Assignment 3: Programming Report

Lucas Machado

Student ID: 400507829

Scientific Programming: Operations on Matrices

Monday, March 25, 2024

1 Solving the Linear System of equations $Ax = b$

To solve the system of linear equations, we will use Gaussian Elimination. For our algorithm to solve the system, we will use forward elimination and backward substitution. However, we will not use row swapping as it is not necessary to implement, and if we were to implement it, then the code would be more complex, and we would have to deal with more factors. To simplify the algorithm's explanation, we will walk through an example, explain how it works, and provide examples to go with the explanations to make it as easy as possible for anyone to understand. To keep things very simple, we will work with an arbitrary 2×2 matrix A and an arbitrary 2×1 vector B to explain the first steps in the algorithm thus,

Given a matrix A (2×2) and a vector B (2×1):

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}, \quad B = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$

1.1 Initial Conditions on Matrices

1. Check to make sure that the matrix A is square $(N1 = M1)$.
2. Verify that the number of rows in Matrix A is equal to the number of rows in Matrix B $(N1 = N2)$.
3. Ensure that Matrix B is a column vector B $(M2 = 1)$.

1.2 Creating Augmented Matrix

Create an augmented matrix: by using matrix A and appending matrix B to the end of it.

$$\text{Augmented Matrix} = \begin{pmatrix} a_{11} & a_{12} & | & b_1 \\ a_{21} & a_{22} & | & b_2 \end{pmatrix}$$

1.3 Normalization of the Pivot Row

We want to normalize each pivot row before we start the forward elimination. This means we will divide every number in the pivot row by the pivot element, which makes the pivot element that we currently work with 1. That is the number on the diagonal of the matrix.

Hence, the pivot number in the first row is a_{11}. Moreover, we divide all of the elements in the pivot row by a_{11}, which makes the first pivot number equal to 1:

$$\text{Row}_1 = \frac{\text{Row}_1}{a_{11}}$$

Then the augmented matrix would look like this:

$$\begin{pmatrix} 1 & \frac{a_{12}}{a_{11}} & | & \frac{b_1}{a_{11}} \\ a_{21} & a_{22} & | & b_2 \end{pmatrix}$$

Doing this will make eliminating the elements below the pivot element easier.

1.4 Forward Elimination Continued

Now, we will make the number below our pivot number 0, a_{21}, and we will do this to all of the elements if there were more. This is done by the following:

$$\text{Row}_2 = \text{Row}_2 - a_{21} \times \text{Row}_1$$

Then the augmented matrix would look like:

$$\begin{pmatrix} 1 & \frac{a_{12}}{a_{11}} & | & \frac{b_1}{a_{11}} \\ 0 & a'_{22} & | & b'_2 \end{pmatrix}$$

with a'_{22} and b'_2 being the changed numbers after the elimination

1.5 Backward Substitution

After we do the forward elimination, we will get the augmented matrix to be in upper triangular form. This makes it very simple to solve for the unknowns, as we start from the last row and then substitute the known variables into the equations we do not know. We keep doing this until we find all of the unknowns.

Form the example that we have been using to find the value of x_2 we perform,

$$x_2 = \frac{b'_2}{a'_{22}}$$

Then, we substitute x_2 into the equation of the first row of the augmented matrix, and this will let us find what x_1 is, and thus, we will solve the system. Now, let us go through how the program would run.

$$x_1 = \frac{b_1 - a_{12}x_2}{a_{11}}$$

1.6 Programming Running Example

Now, we will provide an example of running the program and explain the output and the process of obtaining the solution.

1.6.1 Input

We first run make and then, ./math_matrix 2 2 2 1 solve print in the command line, and then we get a random matrix A and vector B. Please note that for this example, we are looking at the main sections of the code and not the parts that handle errors to ensure that it is easy to understand:

$$A = \begin{pmatrix} 3.302049 & -2.470746 \\ -5.825511 & -9.356506 \end{pmatrix}, \quad B = \begin{pmatrix} 5.202234 \\ -6.052324 \end{pmatrix}$$

1.6.2 Initial Augmented Matrix

The augmented matrix that we get from appending B to A:

$$\text{Augmented Matrix} = \begin{pmatrix} 3.302049 & -2.470746 & | & 5.202234 \\ -5.825511 & -9.356506 & | & -6.052324 \end{pmatrix}$$

