   2.000   3.000   4.000
5.000   6.000   7.000   8.000
9.000   10.000  11.000  12.000
MatrixSize = [4, 3]
1.000   5.000   9.000
2.000   6.000   10.000
3.000   7.000   11.000
4.000   8.000   12.000
```

6.2.4 矩阵拷贝

矩阵拷贝分为软拷贝和硬拷贝，其中软拷贝只拷贝矩阵的地址，而硬拷贝将拷贝矩阵的值。

`void copyMatrixbyRef(Matrix **dest_matrix, Matrix **src_matrix);`库采用该函数接口实现矩阵的软拷贝。需要输入目标矩阵地址的地址，以及源矩阵地址的地址。

```c
#include "matrix.h"
#include "stdio.h"
#include "math.h"
int main() {
    Matrix *a = createMatrixZeros(3, 4);
    Matrix *b = createMatrixString("1 2 3 4; 5 6 7 8; 9 10 11 12");
    copyMatrixbyRef(&a, &b);
    showMatrix(a);
    modifyMatrixElement(b, 1, 1, -1);
    showMatrix(a);
    return 0;
}
```

问题　输出　调试控制台　终端　JUPYTER

```
MatrixSize = [3, 4]
1.000   2.000   3.000   4.000
5.000   6.000   7.000   8.000
9.000   10.000  11.000  12.000
MatrixSize = [3, 4]
-1.000  2.000   3.000   4.000
5.000   6.000   7.000   8.000
9.000   10.000  11.000  12.000
```

在上面的例子中，可以说明使用`copyMatrixbyRef`函数是软拷贝的。

`void copySubMatrix(const Matrix *dest_matrix, const int dest_row_id1, const int dest_col_id1, const int dest_row_id2, const int dest_col_id2, const Matrix *src_matrix, const int src_row_id1, const int src_col_id1, const int src_row_id2, const int src_col_id2);`

该函数实现一个硬拷贝的函数接口，将目标矩阵`dest_matrix`左上角在`(dest_row_id1, dest_col_id1)`右下角在`(dest_row_id2, dest_col_id2)`的子矩阵使用源矩阵`src_matrix`左上角在`(src_row_id1, src_col_id1)`右下角在`(src_row_id2, src_col_id2)`的子矩阵的值进行硬拷贝。

```c
#include "matrix.h"
#include "stdio.h"
#include "math.h"
int main() {
    Matrix *a = createMatrixString("1 2 3 4 5; 6 7 8 9 10; 11 12 13 14 15; 16 17 18 19 20");
    Matrix *b = createMatrixString("-1 -2 -3 -4; -5 -6 -7 -8; -9 -10; -11 -12; -13 -14 -15 -16");
    showMatrix(a);
    showMatrix(b);
    copySubMatrix(a, 2, 2, 3, 4, b, 1, 1, 2, 3);
    showMatrix(a);
    modifyMatrixElement(b, 1, 1, 0);
    showMatrix(a);
    return 0;
}
```

问题　输出　调试控制台　终端　JUPYTER

```
MatrixSize = [4, 5]
1.000   2.000   3.000   4.000   5.000
6.000   7.000   8.000   9.000   10.000
11.000  12.000  13.000  14.000  15.000
16.000  17.000  18.000  19.000  20.000
MatrixSize = [5, 4]
-1.000  -2.000  -3.000  -4.000
-5.000  -6.000  -7.000  -8.000
-9.000  -10.000 0.000   0.000
-11.000 -12.000 0.000   0.000
-13.000 -14.000 -15.000 -16.000
MatrixSize = [4, 5]
1.000   2.000   3.000   4.000   5.000
6.000   -1.000  -2.000  -3.000  10.000
11.000  -5.000  -6.000  -7.000  15.000
16.000  17.000  18.000  19.000  20.000
MatrixSize = [4, 5]
1.000   2.000   3.000   4.000   5.000
6.000   -1.000  -2.000  -3.000  10.000
11.000  -5.000  -6.000  -7.000  15.000
16.000  17.000  18.000  19.000  20.000
```

该函数的实现稍显繁琐，库中还实现了一个函数接口`void copyMatrix(const Matrix *dest_matrix, const Matrix *src_matrix);`用来简单地实现等大矩阵的硬拷贝。

