UNIVERSITY OF CALIFORNIA, SANTA CRUZ
BOARD OF STUDIES IN COMPUTER ENGINEERING

CSE013E: COMPUTER SYSTEMS AND "C" PROGRAMMING

Lab 3 - Matrix Math
12 Points

Introduction

In this lab you will write a library for performing operations on NxM 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

Required Files:

- MatrixMath.c -- Implement the functions that are specified in MatrixMath.h
- mml_test.c -- A test harness that you write for the Matrix Math library.
- README.md

Provided files:

- **Edit these files:**
  - mml_test.c – This template file contains `main()` and a very trivial example test to get you started.
- **Do NOT edit these 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.

Assignment requirements:

- Create a new file called MatrixMath.c that implements all of the functions whose prototypes are in the header file MatrixMath.h.
  - All functions should use loops to iterate over elements of the array (with the exceptions of `MatrixSubmatrix()` and `MatrixDeterminant()`). Most functions should have one long loop or two nested loops.
  - The only function in MatrixMath.c that should use `printf()` is `MatrixPrint()`. No functions in MatrixMath.c should call `MatrixPrint()`;
- The testing of your functions happens within the `main()` of mml_test.c. All printing should happen within `main()`, or if you wish, within helper functions defined in mml_test.c.
  - Your tests should be thorough and diverse. Each of the above (except perhaps `MatrixPrint()`) should be tested in *at least* two ways. The two ways should be meaningfully different from each other:
    - Testing "trivial" cases like adding 0 or multiplying by 1 are useful basic tests because they are easy to implement. These tests can make sure your functions execute successfully. Follow up with more complicated tests to ensure that your functions use correct mathematics.
    - Try testing for mathematical properties of matrices. For example, A + B = B + A is always true, so make sure that that your code produces that result. But A*B =B*A is NOT always true, so make sure that you can produce and test that situation too.
  - `MatrixEquals()` will form the foundation for most of your other tests, so it's important to get it right! It should have several tests.
    - Be especially sure to test for Type II errors! Many students correctly check that `MatrixEquals()` returns true for two equal matrices, but fail to check that it returns false for non-equal matrices.
  - Also, demonstrate that your `MatrixPrint()` function works correctly by calling it on a non-empty matrix at some point in your code.
    - `MatrixPrint()` should only be invoked once or twice in the final version of your code.
- `main()` should output all of the test results from these unit tests (again, no `printf()` calls should exist inside your MatrixMath.c file, except in `MatrixPrint()`).
  - The output for unit tests should be concise and easy to read.

    - The idea here is that, if there is an error in your library, it would be (relatively) easy to locate the error by reading your output.
- Your README file should be readable and well-develped. In it, you should include:
    - Your name & the names of colleagues who you have collaborated with[1], and a description of what you discussed together.
    - 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.
    - Describe your approach to the lab. 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?
    - Give us some feedback. 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?

Doing this Lab on Linux:

This lab works very well on linux as it has no input. You can compile your project (and give it the name mml_test) using the command below.

```
$ gcc -Wall mml_test.c MatrixMath.c BOARD.c -lm -o mml_test
```

[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.

Grading

This assignment consists of 13 points, though it is only scored for 12 points, so you have a 1-point "cushion."

- 9 points: Autograded components:
    - 2.0 points - MatrixEquals()
    - 1.0 point - MatrixAdd()
    - 1.0 point - MatrixMultiply()
    - 0.5 point - MatrixScalarAdd()
    - 0.5 point - MatrixScalarMultiply()
    - 1.0 point - MatrixTrace()
    - 1.0 point - MatrixTranspose()
    - 1.0 point – MatrixSubmatrix()
    - 0.5 point - MatrixDeterminant()
    - 0.5 point – MatrixInverse()
- 4 points: Human –graded components:
    - 1 points – MatrixPrint()
    - 2 points --Test harness
        - 1 point for readability and usefulness of readout
        - 1 point for thorough and diverse tests
    - 1 point – README.md
        - 0.5 points for completeness
        - 0.5 points for clear thinking and communication
- Possible Deductions:
    - Code style and readability (we expect you to follow CSE13E_StyleGuidelines)
    - Any compiler warnings
    - Runtime errors when executing your functions
    - Hard-coding your functions, rather than using loops. (Hard coding in mml_test.c is acceptable)
    - Graders may deduct points for "bad form": things like bad program flow, disregard for the instructions, or anything that makes their jobs unnecessarily hard.
    - It must compile within MPLABX. For this lab in particular do not use for loop variable declarations, they are not part of the C89 spec.

