Scientific Programming: Operations on Matrices

George Ghiugan

March 25, 2024

Part A: Matrix Solver Algorithm

The `solveLinearSystem()` function is used to solve system of linear equations using Gauss Jordan Elimination method. The algorithm validates the input sizes of the given matrices and then row reduces the matrix to the identity matrix in order to solve the linear system of equations (Ax = B for x). The algorithm works as follows:

Compatibility Check: The if statement checks if the first matrix A is square and that matrix B has same rows as matrix A:

```
void solveLinearSystem(int N1, int M1, double A[N1][M1], int N2,
   int M2, double B[N2][M2], double x[N1][M2])
{
    // Check if the size of the matrices are compatible
    if (N1 == M1 && N1 == N2 && M2 == 1)
    {
        // Algorithm starts here...
    }
}
```

Pivot Validation: This part checks if the pivot element of the current row in matrix A is zero. If it's zero, it prints an error message indicating that division by zero was encountered and exits the loop:

```
        // Iterate through each row of matrix A
        for (int i = 0; i < N1; i++)
        {
            // Check if each pivot is valid
```

```
if (A[i][i] == 0) {
    printf("Division by zero encountered: No Solution."
       );
    break;
}
```

Pivot Scaling: In this step, the pivot value is stored, and then each entry in the current row of matrix A is divided by the pivot value. The corresponding entry in matrix B is also divided by the pivot value accordingly:

```
// Divide each entry in the row by the pivot value
// Adjust the corresponding value in matrix B
   accordingly
float pivotValue = A[i][i];
B[i][0] /= pivotValue;
for (int j = 0; j < M1; j++)
{
    A[i][j] /= pivotValue;
}
```

Row Operations: This is the step which row reduces the matrix to the identity matrix in order to find the solution vector for x. It starts by looping through each row except the current row. Then it gets the elimination factor which is the number above or below the pivot. It then multiplies the current row by the elimination factor and subtracts two rows in order to zero out the columns. Note that the procedure for row operations on matrix A is also performed on matrix B. It continues to do this step within the k loop until only the pivot entries remain, thus reaching an identity matrix:

```
// Perform row operations to eliminate entries above
   and below each pivot
for (int j = 0; j < N1; j++)
{
    // Skip the current row
    if (j == i)
    {
        continue;
    }

    // The elimination factor is the matrix entry
       directly above or below the pivot
```

```
                float eliminationFactor = A[j][i];

                // Multiply the i-th row by the eliminationFactor
                   and then subtract each row.
                for (int k = 0; k < M1; k++)
                {
                    A[j][k] -= A[i][k] * eliminationFactor;
                }
                B[j][0] -= B[i][0] * eliminationFactor;
            }

        }
```

Solution Transfer: This step transfers the values obtained in B to the solution vector:

```
        // Transfer solved matrix values from matrix B over to the
           solution matrix.
        for (int i = 0; i < N2; i++)
        {
            x[i][0] = B[i][0];
        }
```

Error Handling: If the matrices are not compatible as determined in the first if statement, it jumps here where the program prints an error message indicating that the input matrices must be a square matrix and a vector:

```
    // Error message for incompatible matrix sizes
    else
    {
        printf("Invalid matrices. The input must be a square matrix
            and a vector.\n");
    }
```

Part B: Segmentation Fault Debugging

This section of the report talks about the debugging of my code using GDB and the key findings in regards to the Segmentation Fault Error. I tested values of different size of matrices using the `./math_matrix N N N N add` syntax where N represents the size of the matrices being added. While debugging, I found three specific points of interest where the code breaks depending on the size of the matrices inputted:

First Point:

```
Program received signal SIGSEGV, Segmentation fault.
0x0000555555555943 in main (argc=6, argv=0x7fffffffde68) at
   math_matrix.c:64
64              double resultMatrix[rowsA][colsA];
```

When the matrix size is 592x592 the program breaks down at line 61. This line is where the third matrix, `resultMatrix[rowsA][colsA];` of the program is declared. When printing the matrix in GDB it says:

`value requires 2803712 bytes, which is more than max-value-size`.

This message means that the size of the matrix is to big and it exceeds the amount of memory that has been allocated to it which crashes the program.

Second Point:

```
Program received signal SIGSEGV, Segmentation fault.
0x00005555555556ac in main (argc=6, argv=0x7fffffffde48) at
   math_matrix.c:43
43          double matrixTwo[rowsB][colsB];
```

When the matrix size is increased to 724x724 dimension, the program breaks down at line 43. This line is where the **second** matrix, `matrixTwo[rowsA][colsA];` of the program is declared. When printing `matrixTwo` in GDB it says:

`value requires 4193408 bytes, which is more than max-value-size`.