```c
#include "matrix.h"
#include "stdio.h"
#include "math.h"
int main() {
    Matrix *a = createMatrixZeros(3, 4);
    Matrix *b = createMatrixString("1 2 3 4; 5 6 7 8; 9 10 11 12");
    copyMatrix(a, b);
    showMatrix(a);
    modifyMatrixElement(b, 1, 1, -1);
    showMatrix(a);
    return 0;
}
```

问题 输出 调试控制台 终端 JUPYTER

```
MatrixSize = [3, 4]
1.000   2.000   3.000   4.000
5.000   6.000   7.000   8.000
9.000   10.000  11.000  12.000
MatrixSize = [3, 4]
1.000   2.000   3.000   4.000
5.000   6.000   7.000   8.000
9.000   10.000  11.000  12.000
```

6.2.5 矩阵拼接

库中还实现了两个接口函数，用于实现矩阵的拼接。

`Matrix *MatrixHorizontalStack(Matrix *matrix1, Matrix *matrix2);` 两个矩阵进行水平拼接。

`Matrix *MatrixHorizontalStack(Matrix *matrix1, Matrix *matrix2);` 两个矩阵进行垂直拼接。

```c
#include "matrix.h"
#include "stdio.h"
#include "math.h"
int main() {
    Matrix *a = createMatrixString("1 2; 3 4");
    Matrix *b = createMatrixString("5 6; 7 8");
    showMatrix(a);
    showMatrix(b);
    showMatrix(MatrixHorizontalStack(a, b));
    showMatrix(MatrixVerticalStack(a, b));
    return 0;
}
```

问题 输出 调试控制台 终端 JUPYTER

```
MatrixSize = [2, 2]
1.000   2.000
3.000   4.000
MatrixSize = [2, 2]
5.000   6.000
7.000   8.000
MatrixSize = [2, 4]
1.000   2.000   5.000   6.000
3.000   4.000   7.000   8.000
MatrixSize = [4, 2]
1.000   2.000
3.000   4.000
5.000   6.000
7.000   8.000
```

7 异常处理

在本库中，异常分为`警告 warning`和`错误 error`。

对于`警告 warning`异常，仅在控制台输出警告信息用于提示。

对于`错误 error`异常，在控制台输出警告信息的同时，根据错误处理模式`kErrorHandleMode`的数值确定处理办法。

- 当`kErrorHandleMode = 0`（默认情况），直接从库返回异常值至主程序，如返回值是`float`的返回`nan`，返回值是指针的，返回`NULL`，不在库中截断程序。
- 当`kErrorHandleMode = 1`（可手动设置），报出 error，并接着使用`exit(EXIT_FAILURE);`在库中截断程序，以避免后续程序执行造成更严重的错误。

使用者可以根据实际使用情况进行选择。

7.1 警告示例

```c
#include "matrix.h"
int main() {
  createMatrixFile("");
  Matrix *a = createMatrixString("1 2 3; 4 5 6; 7 8 9");
  Matrix *b = createMatrixString("1 2 3");
  addMatrix(a, b);
  Matrix *c = createMatrixZeros(1, 1);
  MatrixInv(c);
  return 0;
}
```

在控制台输出：

```
问题  输出  调试控制台  终端  JUPYTER

warning: [Illegal File Access] File does not exist.
warning: [Broadcast Matrix Operation] The broadcast mechanism is triggered due to the different sizes of the two matrices.
warning: [Matrix Inverse NULL] The inverse of the matrix does not exist.
```

7.2 错误示例

