CS205 C/ C++ Programming – Project 3

Name: 凌硕（Ling Shuo）
SID：11912409

Part 1 – Analysis

The project is a Library for Matrix Operations in programming language C. To finish the work given by the requirements, I wrote code to solve the following subproblems.

1 – Struct for matrices

Consider the struct of a matrix, I should put the numbers of rows and columns in it. They are trivial since they're definitely integers. And I should also handle the elements in a matrix which can have size with arbitrary non-negative numbers. So I used a double pointer to describe the address of the elements of a matrix. And that makes sense because if `a[3][4]` is a two-dimensional array, and `p` is the double pointer such that `int (*p)[4] = a`, then we have `a[i][j]` = `*(*(p+i)+j)` = `p[i][j]`. Above, the struct contains `int rows`, `int columns` and `float ** elements`.
Please see the code in Part 2. And this part of the code is written in file ***matrix.h***.

2 – Creating a matrix

In this function, I want the user to be free to enter the elements of the matrix they want to create. Thus I set the `rows` and `columns` according to the arguments `int r, int c` firstly. Then I apply a block of memory for the `elements` of this matrix, and store the entries entered by `scanf`.
Remark 1 : It is worth mentioning that if the arguments input are illegal, I will set the `rows` and `columns` of the matrix to 0 and set the pointer `elements` to NULL.
Please see the code in Part 2. And this part of the code is written in file ***matrix.c***. The function declaration is `void createMatrix(int r, int c, struct Matrix * a)`.

3 – Deleting a matrix

Since the member variables of the struct involve pointers and memory management, we need to free the corresponding memory when we have finished using the matrix. Consider that the pointer we used is a double pointer, I free memory gradually.
Remark 2 : The input argument of this function is a pointer of struct matrix, so it is necessary to check the pointer before using it. The same is true for other functions that take pointers as arguments.

Please see the code in Part 2. And this part of the code is written in file ***matrix.c***. The function declaration is `void deleteMatrix(struct Matrix * a)`.

4 – Printing a matrix

Before I get to the function that the problem requires, I write a function that prints matrices. If it is a legal matrix (the `rows` and `columns` are not zeros), I will print it as the normal format. And if the size of the matrix is large, I will add '`;`' after each rows.
Remark 3 : In a function, if we do not want to modify the matrices pointed by the pointers which are given by the input arguments, it is better to use constant input arguments, for example, `const struct Matrix * a`. The same is true for other functions if necessary.
Please see the code in Part 2. And this part of the code is written in file ***matrix.c***. The function declaration is `void printMatrix(const struct Matrix * a)`.

5 – Copying a matrix

To copy the elements of a matrix `a` to matrix `b`, I use `struct Matrix *b, const struct Matrix * a` as the input arguments. In fact, it is logically similar with creating a matrix which has the input arguments `int r, int c, struct Matrix * a`. The difference is that we should define the member variables of the matrix `b` in terms of the member variables of the matrix `a`. Note that we should apply a block of memory for the `elements` of matrix `b` and set the each entries by matrix `a` rather than easy assignment for pointers.
Please see the code in Part 2. And this part of the code is written in file ***matrix.c***. The function declaration is `void copyMatrix(struct Matrix *b, const struct Matrix * a)`.

6 – Adding two matrices

Assume the two matrices are `a` and `b`, the result matrix is `result`. Then we have $result[i][j] = a[i][j] + b[i][j]$.
Remark 4 : We should check the sizes of two matrices to insure they can be added before adding them. The same is true for other functions if necessary.
Please see the code in Part 2. And this part of the code is written in file ***matrix.c***. The function declaration is `struct Matrix addMatrix(const struct Matrix * a, const struct Matrix * b)`.

7 – Subtraction of two matrices

Assume the two matrices are `a` and `b`, the result matrix is `result`. Then we have $result[i][j] = a[i][j] - b[i][j]$.
Please see the code in Part 2. And this part of the code is written in file ***matrix.c***. The function declaration is `struct Matrix subtractMatrix(const struct Matrix * a, const struct Matrix * b)`.

8 – Adding a scalar to a matrix

Assume the matrix is `a` and the number is `b`, the result matrix is `result`. Then we have $result[i][j] = a[i][j] + b$.
Please see the code in Part 2. And this part of the code is written in file ***matrix.c***. The function declaration is `struct Matrix addScalarWithMatrix(const struct Matrix * a, float b)`.