The code that dose this is below as we simply add the values to A and then append B at the end,

```
for (int row = 0; row < N1; row++) {
    for (int col = 0; col < M1; col++) {
        augmentedMatrix[row][col] = A[row][col];
    }
    augmentedMatrix[row][M1] = B[row][0];
}
```

1.6.3 Normalizing Row 1

Then, the first row is normalized by dividing each element in the pivot row by the pivot number, which is 3.302049:

$$\begin{pmatrix} 1.000000 & -0.748246 & | & 1.575457 \\ -5.825511 & -9.356506 & | & -6.052324 \end{pmatrix}$$

The code that does this is below. We iterate through the pivot row and divide every element by this first pivot value.

```
double pivot = augmentedMatrix[pivotRow][pivotRow];
for (int col = pivotRow; col <= M1; col++) {
    augmentedMatrix[pivotRow][col] /= pivot;
}
```

1.6.4 Forward Elimination for Row 1

After normalization, the program begins the forward elimination to make the numbers below the pivot zero. The matrix then turns into the following:

$$\begin{pmatrix} 1.000000 & -0.748246 & | & 1.575457 \\ 0.000000 & -13.715423 & | & 3.125515 \end{pmatrix}$$

This step includes subtracting the correct multiple of the first row from the second row, and the code that does this is below:

```
for (int elimRow = pivotRow + 1; elimRow < N1; elimRow++) {
    double factor = augmentedMatrix[elimRow][pivotRow];
    for (int col = pivotRow; col <= M1; col++) {
        augmentedMatrix[elimRow][col] -= factor * augmentedMatrix[pivotRow][col];
    }
}
```

1.6.5 Normalizing Row 2

Then the program normalizes the second row just like we did the first time, but now we are dividing the second pivot row by the pivot number, which is -13.715423, and we get:

$$\begin{pmatrix} 1.000000 & -0.748246 & | & 1.575457 \\ 0.000000 & 1.000000 & | & -0.227883 \end{pmatrix}$$

This ensures the pivot element of the second row is also 1.

1.6.6 Back Substitution

Now, after the forward substitution is done, we have the augmented matrix in the upper triangular form:

$$\begin{pmatrix} 1.000000 & -0.748246 & | & 1.575457 \\ 0.000000 & 1.000000 & | & -0.227883 \end{pmatrix}$$

Now, the program starts the back substitution from the last row to solve for the variables. And the last row corresponds to the following equation $0 \cdot x_1 + 1 \cdot x_2 = -0.227883$, then we solve for x_2:

$$x_2 = -0.227883$$

Then the program moves to the first row which is $1 \cdot x_1 - 0.748246 \cdot x_2 = 1.575457$, and then the program substitutes the value of x_2 to solve for x_1:

$$x_1 = 1.575457 + 0.748246 \times (-0.227883)$$

This results in:

$$x_1 = 1.404944$$

Then the program is done all of the steps, and the result is the solution to the system of linear equations, which is, $Ax = B$ is:

$$x = \begin{pmatrix} 1.404944 \\ -0.227883 \end{pmatrix}$$

The code that performs the back substitution is below. It starts by initializing each x_i with the value from the augmented matrix corresponding to b_i. Then, it iteratively subtracts the known variables multiplied by their coefficients and then divides the coefficient of the unknown that is currently being solved if it is not normalized to 1 already.

```
for (int row = N1 - 1; row >= 0; row--) {
    x[row] = augmentedMatrix[row][M1];
    for (int col = row + 1; col < N1; col++) {
        x[row] -= augmentedMatrix[row][col] * x[col];
    }
    if (augmentedMatrix[row][row] != 1) {
        x[row] /= augmentedMatrix[row][row];
    }
}
```

2 Segmentation Fault

Please note that I am on Mac OS using a MacBook M2 Pro, and using LLDB as GDB is not an option; due to this, debugging with LLDB is different than with GDB, and it needs to be more specific. I kept getting the following error, `queue = 'com.apple.main-thread', stop reason = EXC_BAD_ACCESS (code=2, address=0x16f603ff8)`. This was a common error for those trying to debug a segmentation fault across the internet using LLDB. Despite getting this error, I used Replit to try and use the GDB, and it worked, but it still needed to be specific. Despite this, I was able to get more information about the error. Now, let us explain what is happening.

