CS205-2022Fall Project 2

A Library for Matrix Operations

Name: 陈康睿

SID: 12110524

Part 1 - Analysis & Implementation

This project require us to implement a library for matrix operation in C, which means we can't use `struct` as a class or overload operations and functions. We can only process data via pointers and memory directly, Though implementing basic operations are relatively easy compared to the last project, to maintain safety and convenience for users is much harder.

Include, Typedef & Define

```
#include <stddef.h>
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <stdio.h>
```

```
typedef float entry_t; // The data type of matrix entries
#define entry_place_holder "%.2f\t" // The precision used for output
```

Global Settings, Variables

To make the lib more flexible, I create a `typedef` that use `entry_t` to signify floating data types, `float` by default. If users need this lib to provide more precise calculation, they can easily change `float` into `double` here rather than dive into every detail structure and function. Also, the `entry_place_holder` defines a format of outputs, which is also customizable.

```
typedef float entry_t;
#define entry_place_holder "%.2f\t"
```

Data Structure

Main Struct: Matrix

`struct Matrix` simply stores the basic informations for a matrix: two `size_t` variables for the number of rows and columns, and an `entry_t` pointer point to the array storing the entries.

```
typedef struct Matrix{
    size_t row, col;
    entry_t *entry;
} Matrix;
```

Support Struct: MatrixPointer

To store the valid `Matrix` pointers, I chose to implement a linked-list (Matrix Pointer List, MPL for short in the following), which can add, check, delete pointer information through several functions. It also provide a function `void MPL_clear()` to release all memory used by matrices and the list itself, which offers user a simple insurance against memory leak.

```
typedef struct MatrixPointer{
    Matrix *mat;
    struct MatrixPointer *next;
} MatrixPointer;
```

```
// Build a matrix pointer list globally
MatrixPointer MPL_head; // The head of the matrix pointers list

// Push a matrix into MPL
void MPL_push_front(Matrix *const mat) {
    MatrixPointer *mp = (MatrixPointer *)malloc(sizeof(MatrixPointer));
    mp->mat = mat;
    mp->next = MPL_head.next;
    MPL_head.next = mp;
}

// Delete a matrix from MPL
void MPL_erase(const Matrix *const mat) {
    MatrixPointer *prev = &MPL_head, *mp;
    while (prev->next->mat != mat)
        prev = prev->next; // find the previous pointer in the list
    mp = prev->next;
    prev->next = mp->next;
    free(mp);
}

// Clear MPL, release the memory of all matrices
void MPL_clear() {
    for (MatrixPointer *cur = MPL_head.next, *next; cur; cur = next) {
        next = cur->next;
        free(cur->mat->entry);
        free(cur->mat);
        free(cur);
    }
}

// Check the validity of a Matrix pointer
bool check_matrix_pointer(const Matrix *const mat) {
    if (!mat) return false; // empty pointer
    for (MatrixPointer *cur = &MPL_head; cur; cur = cur->next)
        if (cur->mat == mat) return true; // in MPL, valid
    return false; // not in MPL, invalid
}
```

Operations

Create

To create a matrix and push the created pointer into MPL. (Besides, I also implements a function `createIMatrix()` to generate identity matrix for convenience.)

```
/**
 * @brief Create a matrix with initialization
 * @param row The number of rows
 * @param col The number of columns
 * @param entry A pointer points to the array of entries
 * @return A pointer point to the created matrix
*/
```

```
Matrix *createMatrix(const size_t row, const size_t col, const entry_t *const
entry) {
    if (!row || !col) {
        puts("Error in createMatrix(): Invalid dimentions!");
        return NULL;
    }
    if (!entry) {
        puts("Error in createMatrix(): Invalid entry array pointer!");
        return NULL;
    }
    Matrix *mat = (Matrix *)malloc(sizeof(Matrix));
    mat->row = row;
    mat->col = col;
    size_t siz = row*col;
    mat->entry = (entry_t *)malloc(sizeof(entry_t)*siz);
    for (size_t i = 0; i < siz; ++i)
        mat->entry[i] = entry[i];
    MPL_push_front(mat);
    return mat;
}

/**
 * @brief Create an identity matrix
 * @param n The size of the matrix
 * @return A pointer point to the created matrix
*/
Matrix *createIMatrix(const size_t n) {
    if (!n) {
        puts("Error in deleteMatrix(): Invalid matrix demensions!");
        return NULL;
    }
    size_t siz = n*n;
    entry_t entry[siz];
    memset(entry, 0, sizeof(entry));
    for (size_t i = 0; i < n; ++i)
        entry[i*n+i] = 1;
    return createMatrix(n, n, entry);
}
```