9 – Subtracting a scalar from a matrix

Assume the matrix is `a` and the number is `b`, the result matrix is `result`. Then we have $result[i][j] = a[i][j] - b$.
Please see the code in Part 2. And this part of the code is written in file ***matrix.c***. The function declaration is `struct Matrix subtractScalarWithMatrix (const struct Matrix * a, float b)`.

10 – Multiplying a matrix with a scalar

Assume the matrix is `a` and the number is `b`, the result matrix is `result`. Then we have $result[i][j] = a[i][j] \times b$.
Please see the code in Part 2. And this part of the code is written in file ***matrix.c***. The function declaration is `struct Matrix struct Matrix mulScalarWithMatrix(const struct Matrix * a, float b)`.

11 – Multiplying two matrices

Assume the matrices are `a` and `b`, the result matrix is `result`. Then we have $result[i][j] = \sum_{k=1}^{n} a[i][k] \times b[k][j]$.
Please see the code in Part 2. And this part of the code is written in file ***matrix.c***. The function declaration is `struct Matrix mulMatrix(const struct Matrix * a, const struct Matrix * b)`.

12 – Finding the minimal and maximal values of a matrix

These things are so trivial that we can just go through each entries of the matrix.
Please see the code in Part 2. And this part of the code is written in file ***matrix.c***. The function declaration is `float matrix_max(const struct Matrix * a)` and `float matrix_min(const struct Matrix * a)`.

13 – Finding the transpose of a matrix

To transpose matrix `a` to get matrix `b`. it is logically similar with copying a matrix except $b[i][j] = a[j][i]$.

Please see the code in Part 2. And this part of the code is written in file ***matrix.c***. The function declaration is `void transpose(struct Matrix *b, const struct Matrix * a)`.

14 – Finding the determinant of a matrix

We first need to check that the input matrix is a square matrix. Then we achieve the function by recursion. For matrix $A = (a_{i,j})_{i,j=1,\dots n}$, $\det(A) = \sum_{i=1}^{n}(-1)^{1+i}\det(A_{1,i})$, where $A_{j,i}$ is the matrix given by removing the $j-th$ row and the $i-th$ column from A. Since the size of $A_{1,i}$ is small than A, the recursion makes sense.
Please see the code in Part 2. And this part of the code is written in file ***matrix.c***. The function declaration is `float det(struct Matrix a)`.

15 – Finding the inverse of a matrix

We first need to check that the input matrix is a square matrix. For square matrix $A = (a_{i,j})_{i,j=1,\dots n}$, if $\det(A) = 0$, A has no inverse. Else, $A^{-1} = \frac{A^*}{\det(A)}$, where $A^* = ({a^*}_{i,j})_{i,j=1,\dots n}$ and ${a^*}_{i,j} = (-1)^{i+j}\det(A_{j,i})$ which $A_{j,i}$ is defined in Part1-14.
Please see the code in Part 2. And this part of the code is written in file ***matrix.c***. The function declaration is `struct Matrix inverse(struct Matrix a)`.

16 – Testing

To verify the correctness of the code and demonstrate how to use it, I wrote test code in the main function.
Please see the code in Part 2. And this part of the code is written in file ***main.c***.

17 – CMakeLists.txt

To manage files and get an executable file called "*test*", there is content in the CMake file:

```
cmake_minimum_required(VERSION 3.10)
project(test)
aux_source_directory(. DIR_SRCS)
add_executable(test ${DIR_SRCS})
```

Part 2 – Code

In this part, I put some important code.

1 – struct Matrix in matrix.h

```
struct Matrix
```

```
{
    int rows;
    int columns;
    float ** elements;
};
```

