Matrix Library ABFT

Michael Dandrea

***Abstract*—This report covers a series of eight projects for CSCI 207 at Coastal Carolina University in the fall semester of 2023. These projects simulate different operations of matrices, and also includes the ability to corrupt, detect and correct a bit flip in a matrix index. The final project in the group simulates Hamming(7,4) encoding. ChatGPT was used to provide outlines for how to implement the functions in the utilities.c file.**

I. Project 1: C4

This first project was to create three driver programs as follows:

- make data 2d: Create a matrix of specified dimensions with values in a given limit and write it to a specified output file.
- read data 2d: Read a matrix from an input file and print it to the screen.
- add data 2d: Add two matrices from input files and write the result to a specified output file.

The programs can be run like this:

- ./make_data_2d [rows] [cols] [high [low] [outfile]
- ./read_data_2d [infile]
- ./add_data_2d [infile A] [infile B] [outfile]

In order for two matrices to be added, they must have the same dimensions.

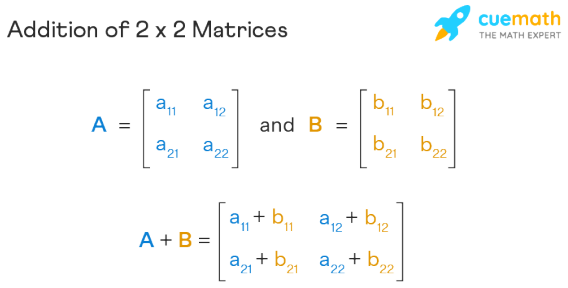

Figure 1: Example of matrix addition

Both matrices A and B need to be checked to make sure they are the same size. If they are not, they matrices cannot be added. Because of this, the program to add the matrices has error checking to make sure the matrices are the same size.

A. Valgrind

Because we are dealing with pointers and 2D Arrays, it is very common for segmentation faults to occur. One way to check for and troubleshoot these issues is to use a debugger like Valgrind. Normally, a program will just exit due to a segmentation fault, and not give us any information about where it occurs in the program. Valgrind tells us what line caused a segmentation fault to occur. Using this, we can go back and easily fix any segmentation faults, rather than trying to figure out where they are on our own.

II. Project 2: C5

Project 2 was to add four more driver programs to this project. The previous three can (and will) be used as well. The four driver programs added were:

- mul_data_2d: Multiply two matrices together and write the result to a specified output file.
- cs_rows_data_2d: Checksum the rows of the input matrix and write the result to a specified output file.
- cs_cols_data_2d: Checksum the columns of the input matrix and write the result to a specified output file.
- cs_data_2d: Checksum both the rows and columns of of an input matrix and write the result to a specified output file.

The programs can be run like this:

- ./mul_data_2d [infile A] [infile B] [outfile]
- ./cs_rows_data_2d [infile] [outfile]
- ./cs_cols_data_2d [infile] [outfile]
- ./cs_data_2d [infile] [outfile]

Matrix multiplication is a bit more complicated than matrix addition. Using the inputs above, the number of rows in matrix B must equal the number of columns in matrix A. Once again, proper error

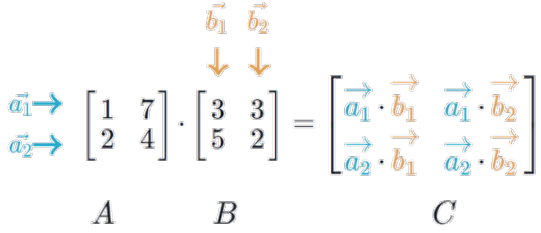

Figure 2: Example of matrix multiplication

checking was included to make sure the matrices are the correct dimensions to be multiplied. If they were not the correct size, a segmentation fault would have occurred. This could have easily been fixed using Valgrind, like stated before.

A. Checksum

Computing the row or column checksum of a matrix means to add the values in the row or column, and put the results in a new column (if you are checksumming the rows) or row (if you are checksumming the columns). We do this for each row row or column. For example, if the input matrix is a 2x2 matrix and we checksum the rows, the resulting matrix would be a 2x3 matrix after adding the column that contains the row checksums.