Delete

To delete a matrix and erase in from MPL. (Noted that it is also needed to release the memory of the entry array.)

```
/**
 * @brief Delete a matrix
 * @param mat The pointer points the matrix to be deleted
*/
void deleteMatrix(Matrix *const mat) {
    if (!check_matrix_pointer(mat)) {
        puts("Error in deleteMatrix(): Invalid matrix pointer!");
        return;
    }
    MPL_erase(mat);
    free(mat->entry);
    free(mat);
}
```

Copy

To copy data from source matrix to destination matrix. (Noted that the entry array memory of destination matrix needs reallocation.)

```
/**
 * @brief Copy data from a matrix to another
 * @param a A pointer point to the DST matrix
 * @param b A pointer point to the SRC matrix
*/
void copyMatrix(Matrix *const a, const Matrix *const b) {
    if (!check_matrix_pointer(a) || !check_matrix_pointer(b)) {
        puts("Error in copyMatrix(): Invalid matrix pointers!");
        return;
    }
    a->row = b->row;
    a->col = b->col;
    size_t siz = b->row*b->col;
    a->entry = (entry_t *)realloc(a->entry, sizeof(entry_t)*siz);
    memcpy(a->entry, b->entry, sizeof(entry_t)*siz);
}
```

Arithmetic

Between Matrices

To calculate two matrice using tree matrix computations rules: addition, substraction, and multiplication. (Noted that these functions return new matrix pointers supposed to be handled by users.)

```
/**
 * @brief Matrix addition
 * @param a A pointer point to the augend matrix
 * @param b A pointer point to the addend matrix
 * @returns A pointer point to the result of the addition
*/
Matrix *addMatrix(const Matrix *const a, const Matrix *const b) {
    if (!check_matrix_pointer(a) || !check_matrix_pointer(b)) {
```

```
        puts("Error in addMatrix(): Invalid matrix pointers!");
        return NULL;
    }
    if (a->row != b->row || a->col != b->col) {
        puts("Error in addMatrix(): Inequal dementions!");
        return NULL;
    }
    size_t size = a->row*a->col;
    entry_t entry[size];
    for (size_t i = 0; i < size; ++i)
        entry[i] = a->entry[i]+b->entry[i];
    return createMatrix(a->row, a->col, entry);
}

/**
 * @brief Matrix substraction
 * @param a A pointer point to the subtrahend matrix
 * @param b A pointer point to the subtractor matrix
 * @returns A pointer point to the result of the substraction
*/
Matrix *substractMatrix(const Matrix *const a, const Matrix *const b) {
    if (!check_matrix_pointer(a) || !check_matrix_pointer(b)) {
        puts("Error in substractMatrix(): Invalid matrix pointers!");
        return NULL;
    }
    if (a->row != b->row || a->col != b->col) {
        puts("Error in substractMatrix(): Inequal dementions!");
        return NULL;
    }
    size_t size = a->row*a->col;
    entry_t entry[size];
    for (size_t i = 0; i < size; ++i)
        entry[i] = a->entry[i]-b->entry[i];
    return createMatrix(a->row, a->col, entry);
}

/**
 * @brief Matrix multiplication
 * @param a A pointer point to the multiplicand matrix
 * @param b A pointer point to the multiplier matrix
 * @returns A pointer point to the result of the multiplication
*/
Matrix *multiplyMatrix(const Matrix *const a, const Matrix *const b) {
    if (!check_matrix_pointer(a) || !check_matrix_pointer(b)) {
        puts("Error in multiplyMatrix(): Invalid matrix pointers!");
        return NULL;
    }
    if (a->row != b->col || a->col != b->row) {
        puts("Error in multiplyMatrix(): Unpaired dementions!");
        return NULL;
    }
    size_t size = a->row*b->col;
    entry_t entry[size];
    for (size_t i = 0; i < a->row; ++i)
        for (size_t j = 0; j < b->col; ++j) {
```

```
            entry[i*b->col+j] = 0.0;
            for (size_t k = 0; k < a->col; ++k)
                entry[i*b->col+j] += a->entry[i*a->col+k]*b->entry[k*b->col+j];
        }
    return createMatrix(a->row, b->col, entry);
}
```