2 – code for partial functions in matrix.c

```
void createMatrix(int r, int c, struct Matrix * a)
{
    if(a==NULL)
    {
        exit(0);
    }
    if(r<1||c<1)
    {
        a->rows=0;
        a->columns=0;
        a->elements=NULL;
    }
    else
    {
        a->rows=r;
        a->columns=c;
        a->elements = (float **)malloc(sizeof(float *)*r);
        for(int i = 0; i < r; i++)
        {
            a->elements[i] = (float *)malloc(sizeof(float)*c);
        }

        printf("To create a matrix(%d, %d), please input elements:\n", r, c);
        printf("Note: Please enter elements from left to right and then top to bottom (Press 'Enter'
between each two elements).\n");
        for(int i = 0; i < r; i++)
        {
            for(int j = 0; j < c; j++)
            {
                scanf("%f", &a->elements[i][j]);
            }
        }
    }
    printf("\n");
}
```

```c
void deleteMatrix(struct Matrix * a)
{
    if(a==NULL)
    {
        exit(0);
    }
    for(int i = 0; i < a->rows; i++)
    {
        free(a->elements[i]);
        a->elements[i] = NULL;
    }
    free(a->elements);
    a->rows=0;
    a->columns=0;
    a->elements = NULL;
    printf("The matrix was successfully deleted!\n");
    printf("\n");
}

void printMatrix(const struct Matrix * a)
{
    if(a==NULL)
    {
        exit(0);
    }
    if(a->rows<1)
    {
        printf("Sorry! This is not a legal matrix or does not exist!\n");
    }
    else if(a->columns<=10)
    {
        printf("This is a %d by %d matrix!\n",a->rows,a->columns);
        for(int i = 0; i < a->rows; i++)
        {
            for(int j = 0; j < a->columns; j++)
            {
                printf("%f ", a->elements[i][j]);
            }
        printf("\n");
        }
    }
    else
    {
        printf("This is a %d by %d matrix!\n",a->rows,a->columns);
```

```
        for(int i = 0; i < a->rows; i++)
        {
            for(int j = 0; j < a->columns; j++)
            {
                printf("%f ", a->elements[i][j]);
            }
        printf(";\n");        //for matrix with large number of columns, use ';' to tell the
different rows.
        }
    }
    printf("\n");
}

void copyMatrix(struct Matrix *b,const struct Matrix * a)  //get a new matrix b as the same as a
{
    if(a==NULL)
    {
        exit(0);
    }
    b->rows=a->rows;
    b->columns=a->columns;
    b->elements = (float **)malloc(sizeof(float *)*a->rows);
    for(int i = 0; i < a->rows; i++)
    {
        b->elements[i] = (float *)malloc(sizeof(float)*a->columns);
    }
    for(int i=0;i<a->rows;i++)
    {
        for(int j=0;j<a->columns;j++)
        {
            b->elements[i][j]=a->elements[i][j];
        }
    }
    printf("The matrix was successfully copied!\n");
    printf("\n");
}

struct Matrix addMatrix(const struct Matrix * a, const struct Matrix * b)
{
    if(a==NULL||b==NULL)
    {
        exit(0);
    }
    if((a->rows!=b->rows)||(a->columns!=b->columns)||(a->columns==0)||(b->columns==0))
```

```
    {
        struct Matrix result;
        result.rows=0;
        result.columns=0;
        result.elements=NULL;
        return result;
    }
    else
    {
        struct Matrix result;
        result.rows=a->rows;
        result.columns=a->columns;
        result.elements = (float **)malloc(sizeof(float *)*result.rows);
        for(int i = 0; i < result.rows; i++)
        {
            result.elements[i] = (float *)malloc(sizeof(float)*result.columns);
        }
        for(int i = 0; i < result.rows; i++)
        {
            for(int j = 0; j < result.columns; j++)
            {
                result.elements[i][j] = a->elements[i][j] + b->elements[i][j];
            }
        }
        return result;
    }
}

struct Matrix addScalarWithMatrix(const struct Matrix * a, float b)
{
    if(a==NULL)
    {
        exit(0);
    }
    if(a->columns==0||a->rows==0)
    {
        struct Matrix result;
        result.rows=0;
        result.columns=0;
        result.elements=NULL;
        return result;
    }
    else
    {
```

```
        struct Matrix result;
        result.rows=a->rows;
        result.columns=a->columns;
        result.elements = (float **)malloc(sizeof(float *)*result.rows);
        for(int i = 0; i < result.rows; i++)
        {
            result.elements[i] = (float *)malloc(sizeof(float)*result.columns);
        }
        for(int i = 0; i < result.rows; i++)
        {
            for(int j = 0; j < result.columns; j++)
            {
                result.elements[i][j] = a->elements[i][j] + b;
            }
        }
        return result;
    }
}

struct Matrix mulMatrix(const struct Matrix * a, const struct Matrix * b)
{
    if(a==NULL||b==NULL)
    {
        exit(0);
    }
    if(a->columns == b->rows && a->rows!=0 && b->rows!=0)
    {
        struct Matrix result;
        result.rows = a->rows;
        result.columns = b->columns;
        result.elements = (float **)malloc(sizeof(float *)*result.rows);
        for(int i = 0; i < result.rows; i++)
        {
            result.elements[i] = (float *)malloc(sizeof(float)*result.columns);
        }
        for(int i = 0; i < result.rows; i++)
        {
            for(int j = 0; j < result.columns; j++)
            {
                result.elements[i][j] = 0;
            }
        }
        for(int i = 0; i < result.rows; i++)
        {
```

```
            for(int j = 0; j < result.columns; j++)