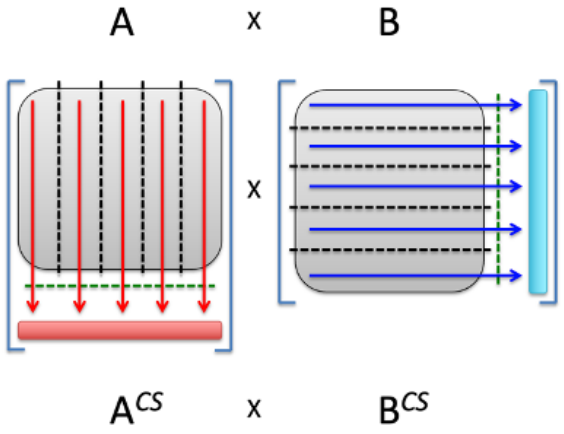

Figure 3: An example of column and row checksums

We can use these two checksummed matrices to create a fully checksummed matrix by multiplying them together.

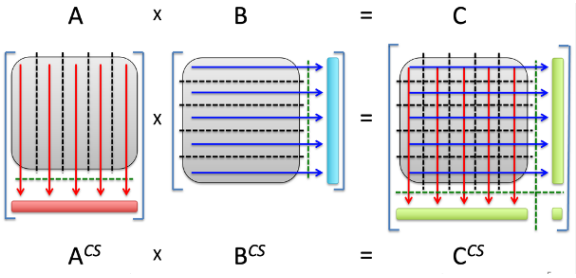

Figure 4: Getting a fully checksummed matrix through multiplication

This way, the resulting matrix C is a fully checksummed matrix that has already been multiplied. However, there is another way to do this using one of the driver programs we created for this project. The cs_data_2d program fully checksums a matrix, so it is possible to multiply A and B, then use cs_data_2d on the resulting matrix to fully checksum it (so it would only be matrix C in Figure 4).

III. Project 3: C6

The goal of project 3 was to update all of the driver programs with command line argument parsing using getopt() and optarg. Using getopt() and optarg lets us use flags to specify what each argument is, rather than having random numbers or file names. This also allows us to change the order of the arguments while running the command, as the flags specify what each input is rather than needing them in a specific order. The new commands to run the driver programs are as follows:

- ./make_data_2d -r [rows] -c [cols] -l [low] -h [high] -o [outfile]
- ./read_data_2d -i [infile]
- ./add_data_2d -a [infile a] -b [infile b] -c [outfile]
- ./mul_data_2d -a [infile a] -b [infile b] -c [outfile]
- ./cs_rows_data_2d -i [infile] -o [outfile]
- ./cs_cols_data_d2 -i [infile] -o [outfile]
- ./cs_data_2d -i [infile] -o [outfile]

All programs (except for read) also have an optional -p flag which will print the result of a program as well as all matrices involved, as well as the time took for the program to run like normal. For example, if -p is provided to the add program, matrix a, matrix b and matrix c will all be printed to the screen. Some of these programs have default values as well. This means that if no option is provided, no error will be thrown and instead they will be set to a default value. The defaults are as follows:

- -p if not specified, it is assumed to be off (i.e. this one is optional)
- -r if not specified, assumed to be 3 (i.e. this one is optional)
- -c if not specified, assumed to be 4 (i.e. this one is optional)
- -l if not specified, assumed to be 0 (i.e. this one is optional)
- -h if not specified, assumed to be 9 (i.e. this one is optional)
- -i if not specified, this is an error, must be specified (i.e. this one is required)
- -o if not specified, this is an error, must be specified (i.e. this one is required)
- -a if not specified, this is an error, must be specified (i.e. this one is required)
- -b if not specified, this is an error, must be specified (i.e. this one is required)
- -c if not specified, this is an error, must be specified (i.e. this one is required)

A. getopt() and optarg

The getopt() function and the optarg variable are built into C and are commonly used for parsing command-line arguments. They are part of the **unistd.h** library. The getopt() function has three parameters:

- argc: Number of command-line arguments.
- argv: Array of command-line argument strings.

- opstring: A string specifying the allowed options. If an option requires an argument, a colon (':') is placed after the option character in optstring.

The optarg variable is a global variable defined in the <unistd.h> header file. It points to the argument associated with the last processed option when using getopt(). If an option requires an argument, the argument value can be accessed using the optarg variable. Here is an example of parsing the command line arguments using my function that parses the arguments for make_data_2d:

```
void parseMake(int argc, char *argv[], int *rows, int *cols, int *min, int *max, char **fname, int *printFlag)
{
    *rows = 3; // Default value for rows.
    *cols = 4; // Default value for cols.
    *min = 0;  // Default value for min.
    *max = 9;  // Default value for max.

    int opt;
    while ((opt = getopt(argc, argv, "r:c:l:h:o:p")) != -1)
    {
        switch (opt)
        {
        case 'r':
            *rows = atoi(optarg);
            break;
        case 'c':
            *cols = atoi(optarg);
            break;
        case 'l':
            *min = atoi(optarg);
            break;
        case 'h':
            *max = atoi(optarg);
            break;
        case 'o':
            *fname = optarg;
            break;
        case 'p':
            *printFlag = 1;
            break;
        default:
            printf("Usage: %s -r <#rows> -c <#cols> -l <low> -h <high> -o <output filename> [-p]\n", argv[0]);
            exit(1);
        }
    }
}
```

Figure 5: Example command line argument parsing

The command line parsing for all of the driver functions looks similar to this, just with different default values and different flags.

