University of California, Santa Cruz
Board of Studies in Computer Engineering

CMPE-13/L: Computer Systems and "C" Programming

Lab 3 - Matrix Math

Commit ID Form: https://goo.gl/forms/H0BHXjoTOJh2rbAY2

12 Points

Introduction

In this lab you will write a library for performing operations on 3x3 matrices. You will also develop a small program that tests some of these functions to verify their correctness. This lab will show you how libraries are built, tested, and then used by other people in their projects. Since this lab does not require any user input, it can be run entirely within the simulator, which will be faster than running it on the actual hardware.

Reading

- **K&R** – Sections 4.1, 4.2, 4.5, 5.7

Concepts

- Libraries
- Passing by reference
- Arrays
- Multidimensional Arrays
- Unit Testing

Provided files

- MatrixMath.h – Describes the functions you will implement in MatrixMath.c and contains the function prototypes that your functions will implement. Add this file to your project directly. You will not be modifying this file at all!
- BOARD.c/h – Contains initialization code for the UNO32 along with standard #defines and system libraries used.
- mml.c – This file contains `main()` and all the support code you'll need. You will add tests for every function in the MatrixMath library in this file.

Assignment requirements

- Create a new file called MatrixMath.c that implements all of the functions whose prototypes are in the header file MatrixMath.h (excluding `MatrixAdjugate()`, which is extra credit) according to the specifications given therein.
    - All of your functions in MatrixMath.c must use for-loops except the following two:
        - `MatrixDeterminant()`
        - `MatrixInverse()`
    - The only function that should use `printf()`, or really perform any non-mathematical operation, is `MatrixPrint()`, everything else should only be doing mathematical calculations.
    - The testing of your functions happens within `main()` and that's when most things are printed to the user.
- Expand `main()` in mml.c to include more unit tests that test every function in MatrixMath.h **at least twice**. This means using substantially different tests! Remember to include proper testing for floating point equality.
    - Also, demonstrate that your `MatrixPrint()` function works correctly by calling it on a non-empty matrix at some point in your code. You should also hard-code a printout of that same matrix as a reference so that any tester could easily see if `MatrixPrint()` failed.
- main() should output all of the test results from these unit tests (no printf() calls should exist inside your MatrixMath.c file). The output for unit tests should be grouped per function tested so that a list of all tested functions is presented along with a PASSED or FAILED depending on if they passed all tests or failed some, and the number of tests run for that function. The Example Output section demonstrates suitable output.
- **Extra credit:** Implement the `MatrixAdjugate()` function defined in MatrixMath.h without hard-coding any values and using for-loops. At least two tests for this function should also be added to your main() to get the point. You must also use this function within `MatrixInverse()`.
- Create a readme file named README.txt containing the following items. Spelling and grammar count as part of your grade so you'll want to proof-read this before submitting. This will follow the same rough outline as a lab report for a regular science class. It should be on the order of three paragraphs with several sentences in each paragraph.

    - First you should list your name & the names of colleagues who you have collaborated with[1].
    - In the next section you should provide a summary of the lab in your own words. Highlight what you thought were the important aspects of the lab. If these differ from how the lab manual presents things, make a note of that.
    - The following section should describe your approach to the lab. What was your general approach to the lab? Did you read the manual first or what were your first steps? What went wrong as you worked through it? What worked well? How would you approach this lab differently if you were to do it again? Did you work with anyone else in the class? How did you work with them and what did you find helpful/unhelpful? (see footnote on previous page)
    - The final section should describe the results of you implementing the lab. How did it end up finally? How many hours did you end up spending on it? What'd you like about it? What did you dislike? Was this a worthwhile lab? Do you have any suggestions for altering it to make it better? What were the hardest parts of it? Did the points distribution for the grading seem appropriate? Did the lab manual cover the material in enough detail to start you off? Did examples or discussions during class help you understand this lab or would more teaching on the concepts in this lab help?
- Format your code to match the style guidelines that have been provided.
- **Submit MatrixMath.c, mml.c, and README.txt.**

Grading

This assignment consists of 12 points:

- 1 point for each implemented function with tests (half is for the function implementation and the other half is for having effective tests for it).
    - Note that testing for your `MatrixPrint()` function only requires that you print out a 3x3 matrix with nice formatting like the example given at the end of the lab manual.
- 1 point - Followed the guidelines for program output
- 1 point - <= 5 coding/style errors in submitted code
- 1 point - Provided a README.txt file
- **Extra credit (1 point)** - Implement `MatrixAdjugate()`.