Between a matrix and a scalar

To calculate each matrix entry indipendently with a given scalar using four basic computations: addition, substraction, multiplication and division. (Noted that there is no new matrix created.)

```
/**
 * @brief Add a scalar to a matrix
 * @param mat A pointer point to the augend matrix
 * @param x The addend scalar
*/
void addScalar(const Matrix *mat, const entry_t x) {
    if (!check_matrix_pointer(mat)) {
        puts("Error in addScalar(): Invalid matrix pointer!");
        return;
    }
    for (size_t i = 0; i < mat->row*mat->col; ++i)
        mat->entry[i] += x;
}

/**
 * @brief Substract a scalar from a matrix
 * @param mat A pointer point to the subtrahend matrix
 * @param x The substractor scalar
*/
void substractScalar(const Matrix *mat, const entry_t x) {
    if (!check_matrix_pointer(mat)) {
        puts("Error in substractScalar(): Invalid matrix pointer!");
        return;
    }
    for (size_t i = 0; i < mat->row*mat->col; ++i)
        mat->entry[i] -= x;
}

/**
 * @brief Multiply a matrix and a scalar
 * @param mat A pointer point to the multiplicand matrix
 * @param x The multiplier scalar
*/
void multiplyScalar(const Matrix *mat, const entry_t x) {
    if (!check_matrix_pointer(mat)) {
        puts("Error in multiplyScalar(): Invalid matrix pointer!");
        return;
    }
    for (size_t i = 0; i < mat->row*mat->col; ++i)
        mat->entry[i] *= x;
}

/**
```

```
 * @brief Divide a matrix by a scalar
 * @param mat A pointer point to the dividend matrix
 * @param x The divisor scalar
*/
void divideScalar(const Matrix *mat, const entry_t x) {
    if (!check_matrix_pointer(mat)) {
        puts("Error in multiplyScalar(): Invalid matrix pointer!");
        return;
    }
    for (size_t i = 0; i < mat->row*mat->col; ++i)
        mat->entry[i] /= x;
}
```

Functions

Max & Min Entry

To find the minimal or the maximal entry of a matrix.

```
/**
 * @brief Find the minimal entry of a matrix
 * @param mat The target matrix
 * @returns The minimal entry of the matrix
*/
entry_t minEntry(const Matrix *const mat) {
    if (!check_matrix_pointer(mat)) {
        puts("Error in minEntry(): Invalid matrix pointer!");
        return NAN;
    }
    entry_t result = mat->entry[0];
    for (size_t i = 1; i < mat->row*mat->col; ++i)
        result = (result <= mat->entry[i] ? result : mat->entry[i]);
    return result;
}

/**
 * @brief Find the maximum entry of a matrix
 * @param mat The target matrix
 * @returns The maximum entry of the matrix
*/
entry_t maxEntry(const Matrix *const mat) {
    if (!check_matrix_pointer(mat)) {
        puts("Error in maxEntry(): Invalid matrix pointer!");
        return NAN;
    }
    entry_t result = mat->entry[0];
    for (size_t i = 1; i < mat->row*mat->col; ++i)
        result = (result >= mat->entry[i] ? result : mat->entry[i]);
    return result;
}
```

