1 Reading

Reading the csv files was an interesting challenge, since I was unable to solve the problem for lab 8. I re-discovered a section from lab 5 describing how to read a csv file. This, along with searching online, proved to be very useful. The method I employed involved storing the opened file as a variable. I would then search one line of the file for a comma. When the comma was reached I stored the previous characters as an integer. I repeated the process of searching for commas and storing the elements in an array until the end of the line was reached. At this point I would move store the array in a larger array and move onto the next line of the csv file.

Allowing files to be specified in the command line just involved invoking *int argc, char* argv[]* after specifying main. I then read the file specified in the first argument, stored it in a matrix, closed it and repeated the process for the second file. To ensure I only had two files specified in the command line I checked that *argc == 3*, responding with an error if this was false.

2 Sanitization

Input sanitization followed reading the files. To check that the input arguments were files I wrote a function that checked for a decimal point. It then ensured that "csv" followed the decimal point. If the command line argument did not end immediately after that then the input was rejected as not being a csv file. I made sure the files were readable by checking that the files were not null. If the file could not be opened the *fopen* function would not work and an error would result from the null check. I searched for commas when storing the file data in the elements. This, in conjunction with other checks, ensured that the matrices were stored in the correct format.

I checked that the matrices were the expected sizes by checking the length of each array. If the length of one row of the matrix was different than the length of any other row of the matrix then the program was ended for the matrices being improperly formed. I checked that each input was a number using another function. It ensured that each character of the element was either a number, one negative sign, or one decimal point. If the element was not a matrix the program ended.

I allocated memory to the matrices based on the size of the files. I checked the number of columns in the first row of the files. I then checked the number of rows in the files. I used these results to dynamically allocate the matrices. Since the matrices were allocated as floats I made sure that any element did not exceed the limits of a float. If the element overflowed or underflowed a float the program ends.

I checked that the matrices could be multiplied by ensuring that the number of columns of the first matrix equaled the number of rows of the second matrix. As long as this was true, and the other checks passed, then the matrices could be multiplied.

3 Multiplication

To multiply the matrices I invoked the equation

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \tag{1}$$

Which would multiply the an element of a given row from matrix one by the corresponding element of a given column from matrix two. All of the products of the elements from the given row and column are then summed. This provides one element of the matrix product, the process is repeated for each row and column combination to give the entire matrix product.

While this process works to give the cross product, it is inefficient. The first matrix will receive a cache hit every time as it is being accessed. The second matrix, however, receives a cache miss every time since the matrix is accessed one column at a time. To circumvent this I rotated the second matrix. However, C++ does not allow you to simply rotate a matrix. Instead I simply stored the second matrix column first. This may just move the problem ahead instead of correcting it, but at least the computer is not waiting on the second matrix to provide a cache hit to perform a multiplication.

The time complexity of this process is dependent on the sizes of both matrices. Increasing the size of the first matrix by one row will increase the number of multiplications by the number of columns in the second matrix. This corresponds to a linear growth rate with the size of the data. As such this code has a complexity of O(n) and a linear run time.

4 Writing

To output the matrix I first created a write-able csv file. I then stored each element of the matrix in the file use *fprintf*. Between each element I would insert a comma. At the end of a row I went to a new line in the file and repeated the process for the next row. When all of the elements were stored I closed the file.

5 Makefile

Since all of the code is stored in one file the makefile is quite simple. I called the code to make the .o file. I then used that file to create the executable. Since no external libraries were used I did not need to make any specific calls to them.