[1] NOTE: collaboration != copying. If you worked with someone else, be DETAILED in your description of what you did together. If you get flagged by the code checker, this is what will save you.

You will lose points for the following:

- no points for any code that doesn't compile[2]
- -2 points: any compiler warnings
    - At this point warnings of any kind are unacceptable. If you're unsure how to fix some errors or warnings, see the Compiler Errors document on the course website.
- -2 points: if gotos were used

Passing by reference

In C, functions are normally passed their arguments "by value". This means that while the data the function receives is the same as what you passed it, the actual variable itself isn't. This means that in the following code, the variable "d" never changes value:

```
char SomeCalculation (char type, int b)
{
	while (b--) {
		type += 2;
	}
	return type;
}

int main()
{
	int d = 5;
	SomeCalculation('*', d);
}
```

What happens when `SomeCalculation()` is actually called is that a copy of "d" is passed to it as the variable "b". Since "d" was copied, the decrementing of "b" within `SomeCalculation()` does not affect the original variable "d".

Sometimes you don't want to pass a copy to a function, so you can instead pass an argument 'by reference'. This means that you pass the actual variable. If we did this when calling `SomeCalculation()` in the example code above then "b" and "d" would actually be the same variable and any changes to "b", like the decrementing done within `SomeCalculation()`, would affect "d" as well.

[2] There is NEVER any reason to submit code that does not compile. Make sure you take the 30 seconds to check this before submitting your code. We will not attempt to fix code that does not compile; you will simply get a zero.

While for most data types you have the option to pass by reference or by value it is important to understand that arrays are special: *you cannot pass them by value*. This means if you alter the array within a function that change will persist in that variable even after the function returns.

Passing by reference is also the only way to retrieve the output of a function call if it's an array without doing rather complicated memory management. The way to do this is to pass an additional array as an argument to the function and then for that function to perform calculations and place the results in that array. In fact you have already seen all of these uses of pass-by-reference when using the venerable `printf()` and `scanf()` functions!

Arrays

If you think of a variable as a tool to move around a single piece of data, then arrays are easy to understand. Arrays are used to move around multiple pieces of data of the same type around as a single variable. For example, the following statement:

```
int nameOfArray[3];
```

declares a one dimensional array with three pieces of data of type `int` (this is just like the declaration statement of a variable).

When thinking of a variable it is important to initialize it to a value before using it, and this concept is even more important with arrays because arrays are not initialized on declaration like some variables (also *never count on a variable to be initialized to 0 automatically* because some compilers don't play nicely). To initialize all the elements in an array to 0 empty braces can be used like the following:

```
nameOfArray[3] = {};
```

To change the values of this array a series of statements like the following could be used (note that the first element is at index 0):

```
nameOfArray[0] = 1;
nameOfArray[1] = 2;
nameOfArray[2] = 3;
```

After the change your array can be visualized as [1, 2, 3] (you can also use a series of statements like these or a for-loop to initialize the values of your array).

Notice that the first element of your array is accessed with the index value of 0 instead of 1[3]. This means that the index value of the last element in your array will be one less than the length of the array in your declaration statement. To avoid some of this confusion the declaration and initialization of an array can be combined into a single statement:

```
int nameOfArray[3] = {1, 2, 3};
```

or even

```
int nameOfArray[] = {1, 2, 3};
```

Both of these statements are equivalent, the second just uses the number of initialization values to declare the size of the array, and is actually preferred. By utilizing the second method, there is no chance of making a mistake because the math is left to the compiler. Arrays are of a fixed size; this means that the initializations for your array must match up with the size of the array. For instance, the statement:

```
int nameOfArray[4] = {1, 2, 3};
```

Will declare and initialize an array with the following representation: [1, 2, 3, 0] which is probably not what you want. An array that has fewer elements in its initialization than you declare for the size of the array will have the extra elements initialized to 0.

Multidimensional Arrays

The important concept to remember when working with multidimensional arrays is that they are row major, this means that all of the elements in the first row are accessed before moving on to the next row. This concept can be applied to both initializing the values of the array, and accessing the elements of the array. For instance, the statement:

```
int exampleArray[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
```

Initializes a two dimensional array of type int with three rows and three columns.

This array can be visualized as:

[3] If you remember your CE12 lectures about memory, you can easily understand why using offset addressing the first element of the array is the [0] address.

	col 0	col 1	col 2
row 0	1	2	3
row 1	4	5	6
row 2	7	8	9

Notice the expression ‘twoDimArray[2][1]’ will refer to the element with the value 8 (remember the index value starts at 0).

