Lab #12

Physics 91SI Spring 2013

Objective: In this lab, you will write some basic programs in C, preparing you for integrating fast C code into a Python program.

As usual, log on to `corn` and clone over the starter repository:

```
hg clone /afs/ir.stanford.edu/class/physics91si/src/lab12 lab12
```

NOTE ABOUT COMPILING C CODE: A Makefile is included in the lab12 folder. A Makefile is a sort of shell script for using gcc to compile C code. For your use, you can type `make factorial` and this will take the code in `factorial.c` and create a `factorial` file that you can run by typing `./factorial` (this applies for the other .c files as well).
Remember to `hg commit` often to save your changes, and submit your code at the end of class.

Part 1: Factorial in C

A `factorial.c` file has been started for you. Look at the basic skeleton that has been set up for you, ie how the main function is declared and structured, how variable declaration is done, if statement formatted, etc. The file includes plenty of comments to explain what's going on in this new syntax.

A factorial function has been declared for you, but has no substantial code in its body. Replace the temporary "`return 0;`" with your implementation of a factorial function. You may implement it however you'd like (recursively or iteratively), as long as it works in C and returns the correct value.

Once you're done editing the file, run `make factorial` and then run `./factorial n`, where n is a numeric command-line argument. Does your code compile when you run make factorial? Does it work properly when you run it for a given number?

Part 2: Matrix Multiplication in C

The matrixmult.c file has been set up to create two 10 x 10 matrices by reading in values from two files and at the end of the main function, it prints the values in the result matrix. Add code to the file that will perform matrix multiplication, mat1 * mat2, and will save the answer to the result matrix. You should update `result[ i ][ j ]` after each time you calculate an element of the product, ie don't try to assign anything to `result` itself, but rather, make assignments to specific elements in `result`.

Part 3: Taylor Series Approximation of Sine in C

A taylorSine.c file has been started for you. The goal here is to write a function called TaylorSine, which takes two arguments provided from the command-line, the angle in radians, and the polynomial order of the error in the approximation, and returns an approximation to sin(x) up to O(x^n) corrections.

If you don't know the *taylor series for sin(x) at x=0*, you can just type the italics into WolframAlpha to get a nice expression, and click on "more terms" to see a larger trend. Notice that the terms are alternating in sign (+/-) and each term has x raised to an exponent that was +2 away from the exponent in the previous term. These are details you have to account for as you sum terms to get a final answer. Also, each term is divided by (exponent)!, so you should copy your factorial function into this new file. Lastly, to raise something to an exponent in C, we included the cmath library for you, so you can just call `pow(base, exponent)`.

Once you've finished editing the file, compile it using make and run it, just like you did with factorial.c, and compare your program's approximation to sin(0.1) to the actual value.