COMPSCI 1XC3: DEVELOPMENT BASICS

Scientific Programming: Operations on Matrices

Gabriel Syriani

400176049

Contents

1 Introduction

A matrix is a rectangular array of elements arranged into a specific number of rows and columns. Such elements, also called entries, most commonly consist of numbers, but can also range from symbols to mathematical expressions. Traditional algebraic operations, such as addition, subtraction and multiplication can be performed on a set of matrices. Additionally, one can solve a linear system of equations using matrices (i.e. solving for x in $Ax = B$ where A and B are matrices).

Since the number of elements in a matrix can quickly become overwhelming, such operations can be extremely tedious if performed by hand. As such, efficient programming algorithms are commonly used to handle computation on matrices, eliminating the complexity of manual calculations as well as significantly improving the speed at which they are performed. This report will provide a comprehensive breakdown of a code written in C, containing functions implemented for each of the traditional operations.[1]

2 Operations on Matrices in C

In order to implement different functions in C for the purpose of performing operations on matrices, the code will first generate two (2) random matrices with each entry consisting of a number ranging from -10 to 10. The number of rows and columns for said matrices is expected to be provided by the user as arguments passed directly through the terminal. The input will need to be formatted as shown below.

```
./math_matrix nrows_A ncols_A nrows_B ncols_B <operation> [print]
```

where nrows_A and ncols_A represent the number of rows and columns of Matrix A, and nrows_B and ncols_B represent the number of rows and columns of Matrix B, respectively. The <operation> placeholder will take either add, subtract, multiply or solve as arguments. Finally, print is an optional argument that can be added by the user if they wish to print Matrix A, Matrix B, and the resulting matrix.

Additionally, below is a breakdown of the common variables that will be used within the C program. These variables are a crucial part of understanding how the algorithm will operate.

i. N1: number of rows in Matrix A;

ii. M1: number of columns in Matrix A;

iii. N2: number of rows in Matrix B

iv. M1: number of columns in Matrix B;

[1] Course material taught in COMPSCI 1XC3: Development Basics, as well as ChatGPT, were used in the writing of the codes. Raw codes are copy-pasted in Appendix A.

2.1 generateMatrix()

This function will be called by the computer to generate the matrices prior to performing any operations. The number of rows and columns of each matrix are given by the user through the terminal. The code is as follows:

Listing 1: C Code for Matrix Generation

```
void generateMatrix(int rows, int cols, double matrix[rows][cols]) {

    double min = -10.0;
    double max = 10.0;

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            double randDouble = (double)rand() / RAND_MAX;
            matrix[i][j] = min + randDouble * (max - min);
        }
    }
}
```

2.1.1 Code Breakdown

The function takes as inputs the number of rows and columns given by the user, as well as a pre-initalized matrix in which random numbers ranging from -10 to 10 (of double precision) will be inserted.

The code loops through each cell of the matrix, generates a random number using the rand() function (which was adjusted to only produce values between the min and max which are defined as -10 and 10) and appends the generated value to the cell.

2.2 printMatrix()

The purpose of this function is to print the generated matrices as well as the resulting matrix should the user opt to include the print argument inside the terminal when executing the compiled code. Below is the code:

Listing 2: C Code for Printing a Matrix

```
void printMatrix(int rows, int cols, double matrix[rows][cols]) {

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("%lf ", matrix[i][j]);
        }
        printf("\n");
    }
    printf("\n");
}
```

2.2.1 Code Breakdown

The inputs of the printMatrix() function are the number of rows and columns of the matrix being printed, as well as the matrix itself. The function loops through each cell and prints the value using %lf as the placeholder as the values are of double precision. The function adds a new line after each row for formatting purposes.

2.3 addMatrices() & subtractMatrices()

The function implementation for the Addition and Subtraction operations, denoted by addMatrices() and subtractMatrices(), is given in the code below:

Listing 3: C Code for Matrix Addition and Subtraction

```
void addMatrices(int N1, int M1, double A[N1][M1], int N2, int M2, double B
    [N2][M2], double result[N1][M1]) {

    if (N1 != N2 || M1 != M2) {
        printf("Cannot complete addition: please input matrices of the same
            size!\n");
        return;
    }

    for (int i = 0; i < N1; i++) {
        for (int j = 0; j < M1; j++) {
            result[i][j] = A[i][j] + B[i][j];
        }
    }
}

void subtractMatrices(int N1, int M1, double A[N1][M1], int N2, int M2,
    double B[N2][M2], double result[N1][M1]) {

    if (N1 != N2 || M1 != M2)
        printf("Please input matrices of the same size!\n");

    for (int i = 0; i < N1; i++){
        for (int j = 0; j < M1; j++) {
            result[i][j] = A[i][j] - B[i][j];
        }
    }

    return;
}
```

2.3.1 Code Breakdown

Both functions addMatrices() and subtractMatrices() take the following as inputs, in order: N1, M1, Matrix A, N2, M2, Matrix B, Result Matrix.