A multidimensional array can also be initialized to contain all 0’s similar to one dimensional arrays with a statement like:

```
int exampleArray2[3][3] = {{}, {}, {}};
```

The most common way to access the values in a multidimensional array is using nested loops. Here is a way to declare and initialize the above two dimensional array with nested for-loops:

```
int arrayElementValue = 1;           // Value stored in array element
int i, j;                            // Array index counters
int twoDimArray[3][3];               // Array declaration
for (i = 0; i < 3; i++) {            // First loops counts through rows
       for (j = 0; j < 3; j++) {     // Next loop counts through cols
             twoDimArray[i][j] = arrayElementValue;        // Set val
             arrayElementValue++;                   // Increment val
       }
}
```

Working with matrices (linear algebra)

The numbers you are used to using are also known as scalars, which can also be thought as a 1x1 matrix. They are also the values that make up matrices.

Matrices are a two-dimensional array of numbers. They’re used in mathematics for many different things, but one of the more common uses in programming is for rendering 3D scenes. All of the video games you have played rely on matrices to describe how the game world or the objects within it will be altered. We won’t be going into any more detail than this but plenty of reading material is available online.

For this assignment we’re only going to be talking about 3x3 matrices and scalars. As a matrix is two-dimensional it consists of both rows and columns. A 3x3 matrix therefore has 3 rows and 3 columns:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

As you can see this matrix contains 9 elements, a_{ij}, with the subscript denoting the element's location: i denotes its row and j denotes its column. If you think this sounds like a similar way of referencing items as in an array in C, then you're spot on. Remember that array indexing starts at 0 in C! You will be using the two-dimensional arrays mentioned earlier for storing the matrices.

So now we know what 3x3 (pronounced 3-by-3) matrices are and how to define them in C, but what are we going to do with them? Well, the answer is that we will want to perform basic matrix arithmetic on them.

Matrix equality: The most important aspect of matrices is equality. How do we know when two matrices are equal? We know what it means for two numbers to be equal, but how is this extended to an entire matrix? In linear algebra equality is based on two conditions: that the matrices have the same dimensions and that the elements at equivalent locations within both matrices are equal. This means that for matrix A and B to be identical element a_{11} from matrix A and element b_{11} from matrix B need to be equal and so on for every element of A and B with no unpaired elements.

Since this library only works with 3x3 matrices the first equality condition will always be true, so only the second one has to be checked. Since this library is working exclusively with floating-point numbers we need to discuss something called round-off error[4]. What this means is that some numbers cannot be perfectly represented in the binary format that the computer stores numbers in. 0.1 is one such number that can't be perfectly represented in a binary format with a finite number of digits. In case you were wondering, only numbers that can be represented as fractions with a denominator that is a multiple of two can be represented exactly in the binary system.

And here's the cincher about all this round-off error: it gets worse as you continue to do mathematical operations. You start with a little round-off error in a number and then do a lot of math with that number, say use it in a matrix multiplication operation, and then there's round-off error in the result of that operation! So round-off error keeps compounding. This can result in very inaccurate calculations. More commonly, however, it just means that you can't compare floating-point types (floats and doubles in C) directly using the equality operator.

[4] Again, remember your CE12 IEEE Floating Point number discussion, and how a fixed number of bits are used in the mantissa and significand. Think about the conversion process to binary.

When dealing with floating point numbers you'll want to take the round-off error into account. This will be done by just checking whether two numbers are within some delta of each other, where you choose delta to be quite small, say 0.0001 (if you look in the MatrixMath.h file you will see that a constant with this value has already been defined for you). If those numbers are that close to each other you can say they are equal. While this is not the proper way to test equality when doing high-precision scientific calculations it's quite commonly used in less-demanding situations like this.

Now to determine how close two numbers are relies on a simple subtraction. The only issue then is that you either have a positive or negative result. You can check that this number is either less than delta AND greater than –delta.