Gaussian Elimination

To elimination a matrix using Gaussian elimination.

```
/**
 * @brief Process a matrix with Gaussian elimination
 * @param mat The target matrix
*/
void GaussianEliminate(Matrix *const mat) {
    if (!check_matrix_pointer(mat)) {
        puts("Error in GaussianElimination(): Invalid matrix pointer!");
        return;
    }
    size_t row = mat->row, col = mat->col;
    entry_t *entry = mat->entry;
    for (size_t i = 0, j = 0, k; i < col, j < row; ++i) {
        for (k = j; k < row; ++k) // find a row whose entry is not 0
            if (entry[j*col+i] != 0) break;
        if (k == row) continue; // all are 0
        if (k != j) // swap row
            for (size_t t = i; t < col; ++t) {
                entry_t temp = entry[j*col+t];
                entry[j*col+t] = entry[k*col+t];
                entry[k*col+t] = temp;
            }
        for (k = j+1; k < row; ++k)
            if (entry[k*col+i] != 0) {
                entry_t coe = entry[k*col+i]/entry[j*col+i];
                entry[k*col+i] = 0;
                for (size_t t = i+1; t < col; ++t)
                    entry[k*col+t] -= coe*entry[j*col+t];
            }
        ++j;
    }
}
```

Trace

To Calculate the trace (the sum of diagonal entries) of a square matrix.

```
/**
 * @brief Calculate the trace of a matrix
 * @param mat The target matrix
 * @return Trace of the matrix
*/
entry_t trace(const Matrix *const mat) {
    if (!check_matrix_pointer(mat)) {
        puts("Error in trace(): Invalid matrix pointer!");
        return NAN;
    }
    if (mat->row != mat->col) {
        puts("Error in trace(): Invalid matrix dementions!");
        return NAN;
    }
    entry_t result = 0.0;
```

```
    for (size_t i = 0; i < mat->row; ++i)
        result += mat->entry[i*mat->col+i];
    return result;
}
```

Determinant

To calculate the determinant of a matrix by expand along it's first column recursively. (Noted that it is needed to invoking `deleteMatrix()` after finishing using temporary matrices; and using `MPL_push_front()` in `createMatrix()` ensures the cost of `check_matrix_pointer()` and `MPL_erase()` is $\Theta(1)$ according to recursion stack.)

```
/**
 * @brief Calculate the determinant of a matrix
 * @param mat The target matrix
 * @return Determinant of the matrix
*/
entry_t det(const Matrix*const mat) {
    if (!check_matrix_pointer(mat)) {
        puts("Error in det(): Invalid matrix pointer!");
        return NAN;
    }
    if (mat->row != mat->col) {
        puts("Error in det(): Invalid matrix dementions!");
        return NAN;
    }
    size_t row = mat->row, col = mat->col, siz = (mat->row-1)*(mat->col-1);
    entry_t *entry = mat->entry, temp[siz];
    switch (row) {
        case 1: return entry[0];
        case 2: return entry[0]*entry[3]-entry[1]*entry[2];
        default: // expand the first column of the matrix
            entry_t ret = 0;
            for (size_t k = 0; k < row; ++k) {
                size_t idx = 0;
                for (size_t i = 0; i < row; ++i) {
                    if (i == k) continue;
                    for (size_t j = 1; j < col; ++j)
                        temp[idx++] = entry[i*col+j];
                }
                Matrix *sub = createMatrix(row-1, col-1, temp);
                ret += (k&1 ? -entry[k*col]*det(sub) : entry[k*col]*det(sub));
                deleteMatrix(sub);
            }
            return ret;
    }
}
```