```c
#include "stdio.h"
#include "matrix.h"
#include "math.h"
int main() {
    Matrix *a = NULL;
    MatrixSize(a);
    Matrix *b = createMatrixZeros(1, 1);
    MatrixElement(b, 1, 2);
    createMatrixEmpty(-1, 1);
    createMatrixEmpty(1, -1);
    createMatrixEmpty(1000000000, 1000000000);
    createMatrixString("1..1");
    reshapeMatrix(a, 1, 1);
    reshapeMatrix(b, -1, 1);
    reshapeMatrix(b, 1, -1);
    reshapeMatrix(b, 1000000000, 1000000000);
    reshapeMatrix(b, 1, 2);
    a = createMatrixZeros(3, 4);
    b = createMatrixOnes(4, 3);
    copyMatrix(a, b);
    deleteMatrix(&a);
    operatorMatrix(a, b, pow);
    a = createMatrixOnes(4, 4);
    operatorMatrix(a, b, pow);
    deleteMatrix(&a);
    addScalarToMatrix(a, 1);
    subtractScalarToMatrix(a, 1);
    multiplyScalarToMatrix(a, 1);
```

```
    devideScalarToMatrix(a, 1);
    modScalarToMatrix(a, 1);
    multiplyMatrix(a, b);
    a = createMatrixZeros(4, 3);
    multiplyMatrix(a, b);
    deleteMatrix(&a);
    MatrixMaxElement(a);
    MatrixMinElement(a);
    MatrixElementSum(a);
    transposeMatrix(a);
    elementProcessMatrix(a, sin);
    MatrixPower(a, 10);
    a = createMatrixZeros(4, 3);
    MatrixPower(a, 10);
    deleteMatrix(&a);
    MatrixBegin(a);
    MatrixEnd(a);
    MatrixIterators(a);
    MatrixHorizontalStack(a, b);
    MatrixVerticalStack(a, b);
    a = createMatrixZeros(1, 2);
    b = createMatrixZeros(3, 4);
    MatrixHorizontalStack(a, b);
    MatrixVerticalStack(a, b);
    deleteMatrix(&a);
    MatrixDet(a);
    a = createMatrixZeros(1, 2);
    MatrixDet(a);
    deleteMatrix(&a);
    MatrixInv(a);
    a = createMatrixZeros(1, 2);
    MatrixInv(a);
    return 0;
}
```

在控制台输出：（默认情况 `kErrorHandleMode = 0`）

问题　输出　调试控制台　终端　JUPYTER