Matrix-matrix addition: Matrix addition works similarly to regular integer addition, you just have to do more of it. To add two matrices together you just add all of the corresponding entries together:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} a_{11} + b_{11} & a_{12} + b_{12} & a_{13}{+}b_{13} \\ a_{21} + b_{21} & a_{22} + b_{22} & a_{23} + b_{23} \\ a_{31} + b_{31} & a_{32} + b_{32} & a_{33} + b_{33} \end{bmatrix}$$

Matrix-scalar addition: Another common operation is matrix-scalar addition. This is the addition of a single number, a scalar, to a matrix. The way this operation works is to add that number to every entry in the matrix:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} + g = \begin{bmatrix} a_{11} + g & a_{12} + g & a_{13} + g \\ a_{21} + g & a_{22} + g & a_{23} + g \\ a_{31} + g & a_{32} + g & a_{33} + g \end{bmatrix}$$

Matrix-scalar multiplication: Matrix-scalar multiplication follows similarly from matrix-scalar addition:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} * g = \begin{bmatrix} a_{11} * g & a_{12} * g & a_{13} * g \\ a_{21} * g & a_{22} * g & a_{23} * g \\ a_{31} * g & a_{32} * g & a_{33} * g \end{bmatrix}$$

Matrix-matrix multiplication: Where things first start to get more complicated is during matrix-matrix multiplication. Each element in the final matrix is going to be the sum of all of the elements in this element's row from the first matrix multiplied by all of the elements of this element's column from the second matrix. The best way to clarify this is with an example:

$$c_{13} = [a_{11} \quad a_{12} \quad a_{13}] * \begin{bmatrix} b_{13} \\ b_{23} \\ b_{33} \end{bmatrix} = a_{11} * b_{13} + a_{12} * b_{23} + a_{13} * b_{33}$$

So to calculate the element in the first row, third column of the matrix that is the product of matrix A and matrix B is to multiply all the elements in the first row of A by all of the elements in the third column of B. This process continues for every element of the final matrix (all 9 of them in the case of a 3x3 matrix).

All the matrix operations just described were binary operations: they operated on two different items, either a matrix and another matrix or a matrix and a scalar. There are four more important matrix operations that are known as unary, as they only operate on a single operand, a single matrix:

Trace: The trace of a matrix (denoted tr(A)) is the sum of all of the diagonal elements of a matrix. Now when someone says diagonal in reference to a matrix they mean all of the elements that start from the upper-left corner of the matrix and go down and to the right by one in every subsequent row. So the elements that are used for calculating the trace are highlighted below:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

Determinant: The determinant of a matrix is an important property that is used for various things in linear algebra. For 3x3 matrices it is defined as the product of the three left-to-right diagonals starting from the first row of elements of the matrix minus the three right-to-left diagonals that start from the first row of elements (the diagonals wrap around). The picture below better illustrates this (the 3x3 elements on the left of the dashed line are the original matrix and the two leftmost columns are repeated on the other side to simplify reading):

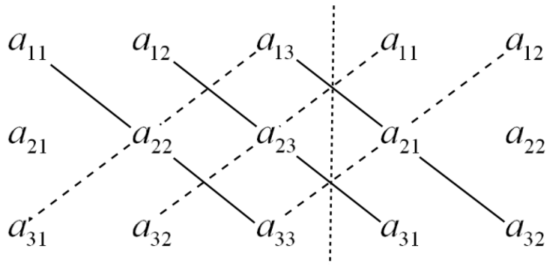


Figure - Sarrus' rule for the determinant of a 3x3 matrix

So the products of each solid-line diagonal are added together. Then the products of each dashed-line diagonal are subtracted from this total to give the final value of the determinant.

Transpose: The transpose of a matrix is just the mirroring of all of its elements across its diagonal, the line connecting the upper-left corner and the bottom-right corner. The transpose therefore doesn't affect the elements along this diagonal line, only the ones not on it. So elements a_{21} and a_{12} will be switched in the transpose of matrix A, but a_{11} will stay the same. The notation to describe the transpose of a matrix is with a superscript T:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}^T$$

Inverse: The inverse of a matrix is an abstraction of the inverse of a number. While a number times its inverse equals one, a matrix times its inverse equals the identity matrix (a matrix with 1s along the diagonal and 0s everywhere else, shown below).

$$\mathrm{I}_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

And much like a number's inverse, the inverse of a matrix A is denoted by A^{-1}. While this is the definition, the calculation of the inverse for 3x3 matrices is trivial if we build on the calculations of the determinant and transpose discussed earlier.