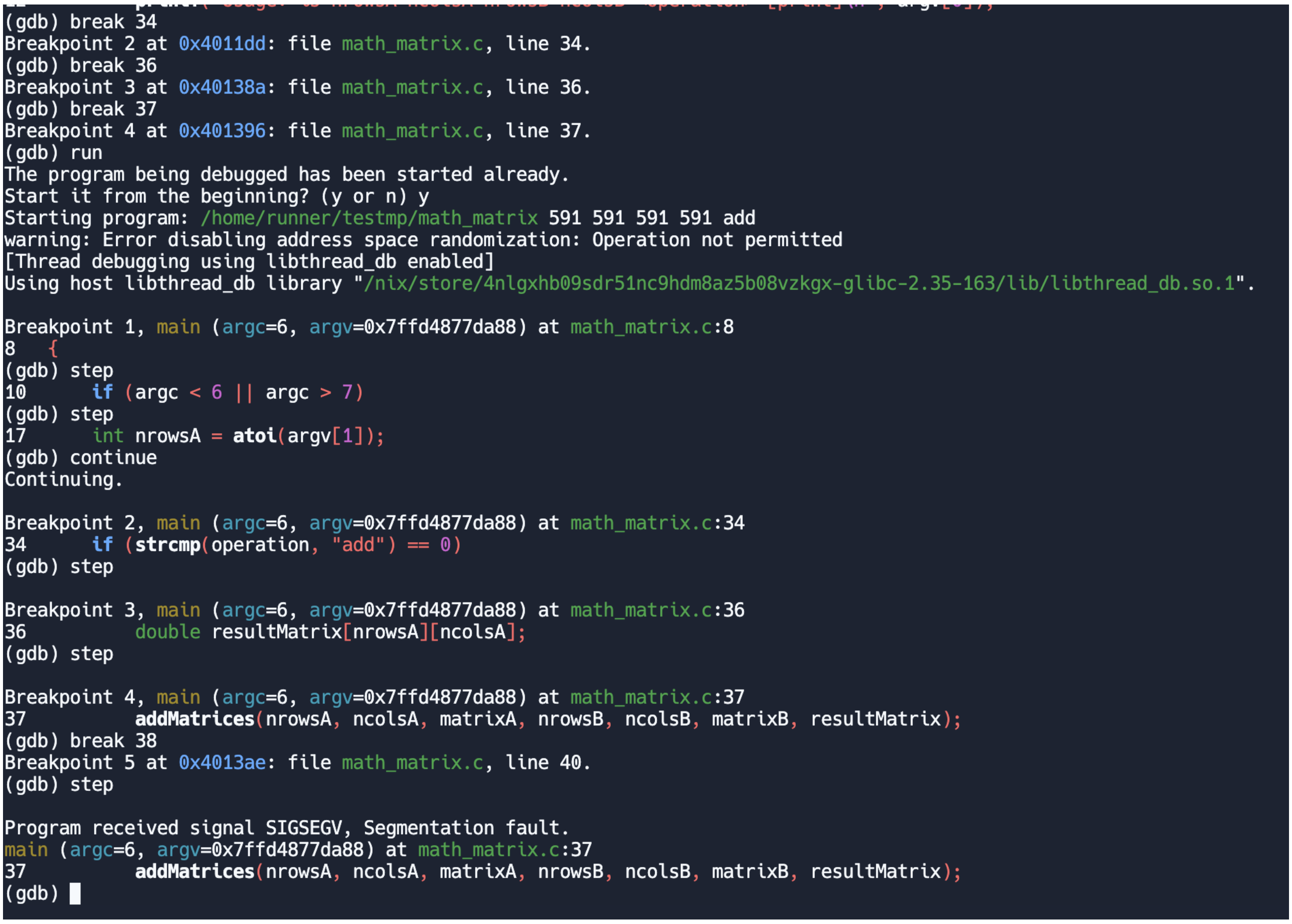

Figure 1: Segmentation fault, terminal GDB debugging

As we can see from the figure, the line causing the is 37, but since that is the Replit file, the line is off from the actual file, and in my VScode, the line is 41. If we were using GDB on Windows, we could get more information, but this tells us enough information, and we can break down why this is happening. We are calling ./math matrix 591 591 591 591 add, which is creating a vast matrix that is 591*591 sized matrix A and 591*591 sized matrix B, and the program is trying to add these two matrices and stores them in a result matrix of the same size. The main issue is why we are getting the segmentation fault error issues with stack memory overflow. This is because of the allocation of a large matrix as local variables in the primary function exceed the stack size limit of my environment.The allocated memory for the matrices in the `addMatrices` function is substantial. Specifically, for three matrices each of size 591×591 doubles, the required memory is:

$$3 \times 591^2 \times 8\,\text{bytes}$$

With the notion that each double usually occupies 8 bytes, the total memory allocation is approximately 24 MB. This substantial allocation on the stack can easily lead to a segmentation fault due to exceeding the stack's memory limits. One potential solution is to use dynamic memory allocation, which allocates memory on the heap instead of the stack.

3 Running The Program

In order to run the program, follow the commands below:

1. Compile the program:

```
make
```

2. Run the program:

```
./math_matrix <nrowsA> <ncolsA> <nrowsB> <ncolsB> <operation> [print]
```

4 Appendix

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void generateMatrix(int rows, int cols, double matrix[rows][cols])
{
    // Iterate over each row of the matrix
    for (int i = 0; i < rows; i++)
    {
        // Within each row iterate over each column
        for (int j = 0; j < cols; j++)
        {
            // Assign a random double from -10.0 to 10.0 to the current matrix element
            matrix[i][j] = ((double)rand() / (double)(RAND_MAX)) * 20.0 - 10.0;
        }
    }
}

// Prints a matrix with specified dimensions