Passing by reference

So far in this course, you've mostly worked with functions that pass their arguments "by value". This means that the body of the function uses a copy of the variable that you passed it, and cannot modify the original value. So, in the following code, the variable "d" never changes value:

```
char TryToChangeA(int A)
{
      A = A+1;
      return A;
}

int main()
{
      int a = 5;
      TryToChangeA(a);
      printf("%d",a);
}
```

What happens when `SomeCalculation()` is actually called? The C runtime environment makes a copy of `a`, which it stores in a memory location that `TryToChangeA()` can access using the name `A`. `TryToChangeA()` modifies that copy, but when it returns, that copy is destroyed. Since the `main()` code doesn't save that returned `A`, and it is gone forever. Only `a` remains.

Sometimes, you cannot use a `return` to get information out of a function. Perhaps you want to pass more than one variable out of the function, or perhaps you need to use the return value to report an error during function execution (this is a standard convention in C).

Instead, you can pass an argument 'by reference'. This means that you pass the *address* of a variable, allowing the function to find the original variable "where it lives," and modify it there. If we did this when calling `SomeCalculation()` in the example code, it would look like this:

```
void ChangeA(int * A)
{
      *A = *A+1;
}

int main()
{
      int a = 5;
      ChangeA(&a);
      printf("%d",a);
}
```

Note that `ChangeA()` returns void (since it is using another method to share information with other code). As an argument, it now takes `*A`, which in this context means that `A` is the *address* of an integer. In the code body, it refers to `*A`, which in this context means that our code is operating on the integer stored at the address `A`. In `main()`, we pass `&a`, using `&` to find the address of `a`.

In other words, A is now a *pointer.* ' `*` ' is called the "dereferencing" operator, and it is used to transform an address into a value. ' `&` ' is called the "referencing" operator, and it is used to turn a value into an address.[2]

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 or return them by value*. To create or alter an array using a function, you *must* pass by reference.

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()`[3] and `scanf()` functions!

---

[2] Note that * has subtly different meanings when we use it as a declaration or in an argument, versus when we use it in an expression. In a declaration or an argument, * tells the compiler that the variable in question should be treated as an address. In an expression, * tells the compiler to use the value at that address.

[3] If you are wondering how `printf()` uses pass-by-reference, recall that strings in C are really just pointers to arrays of chars.

Arrays

If you think of a variable as a tool to handle a piece of data, then arrays are easy to understand. Arrays are used to handle multiple pieces of data (as long as all the pieces are the same type). For example, the following statement:

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

declares a one-dimensional array (that is, a vector or a list) with three pieces of data of type `int`. This is very similar to the declaration statement of a single variable, but in this case, `nameOfArray` is actually a *pointer* to the first (or rather, zero-th) value in the array.

It is almost always important to initialize a variable to a value before using it, and arrays are not an exception. 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 for-loop to initialize the values of your array, or use a statement like

```
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.

Arrays have a fixed size, so 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 may not be what you want.

A couple final notes on one-dimensional arrays:

First, strings are arrays in C. They are one-dimensional arrays of type 'char'. So the following two pieces of code are completely equivalent (check your ASCII table):

```
char hwString[10] = "Hello!"
printf('%c', hwString[5]);

char hwString[10] = { 72, 101, 108, 108, 111, 33}
printf('%c', hwString[5]);
```

and in both cases print ' ! '

Secondly, there is another way to declare matrices. We will not use it in this lab, but it is included here so you can recognize it later:

```
int * fooArray = {1,2,3}
char * barString = "Hello!"
```

Multidimensional Arrays

If we can group together many items of the same type into an array we can also group many arrays of the same type together to create multidimensional arrays.

The important concept to remember when working with multidimensional arrays is that they are *row major*, meaning 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 twoD[3][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}};
```

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

This array can be visualized as:

	col 0	col 1	col 2	col 3
row 0	1	2	3	4
row 1	5	6	7	8
row 2	9	10	11	12

The expression '`twoD[2][1]`' will refers to the element with the value 10. The indexes can be variables, so the expression `twoD[ r, c]` will access the element in row `r` and column `c`.