Rank

To calculate the rank of a function by counting the non-zero rows after elimination.

```
/**
 * @brief Calculate the rank of a matrix
```

```
 * @param mat the target matrix
 * @return Rank of the matrix
*/
size_t rank(const Matrix *const mat) {
    if (!check_matrix_pointer(mat)) {
        puts("Error in rank(): Invalid matrix pointer!");
        return 0;
    }
    Matrix *temp = createIMatrix(1);
    copyMatrix(temp, mat);
    GaussianEliminate(temp);
    size_t row = mat->row, col = mat->col, ret = 0;
    for (size_t i = 0; i < row; ++i) {
        bool flag = false;
        for (size_t j = 0; j < col; ++j)
            flag |= (temp->entry[i*col+j] != 0);
        ret += flag;
    }
    deleteMatrix(temp);
    return ret;
}
```

Part 2 - Results & Verification

Demonstration

```
int main () {
    entry_t e1[] = {1, 0,
                    0, 1,
                    1, 1},
            e2[] = {1, 1, 0.5,
                    0.5, 0.5, 1};
    Matrix *m1 = createMatrix(3, 2, e1), *m2 = createMatrix(2, 3, e2);

    GaussianEliminate(m1);
    printMatrix(m1);
    printMatrix(m1+1);

    substractScalar(m2, 0.5);
    Matrix *result = multiplyMatrix(m1, m2);
    printf("rank = %zu\n", rank(result));

    copyMatrix(m1, result);
    printMatrix(m1);

    deleteMatrix(result);
    MPL_clear();
    return 0;
}
```

```
root@ArtanisaxLEGION:~/GitHub/CS205-2022Fall/Project3/build# ./Matrix
row = 3, col = 2
1.00    0.00
0.00    1.00
0.00    0.00
Error in printMatrix(): Invalid matrix pointer!
rank = 2
row = 3, col = 3
0.50    0.50    0.00
0.00    0.00    0.50
0.00    0.00    0.00
```