void printMatrix(int rows, int cols, double matrix[rows][cols])
{
    // Loop through each row.
    for (int i = 0; i < rows; i++)
    {
        // Loop through each column in the current row
        for (int j = 0; j < cols; j++)
        {
            // Print the current element with 6 decimal places
            printf("%.6f ", matrix[i][j]);
        }
        printf("\n");
    }
}
```

```c

// Adds two matrices A and B and then stores result in the result matrix
void addMatrices(int N1, int M1, double A[N1][M1], int N2, int M2, double B[N2][M2], double result[N1][M1])
{
    // Check if dimensions of both matrices are the same
    if ((N1 == N2) && (M1 == M2))
    {
        for (int i = 0; i < N1; i++)
        {
            for (int j = 0; j < M1; j++)
            {
                // Add corresponding elements and store in result
                result[i][j] = A[i][j] + B[i][j];
            }
        }
    }
    else
    {
        printf("Cannot add the matrices because of incompatible matrix sizes\n");
    }
}

// Subtracts matrix B from matrix A and stores in a result matrix
void subtractMatrices(int N1, int M1, double A[N1][M1], int N2, int M2, double B[N2][M2], double result[N1][M1])
{
    // check that matrices A and B have the same dimensions
    if ((N1 == N2) && (M1 == M2))
    {
        for (int i = 0; i < N1; i++)
        {
            for (int j = 0; j < M1; j++)
            {
                // Subtract element of B from A and store in result
                result[i][j] = A[i][j] - B[i][j];
            }
        }
    }
    else
    {
        printf("Cannot subtract the matrices due to incompatible sizes\n");
    }
}

// Function to multiply two matrices and stores in result in result matrix
void multiplyMatrices(int N1, int M1, double A[N1][M1], int N2, int M2, double B[N2][M2], double result[N1][M2])
{
    // checking to make sure that the matrix multiplication can be done
    if (M1 == N2)
    {
        // iterate through the rows of the matrix
        for (int rowA = 0; rowA < N1; rowA++)
        {
            // iterate through the cols of the matrix
            for (int colB = 0; colB < M2; colB++)
            {
                double sum = 0.0;
                /*
```

