Lab 2 – A matrix math library

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.

Required reading

- K&R Chapter 1, 4.1, 4.2, 4.5, 5.7
- Style guidelines discussed in Lab 0

What we provide

- 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!
- lab2.c – This file contains main() and all the support code you’ll need. Includes a single test of MatrixMultiply. You will add tests for each other function that you implement in this file. (This will utilize the Proteus VSM within MPLAB and require no changes to the Explorer 16.DSN model as with lab 1).

Assignment requirements

- Create a new file called MatrixMath.c that implements all of the functions whose prototypes are in the header file MatrixMath.h.
- Expand main() in lab2.c to include more unit tests that test every function in MatrixMath.h at least once. Start by testing MatrixEquals() and continue from there (remember to include proper testing for floating point equality).
- Add the following to the top of your MatrixMath.c file as comments:
  - Your name
  - The names of colleagues who you have collaborated
- Format your code to match the style guidelines that have been provided.
- Make sure that your code triggers no errors or warnings when compiling. Compilation errors will result in NO credit. Compilation warnings will result in lost points.
- Submit both MatrixMath.c and lab2.c via eCommons before the due date.
Grading

This assignment again consists of 10 points:

- One point each for each function you needed to implement as well as a working test for it.
- One point for adhering to the style guidelines and including the required comments in MatrixMath.c.

Passing by reference

In C functions are normally passed 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:

```c
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”.

Now there’s another way to pass variables as 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.

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 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! Hopefully now the way these functions were used makes more sense.

**Working with matrices (linear algebra)**

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 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. As a matrix is two-dimensional it consists of rows and columns. A 3x3 matrix therefore has 3 rows and 3 columns (it is also big enough to be used for the 3D game worlds mentioned previously). Without further ado, this is what a 3x3 matrix looks like:

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

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. The way to define a matrix like this in C would be: “float a[3][3];”. The first subscript refers to the row (like \( i \) did) and the second refers to the column (as \( j \) did). To refer to a specific element you then reference it like “\( a[0][0] \)”, with references the upper-leftmost element in the matrix. Remember that array indexing starts at 0 in C!

Initializing a 2-dimensional array in C is a little different than for one-dimensional arrays, however as you need to use an inner set of curly-braces:

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

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 arithmetic on them: multiplication and addition primarily.

**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 for 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.

Since this library only works with 3x3 matrices the first condition will always be true. But it’s a little trickier for comparing all of the elements. Since this library is working exclusively with floats we need to discuss something called round-off error. What this
means is that some numbers cannot be perfectly represented in the base-2 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 perfectly in the binary base-2 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 some inaccurate calculations sometimes. 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 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 regular situations like this.

Now to determine how close two numbers are together you can just subtract them. The only issue then is that you either have a positive or negative result. You can either use the fabs() function from math.h or check that this number is either less than delta AND greater than –delta.

**Matrix-matrix addition:** First let’s start with 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:

\[
\begin{bmatrix}
a_{11} & a_{12} & a_{13} \\
a_{21} & a_{22} & a_{23} \\
a_{31} & a_{32} & a_{33}
\end{bmatrix} + B =
\begin{bmatrix}
a_{11} + B & a_{12} + B & a_{13} + B \\
a_{21} + B & a_{22} + B & a_{23} + B \\
a_{31} + B & a_{32} + B & a_{33} + B
\end{bmatrix}
\]

**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 the row of this element from the first matrix multiplied by all of the elements of the column of this element from the second matrix. The best way to clarify this is with an example:

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

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 1 - Sarrus' rule for the determinant of a 3x3 matrix](image)

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 $a_{12}$ and $a_{11}$ 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).

$$
\begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{bmatrix}
$$

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 adjugate matrix divided by its determinant:

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

where the right-hand matrix is defined as follows:

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

**Writing a library**

In programming libraries refer to existing code that can be easily incorporated into your own. This will 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 inputs and outputs 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! 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. This now leads us into the discussion about testing.

**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 lab2.c). It is used by people when developing large amounts of code to do two things: 1) confirm that the code operates as expected and 2) that any future changes to this code also work as expected.

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

1. Generate input parameters
2. Call the function that you’re testing
3. Verify the output against what you expected
4. Log the results

The single unit test that has been included in `main()` follows this paradigm. It is well-documented and should provide an example of good testing.

When writing code and unit tests for it keep in mind that the proper way to do it is to write the tests first and then write the function. This serves a couple of purposes: first it confirms that you know how the function is supposed to work and can therefore code it and second it results in more correct tests. Since you aren’t thinking about writing code at all yet you aren’t worried about mentally checking the code as you write the test. This can cause you to write tests that your code passes versus writing correct tests and then seeing if your code can pass them.
Also 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. I recommend starting with MatrixEquals() and then moving on to MatrixMultiply() and the first test you were provided.

One thing to be careful about is whether your tests are 100% correct. If either your test’s inputs or the known good 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 means that one of the first things you should check if a test fails is whether the input and output is correct. If it is, then it is definitely your function that is at fault and you’ll find the problem inside of it.

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. This is already done in an easily extensible way in the provided code, so just continue to use this framework.