An if statement is included at the beginning of each function to check if the given number of rows and columns of the matrices are valid for addition/subtraction, as these operations can only be performed on matrices if their number of rows and columns are equal. If the number of rows and/or columns do not match, the code will print an error statement and return to terminate the program.

If the inputs are valid, the program will loop through each cell of the matrices, and add (or subtract) their values together, appending the result to the corresponding cell in the Result Matrix.

2.4 multiplyMatrices()

This function will be used to multiply the two (2) matrices generated by the computer.

Listing 4: C Code for Matrix Multiplication

```
void multiplyMatrices(int N1, int M1, double A[N1][M1], int N2, int M2,
    double B[N2][M2], double result[N1][M2]) {

    if (M1 != N2) {
        printf("Please input valid matrices where the number of columns in
            Matrix A is equal to the number of rows in Matrix B!\n");
        return;
    }

    for (int i = 0; i < N1; i++) {
        for (int j = 0; j < M2; j++) {
            result[i][j] = 0;
            for (int k = 0; k < M1; k++) {
                result[i][j] += A[i][k] * B[k][j];
            }
        }
    }

    return;
}
```

2.4.1 Code Breakdown

The function takes the same inputs as the functions for addition/subtraction (refer to section 2.1).

An if statement is included at the beginning of the multiplyMatrices() function to check if the given number of rows and columns of the matrices are valid, as multiplication can only be performed on matrices if the number of columns in Matrix A is equal to the number of Rows in Matrix B. If these numbers do not match will print an error statement and return to terminate the program.

If the inputs are valid, the program will loop through the rows of Matrix A, and the columns of Matrix B, multiplying each corresponding cell together, summing up the products as it does so and appending the resulting value to the corresponding cell in the Result Matrix. This is because each element of the Result Matrix in matrix multiplication is found by taking the dot product of the corresponding row of Matrix A and the corresponding column of Matrix B.

2.5 solveLinearSystem()

This function will be called to solve for x in $Ax = B$ where A and B are the randomly generated matrices. The code is given in Listing 5 below:

Listing 5: C Code for Solving a Linear System of Equations

```
void solveLinearSystem(int N1, int M1, double A[N1][M1], int N2, int M2,
    double B[N2][M2], double x[N1][M2]) {

    if (N1 != M1 || N2 != N1 || M2 != 1) {
        printf("Invalid input! Matrix A must be square; the number of rows
            in Matrix B should be equal to Matrix A and the number of
            columns must be equal to 1.\n");
        return;
    }

    for (int i = 0; i < N1 - 1; i++) {
        for (int j = i + 1; j < N1; j++) {
            double factor = A[j][i] / A[i][i];
            for (int k = 0; k < M1; k++) {
                A[j][k] -= factor * A[i][k];
            }
            B[j][0] -= factor * B[i][0];
        }
    }

    for (int i = N1 - 1; i >= 0; i--) {
        x[i][0] = B[i][0];
        for (int j = i + 1; j < N1; j++) {
            x[i][0] -= A[i][j] * x[j][0];
        }
        x[i][0] /= A[i][i];
    }
}
```

2.5.1 Code Breakdown

The function takes the same inputs as the functions for addition/subtraction (refer to section 2.3). It should be noted that the code only supports square matrices as input for Matrix A, and a column vector for Matrix B (as it will be used for the purpose of solving a linear system of equations).

To check if the given inputs are valid, the code ensures that Matrix A is square, that the number of rows in Matrix A and B are equal, and that Matrix B is a column vector. If any of those conditions are not met, an error statement is printed and the code returns to terminate the function.

The first set of nested for-loops are implemented to perform Gaussian Elimination on Matrices A and B. The elimination factor is calculated by taking the ratio of each cell in the lower triangle of the matrix to the one above. Subsequently, the code loops through the corresponding row and subtracts the product of the factor and the corresponding cell above it. The result is a matrix of upper-triangle form.