```
error: [Illegal Matrix Access] Matrix does not exist or has been deleted.
error: [Illegal Element Access] The element does not exist in the matrix.
error: [Illegal Matrix Definition] The number of rows is not a positive number.
error: [Illegal Matrix Definition] The number of cols is not a positive number.
error: [Illegal Matrix Definition] The number of matrix elements is too large.
error: [Illegal Matrix Definition] Illegal element format.
error: [Illegal Matrix Reshape] Matrix does not exist or has been deleted.
error: [Illegal Matrix Reshape] The number of rows is not a positive number.
error: [Illegal Matrix Reshape] The number of cols is not a positive number.
error: [Illegal Matrix Reshape] The number of matrix elements is too large.
error: [Illegal Matrix Reshape] The number of matrix elements is different before and after the reshape.
error: [Illegal Matrix Copy] Destination matrix and source matrix are different in size.
error: [Illegal Matrix Operation] Matrix does not exist or has been deleted.
error: [Illegal Matrix Operation] The two matrix sizes should match.
error: [Illegal Matrix Add a Scalar] Matrix does not exist or has been deleted.
error: [Illegal Matrix Subtract a Scalar] Matrix does not exist or has been deleted.
error: [Illegal Matrix multiply a Scalar] Matrix does not exist or has been deleted.
error: [Illegal Matrix Devide a Scalar] Matrix does not exist or has been deleted.
error: [Illegal Matrix Mod a Scalar] Matrix does not exist or has been deleted.
error: [Illegal Matrix Multiply] Matrix does not exist or has been deleted.
error: [Illegal Matrix Multiply] The two multiplied matrices do not match in size and cannot be multiplied.
error: [Illegal Matrix Max Element Find] Matrix does not exist or has been deleted.
error: [Illegal Matrix Min Element Find] Matrix does not exist or has been deleted.
error: [Illegal Matrix Sum Find] Matrix does not exist or has been deleted.
error: [Illegal Matrix Transpose] Matrix does not exist or has been deleted.
error: [Illegal Matrix Element Process] Matrix does not exist or has been deleted.
error: [Illegal Matrix Power] Matrix does not exist or has been deleted.
error: [Illegal Matrix Power] It must be a square matrix to be exponentiated.
error: [Illegal Matrix Begin] Matrix does not exist or has been deleted.
error: [Illegal Matrix End] Matrix does not exist or has been deleted.
error: [Illegal Matrix Iterator] Matrix does not exist or has been deleted.
error: [Illegal Matrix Horizontal Stack] Matrix does not exist or has been deleted.
error: [Illegal Matrix Vertical Stack] Matrix does not exist or has been deleted.
error: [Illegal Matrix Horizontal Stack] The number of rows of two matrices does not match.
error: [Illegal Matrix Vertical Stack] The number of columns of two matrices does not match.
error: [Illegal Matrix Determinant] Matrix does not exist or has been deleted.
error: [Illegal Matrix Determinant] Number of row and column of two matrices does not match.
error: [Illegal Matrix Inverse] Matrix does not exist or has been deleted.
error: [Illegal Matrix Inverse] Number of row and column of two matrices does not match.
```

在控制台输出：（手动设置情况 `kErrorHandleMode = 1`）

问题　输出　调试控制台　终端　JUPYTER

```
error: [Illegal Matrix Access] Matrix does not exist or has been deleted.
```

8　项目特色

8.1　面向用户编程理念

在实现库的过程中，始终将面向用户作为一个重要的编程理念，包括但不限于如下几个方面。

甚至可以说，在库的设计中，处处体现“面向用户”的编程理念。

8.1.1　多函数创建矩阵

多创建函数，便于用户选用合适的矩阵创建方法，提高数据组织的效率。

用户既可以将矩阵批量复制，也可以构建出随机矩阵，也可以通过文件构建矩阵，也可以通过特定格式的字符串构建矩阵，也可以通过数组构建矩阵。

8.1.2 广播机制

由于广播机制能够大大化简用户在使用库时的编程复杂程度，本库仿造 `Numpy` 的广播机制扩展了矩阵间对应位置函数计算。

8.1.3 多函数支持

通过传递函数指针的方法，可以将用户自定义的函数引入库中，进行运算，方便用户自定义。

8.2 运算效率高

库中所有参数传递都通过指针进行，使得数据传输的效率较高，速度较快。

8.3 多函数支持

项目中支持相当一部分实数矩阵运算，包括但不限于求逆、求转置、求幂、求行列式等，可以承担相当一部分数值计算工作。

8.4 错误信息提示

将函数处理全过程中的异常信息都考虑到，使得程序能够稳健运行，鲁棒性强。

8.5 格式规范便于维护

代码采用 `Google` 标准，通过 `cpplint.py` 校验，已经在`github` 上以项目形式开源，便于维护。

9 项目及代码

本项目已经上传到 `GitHub` 上，推荐您进入该项目阅读我的代码。

当然，您也可以查看随报告一并提交的`main.c` `matrix.c` `matrix.h` 文件查看代码。

项目链接：https://github.com/Maystern/SUSTech_cpp_Project03_a-libray-for-matrix-operationsin-C。

您可以在 `src` 中查看我的代码。其中，

- `main.c` 是一个测试文件，您可以参考其中的注释和输出结果。
- `matrix.h` 是矩阵库的头文件。
- `matrix.c` 是矩阵库的具体实现。

关于GitHub上的项目介绍，请参阅`README.md`文档，关于项目如何运行，介绍如下：

项目推荐在 `Ubuntu` 环境下运行。

1. 使用命令 `git clone https://github.com/Maystern/SUSTech_cpp_Project03_a-libray-for-matrix-operations-in-C.git` 将项目下载到当前目录。
2. 在当前目录下执行`cd SUSTech_cpp_Project03_a-libray-for-matrix-operations-in-C` 进入项目根目录。
3. 在项目根目录执行`sh Run_Matrix.sh` 命令，该命令执行后，将在`./build`中自动使用 `cmake` 编译原代码文件，并打开位于`./build/bin`目录下的二进制可执行程序`matrix_demo`。
4. 终端中会出现一个使用该库的示例程序的执行成果，该程序在 `src` 目录下的 `main.c` 中可以查阅。
5. 如果您仅仅想获得一个动态库，请您进入 `./build/bin`目录下取用`libmatrix.so`
6. 如果您仅仅想获得一个静态库，请您修改`src` 中的`CMakeLists.txt` 将第一条语句的`ON` 更改为 `OFF`，重新进行编译后，进入 `./build/bin`目录下取用`libmatrix.a`。

项目代码通过`cpplint.py` 检测，可以认为格式符合`Google` 标准。