```
                iterate along the rows and down the columns to perform the
                    multiplication
                by multiplying the values and then adding them up
                */
                for (int k = 0; k < M1; k++)
                {
                    sum += A[rowA][k] * B[k][colB];
                }
                // assign the sum to the result matrix
                result[rowA][colB] = sum;
            }
        }
    }
    else
    {
        printf("Cannot add the matrices because of incompatible matrix sizes
            \n");
    }
}

void solveLinearSystem(int N1, int M1, double A[N1][M1], int N2, int M2,
    double B[N2][M2], double x[N1])
{
    /*
    Check if the dimesnsions are right to be able to sovle the linear system
        ,
    if not then we will print a message saying that we cannot solve the
        system becuase of this
    */
    if (N1 == M1 && N1 == N2 && M2 == 1)
    {
        // initalize an augmented matrix that holds matrix A and the vector
            B
        double augmentedMatrix[N1][M1 + 1];

        // use a nested for loop to create the augmented matrix by adding
            the values to the matrix
        for (int row = 0; row < N1; row++)
        {
            for (int col = 0; col < M1; col++)
            {
                augmentedMatrix[row][col] = A[row][col];
            }
            // Setting b as the last column in the augmented matrix,
            //  after filling in the values from matrix A
            augmentedMatrix[row][M1] = B[row][0];
        }

        // Forward elimination to put the augmented matrix in an upper
            triangular form
        for (int pivotRow = 0; pivotRow < N1; pivotRow++)
        {
            // Check if the pivot element is 0 if it is 0 then there will be
                division by zero
            // thus we cannot do that
            if (augmentedMatrix[pivotRow][pivotRow] != 0)
            {
                // Normalize the pivot row by dividing by the pivot element
                double pivot = augmentedMatrix[pivotRow][pivotRow];
                for (int col = pivotRow; col <= M1; col++)
                {
```

```c
                    augmentedMatrix[pivotRow][col] /= pivot;
                }

                // Eliminate elements below the pivot to create zeros
                for (int elimRow = pivotRow + 1; elimRow < N1; elimRow++)
                {
                    double factor = augmentedMatrix[elimRow][pivotRow];
                    for (int col = pivotRow; col <= M1; col++)
                    {
                        augmentedMatrix[elimRow][col] -= factor * augmentedMatrix[pivotRow][col];
                    }
                }
            }
            else
            {
                printf("Math error a zero pivot the solution may not be correct\n");
            }
        }