IV. Project 4: C7

The goal of the fourth project was to add two more driver programs, corrupt_data_2d and detect_data_2d. Corrupt data takes an input file and corrupts the number at position (x,y) in the matrix. It does this by flipping the bit indicated by argument z. Detect data opens a given input file and determines in there has been a corruption by computing the row and column checksums to see if they match the last row and column of the matrix (this can only be done if the input matrix is checksummed). If no error is detected, it prints "NO ERROR DETECTED" to the screen. If there is an error, it prints "ERROR DETECTED AT ROW [rowindex] AND COLUMN [colindex]". The programs can be run as follows:

- ./corrupt_data_2d -i [infile] -x [row index] -y [col index] -z [bit position] -o [outfile]
- ./detect_data_2d -i [infile]

These programs also include the optional -p flag mentioned before, as well as new defaults for the corrupt program. They are:

- -x if not provided, x = UNIF[0, rows-1] (i.e. this one is optional)
- -y if not provided, y = UNIF[0, cols-1] (i.e. this one is optional)
- -z if not provided, z = UNIF[0, 31] (i.e. this one is optional)

where UNIF[p, q] means a uniformly distributed random number between p and q inclusive. So, if the number at the (x,y) was 5 for example, an we told it to flip bit 1, the resulting number would be 4 (0101 flipped at position zero is 0100). I accomplished this by XORing the number at (x,y) with (1 ¡¡ [bit]). The left shift guarantees that we are flipping the correct bit.

```
void corrupt(dtype **A, dtype **B, int rows, int cols, int crow, int ccol, int cbit)
{
    for(int i = 0; i < rows; i++){
        for(int j = 0; j < cols; j++){
            B[i][j] = A[i][j];
        }
    }

    B[crow][ccol] = B[crow][ccol] ^ (1 << cbit);
}
```

Figure 6: My function for corrupting the data

To check if there was a corruption, I summed the current row and column and if that sum did not equal the checksum, that is the row or column that has corrupted data. My code looks like this:

```
for(int i = 0; i < rows; i++){
    rowchecksum = A[i][cols - 1];
    rowsum = 0;
    for(int j = 0; j < cols - 1 ; j++){
        rowsum += A[i][j];
    }
    if(rowsum != rowchecksum){
        rowindex = i;
        break;
    }
}

for(int i = 0; i < cols; i++){
    colchecksum = A[rows - 1][i];
    colsum = 0;
    for(int j = 0; j < rows - 1; j++){
        colsum += A[j][i];
    }
    if(colsum != colchecksum){
        colindex = i;
        break;
    }
}

if(rowindex != -1 && colindex != -1){
    printf("Error detected at row %d, column %d\n", rowindex, colindex);
}else{
    printf("No error detected.\n");
}
```

Figure 7: My function for detecting a corruption

Since this compares the sum of the rows and columns with their checksums, this only works if the input matrix has been checksummed. Otherwise, it will always say an error has occurred.

V. Project 5: C8

The goal of this project was to be able to identify a corruption in a matrix and correct it. To do this, we made a new driver program called correct_data_2d. This takes an input file and if there is an error to correct, corrects it, and writes the result to an output file. If not, it states that there is no error, and still writes to an output file. Again, this has an optional -p

flag which will print both the corrupted and corrected matrices to the screen. The program can be run like this:

- ./correct_data_2d -i [infile] -o [outfile]