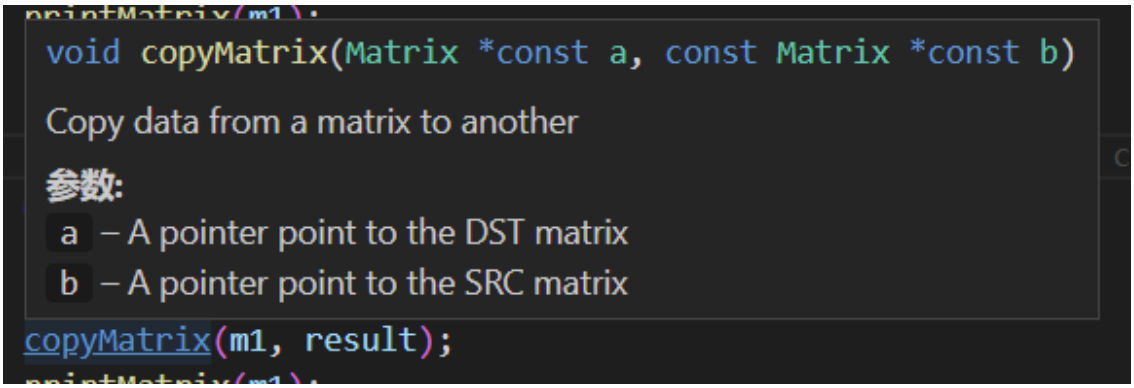
User-friendly Coding

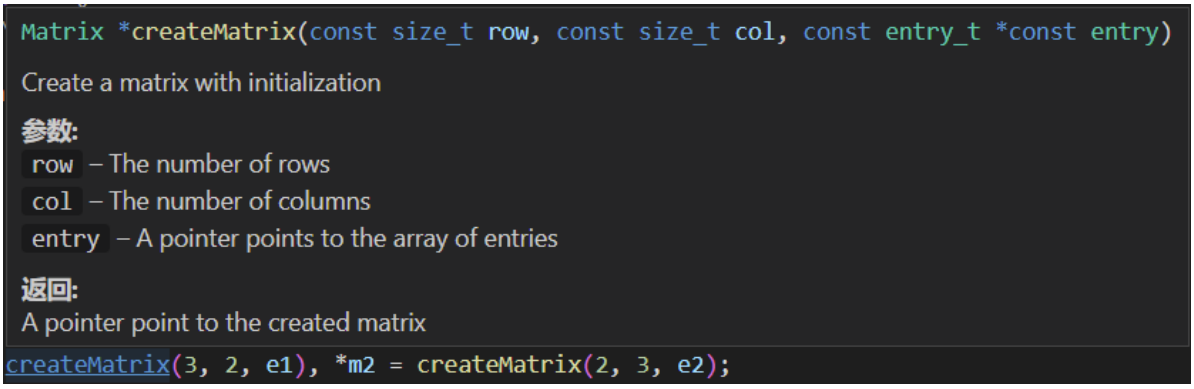
1. Using `const` to modify function parameters, so users will know whether the matrix is changed in different functions.

```
Matrix *substractMatrix(const Matrix *const a, const Matrix *const b);

Matrix *multiplyMatrix(const Matrix *const a, const Matrix *const b);

void addScalar(const Matrix *mat, const entry_t x);

void substractScalar(const Matrix *mat, const entry_t x);

void multiplyScalar(const Matrix *mat, const entry_t x);

void divideScalar(const Matrix *mat, const entry_t x);

entry_t minEntry(const Matrix *const mat);

entry_t maxEntry(const Matrix *const mat);

void GaussianEliminate(Matrix *const mat);

entry_t trace(const Matrix *const mat);

entry_t det(const Matrix *const mat);

size_t rank(const Matrix *const mat);
```

2. Using annotations `@` in comments to elaborate functions, so users can get what do these functions do and the meanings of parameters and return values quickly as well as

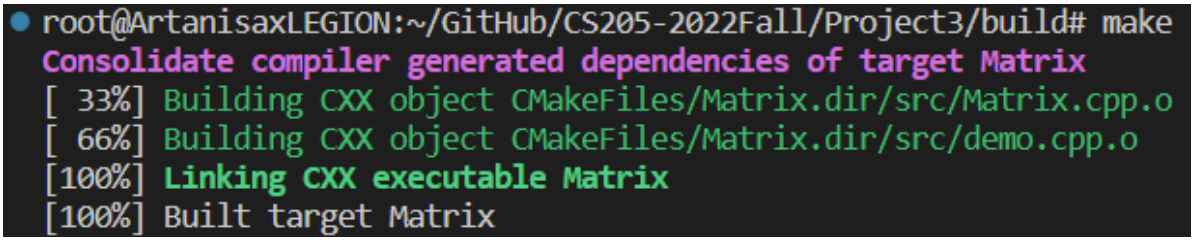
comfortably.





C++ Compability

This lib can be used in C++ by simply change `Matrix.c` into `Matrix.cpp`. Noted that to achieve this, I convert `void *` into `Matrix *` by force after invoking `malloc()` or `realloc()`, so it satisfies strong-type requirement and can go through C++ compilers. If user want to use this lib in C++ environment, they can simply rename `Matrix.c` into `Matrix.cpp`. (If I put function implementation all in `Matrix.h`, the rename step can be skipped.)



Part 3 - Self-review

1. To use `struct` only without `class`, I used a global variable to create MPL, but it seems is not recommended to use global variables.
2. I have no means of preventing user from messing up with the pointers out side of my lib.

Part 4 - Codes & Comments

Please see other files uploaded on BB.

(They are also available on My GitHub Repository)