```
问题  输出  调试控制台  终端  JUPYTER

Ignoring ../src/CMakeLists.txt; not a valid file name (cc, cxx, hxx, c++, cpp, hpp, h, c, h++)
Done processing ../src/CMakeLists.txt
Done processing ../src/main.c
Done processing ../src/matrix.c
Done processing ../src/matrix.h
Total errors found: 0
```

项目代码量采用 `cloc .` 进行统计。

```
问题  输出  调试控制台  终端  JUPYTER

(base) root@MaysternLaptop:/home/cpp_fall2022/SUSTech_cpp_Project03_a-libray-for-matrix-operations-in-C# cloc .
      50 text files.
      48 unique files.
      23 files ignored.

github.com/AlDanial/cloc v 1.90  T=0.04 s (659.5 files/s, 242945.3 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Python                           1            895           2644           2836
C                                3            193            126           1660
C++                              1            129             64            598
CMake                           13             82             43            377
make                             2            108            118            246
D                                2              0              0             98
C/C++ Header                     1              0              6             61
Markdown                         1              6              0             12
Bourne Shell                     2              0              0              8
TypeScript                       2              0              0              4
-------------------------------------------------------------------------------
SUM:                            28           1413           3001           5900
-------------------------------------------------------------------------------
```

10 程序实现效果

由于本项目说明 `Project03.pdf` 中没有给出样例数据，根据前面的代码运行截图，可以反映本项目所做的工作，所以请见前面的代码及其执行结果。

11 一些问题

在功能上，项目虽然实现了很多功能，但在如下方面可供探讨和完善。

- 更快速的矩阵相关运算：可以采用并行计等方式优化矩阵运算速度。
- 更多的矩阵相关数值计算函数支持：目前项目针对矩阵数值计算支持还是不高，应该添加更丰富的函数。
- 关于拷贝函数，是否可以设计更复杂的矩阵结构，使得硬拷贝通过软拷贝的方式进行实现。

12 后记

本次项目和之前的项目相比，有如下明显提高：

1. 编程风格严格遵循 `Google` 标准，使用了 `cpplint.py` 进行校验和修正。
2. 指针和内存管理方面更加自信了，使用指针进行内存管理的能力得到了加强。
3. 能较为熟练地使用 `Cmake` 组织管理项目了。
4. `GitHub` 使用更为熟练了。
5. 通过写 BUG 和 DEBUG 越来越认识到 C 和 C++的不同，以及 C 的局限性。

希望老师对我的项目多提FeedBack，帮助我提高自己的C/C++编程水平和项目能力。

项目报告较长，希望老师理解，也非常感谢您能看到这里，谢谢！