The second set of nested for-loops allow the computer to perform back substitution on the upper-triangle matrix found in the step above. It achieves this by calculating the value of the last entry in the Result Vector and working its way upwards. Since the code works upwards through an upper-triangle matrix, each step contains only one unknown, and is therefore straightforward to compute. In other words, each value in the solution vector (working upwards) is calculated using only the previously determined values, ensuring accuracy in the operations.

3 Calling the Functions in main()

Once all the functions discussed in Section 2 have been implemented (in a `functions.c` file), they can be called in main() (in a separate file called `math_matrix.c`) in order to finalize the algorithm which takes its inputs through command-line arguments.

3.1 main(int argc, char *argv[])

Below is the code written in main, including the necessary packages.

Listing 6: C Code in main()

```
#include <string.h>
#include <stdio.h>
#include "functions.h"
#include <time.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {

    if (argc < 6 || argc > 7) {
        printf("Usage: %s nrows_A ncols_A nrows_B ncols_B <operation> [
            print]\n", argv[0]);
        return 1;
    }

    for (int i = 1; i < 5; i++) {
        if (atoi(argv[i]) <= 0) {
            printf("Please input positive integers for matrix dimensions!")
                ;
            return 1;
        }
    }

    int N1 = atoi(argv[1]);
    int M1 = atoi(argv[2]);
    int N2 = atoi(argv[3]);
    int M2 = atoi(argv[4]);

    double A[N1][M1];
    double B[N2][M2];

    srand(time(NULL));
    generateMatrix(N1, M1, A);
    if (argc == 7 && strcmp(argv[6], "print") == 0) {
        printf("Matrix A:\n");
        printMatrix(N1, M1, A);
    }


    srand(time(NULL) + 1);
    generateMatrix(N2, M2, B);

    if (argc == 7 && strcmp(argv[6], "print") == 0) {
        printf("Matrix B:\n");
        printMatrix(N2, M2, B);
    }

    if (strcmp(argv[5], "add") == 0) {
        double result[N1][M1];
```

```
        addMatrices(N1, M1, A, N2, M2, B, result);
        if (argc == 7 && strcmp(argv[6], "print") == 0) {
            if (N1 != N2 || M1 != M2) {
                return 1;
            }
            printf("Result of A + B:\n");
            printMatrix(N1, M1, result);
        }
    }

    else if (strcmp(argv[5], "subtract") == 0) {
        double result[N1][M1];
        subtractMatrices(N1, M1, A, N2, M2, B, result);
        if (argc == 7 && strcmp(argv[6], "print") == 0) {
            if (N1 != N2 || M1 != M2) {
                return 1;
            }
            printf("Result of A - B:\n");
            printMatrix(N1, M1, result);
        }
    }

    else if (strcmp(argv[5], "multiply") == 0) {
        double result[N1][M2];
        multiplyMatrices(N1, M1, A, N2, M2, B, result);
        if (argc == 7 && strcmp(argv[6], "print") == 0) {
            if (M1 != N2)
                return 1;
            else {
                printf("Result of A * B:\n");
                printMatrix(N1, M2, result);
            }
        }
    }

    else if (strcmp(argv[5], "solve") == 0) {
        double result[N1][1];
        solveLinearSystem(N1, M1, A, N2, M2, B, result);
        if (argc == 7 && strcmp(argv[6], "print") == 0) {
            if (N1 != M1 || N2 != N1 || M2 != 1)
                return 1;
            else {
                printf("Result of x = B/A:\n");
                printMatrix(N1, M2, result);
            }
        }
    }

    else
        printf("Please enter a valid operation (e.g. add, subtract,
            multiply, solve)!\n");

    return 0;

}
```

3.1.1 Code Breakdown

An if-statement is included at the top of the code to ensure that the number of command-line arguments that were passed by the user when running the executable is either six (6) or seven (7), as per the format of the usage. As a reminder, the format consists of six (6) required inputs, with an optional print statement bringing the total to a possible number of seven (7). A second if-statement was also added to ensure that the first four (4) arguments (after ./math_matrix) are all positive integers, as they should be representing matrix dimensions. These conditions are both checked, and if the computer finds that the format was not correctly followed, it will print an error statement reminding the user of the specific input requirements and will return to terminate.