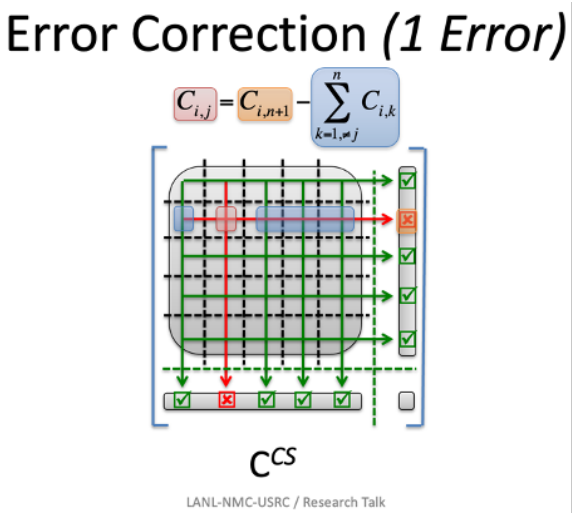

Figure 8: How to correct a matrix with one error

In order to correct the matrix, we must first find out where the corruption is. This can easily be done using the detect function we made previously, having it return the row and column index of the corruption. I did this by making the function return a struct which includes three ints, the row index, column index and bit position of the error. Then, we can sum all of the values in the row or column excecpt the one that has the error, and subtract that from the checksum of the row or column. The resulting value is what the correct value should be. However, in this case, since the corruption is caused by flipping a bit, we can correct it by flipping the same bit:

```
void correct(dtype **A, dtype **B, int rows, int cols)
{
    Index index = detect(A, rows, cols);
    int correctRow = index.row;
    int correctCol = index.col;
    int correctBit = index.bit;

    if (correctRow == -1 && correctCol == -1)
    {
        return;
    }

    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < cols; j++)
        {
            B[i][j] = A[i][j];
        }
    }

    B[correctRow][correctCol] = B[correctRow][correctCol] ^ (1 << correctBit);
}
```

Figure 9: My code to do the correction

This code sets the corrected matrix equal to the to corrupted matrix, then corrects the bit in the corrected matrix, so it can print both the corrupted and corrected matrices if the -p flag is provided.

VI. Project 6: C9

This project was different from the rest. It did not include any of the previous program. Instead, the goal of this program was to simulate a masked benign error when a random bit in a 32-bit unsigned integer is flipped. A masked benign error is when a bit gets corrupted but it results in the same number. To do this, we created a new driver program called simple_mbe_test. Once again, this program has an optional -p that will print every mbe to the screen, as well as always printing the probability of an mbe occurring. It can be ran like this:

- ./simple_mbe_test -n [number of trials]

This output could also be redirected to a csv file in order to make plots of the resulting data:

- ./simple_mbe_test -n [number of trials] -p > logfile.csv

This information could then be used in two python programs written for this project to create two graphs, one for making a histogram of the frequency an mbe occurs at which bit position, and another for showing the probability of an mbe in a given number of trials:

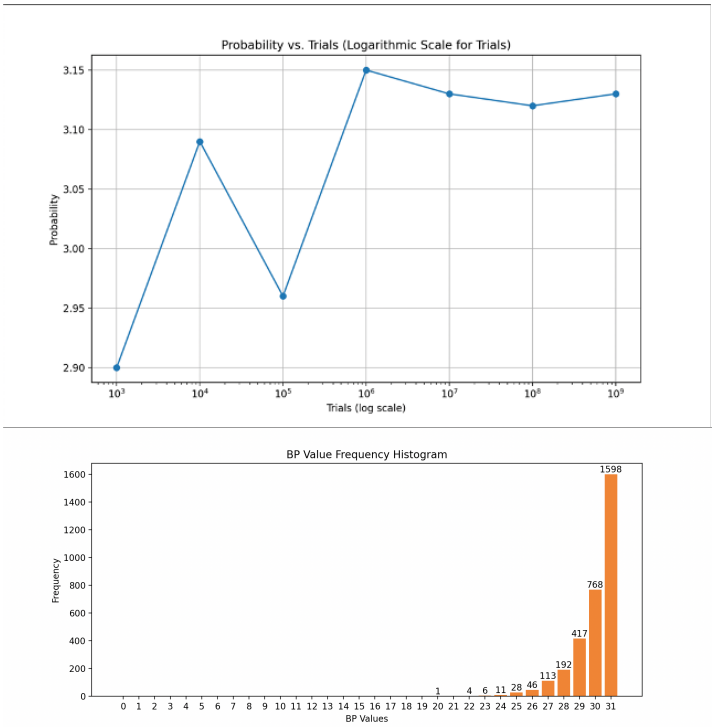

Figure 10: Examples of graphs created.