        // Back substitution to solve for variables
        for (int row = N1 - 1; row >= 0; row--)
        {
            // Initialize with the value from the augmented matrix
            x[row] = augmentedMatrix[row][M1];
            for (int col = row + 1; col < N1; col++)
            {
                // subtract the known variables
                x[row] -= augmentedMatrix[row][col] * x[col];
            }
            // make sure that the row is normalized
            if (augmentedMatrix[row][row] != 1)
            {
                x[row] /= augmentedMatrix[row][row];
            }
        }
    }
    else
    {
        printf("Incompatible dimensions.\n");
    }
}
```

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

int main(int argc, char *argv[])
{
    // Check if the number of arguments is correct
    if (argc < 6 || argc > 7)
    {
        printf("Usage: %s nrowsA ncolsA nrowsB ncolsB <operation> [print]\n", argv[0]);
        return 1;
    }

    // Creating variables based on the input form the command line
```

```
    int nrowsA = atoi(argv[1]);
    int ncolsA = atoi(argv[2]);
    int nrowsB = atoi(argv[3]);
    int ncolsB = atoi(argv[4]);
    char *operation = argv[5];

    // Seed the random number generator
    srand(time(NULL));

    // Initalize two sizes for matrix A and matrix B
    double matrixA[nrowsA][ncolsA];
    double matrixB[nrowsB][ncolsB];

    // Fill in matrix A and matrix B with random values
    generateMatrix(nrowsA, ncolsA, matrixA);
    generateMatrix(nrowsB, ncolsB, matrixB);

    // Checking which operation the user wants to perform (add)
    if (strcmp(operation, "add") == 0)
    {
        // Initializing the side for the result matrix
        double resultMatrix[nrowsA][ncolsA];

        // Calling the add matrices funtion to add matrix A and B
        addMatrices(nrowsA, ncolsA, matrixA, nrowsB, ncolsB, matrixB,
            resultMatrix);

        /*
        Check if the user wants to print the result matrix,
        and that the matrix addition is possible, if so
        then print out the matrix A and B and the result matrix
        */
        if (argc == 7 && strcmp(argv[6], "print") == 0 && (nrowsA == nrowsB)
             && (ncolsA == ncolsB))
        {
            printf("Matrix A:\n");
            printMatrix(nrowsA, ncolsA, matrixA);

            printf("\nMatrix B:\n");
            printMatrix(nrowsB, ncolsB, matrixB);

            printf("\nResult of A + B:\n");
            printMatrix(nrowsA, ncolsA, resultMatrix);
        }
    }
    // Checking which operation the user wants to perform (subtract)
    else if (strcmp(operation, "subtract") == 0)
    {
        // Initializing the side for the result matrix
        double resultMatrix[nrowsA][ncolsA];
        // Calling the function to subtract matrix A and B
        subtractMatrices(nrowsA, ncolsA, matrixA, nrowsB, ncolsB, matrixB,
            resultMatrix);

        /*
        Check if the user wants to print the result matrix,
        and that the matrix subtraction is possible, if so
        then print out the matrix A and B and the result matrix
        */
        if (argc == 7 && strcmp(argv[6], "print") == 0 && (nrowsA == nrowsB)
             && (ncolsA == ncolsB))
```

```
        {
            printf("Matrix A:\n");
            printMatrix(nrowsA, ncolsA, matrixA);

            printf("\nMatrix B:\n");
            printMatrix(nrowsB, ncolsB, matrixB);

            printf("\nResult of A - B:\n");
            printMatrix(nrowsA, ncolsA, resultMatrix);
        }
    }
    // Checking which operation the user wants to perform (multiply)
    else if (strcmp(operation, "multiply") == 0)
    {
        // Initializing the side for the result matrix
        double resultMatrix[nrowsA][ncolsB];

        // call the multiply matrices function
        multiplyMatrices(nrowsA, ncolsA, matrixA, nrowsB, ncolsB, matrixB,
            resultMatrix);

        /*
        Check if the user wants to print the result matrix,
        and that the matrix addtion is possible, if so
        then print out the matrix A and B and the result matrix
        */
        if (argc == 7 && strcmp(argv[6], "print") == 0 && (ncolsA == nrowsB)
            )
        {
            printf("Matrix A:\n");
            printMatrix(nrowsA, ncolsA, matrixA);
            printf("\nMatrix B:\n");
            printMatrix(nrowsB, ncolsB, matrixB);

            printf("\nResult of A * B:\n");
            printMatrix(nrowsA, ncolsB, resultMatrix);
        }
    }

    // Checking which operation the user wants to perform (solve)
    else if (strcmp(operation, "solve") == 0)
    {
        //initialized vector to store the solution
        double resultVector[nrowsA][1];

        // Call the solveLinearSystem function
        solveLinearSystem(nrowsA, ncolsA, matrixA, nrowsB, ncolsB, matrixB,
            resultVector);

        /*
        Check if the user wants to print the result and
        that the input matrices are valid; if so, then print out the
            matrices A and B and the result matrix
        */
        if ((argc == 7 && strcmp(argv[6], "print") == 0) && (nrowsA ==
            ncolsA && nrowsA == nrowsB && ncolsB == 1))
        {
            printf("Matrix A:\n");
            printMatrix(nrowsA, ncolsA, matrixA);

            printf("\nMatrix B:\n");
```

```c
                printMatrix(nrowsB, ncolsB, matrixB);

                // print out the solution
                printf("\nResult of x=B/A:\n");
                for (int i = 0; i < nrowsA; i++)
                {
                    printf("%f\n", resultVector[i][0]);
                }
            }
        }

        else
        {
            printf("Invalid operation specified.\n");
        }

        return 0;
}
```

```makefile
# Define the compiler
CC = gcc

# Define any compile-time flags
CFLAGS = -Wall -g

# Define the source file names
SRC = math_matrix.c functions.c

# Define the executable file name
EXEC = math_matrix

# Default target
all: $(EXEC)

$(EXEC): $(SRC)
	$(CC) $(CFLAGS) -o $(EXEC) $(SRC)

# Clean target for removing compiled binary and object files
clean:
	rm -f $(EXEC) *.o

# Phony targets for commands that do not represent files
.PHONY: all clean
```