The inverse of a 3x3 matrix is the transpose of its cofactor matrix (also known as its adjugate matrix) divided by its determinant:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}^{-1} = \frac{1}{detA} \begin{bmatrix} b & c & d \\ e & f & g \\ h & k & l \end{bmatrix}^T$$

where the right-hand matrix is defined as follows:

$$b = a_{22} * a_{33} - a_{23} * a_{32} \quad c = a_{23} * a_{31} - a_{21} * a_{33} \quad d = a_{21} * a_{32} - a_{22} * a_{31}$$
$$e = a_{13} * a_{32} - a_{12} * a_{33} \quad f = a_{11} * a_{33} - a_{13} * a_{31} \quad g = a_{12} * a_{31} - a_{11} * a_{32}$$
$$h = a_{12} * a_{23} - a_{13} * a_{22} \quad k = a_{13} * a_{21} - a_{11} * a_{23} \quad l = a_{11} * a_{22} - a_{12} * a_{21}$$

Adjugate (extra credit): The adjugate of a 3x3 matrix is the transpose of the cofactor matrix. In the 3x3 case, it is what multiplies the inverse of the determinant when finding a matrix inverse, so refer to the equations for calculating the matrix inverse.

To get the extra credit you must:

- Correctly implement the `MatrixAdjugate()` function.

- Implement the `MatrixAdjugate()` function WITHOUT hard coding anything. This means all values should be determined purely based on the indices of the current matrix location.
- Use the `MatrixAdjugate()` function within your `MatrixInverse()` function.
- Implement 2 correct tests for the `MatrixAdjugate()` function.

Writing a library

Software libraries refer to existing modular code that can be easily incorporated into your own. This will usually just require using a #include directive and linking to their library (don't worry about this last part). Oftentimes this code will perform very specific functions that make sense to distribute separately from the applications that use them, such as math libraries like the one you'll write for this lab.

Libraries are commonly broken into three parts: the library description in one or many header files, the library itself, and a testing framework of some kind. For this assignment you're given the library description as a header file and have to implement the library code as well as the testing framework.

The header file will be used by others who want to use the library. They'll include this header in their program so that the compiler knows what functions are included in the library. This is also commonly used as a form of documentation and describes all of the various functions, their arguments and some details of how they work. This is how the header file is used for this library.

One of the advantages of putting the documentation in the header file is that someone else can then implement their own version of your library if they so choose. A common use case would be if your project used some library but that library was slow and no longer maintained. What you would do is write your own that followed the same function prototypes as and you wouldn't need to change any code in your project to use this new in-house implementation![5] Pretty awesome, right?

So libraries are very useful for separating out functions that are useful for other people or other projects. And let you be forewarned now that you may be required to use someone else's matrix math library in one of the later labs and someone will use yours. This is the downside to writing a library: someone else may use your code and complain to you about it. In the real world that's not too big of a deal, but in these labs you'll be docked points later for a library that doesn't work.

[5] This is a basic result of what is called "information hiding" and is central to good coding practices.

Unit testing

Unit testing refers to the practice of writing code that, on a function-by-function basis, tests other code (in this case it refers to the code you'll add to `main()` in mml.c). It is used when developing large amounts of code to do two things: confirm that the code operates as expected and make sure that any future changes to this code also work as expected. This last point isn't crucial for these labs but becomes important when working on large code bases with many developers. Generally functions will require several different tests that vary the input over a range of valid and invalid values before the function being tested is considered working. For this lab we require you to implement at least 2 non-trivial test cases for each function in the library.

The way you will write unit tests will look like the following.

1. Manually generate input parameters
2. Call the function with these inputs
3. Verify the output against what is expected
4. Log the results

When writing code and unit tests keep in mind that the proper order of implementation is to write the tests first and then write the function. This serves a couple of purposes: it confirms that you know how the function is supposed to work and can therefore code it and it results in more correct tests. Since you aren't thinking about writing code at all you aren't worried about mentally checking the code as you write the test, otherwise it is easy to write tests that your code passes versus writing correct tests and then seeing if your code can pass them.

Also it is best to write tests one at a time along with the functions you're testing. The exact wrong way to do this lab is to implement every function and then have to go through and test them all. It's a lot of code to keep track of and will make your life much harder. Just start with a test for a function and then implement that function and then move on.

One thing to be careful about is whether your tests are 100% correct. If either your test's inputs or the expected result are incorrect then the test that you have just written is worthless. It's actually very harmful because now you are checking that this function produces incorrect output! This is one of the benefits of using at least two tests, as that will generally catch these cases.

This means that one of the first things you should check if a test fails: is whether the input and expected output are correct. If it is, then it is definitely the function that is at fault.

Lastly you'll want to make sure that the results logged by your unit test harness are easy to read. So each test should show whether it succeeded and there should be a tally at the end of how many tests pass. See the example output for reference.