            {
                for(int k = 0; k < a->columns; k++)
                {
                    result.elements[i][j] = result.elements[i][j] + a->elements[i][k] *
b->elements[k][j];
                }
            }
        }
        return result;
    }
    else
    {
        struct Matrix result;
        result.rows=0;
        result.columns=0;
        result.elements=NULL;
        return result;
    }
}

float matrix_max(const struct Matrix * a)
{
    if(a==NULL)
    {
        exit(0);
    }
    if(a->rows==0)
    {
        return 0;
    }
    else
    {
        float max = a->elements[0][0];
        for(int r = 0; r < a->rows; r++)
        {
            for (int c = 0; c < a->columns; c++)
            {
                float val = a->elements[r][c];
                max = ( max > val ? max : val);
            }
        }
        return max;
    }
```

```
}

float det(struct Matrix a)
{
    if(a.rows==a.columns && a.rows!=0)
    {
        float result;
        if(a.rows==1)
        {
            result = a.elements[0][0];
        }
        else
        {
            result = 0;
            for(int i=0;i<a.rows;i++)
            {
                struct Matrix temp;
                temp.rows = a.rows - 1;
                temp.columns = a.columns - 1;
                temp.elements = (float **)malloc(sizeof(float *)*temp.rows);
                for(int j = 0; j < temp.rows; j++)
                {
                    temp.elements[j] = (float *)malloc(sizeof(float)*temp.columns);
                }
                for(int k=0;k<temp.rows;k++)
                {
                    for(int l=0;l < temp.columns;l++)
                    {
                        if(l<i)
                        {
                            temp.elements[k][l] = a.elements[k+1][l];
                        }
                        else
                        {
                            temp.elements[k][l] = a.elements[k+1][l+1];
                        }
                    }
                }
                if((1+i+1)%2==0)
                {
                    result = result + a.elements[0][i] * det(temp);
                }
                else
                {
```

```
                    result = result - a.elements[0][i] * det(temp);
                }
                //free
                for(int ii = 0; ii < temp.rows; ii++)
                {
                    free(temp.elements[ii]);
                    temp.elements[ii] = NULL;
                }
                free(temp.elements);
                temp.elements = NULL;
            }
        }
    }
    else
    {
        return 0;
    }
}

struct Matrix inverse(struct Matrix a)
{
    if(a.rows==a.columns && a.rows>=2 && det(a)!=0)
    {
        struct Matrix result;
        result.rows = a.rows;
        result.columns = a.columns;
        result.elements = (float **)malloc(sizeof(float *)*result.rows);
        for(int j = 0; j < result.rows; j++)
        {
            result.elements[j] = (float *)malloc(sizeof(float)*result.columns);
        }
        for(int i=0;i<result.rows;i++)
        {
            for(int j=0;j<result.columns;j++)
            {
                struct Matrix temp;
                temp.rows = result.rows - 1;
                temp.columns = result.columns - 1;
                temp.elements = (float **)malloc(sizeof(float *)*temp.rows);
                for(int k = 0; k < temp.rows; k++)
                {
                    temp.elements[k] = (float *)malloc(sizeof(float)*temp.columns);
                }
                for(int ii=0;ii<temp.rows;ii++)
```

```
            {
                for(int jj=0;jj < temp.columns;jj++)
                {
                    if(ii<i&&jj<j)
                    {
                        temp.elements[ii][jj] = a.elements[ii][jj];
                    }
                    else if(ii<i&&jj>=j)
                    {
                        temp.elements[ii][jj] = a.elements[ii][jj+1];
                    }
                    else if(ii>=i&&jj<j)
                    {
                        temp.elements[ii][jj] = a.elements[ii+1][jj];
                    }
                    else
                    {
                        temp.elements[ii][jj] = a.elements[ii+1][jj+1];
                    }
                }
            }
            if((i+1+j+1)%2==0)
            {
                result.elements[j][i] = det(temp)/det(a);
            }
            else
            {
                result.elements[j][i] = - det(temp)/det(a);
            }
            //free
            for(int h = 0; h < temp.rows; h++)
            {
                free(temp.elements[h]);
                temp.elements[h] = NULL;
            }
            free(temp.elements);
            temp.elements = NULL;
        }
    }
    return result;
}
else if(a.rows==1&&det(a)!=0)
{
    struct Matrix result;
```

```
        result.rows=1;
        result.columns=1;
        result.elements = (float **)malloc(sizeof(float *)*result.rows);
        for(int j = 0; j < result.rows; j++)
        {
            result.elements[j] = (float *)malloc(sizeof(float)*result.columns);
        }
        result.elements[0][0]=1/a.elements[0][0];
        return result;
    }
    else
    {
        struct Matrix result;
        result.rows=0;
        result.columns=0;
        result.elements=NULL;
        return result;
    }
}
```