Once the if-statements are checked, the program then proceeds to convert the matrix dimensions to integers using the atoi() function on the `argv[]` array, which is the zero-indexed array that stored all the arguments passed by the user as strings. The program then initializes two (2) matrices, Matrix A and Matrix B, each with the dimensions obtained as detailed above. The rand() function is then seeded with the current time before calling generateMatrix() to fill Matrix A. However, before calling generateMatrix() on Matrix B, the seed is updated in order to ensure that duplicate matrices are not produced.

With both matrices now populated with random integers from -10 to 10, a chain of if/else-if statements are incorporated to check the 6^{th} element of the `argv[]` array in order to determine which mathematical operation was requested by the user. The strcmp() command is used to compare the string provided by the user with all of the operations supported by the program (i.e. add, subtract, multiply, solve), and when a match is found, the computer calls the appropriate function to perform the calculations. If no match is found, it means that the 6^{th} element was not any of the operations, a statement is printed to reflect the error.

4 Program Limitations: Segmentation Fault

While this program works as intended for matrices with relatively small dimensions, the computer may run into an error with larger matrices. For example, if the user enters the following command-line arguments (i.e. ./math_matrix 1000 1000 1000 1000 add print), the program returns the following error:

```
Segmentation fault (core dumped)
```

In order to debug this error and obtain more information, gdb will be used with the same arguments. Below is what the computer indicates:

```
Program received signal SIGSEGV, Segmentation fault.
0x0000000008001653 in main (argc=<error reading variable: Cannot access
    memory at address 0x7fffffedf0c>, argv=<error reading variable: Cannot
    access memory at address 0x7fffffedf00>) at math_matrix.c:27
27          double B[N2][M2];
```

As can be seen from the error above, the computer runs into a segmentation fault at line 27 of the `math_matrix.c` file, when trying to initialize Matrix B with 1,000 rows and columns. The reason this is happening for Matrix B and not Matrix A, is because the computer had successfully declared Matrix A, but had taken up most of the memory in the stack in doing so. As such, because the stack size is limited, the computer is experiencing a stack overflow when trying to declare another matrix requiring an overwhelming amount of memory (1000×1000 entries each with double-precision). This can be verified by trying to execute the program with 100,000 rows and columns (i.e. ./math_matrix 10000 10000 10000 10000 add print) for both matrices, and seeing at which line the computer encounters the error. Below is the dbg output for this execution:

```
Program received signal SIGSEGV, Segmentation fault.
0x00000000080014b8 in main (argc=<error reading variable: Cannot access
    memory at address 0x7fffffedf0c>, argv=<error reading variable: Cannot
    access memory at address 0x7fffffedf00>) at math_matrix.c:26
26          double A[N1][M1];
```

As expected, the program fails (by Segmentation fault) at line 26, where Matrix A was declared. To reiterate, this is because stack overflow is encountered at the initialization of the first matrix, as there is not enough memory in the stack to accommodate even one matrix with 10,000 rows and columns.

A Copy-Pasted Codes

A.1 `functions.c` file

Listing 7: C Code in `functions.c`

```c
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include "functions.h"

void generateMatrix(int rows, int cols, double matrix[rows][cols]) {

    double min = -10.0;
    double max = 10.0;

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            double randDouble = (double)rand() / RAND_MAX;
            matrix[i][j] = min + randDouble * (max - min);
        }
    }
}

void printMatrix(int rows, int cols, double matrix[rows][cols]) {

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("%lf ", matrix[i][j]);
        }
        printf("\n");
    }
    printf("\n");
}

void addMatrices(int N1, int M1, double A[N1][M1], int N2, int M2, double B
    [N2][M2], double result[N1][M1]) {

    if (N1 != N2 || M1 != M2) {
        printf("Cannot complete addition: please input matrices of the same
            size!\n");
        return;
    }

    for (int i = 0; i < N1; i++) {
        for (int j = 0; j < M1; j++) {
            result[i][j] = A[i][j] + B[i][j];
        }
    }
}

void subtractMatrices(int N1, int M1, double A[N1][M1], int N2, int M2,
    double B[N2][M2], double result[N1][M1]) {

    if (N1 != N2 || M1 != M2)
        printf("Please input matrices of the same size!\n");

    for (int i = 0; i < N1; i++){
        for (int j = 0; j < M1; j++) {
```

```
            result[i][j] = A[i][j] - B[i][j];
        }
    }

    return;
}

void multiplyMatrices(int N1, int M1, double A[N1][M1], int N2, int M2,
    double B[N2][M2], double result[N1][M2]) {

    if (M1 != N2) {
        printf("Please input valid matrices where the number of columns in
            Matrix A is equal to the number of rows in Matrix B!\n");
        return;
    }

    for (int i = 0; i < N1; i++) {
        for (int j = 0; j < M2; j++) {
            result[i][j] = 0;
            for (int k = 0; k < M1; k++) {
                result[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    return;
}

void solveLinearSystem(int N1, int M1, double A[N1][M1], int N2, int M2,
    double B[N2][M2], double x[N1][M2]) {

    if (N1 != M1 || N2 != N1 || M2 != 1) {
        printf("Invalid input! Matrix A must be square; the number of rows
            in Matrix B should be equal to Matrix A and the number of
            columns must be equal to 1.\n");
        return;
    }

    for (int i = 0; i < N1 - 1; i++) {
        for (int j = i + 1; j < N1; j++) {
            double factor = A[j][i] / A[i][i];
            for (int k = 0; k < M1; k++) {
                A[j][k] -= factor * A[i][k];
            }
            B[j][0] -= factor * B[i][0];
        }
    }

    for (int i = N1 - 1; i >= 0; i--) {
        x[i][0] = B[i][0];
        for (int j = i + 1; j < N1; j++) {
            x[i][0] -= A[i][j] * x[j][0];
        }
        x[i][0] /= A[i][i];
    }
}
```