Doing this lab

Doing this lab should follow from the iterative design methodology that you have already used in the previous labs. For this lab it should actually be easier to follow the iterative design method as you implement single functions for this library. So to do this lab, merely repeat the following step for each function starting with the MatrixEquals() function:

Step 1 to n:

- Calculate multiple input/output values for use with testing the current function.
- Implement the current function.
- Write code for testing all input/output pairings that you precalculated.
- Add output describing if the function passed all tests or failed some.

Step n + 1:

- Implement MatrixAdjugate() if you want to do the extra credit (remember that you can use the MatrixTranspose() function!)
- Write two tests for it to confirm it works.
- Replace hard-coded adjugate calculation in MatrixInverse() with MatrixAdjugate() function.

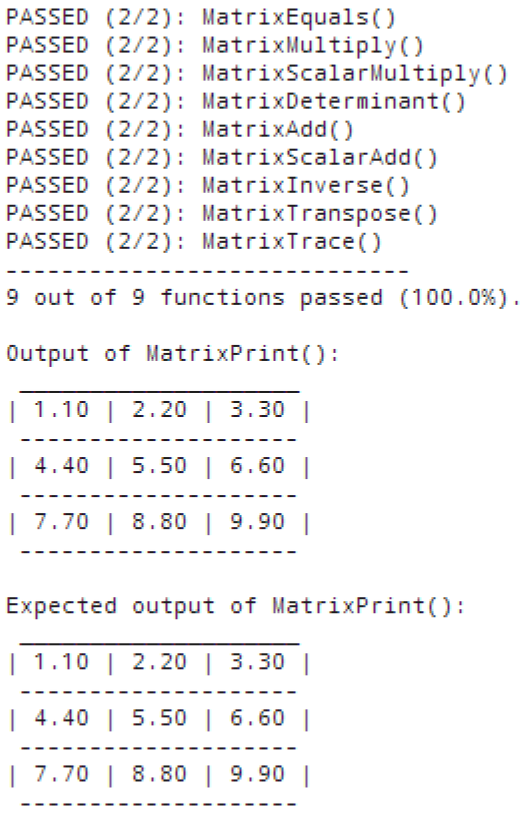
Step n + 2:

- Double-check your lab. Confirm that you implemented all of the functions necessary following any rules. Reread the Assignment Requirements section to check this.
- Create your README.txt file.
- Make sure all of your files are properly named, your code is properly formatted, well-commented, and you have no compilation warnings.
- **Make sure you haven't modified your MatrixMath.h file or we may have problems testing your code resulting in loss of points. Any helper functions within MatrixMath.c need have function prototypes inside your MatrixMath.c file. This means the ONLY functions you can use in mml.c are the ones we have defined for you in MatrixMath.h.**
  - **We do NOT use your MatrixMath.h file when grading this lab.**
- Commit and push your files (mml.c, MatrixMath.c, and README.txt).
- Submit your commit ID through the google form.

Example Output

The test output should look similar to what we have below and output the following:

- For every test: Whether it passed or not, how many of your tests passed out of the total, and the function name.
- At the end the total number of functions passed out of the total should be shown along with a percentage value.
- Since `MatrixPrint()` cannot be tested, a matrix should be hardcoded as output and then `MatrixPrint()` should be called, with the same expected values for both matrices, along with a clarifying label for each output.



```
PASSED (2/2): MatrixEquals()
PASSED (2/2): MatrixMultiply()
PASSED (2/2): MatrixScalarMultiply()
PASSED (2/2): MatrixDeterminant()
PASSED (2/2): MatrixAdd()
PASSED (2/2): MatrixScalarAdd()
PASSED (2/2): MatrixInverse()
PASSED (2/2): MatrixTranspose()
PASSED (2/2): MatrixTrace()
-----------------------------
9 out of 9 functions passed (100.0%).

Output of MatrixPrint():
 ____________________
| 1.10 | 2.20 | 3.30 |
 --------------------
| 4.40 | 5.50 | 6.60 |
 --------------------
| 7.70 | 8.80 | 9.90 |
 --------------------

Expected output of MatrixPrint():
 ____________________
| 1.10 | 2.20 | 3.30 |
 --------------------
| 4.40 | 5.50 | 6.60 |
 --------------------
| 7.70 | 8.80 | 9.90 |
 --------------------
```

Note that we will be running your test harness with our own version of MatrixMath.c (which we know works), and we will also be running your MatrixMath.c with our own test harness (much more comprehensive than the one you wrote).