Similar to the first point the message means that the size of the matrix is to big and it exceeds the amount of memory that has been allocated to it which crashes the program. Notice that the amount of memory to store the values of this bigger matrix requires more bytes.

Third Point:

```
Program received signal SIGSEGV, Segmentation fault.
0x00005555555554e3 in main (argc=6, argv=0x7fffffffde68) at
   math_matrix.c:42
42          double matrixOne[rowsA][colsA];
```

When the matrix size is increased to 1024x1024 dimension, the program breaks down at line 42. This line is where the **first** matrix, `matrixOne[rowsA][colsA];` of the program is declared. When printing `matrixOne` in GDB it says:

`value requires 8388608 bytes, which is more than max-value-size`.

The message means that the size of the matrix is to big and it exceeds the amount of memory that has been allocated to it which crashes the program. Notice that the amount of memory to store the values of this bigger matrix requires more bytes than the first and second points. Using this information from the 1024x1024 error, we are able to determine the amount of memory which is allocated for the double matrices in the program. Refer to the explanation below for a detailed analysis.

Explanation:

Through debugging we were able to distinguish when the size of one matrix crashes the program. When the size of the matrix exceeds 1023x1023 the program crashes at the first matrix declaration so somewhere in the memory space allocated between 1023x1023 and 1024x1024 matrix represents a limit before crashing. Since we know that 1023 does not crash the first matrix declaration we can use this value to calculate the amount of memory allocated. In C, each double takes 8 bytes of memory so a 1023x1023 matrix takes up 1023*1023*8 = 8372232 bytes of memory. Therefore the limit memory allocated for such a matrix is around 8.37 MB. This memory allocation number verifies why the code crashes at the other two points in the program. The program crashes at matrixTwo declaration when the size of the matrices exceed 723x723 dimensions. This is supported by the calculation 724*724*8*2 = 8.387 MB (multiply by two since declaring two matrices before program crashing). Since 8.387 MB $>$ 8.37 MB it verifies why the program crashes at the second matrix declaration. Similarly, the program crashes when creating three matrices of sizes larger than 591x591. The calculation 592*592*8*3 = 8.41 MB exceeds the limit of 8.37 MB, thus crashing the program when creating the matrices.

Note: The code/debugging was done on Windows Subsystem for Linux (WSL) using AMD Ryzen 7 6800HS processor on Windows 11.

Appendix:

math_matrix.c:

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

int main(int argc, char *argv[])
{
    // Seed the random number generator
    srand(time(0));

    // Check if the number of arguments passed in the command line
       is valid. Print message if there are under 6 arguments or
       over 7
    if (argc < 6 || argc > 7) {
        printf("Usage: %s nrows_A ncols_A nrows_B ncols_B <
           operation> [print]\n", argv[0]);
        return 1;
    }

    // Initialize variables for matrix dimensions
    long rowsA;
    long colsA;
    long rowsB;
    long colsB;
    char *p;

    // Retrieve the command line arguments for dimensions of the
       first Matrix
    rowsA = strtol(argv[1], &p, 10);
    colsA = strtol(argv[2], &p, 10);

    // Retrieve the command line arguments for dimensions of the
       second Matrix
```

```
    rowsB = strtol(argv[3], &p, 10);
    colsB = strtol(argv[4], &p, 10);

    // Check for invalid row/column input from the user
    if (rowsA == 0 || rowsB == 0 || colsA == 0 || colsB == 0)
    {
        // Print error message to console
        printf("Invalid row/column input. Please make sure to enter
            a positive integer as row/column number.\n");
        return 1;
    }

    // Initialize the size of the two matrices using the values
       passed in the command line
    double matrixOne[rowsA][colsA];
    double matrixTwo[rowsB][colsB];

    // Generate the Matrices with random values between -10 and 10
       inclusive
    generateMatrix(rowsA, colsA, matrixOne);
    generateMatrix(rowsB, colsB, matrixTwo);

    // Print Matrices to console if user inputs the "print"
       agrument in terminal
    if(argc > 6 && strcmp(argv[6], "print") == 0)
    {
        // Print the first matrix to console
        printf("Matrix A: \n");
        printMatrix(rowsA, colsA, matrixOne);
        // Print the second matrix to console
        printf("Matrix B: \n");
        printMatrix(rowsB, colsB, matrixTwo);
    }

    // Check if fifth argument is "add" and perform matrix
       operation for addition
    if (strcmp(argv[5], "add") == 0 && rowsA == rowsB && colsA ==
       colsB)
    {
```

