Assignment 2: Implementing a Numerical Library and Corresponding Test Harness

James Contini
10.2.22

Necessary Files -

Makefile: Instructs the computer how to compile the necessary executables and destroy unnecessary or unwanted files.

mathlib.c: This file is the library that contains the c implementation of the math functions.

mathlib.h: Header file to include in *mathlib* to prototype all of the functions.

mathlib-test.c: This is the test harness to interpret input from the command line and call the right library functions.

Control Flow -

In my implementation the user will only interact with *mathlib-test.c*'s executable. They will run *mathlib-test* and append command line options to indicate which function from *mathlib.c* they want to test. For example *~:$ ./mathlib-test -s* roughly translates to "ask *mathlib-test* to ask *mathlib* to run its sin function tests".

The user can also ask *mathlib-test* to run all tests with the command line options *-a*. However a test will never be run twice in one execution because the program first checks to see if the command line includes a *-a* before any function calls are made. If option a is discovered then all tests are run automatically, but if there is no *-a* the *mathlib-test* calls each function in the order they are provided in the command line.

Test Harness Implementation -

The test harness, *mathlib-test*, immediately loops through all of the command line options to check if an *-a* is present. If present, a variable *a* is assigned *true,* if *a* is *true* then the loop checking for other command line options is never executed and all tests from mathlib are run. If *a* is *false* we continue to a loop through the command line options, and using a switch execute the math tests chosen by the user.

///When a test is execute executed by the user an array of numbers from

Library Implementation -

my_sin(): This function takes advantage of a taylor series to approximate sin(). First, five variables are initialized: *denom = 1*, *numerator =1*, *zero = 0*, *total = 0*, and *prev_k = 0*. Next, a for loop is initialized with loop variable *n* from range *0 - 11*. This number represents the index of the sum. Inside the loop, a variable *k* is initialized to the value *(n * 2) + 1*. To calculate *denom*, an random variable *c* is initialized with value *k* then iteratively *denom* is assigned to *c * denom*. With each iteration c is decrements and the loop continues until *c* (which has the value of k) equals *prev_k*'s value. Essentially this is just calculating a factorial through dynamic programming instead of recursion. This implementation also has the added benefit that the entire factorial doesn't need to be recalculated every time because the for loop just picks up where it left off last z_n.
numerator is calculated in a similar way. Iteratively, *numerator* is assigned *numerator * x* (*my_sin'*s input) and *prev_k* increments until *prev_k* equals current *k'*s value. All this does is multiply *x* to itself *k - prev_k* iterations. This, just like *denom,* has the same heuristic of remembering the previous value of z_n, so each iteration doesn't have to calculate the entire numerator over and over again.
Lastly, if *n* is even *numerator / denom* is added to *total*. And if *n* is odd, it is subtracted from *total.* And then *prev_k* is assigned current *k* for the next iteration.

my_cos(): This function is exactly the same as *my_sin().* Except the only difference is that *k* which is (*2 * n + 1)* in *my_sin()* is (*2 * n)* in *my_cos().*

my_arcsin(): This function is contingent upon the previous sin and cos functions to work. For the piazza implementation, three variables must be initialized: *double EPSILON = 1-e10; double zn = 0;* and *double new_zn;*. Next a do while loop is created whose run condition is that the difference between new_zn and zn must be greater than EPSILON. Inside of the loop *new_zn* is assigned by computing *my_sin* of *zn* minus the input parameter *x* all divided by *my_cos* of *zn*. Then all of that is subtracted from *zn.* At the bottom of the loop *zn* is reassigned to the value of *new_zn* and the loop continues. Finally when the run condition fails *new_zn* is returned.

my_arcscos(): This function is implemented by returning *pi/2 - my_arcsin()*

my_arctan(): This function can be implemented very easily. First initialize a variable *t* of type double. And then in the next line assign *t* the result of input parameter *x* divided by *my_sqrt()* function with parameters *x* times *x* plus one. Return *t*.

my_log(): Just like *arcsin()* to implement log we will use the Newton Raphson method and a function e^x. First, three variables must be initialized: *double EPSILON = 1-e10; double zn = x* (the input parameter)*;* and *double new_zn;*. Next a do while loop is created whose run condition is that the difference between new_zn and zn must be greater than EPSILON. Inside of the loop *new_zn* is assigned by computing *exp()* of *zn* minus the input parameter *x* all divided by *my_cos* of *zn*. Then all of that is subtracted from *zn.* At the bottom of the loop *zn* is reassigned to the value of *new_zn* and the loop continues. Finally when the run condition fails *new_zn* is returned.