How to Compile and Run

To compile the program, simply type “make” into your terminal. To run the program based on what you would like it to do, simply type in “./myAO <print_flag> <operation> <other inputs>”. The “<print flag>” is the input for whether you would like the output printed or not (1 for yes, 0 for no) and the “<operation> <other inputs>” should be one of the following.

1. Matrix addition: + nA mA nB mB
2. Matrix subtraction: - nA mA nB mB
3. Matrix multiplication: * nA mA nB mB
4. Matrix transposition: T nA mA
5. Solving a system of linear equation (solving for x in Ax = B): s nA mA nB mB

The first portion represents the operation you would like to do and nA, mA, nB, and mB represent the number of columns and rows in A and B, respectively. To clean up output files after running, type “make clean” into your command line.

How the Program Works

Matrix addition function: Takes in two matrices of the same size, creates new matrix of that size, assigns each entry of the new matrix to be the sum of the corresponding entries of the two given matrices, and returns the new matrix.

Matrix subtraction function: Takes in two matrices of the same size, creates new matrix of that size, assigns each entry of the new matrix to be the difference of the corresponding entries of the two given matrices, and returns the new matrix.

Matrix multiplication function: Takes in two matrices where the number of columns in the first equals the number of rows in the second, uses the standard triple-loop algorithm to compute dot products of rows and columns, fills a new matrix with these results, and returns it.

Matrix transposition function: Takes in a matrix, creates a new matrix with dimensions flipped, iterates through each element, assigns each [i][j] to position [j][i], and returns the transposed matrix.

Solving Ax = B function: Takes in matrix A and vector B, performs Gaussian elimination to reduce A to row echelon form while applying the same operations to B, then uses back substitution to find and return the solution vector x.

Appendix

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

double **allocateMatrix(int cols, int rows)
{
    double **matrix = malloc(rows * sizeof(double *));
    if (!matrix)
        return NULL;
    for (int i = 0; i < rows; i++)
    {
        matrix[i] = malloc(cols * sizeof(double));
        if (!matrix[i])
            return NULL;
    }
    return matrix;
}

void freeMatrix(double **matrix, int rows)
{
    for (int i = 0; i < rows; i++)
        free(matrix[i]);
    free(matrix);
}

double **addMatrices(double **A, double **B, int nA, int mA, int nB, int mB)
{
    if (nA != nB || mA != mB)
        return NULL;
    double **C = allocateMatrix(nA, mA);
    if (!C)
        return NULL;
    for (int i = 0; i < mA; i++)
        for (int j = 0; j < nA; j++)
            C[i][j] = A[i][j] + B[i][j];
    return C;
}

double **subtractMatrices(double **A, double **B, int nA, int mA, int nB, int mB)
{
    if (nA != nB || mA != mB)
        return NULL;
    double **C = allocateMatrix(nA, mA);
    if (!C)
```

```
        return NULL;
    for (int i = 0; i < mA; i++)
        for (int j = 0; j < nA; j++)
            C[i][j] = A[i][j] - B[i][j];
    return C;
}

double **multiplicationMatrices(double **A, double **B, int nA, int mA, int nB, int
mB)
{
    if (nA != mB)
        return NULL;
    double **C = allocateMatrix(nB, mA);
    if (!C)
        return NULL;
    for (int i = 0; i < mA; i++)
        for (int j = 0; j < nB; j++)
        {
            C[i][j] = 0;
            for (int k = 0; k < nA; k++)
                C[i][j] += A[i][k] * B[k][j];
        }
    return C;
}

double **transposeMatrices(double **A, int nA, int mA)
{
    double **C = allocateMatrix(mA, nA);
    if (!C)
        return NULL;
    for (int i = 0; i < nA; i++)
        for (int j = 0; j < mA; j++)
            C[j][i] = A[i][j];
    return C;
}

void swapRows(double **matrix, int row1, int row2, int cols)
{
    for (int j = 0; j < cols; j++)
    {
        double tmp = matrix[row1][j];
        matrix[row1][j] = matrix[row2][j];
        matrix[row2][j] = tmp;
    }
}

double **solveAxB(double **A, double **B, int nA, int mA, int nB, int mB)
{
```

```
    if (mA != mB || nB != 1 || nA != mA)
        return NULL;
    double **aug = allocateMatrix(nA + 1, mA);
    if (!aug)
        return NULL;
    for (int i = 0; i < mA; i++)
    {
        for (int j = 0; j < nA; j++)
            aug[i][j] = A[i][j];
        aug[i][nA] = B[i][0];
    }
    for (int k = 0; k < nA; k++)
    {
        int maxRow = k;
        for (int i = k + 1; i < nA; i++)
            if (fabs(aug[i][k]) > fabs(aug[maxRow][k]))
                maxRow = i;
        if (fabs(aug[maxRow][k]) < 1e-9)
        {
            freeMatrix(aug, mA);
            return NULL;
        }
        if (maxRow != k)
            swapRows(aug, k, maxRow, nA + 1);
        for (int i = k + 1; i < nA; i++)
        {
            double f = aug[i][k] / aug[k][k];
            for (int j = k; j <= nA; j++)
                aug[i][j] -= f * aug[k][j];
            aug[i][k] = 0;
        }
    }
    double **x = allocateMatrix(1, mA);
    if (!x)
    {
        freeMatrix(aug, mA);
        return NULL;
    }
    for (int i = mA - 1; i >= 0; i--)
    {
        x[i][0] = aug[i][nA];
        for (int j = i + 1; j < nA; j++)
            x[i][0] -= aug[i][j] * x[j][0];
        x[i][0] /= aug[i][i];
    }
    freeMatrix(aug, mA);
    return x;
}
```