VII. Project 7: C10

This project combines all of the other projects. The goal of this project was to create a new driver program mul_data_2d_faulty that corrupts the data of a matrix DURING MULTIPLICATION, rather than flipping a bit after the multiplication has been done. The program can be run as follows:

- ./mul_data_2d_faulty -a [infile A] -b [infile B] -c [outfile] -x [row index] -y [col index] -z [bit position] -k [which product term in the dot product should be corrupted]

The program takes two matrices from two separate files and multiplies them, but when it reaches the row and column index of the multiplication, the kth term of matrix B (we always corrupted the data from matrix B) in the dot product had one of its bits flipped. As the corruption takes place in the dot product step of matrix multiplication, the corruption takes place in the innermost for loop, as that is the loop that handles the dot products. I used a separate function to corrupt the data, but that is not necessary. This program is identical to the previous matrix multiplication program, except it calls a new function called mul_matrixFaulty to do the multiplication, so the old multiplication program still works correctly. This program also has an optional -p flag that will print all three matirces to the screen.:

```
void mul_matrixFaulty(dtype **A, dtype **B, dtype **C, int rows, int bcols, int acols, int rowIndex, int colIndex, int bit, int term)
{
    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < bcols; j++)
        {
            C[i][j] = 0;
            for (int k = 0; k < acols; k++)
            {
                //C[i][j] += (A[i][k] * B[k][j]);
                if(i == rowIndex && j ==colIndex && k == term){
                    C[i][j] += mul_ints(A[i][k], B[k][j], bit);
                }else{
                    C[i][j] += (A[i][k] * B[k][j]);
                }
            }
        }
    }
}

int mul_ints(int Aik, int Bkj, int bit){
    int num = Aik * (Bkj ^ (1 << bit));
    return num;
}
```

Figure 11: Code for the corrupted multiplication

From here, we can still use the detect and correct programs on the resulting matrix and they will still function as intended, unless an mbe occurred as they are not detected.

VIII. Project 8: H0

The goal of this program was to simulate Hamming(7,4) encoding through matrix multiplication. Hamming(7,4) encoding is a type of error-correcting code that uses matrix multiplication to encode a 4-bit message into a 7-bit codeword.To obtain the 7-bit codeword C, the 4-bit message M is multiplied by the generator matrix G.

$$\mathbf{x} = \mathbf{G}^{\mathbf{T}}\mathbf{p} = \begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 2 \\ 3 \\ 1 \\ 2 \\ 0 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix}$$

Figure 12: Hamming(7,4) encoding using matrix multiplication

The multiplication is performed modulo 2, which means that if the sum of the products in any position is odd, a 1 is placed in that position; otherwise, a 0 is placed. This process helps in creating redundancy in the codeword for error detection and correction. The resulting 7-bit codeword C is then transmitted or stored. During decoding, if errors occur, the parity bits in the codeword can be used to identify and correct the errors. The Hamming(7,4) code is capable of detecting and correcting single-bit errors and detecting two-bit errors within the codeword.

To complete this assignment, I made a new driver program called **ham-encode-4.c** that takes a nibble of hex data as input in the form of 0xa, where a is a valid hex character. Any other input is considered invalid, and would not work. This was done in the following function:

```
void check_input(int argc, char **argv)
{
    if (argc != 2)
    {
        printf("Usage: %s <number in hex>\n", argv[0]);
        exit(1);
    }

    // Check that the input number starts with "0x".
    if (argv[1][0] != '0' || argv[1][1] != 'x')
    {
        printf("Number must start with 0x\n");
        exit(1);
    }

    // Check that the input number is a valid hexadecimal character.
    if (!isxdigit(argv[1][2]))
    {
        printf("Not a valid hex character\n");
        exit(1);
    }

    // Check that the input number is a nibble of hex data.
    if (argv[1][3] != '\0')
    {
        printf("Number must be 1 hex character\n");
        exit(1);
    }
}
```

Figure 13: Function to check if the input is valid

Here are example outputs for correct and incorrect inputs:

```
michaeldandrea@Michaels-MBP H0 % ./ham-encode-4 0xD
CODEWORD = 0x55 = 0b1010101
michaeldandrea@Michaels-MBP H0 % ./ham-encode-4 0xF
CODEWORD = 0x7F = 0b1111111
michaeldandrea@Michaels-MBP H0 % ./ham-encode-4 0xa
CODEWORD = 0x5A = 0b1011010
michaeldandrea@Michaels-MBP H0 % ./ham-encode-4 0x4
CODEWORD = 0x4C = 0b1001100
michaeldandrea@Michaels-MBP H0 % ./ham-encode-4 g
Number must start with 0x
michaeldandrea@Michaels-MBP H0 % ./ham-encode-4 0xg
Not a valid hex character
michaeldandrea@Michaels-MBP H0 % ./ham-encode-4
Usage: ./ham-encode-4 <number in hex>
```

Figure 14: Outputs of the program