CSE13s Assignment 2 Design Document

Liam Murray

Lijamurr

General Idea

Create a series of basic mathematical functions in order to get a better grasp on how the built in C function library works. This coded library will be accompanied by a test file, exemplifying the performance differences between the manually implemented and the built in functions.

This is done by creating a series of lower level mathematical functions, such as the square root and exponent functions, which will serve as building blocks to create the final, more complex functions. The idea behind first creating less complex functions is to simplify the final design, and to modularize reused code to improve debugging and maintenance. The complex functions will be approximated either by using their respective Taylor series, or with the Newton Method as is the case with the "log()" function, as this is an efficient, accurate way to recreate functions without using the original function equation itself. The test file will be a group of simple loops, which correspond to command line inputs and designates which tests to run. The tests will also be a simple loop, running the built in functions then comparing them to the manually implemented functions based on the increment required by each function.

Pseudocode

#Basic Mathematical Functions

#Factorial

#The factorial function is used in the taylor/maclaurin series as part of calculating the "nth" term in the series. This is implemented by running a loop, decrementing the given number by one each pass, and multiplying a counter by each number.

```
    Begin Factorial Function
        Store input in “starting value”
        Set current value = input
        Begin loop
            Set starting value = current value * starting value
            Decrement current value one
            Exit loop when current = 2 #multiplying by one is a waste of time/memory
        Return “starting variable” value
#Exponent Function
#this is used to calculate the log, e^x and sine/cosine functions
Declare function, accepting two inputs
    Store base input in variable “b_input”
    Store exponent input in variable “x_input”
    Begin loop, looping x_input times
        B_input = b_input * b_input
    Return “b_input” as final value
#Square root function. SOURCE: code taken from Professor Miller’s Piazza Post.
Declare function, accepting 1 input
    Check that the input is greater than 0
    Set a scaling factor of one
    “Guess” the square root, starting at 1.
    Make a While loop, while the input is greater than zero
        Divide input by 4 #because rt(4) is 2
        Multiply final value by 2 #storing root value
        Create a for loop to update the guess, and get it within acceptable accuracy
        Return value
# Absolute Value Function
#Fairly simple. This function simply reverses the sine of X if it is negative. It is used primarily in the sine/cosine functions.
    If x is > 0, pass. Else x = -x
#Complex Functions
```

```
#Sine- Taylor Series
    Set x = input
    Set last_input = x #this is the first term of the Taylor series
    Set running_total = x #first term to be added to the running total
    #This is a shortcut for calculating the “nth” term in the taylor series, which is more efficient then calculating it manually each time. By multiplying the terms together and adding them to the running total, one can cut down significantly on the number of loops required to make it function.
    Make a while loop
        Set term 1 = x / (2n)
        Set term 2 = x / ((2n) +1)
        Set current value = (last_input) * term1 * term2
        Take the absolute value of the current value
        If N is even
            current value is positive
        Else
            Current value is negative
        Add current value to running total
        Set last entry to current value
        Exit when the last entry <= Epsilon, the value where adding it is not helpful
    Return running total
#Cosine
#This is a more straightforward, yet less efficient way of calculating the Taylor Series for Cosine. This function calculates successive terms in regards to “n” and “x”, rather than the previous function inputs, and adds it to a running total variable.
Set x = input
Set cos_val = 0 #this is a blank counter
    Make a while loop to run “n” times
        Set term 1 = (-1^n)
        Set term 2 = x ^ (2n)
```

Set term 3 = (2n)!

Add Term 1 * (term 2 / term3) to counter

Exit when the last entry <= Epsilon, the value where adding it is not helpful

Return cos_val

#ArcSine

This is an effective, efficient way to implement the arcsin Taylor series. This function is calculating successive terms based on the previous terms in order to cut down on processing.

Set x = input

Set last value = x #first term in arcsine taylor sequence

Set loop to iterate “n” times

Set Previous Term = last value

Set Term 1 = (The sine of last value) - x #using my sine function

Set Term 2 = The cosine of last value

New term = last value - (term 1 / term 2)

Set last value to new term

End loop when (new term - previous term) < Epsilon

#ArcCosine

this uses the arcsine implementation to calculate ArcCosine in an effective, elegant way

Set x = input

Return (math.pi / 2) - arcsine(x)

#ArcTangent

calculates the arctan based on previously implemented arcsine and arccosine to avoid reusing code

Set x = input

Set Term 1 = x

Set Term 2 = (my_squareroot of ((exponent of x, 2) + 1))

Return arcsine of Term 1 / term 2

#Log()

The Newton method for calculating log, used by finding the inverse of the e^x function.

Set y = input

last value = 1

While loop

Set Term 1 = (y - e^last value) #calculated using my e^x function

Set term 2 = e ^ last value

Set current = last value - (term 1 / term2)

Set last value = current value

End loop when current - last value < Epsilon

Return last value

#e^x

function used to calculate log(). It is a fairly simple, efficient recursive definition.

Set x = input

Starting value = 1 #blank value to begin

Counter = 1

While Loop

New value = Starting Value * (x / counter)

Starting Value = new value

Terminate when (new value / counter) < epsilon

Return Starting Value

#Testfile

Include the Stdio.h files required to run the given functions.

Create a function print_test(function_name, lib_function_name, start, end, increment)

Set initial = start

print("x   function_name lib_function_name,   difference")

Create While loop

print(Start   function_name(initial)   lib_function_name(initial)

abs(sin(initial) - my_sin(initial))

initial += increment

Terminate when initial == end

#By creating a function like this, we can factor our reused code to make implementation much easier and more elegant.

Define main

Create variable opt, initialize it to 0.

Use getopt() to set opt to any given command line arguments

```
If opt is s:
    print_test(my_sine, sin, 0, 2pi, .05pi)
If opt is c:
    print_test(my_cosine, cosine, 0, 2pi, .05pi)
If opt is S:
    print_test(my_arcsine, arcsin, -1, 1, .05)
If opt is C:
    print_test(my_arccosine, arccosine, -1, 1, .05)
If opt is T:
    print_test(my_arctan, arctan, 1, 10, .05)
If opt is l:
    print_test(my_log, log, 1, 10, .05)
If opt is a:
    print_test(my_sine, sin, 0, 2pi, .05pi)
    print_test(my_cosine, cosine, 0, 2pi, .05pi)
    print_test(my_arcsine, arcsin, -1, 1, .05)
    print_test(my_arccosine, arccosine, -1, 1, .05)
    print_test(my_arctan, arctan, 1, 10, .05)
    print_test(my_log, log, 1, 10, .05)
```