A.2 `math_matrix.c` file

Listing 8: C Code in `math_matrix.c`

```c
#include <string.h>
#include <stdio.h>
#include "functions.h"
#include <time.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {

    if (argc < 6 || argc > 7) {
        printf("Usage: %s nrows_A ncols_A nrows_B ncols_B <operation> [
            print]\n", argv[0]);
        return 1;
    }

    for (int i = 1; i < 5; i++) {
        if (atoi(argv[i]) <= 0) {
            printf("Please input positive integers for matrix dimensions!")
                ;
            return 1;
        }
    }

    int N1 = atoi(argv[1]);
    int M1 = atoi(argv[2]);
    int N2 = atoi(argv[3]);
    int M2 = atoi(argv[4]);

    double A[N1][M1];
    double B[N2][M2];

    srand(time(NULL));
    generateMatrix(N1, M1, A);
    if (argc == 7 && strcmp(argv[6], "print") == 0) {
        printf("Matrix A:\n");
        printMatrix(N1, M1, A);
    }


    srand(time(NULL) + 1);
    generateMatrix(N2, M2, B);

    if (argc == 7 && strcmp(argv[6], "print") == 0) {
        printf("Matrix B:\n");
        printMatrix(N2, M2, B);
    }

    if (strcmp(argv[5], "add") == 0) {
        double result[N1][M1];
        addMatrices(N1, M1, A, N2, M2, B, result);
        if (argc == 7 && strcmp(argv[6], "print") == 0) {
            if (N1 != N2 || M1 != M2) {
                return 1;
            }
            printf("Result of A + B:\n");
            printMatrix(N1, M1, result);
```

```
        }
    }

    else if (strcmp(argv[5], "subtract") == 0) {
        double result[N1][M1];
        subtractMatrices(N1, M1, A, N2, M2, B, result);
        if (argc == 7 && strcmp(argv[6], "print") == 0) {
            if (N1 != N2 || M1 != M2) {
                return 1;
            }
            printf("Result of A - B:\n");
            printMatrix(N1, M1, result);
        }
    }

    else if (strcmp(argv[5], "multiply") == 0) {
        double result[N1][M2];
        multiplyMatrices(N1, M1, A, N2, M2, B, result);
        if (argc == 7 && strcmp(argv[6], "print") == 0) {
            if (M1 != N2)
                return 1;
            else {
                printf("Result of A * B:\n");
                printMatrix(N1, M2, result);
            }
        }
    }

    else if (strcmp(argv[5], "solve") == 0) {
        double result[N1][1];
        solveLinearSystem(N1, M1, A, N2, M2, B, result);
        if (argc == 7 && strcmp(argv[6], "print") == 0) {
            if (N1 != M1 || N2 != N1 || M2 != 1)
                return 1;
            else {
                printf("Result of x = B/A:\n");
                printMatrix(N1, M2, result);
            }
        }
    }

    else
        printf("Please enter a valid operation (e.g. add, subtract,
            multiply, solve)!\n");

    return 0;

}
```