```
        // Declare the size of the result matrix to be populated
        double resultMatrix[rowsA][colsA];

        // Call the "addMatrices" function to perform the addition
           of the two matrices
        addMatrices(rowsA, colsA, matrixOne, rowsB, colsB,
           matrixTwo, resultMatrix);

        // Check if the user inputted the argument "print" and
           print the result matrix to the console
        if (argc > 6 && strcmp(argv[6], "print") == 0)
        {
            printf("Result of A + B: \n");
            printMatrix(rowsA, colsA, resultMatrix);
        }

    }

    // Check if fifth argument is "subtract" and perform matrix
       operation for subtraction
    else if (strcmp(argv[5], "subtract") == 0 && rowsA == rowsB &&
       colsA == colsB)
    {
        // Declare the size of the result matrix to be populated
        double resultMatrix[rowsA][colsA];

        // Call the "subtractMatrices" function to perform the
           subtraction of the two matrices
        subtractMatrices(rowsA, colsA, matrixOne, rowsB, colsB,
           matrixTwo, resultMatrix);

        // Check if the user inputted the argument "print" and
           print the result matrix to the console
        if (argc > 6 && strcmp(argv[6], "print") == 0)
        {
            printf("Result of A - B: \n");
            printMatrix(rowsA, colsA, resultMatrix);
        }
    }
```

```
// Check if fifth argument is "multiply" and perform matrix
   operation for multiplication
else if (strcmp(argv[5], "multiply") == 0 && colsA == rowsB)
{
    // Declare the size of the result matrix to be populated
    double resultMatrix[rowsA][colsB];

    // Call the "multiplyMatrices" function to perform the
       multiplication of the two matrices
    multiplyMatrices(rowsA, colsA, matrixOne, rowsB, colsB,
       matrixTwo, resultMatrix);

    // Check if the user inputted the argument "print" and
       print the result matrix to the console
    if (argc > 6 && strcmp(argv[6], "print") == 0)
    {
        printf("Result of A * B: \n");
        printMatrix(rowsA, colsB, resultMatrix);
    }

}

// Check if fifth argument is "solve" and perform operations to
    solve system of linear equations
else if (strcmp(argv[5], "solve") == 0 && rowsA == rowsB &&
   colsB == 1 && rowsA == colsA)
{
    // Declare the size of the solution vector x to be
       populated
    double resultMatrix[rowsA][1];

    //  Call the "solveLinearSystem" function to solve the
       system of linear equations
    solveLinearSystem(rowsA, colsA, matrixOne, rowsB, colsB,
       matrixTwo, resultMatrix);

    // Check if the user inputted the argument "print" and
       print the result vector to the console
```

```
        if (argc > 6 && strcmp(argv[6], "print") == 0)
        {
            printf("Result of x=B/A: \n");
            printMatrix(rowsA, 1, resultMatrix);
        }
    }

    // Invalid operation
    else
    {
        // Print error message
        printf("Invalid operation. Please make sure matrix sizes
           are compatible for add, subtract, multiply, or solving
           linear systems.\n");
    }

    return 0;
}
```

functions.c:

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

// Function that generates a matrix of given size with random
   values between -10 and 10 inclusive
void generateMatrix(int rows, int cols, double matrix[rows][cols])
{
    // Declare and initialize the values representing the range of
       the random numbers
    int min = -10;
    int max = 10;

    // Iterate through double for loop to populate the matrix with
       random values ranging from -10 to 10
    for (int i = 0; i < rows; i++)
    {
```

```
        for (int j = 0; j < cols; j++)
        {

            // Generate a random value between -10 and 10 to be
               placed in the matrix
            matrix[i][j] = min + ((double)rand() / RAND_MAX) * (max
                - min);
        }
    }
}

// Function to print the matrix
void printMatrix(int rows, int cols, double matrix[rows][cols])
{
    // Loop through each element of the matrix
    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < cols; j++)
        {
            // Adjust spacing to maintain alignment when printing
               negative numbers
            if (matrix[i][j] >= 0)
            {
                printf(" %f ", matrix[i][j]);
            }

            else
            {
                printf("%f ", matrix[i][j]);
            }
        }
        // Move to the next row after printing all elements in a
           row
        printf("\n");
    }
}

// Function to add two matrices
void addMatrices(int N1, int M1, double A[N1][M1], int N2, int M2,
```

```
   double B[N2][M2], double result[N1][M1])
{
    // Check if the matrices are of equal size
    if (N1 != N2 || M1 != M2)
    {
        // Error message
        printf("Incompatible matrix sizes.\n");
    }

    else
    {
        // Loop through the matrix and add corresponding entry
           values together
        for (int i = 0; i < N1; i++)
        {
            for (int j = 0; j < M1; j++)
            {
                result[i][j] = A[i][j] + B[i][j];
            }
        }
    }
}

// Function to subtract two matrices
void subtractMatrices(int N1, int M1, double A[N1][M1], int N2, int
    M2, double B[N2][M2], double result[N1][M1])
{
    // Check if the matrices are of the same dimensions
    if (N1 != N2 || M1 != M2)
    {
        // Error message
        printf("Incompatible matrix sizes.\n");
    }

    else
    {
        // Loop through each matrix and subtract the values from
           the corresponding entries
        for (int i = 0; i < N1; i++)
```