Part 3 - Result & Verification

In this part, I run my code to verify the correctness. I will put the input and output screen and then explain them and compare them to the true result if necessary to show the correctness.

```
shuo_lin@LAPTOP-CM9SC1IR:~/project3$ ./test
To create a matrix(2, 3), please input elements:
Note: Please enter elements from left to right and then top to bottom (Press 'Enter' between each two elements).
1
2
3
1.2
2.3
3.4
```

Explanation: Create a matrix $a=[\begin{matrix}1 & 2 & 3\\ 1.2 & 2.3 & 3.4\end{matrix}]$.

```
The matrix was successfully copied!

This is a 2 by 3 matrix!
1.000000 2.000000 3.000000
1.200000 2.300000 3.400000
```

Explanation: Copy `a` to `b` and print `b`.

```
To create a matrix(2, 3), please input elements:
Note: Please enter elements from left to right and then top to bottom (Press 'Enter' between each two elements).
-1
-3
-2
1.6
-5
6
```

Explanation: Create a matrix `c`=$\begin{bmatrix} -1 & -3 & -2 \\ 1.6 & -5 & 6 \end{bmatrix}$.

```
This is a 2 by 3 matrix!
0.000000 -1.000000 1.000000
2.800000 -2.700000 9.400000
```

Explanation: Compute `d = b + c`. The true answer is $\begin{bmatrix} 0 & -1 & 1 \\ 2.8 & -2.7 & 9.4 \end{bmatrix}$ and thus the result given by running the code is correct.

```
This is a 2 by 3 matrix!
2.000000 5.000000 5.000000
-0.400000 7.300000 -2.600000
```

Explanation: Compute `e = b - c`. The true answer is $\begin{bmatrix} 2 & 5 & 5 \\ -0.4 & -7.3 & -2.6 \end{bmatrix}$ and thus the result given by running the code is correct.

```
This is a 2 by 3 matrix!
3.222000 4.222000 5.222000
3.422000 4.522000 5.622000
```

Explanation: Compute `f = b + 2.222`. The true answer is $\begin{bmatrix} 3.222 & 4.222 & 5.222 \\ 3.422 & 4.522 & 5.622 \end{bmatrix}$ and thus the result given by running the code is correct.

```
This is a 2 by 3 matrix!
3.222000 4.222000 5.222000
3.422000 4.522000 5.622000
```

Explanation: Compute `g = b - (- 2.222)`. The true answer is $\begin{bmatrix} 3.222 & 4.222 & 5.222 \\ 3.422 & 4.522 & 5.622 \end{bmatrix}$ and thus the result given by running the code is correct.

```
This is a 2 by 3 matrix!
1.100000 2.200000 3.300000
1.320000 2.530000 3.740000
```

Explanation: Compute `h = b * 1.1` .