```h
#ifndef MYAO_H
#define MYAO_H

// IMPORTANT: allocateMatrix() and freeMatrix() are not limited to our test case, and
they must be here rather than utility.h
// results still need memory allocation and de-allocation, making these functions
primary components of your library.
double **allocateMatrix(int rows, int cols);
void freeMatrix(double **matrix, int rows);

double **addMatrices(double **A, double **B, int nA, int mA, int nB, int mB);
double **subtractMatrices(double **A, double **B, int nA, int mA, int nB, int mB);

// CODE: include ONLY the declaration of primary functions
double **multiplicationMatrices(double **A, double **B, int nA, int mA, int nB, int
mB);
double **transposeMatrices(double **A, int nA, int mA);
double **solveAxB(double **A, double **B, int nA, int mA, int nB, int mB);

#endif
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <stdbool.h>
#include <string.h>
#include "utility.h"
#include "myAO.h"

bool isPositiveInteger(const char *s)
{
    if (!s || !*s)
        return false;
    if (s[0] == '0' && s[1])
        return false;
    for (int i = 0; s[i]; i++)
        if (!isdigit(s[i]))
            return false;
    return true;
}

bool validInput(int argc, char *argv[])
{
    if (argc != 5 && argc != 7)
        return false;
```

```
    if (strcmp(argv[1], "0") && strcmp(argv[1], "1"))
        return false;
    if (strlen(argv[2]) != 1)
        return false;
    char op = argv[2][0];
    if (op != '+' && op != '-' && op != 'T' && op != 's' && op != '*')
        return false;
    for (int i = 3; i < argc; i++)
        if (!isPositiveInteger(argv[i]))
            return false;
    return true;
}

int main(int argc, char *argv[])
{
    if (!validInput(argc, argv))
        return 1;
    int nA = atoi(argv[3]), mA = atoi(argv[4]);
    double **A = allocateMatrix(nA, mA), **result = NULL;
    if (!A)
        return 1;
    char op = argv[2][0];
    if (argc == 5)
    {
        result = transposeMatrices(A, nA, mA);
        freeMatrix(A, mA);
    }
    else
    {
        int nB = atoi(argv[5]), mB = atoi(argv[6]);
        double **B = allocateMatrix(nB, mB);
        if (!B)
            return 1;
        if (op == '+')
            result = addMatrices(A, B, nA, mA, nB, mB);
        else if (op == '-')
            result = subtractMatrices(A, B, nA, mA, nB, mB);
        else if (op == '*')
            result = multiplicationMatrices(A, B, nA, mA, nB, mB);
        else if (op == 's')
            result = solveAxB(A, B, nA, mA, nB, mB);
        freeMatrix(A, mA);
        freeMatrix(B, mB);
    }
    if (!result)
        return 1;
    if (!strcmp(argv[1], "1"))
    {
```

```
        int rows = (argc == 5 ? mA : mA);
        int cols = (op == '*' ? atoi(argv[6]) : (op == 's' ? 1 : nA));
        printMatrix(result, rows, cols);
    }
    freeMatrix(result, mA);
    return 0;
}
```

```
/* The purpose of this source code is implement functions that are NOT directly
related to arithmetic operation library.
The functions implemented here and declared in utility.h are used only for our test
cases.
For example, fillRandom() is used to randomly initialize inputs A and B. However,
usually the A and B are given by program or user */
// CODE: Include necessary libraries
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

void fillRandom(double **matrix, int rows, int cols) {
    // NOTE: if you are testing you might want to generate random numbers that are
reproducible
    // For example with srand(42) and Using 42 (or any fixed number) ensures
repeatable results
    srand(time(NULL));
    for (int i = 0; i < rows; i++)
        for (int j = 0; j < cols; j++)
            matrix[i][j] = (rand() % 21) - 10; // Generates values from -10 to +10
}

void printMatrix(double **matrix, int rows, int cols) {
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++)
            printf("%6.2f ", matrix[i][j]);
        printf("\n");
    }
    printf("\n");
}


// CODE: implement more utility functions here if you need more
```

```
#ifndef UTILITY_H
#define UTILITY_H
```

```
// CODE: declare any function implemented in utility (if you need more)
void fillRandom(double **matrix, int rows, int cols);
void printMatrix(double **matrix, int rows, int cols);


#endif
```