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

// Function to multiply two matrices
void multiplyMatrices(int N1, int M1, double A[N1][M1], int N2, int
    M2, double B[N2][M2], double result[N1][M2])
{
    // Check if the matrix sizes are compatible
    if (M1 != N2)
    {
        // Error message
        printf("Incompatible matrix sizes.\n");
    }

    else
    {
        // Loop through the rows of first matrix and columns of
           second matrix
        for (int i = 0; i < N1; i++)
        {
            for (int j = 0; j < M2; j++)
            {
                // Multiply every entry in the row of matrix one
                   with the entry in the column of matrix two
                // and add them all together. Then add the result
                   to the final matrix.
                double entry = 0;
                for (int k = 0; k < M1; k++)
                {
                    entry += A[i][k] * B[k][j];
                }
                result[i][j] = entry;
            }
```

```
        }
    }
}

// Function to solve a linear system of equations Ax = B
void solveLinearSystem(int N1, int M1, double A[N1][M1], int N2,
   int M2, double B[N2][M2], double x[N1][M2])
{
    // Check if the size of the matrices are compatible
    if (N1 == M1 && N1 == N2 && M2 == 1)
    {
        // Iterate through each row of matrix A
        for (int i = 0; i < N1; i++)
        {
            // Check if each pivot is valid
            if (A[i][i] == 0)
            {
                printf("Division by zero encountered: No Solution."
                   );
                break;
            }

            // Divide each entry in the row by the pivot value
            // Adjust the corresponding value in matrix B
               accordingly
            float pivotValue = A[i][i];
            B[i][0] /= pivotValue;
            for (int j = 0; j < M1; j++)
            {
                A[i][j] /= pivotValue;
            }

            // Perform row operations to eliminate entries above
               and below each pivot
            for (int j = 0; j < N1; j++)
            {
                // Skip the current row
                if (j == i)
                {
```

```
                    continue;
                }

                // The elimination factor is the matrix entry
                   directly above or below the pivot
                float eliminationFactor = A[j][i];

                // Multiply the i-th row by the eliminationFactor
                   and then subtract each row.
                for (int k = 0; k < M1; k++)
                {
                    A[j][k] -= A[i][k] * eliminationFactor;
                }
                B[j][0] -= B[i][0] * eliminationFactor;
            }
        }

        // Transfer solved matrix values from matrix B over to the
           solution matrix.
        for (int i = 0; i < N2; i++)
        {
            x[i][0] = B[i][0];
        }
    }

    // Error message for incompatible matrix
    else
    {
        printf("Invalid matrices. The input must be a square matrix
            and a vector.\n");
    }
}
```

functions.h:

```
#ifndef FUNCTIONS_H
#define FUNCTIONS_H
// Function prototypes
void generateMatrix(int rows, int cols, double matrix[rows][cols]);
```

```c
void printMatrix(int rows, int cols, double matrix[rows][cols]);
void addMatrices(int N1, int M1, double A[N1][M1], int N2, int M2,
   double B[N2][M2], double result[N1][M1]);
void subtractMatrices(int N1, int M1, double A[N1][M1], int N2, int
    M2, double B[N2][M2], double result[N1][M1]);
void multiplyMatrices(int N1, int M1, double A[N1][M1], int N2, int
    M2, double B[N2][M2], double result[N1][M2]);
void solveLinearSystem(int N1, int M1, double A[N1][M1], int N2,
   int M2, double B[N2][M2], double x[N1][M2]);

#endif /* FUNCTIONS_H */
```

Makefile:

```
CC = gcc
CFLAGS = -Wall -Wextra -lm

all: main

main: math_matrix.c functions.c functions.h
 $(CC) -o math_matrix math_matrix.c functions.c $(CFLAGS)

clean:
 rm -f math_matrix
```