The true answer is $\begin{bmatrix} 1.1 & 2.2 & 3.3 \\ 1.32 & 2.53 & 3.74 \end{bmatrix}$ and thus the result given by running the code is correct.

```
To create a matrix(3, 2), please input elements:
Note: Please enter elements from left to right and then top to bottom (Press 'Enter' between each two elements).
10
22
43
12
1.1
22.23

This is a 2 by 2 matrix!
99.300003 112.690002
114.639999 129.582001
```

Explanation: Create `i`=$\begin{bmatrix} 10 & 22 \\ 43 & 12 \\ 1.1 & 22.23 \end{bmatrix}$ and compute `j = b * i`.

The true answer(given by Matlab):

```
>> [1 2 3;1.2 2.3 3.4]*[10 22;43 12;1.1 22.23]

ans =

   99.3000  112.6900
  114.6400  129.5820
```

Thus the result given by running the code is correct.

```
3.400000
1.000000
```

Explanation: Find the max value and min value in matrix `b`.

```
This is a 3 by 2 matrix!
1.000000 1.200000
2.000000 2.300000
3.000000 3.400000
```

Explanation: Find the transpose matrix `k = b`.

```
To create a matrix(4, 4), please input elements:
Note: Please enter elements from left to right and then top to bottom (Press 'Enter' between each two elements).

1
2
3
4
11
23
411
21
54
12
12
9.8
12.3
-32
-231
20

2904884.000000
```

Explanation: Create 4×4 matrix `l`= $\begin{matrix} 1 & 2 & 3 & 4 \\ 11 & 23 & 411 & 21 \\ 54 & 12 & 12 & 9.8 \\ 12.3 & -32 & -231 & 20 \end{matrix}$

and compute `det(l)`.

True answer(given by Matlab):

```
>> det([1, 2, 3, 4;11, 23, 411, 21;54, 12, 12, 9.8;12.3, -32, -231, 20])

ans =

   2.9049e+06
```

Thus the result given by running the code is correct.

```
This is a 4 by 4 matrix!
-0.075766 0.002136 0.018624 0.003784
0.217513 -0.017509 0.005921 -0.028020
-0.019059 0.003053 -0.000458 0.000831
0.174480 0.005931 -0.007273 0.012440
```

Explanation: compute the inverse of `l`.

True answer(given by Matlab):

```
>> [1, 2, 3, 4;11, 23, 411, 21;54, 12, 12, 9.8;12.3, -32, -231, 20]^(-1)

ans =

   -0.0758    0.0021    0.0186    0.0038
    0.2175   -0.0175    0.0059   -0.0280
   -0.0191    0.0031   -0.0005    0.0008
    0.1745    0.0059   -0.0073    0.0124
```

Thus the result given by running the code is correct.

```
The matrix was successfully deleted!
The matrix was successfully deleted!
The matrix was successfully deleted!
The matrix was successfully deleted!
The matrix was successfully deleted!
The matrix was successfully deleted!
The matrix was successfully deleted!
The matrix was successfully deleted!
The matrix was successfully deleted!
The matrix was successfully deleted!
The matrix was successfully deleted!
The matrix was successfully deleted!
The matrix was successfully deleted!
```

Explanation: Delete all matrices.

```
Sorry! This is not a legal matrix or does not exist!
Sorry! This is not a legal matrix or does not exist!
Sorry! This is not a legal matrix or does not exist!
Sorry! This is not a legal matrix or does not exist!
Sorry! This is not a legal matrix or does not exist!
Sorry! This is not a legal matrix or does not exist!
Sorry! This is not a legal matrix or does not exist!
Sorry! This is not a legal matrix or does not exist!
Sorry! This is not a legal matrix or does not exist!
Sorry! This is not a legal matrix or does not exist!
Sorry! This is not a legal matrix or does not exist!
Sorry! This is not a legal matrix or does not exist!
Sorry! This is not a legal matrix or does not exist!
```

Explanation: Print the matrices after deleting them.

Part 4 - Difficulties & Solutions

1. If we use simple pointers to describe matrix entries when creating member variables of a matrix struct, we may face index crossing bound problems when dealing with matrices with very large numbers of elements. So I used double pointers.
2. To reduce program crashes caused by incorrect user input, we checked pointers before using them.
3. Using constant pointer input arguments for functions.
4. Check the validity of the operation before operation.
5. Find the inverse of a square matrix using adjoint matrix.