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 twoD [3][4];
int elementValue = 1;
int r, c;
for (r = 0; r < 3; r++) {
      for (c = 0; c < 4; c++) {
            twoD[r][c] = arrayElementValue;
            arrayElementValue++;
      }
}
```

Linear Algebra (Matrix Math)

If you've never taken a linear algebra class, then you've probably only done algebra on *scalars*. A scalar is a single value, like 0, 1, 2.5, -7, or pi. You can also think of a scalar as a 1x1 matrix. A matrix is a two-dimensional array of scalars.

Matrices are used in mathematics for many different things. One example is 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 and displayed. You'll also find matrixes in physics simulations, economic models, robotic control algorithms, computer vision, neural networks, statistical analysis, and many more topics.

As a matrix is two-dimensional, it consists of both rows and columns. A 2x3 matrix therefore has 2 rows and 3 columns:

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

As you can see this matrix contains 6 elements, each of which can be referred to as a_{ij}, where *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 right! There are a couple superficial differences. For example, mathematics literature usually indexes from 1, while most programming languages index from 0.[4] **To be clear, in this lab we expect you to index from 0 in every case, even if the examples index from 1.**

[4]The study of linear algebra is much, *much* older than the study of computer science, originating in China in the 10th century BCE. The ancient mathematicians who first wrote about liner algebra didn't foresee that the nuances of computer architecture would one day make indexing from 0 more convenient.

Matrices have mathematical properties that allow you to do algebra using them, combining various elementary operations to produce useful results. In this lab, you will write functions that implement the following algebraic operations:

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, two matrixes are equal if ALL of the elements at equivalent locations within both matrices are equal. If A = B, then, a_{11} and b_{11} need to be equal, AND a_{12} and b_{12} need to be equal, and so on for every element of A and B.

Since this library is working exclusively with floating-point numbers, we need to discuss something called round-off error[5]. Most numbers cannot be perfectly represented in the binary format that the computer stores numbers in, so this round-off error is unavoidable. [6]

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.

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.

---

[5] Again, remember your CE12 IEEE Floating Point number discussion, and how a fixed number of bits are used in the mantissa and significand. The floating point numbers 0x40490fdb and 0x40490fdc would both seem like pi to you, but C would interpret them as different numbers!

[6] Only numbers that can be represented as an integer power of 2 can be represented exactly in the binary system. (Why?)

Determining how close two numbers are relies on a simple subtraction. The only issue then is that the difference might be positive, or negative. You can easily solve this by finding the absolute value of the difference.

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:

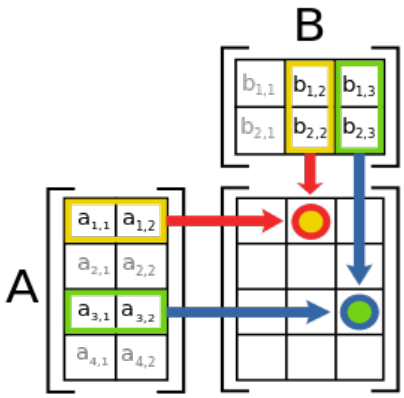
$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} * g = \begin{bmatrix} a_{11} * g & a_{12} * g & a_{13} * g \\ a_{21} * g & a_{22} * g & a_{23} * g \\ a_{31} * g & a_{32} * g & a_{33} * g \end{bmatrix}$$

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. Here, A is a 4x2 matrix and B is a 2x3 matrix. The result, C = A*B, is a 4x3 matrix. Any given element of C (say, c_{12}) can be calculated as:[7]

$$c_{12} = [a_{11} \quad a_{12}] * \begin{bmatrix} b_{12} \\ b_{22} \end{bmatrix} = a_{11} * b_{12} + a_{12} * b_{22}$$



This process is then carried out for every element in C.

Note that matrix multiplication is *not* commutative, so A*B = B*A is usually not true!

Trace:

The trace of a matrix (usually written as tr(A)) is the sum of all of the diagonal elements of a matrix. Here, "diagonal" means 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 of a 3x3 matrix are highlighted below:

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

Transpose:

The transpose of a matrix is just the mirroring of all of its elements across its diagonal. You can think about it as "flipping" the matrix. The notation to describe the transpose of a matrix is with a superscript T:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}^T = \begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix}$$

[7] Picture from wikimedia commons, File:Matrix_multiplication_diagram_2.svg

Another way to think about the transpose is as a matrix that uses the same list of elements, but encodes them in column-major form.

Submatrix:

The *submatrix* of a matrix for element *i,j* is a smaller matrix, formed by removing row *i* and column *j* from the original matrix. For example,

$$\mathrm{A} = \begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix}$$

$$submatrix_{2,3}(\mathrm{A}) = \begin{bmatrix} a & d & - \\ - & - & - \\ c & f & - \end{bmatrix} = \begin{bmatrix} a & d \\ c & f \end{bmatrix}$$

This submatrix isn't particularly useful on its own, but it simplifies many other calculations, such as finding the determinants and inverses of large matrices.

This may be a challenging function for you to implement. Be creative and try things! If worst comes to worst, you may implement this as a large if/else tree, but first try and find a more elegant solution.

Determinant:

Every square matrix has a *determinant*. It is a simple scalar that is a sort of summary of all the information in the matrix. Determinants have a variety of uses: They can be used to *determine* if a physical system is stable or unstable, to *determine* whether matrix multiplication is undo-able, or to *determine* whether an equation with a matrix has a unique solution.

For a 2x2 matrix, the determinant is easily found using a simple formula:

$$\det \begin{bmatrix} a & b \\ c & d \end{bmatrix} = a * d - b * c$$

For higher-dimension matrices, the determinant can be found by selecting a row (or column) of the matrix and calculating a weighted sum of the determinants of the submatrices of each element. For example, for a 3x3 matrix, the determinant can be calculated as:

$$\begin{aligned} \det \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \quad & a_{11} * \det\big(submatrix_{11}(A)\big) \\ & - a_{12} * \det\big(submatrix_{12}(A)\big) \\ & + a_{13} * \det\big(submatrix_{13}(A)\big) \end{aligned}$$

Note that the determinants on the right-hand side are determinants of 2x2 matrices. You will probably find it helpful to create (and test) a helper function called MatrixDeterminant2x2() that calculates 2x2 determinants.

Inverse:

The inverse of a square 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).

$$\mathrm{A} * inverse(\mathrm{A}) = \mathrm{I} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The calculation of the inverse for 3x3 matrices is managable 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, with each element divided by its determinant:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}^{-1} = \frac{1}{\det(A)} * \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}^{T}$$

where each element of cofactor matrix is defined as follows:

$$c_{i,j} = \begin{cases} \det\left(submatrix_{i,j}(A)\right) & \text{if } (i+j) \text{ is even} \\ -1 * \det\left(submatrix_{i,j}(A)\right) & \text{if } (i+j) \text{ is odd} \end{cases}$$

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:

1. the library description, detailed in one or more header files,

2. 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. You could write your own code that followed the same function prototypes, and you wouldn't need to change any code in your project to use this new in-house implementation![8]

So libraries are very useful for separating out functions that are useful for other people or other projects. Of course, there is a downside to writing a library: someone else may use your code and complain to you about it.

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

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

It may seem counterintuitive, but you should write the tests BEFORE you 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 write every function before doing any testing. You'll inevitably find yourself re-writing most of your code. Instead, get into this rhythm: Write a function, run some tests, write a function, run some tests, etc…

You have to be extra careful when writing tests. A test with a bug in it is often worse than no tests, since it can lead you away from bugs in the code it is supposed to be testing. The best strategy for this is to write a diverse set of tests, that use different approaches to test each function.

Lastly you'll want to make sure that the results logged by your unit test harness are easy to read. Make sure the output formatting is correct, and make sure that you can easily identify both *whether* an error was detected, and *where* that error was detected.

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:

- Double-check your lab. Confirm that you implemented all of the functions necessary following any rules. Reread the Assignment Requirements section to check this.
- Finish your README.md 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_test.c, MatrixMath.c, and README.txt).
- Tag your submitted commit.

Example Output

Your test output should look something like this:

- For every function: 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 |
 -------------------
```

Your test harness does not need to look exactly like this. If you want to add additional output above this final summary, go ahead.

Make sure your output is readable, and that it does the main thing that unit test should do: Help a user localize any bugs in your code.

Note that we may intentionally break your code, and then find out if your test harness will catch it!