Areeb Amjad
afamjad@ucsc.edu
4/7/21

CSE 13S Spring 2021

Assignment 2: A Small Numerical Library
Design Document

This assignment first involves the replication of four functions available in the <math.h> library: asin(x), acos(x), atan(x), and log(x). These functions all take a floating-point number as an argument and return their respective values provided the input value is within the functions' domains. The assignment then requires that the replicated functions be compared to those in the math.h library. The functions will be tested as follows:

1. arcSin() and arcCos() will be tested in the range [-1, 1] with steps of 0.1
2. arcTan() and Log() will be tested in the range [1, 10] with steps of 0.1

The output of this program will be the comparison of the functions in table format (two examples are provided in the assignment document).

TOP LEVEL:

This is the initial top-level design for mathlib.c:

```
define a constant EPSILON and set it to 10 x 10^-10

Fact(double x):
    set result to 1
    while x is greater than 0:
        result *= x
        decrement x
    return result

Pow(double x, int y):
    define result variable and set to 1
    for an integer i = 0; as long as i < y; increment i:
        sum *= x
    return result

Abs(double x):
    return -x if x < 0 else return x

arcSin(double x):
    define a term and sum variable and set both to 1
    for an integer k = 0; as long as absolute value of term greater than EPSILON;
    increment k:
        term *= Fact(2k) / (Pow(2, 2k) * Pow(Fact(k), 2))
        term *= Pow(x, 2k + 1) / (2k + 1.0)
        sum += term
    return sum
```

```
arcCos(double x):
    return pi/2 - arcSin(x)

arcTan(double x):
    return arcSin(x / Sqrt(Pow(x, 2) + 1.0))

Log(double x):
    check that x > 0
    define guess variable and set to 1
    while true:
        define guess_update = guess + (x - Exp(guess)) / Exp(guess)
        if absolute value of guess_update - guess less than EPSILON:
            break
        set guess to guess_update to update guess
    return guess
```

I have added helper functions to make writing the four functions easier.

`Fact:`
This is the factorial function that, given a `double x`, returns the factorial of the number. It is pretty simple; a variable `result` is set to 1, and then we multiply `result` by `x` and decrement `x` by 1 while `x` is greater than 0. If `x` is 5, then we multiply `result` by 5 and store it in `result`, which gives us 5. We then decrement `x` by 1, so it is now 4, and then multiply 5 by 4 and so on until `x` is 0. The result is: 5 x 4 x 3 x 2 x 1, which is 5!.

`Pow:`
This is the power function. While I could multiply the term by itself, I figured I should write a function to make my code more readable. Given a `double x` and an `unsigned integer y`, `Pow` returns x^y. It defines a `result` variable and then enters a for loop which runs `y` times. In it, `sum` is multiplied by `x` and stored in `sum`. As an example, $2^2 = 2 * 2 = 4$. The program multiplies `x` by itself `y` number of times.

`Abs:`
This is the absolute value function and it is one line. It returns `-x` if `x` is negative; otherwise, it returns `x`.

`arcSin:`
This is the $\sin^{-1}$ function. It approximates $\sin^{-1}$ using the Taylor Series approximation. A Taylor Series is defined to be an infinite sum of the function's derivatives about a point α (this series is centered about 0). Since it is impossible to sum an infinite number of terms, the program sums the terms until the current term is less than the epsilon constant, 10^{-10}. Inside the for loop, `term` is calculated in two lines for readability.

`arcCos:`
Once `arcSin` is calculated, it is easy to calculate `arcCos`. It is $\pi / 2 -$ `arcSin(x)`. This is due to an algebraic rearranging of the trigonometric identity: $sin(\theta) = cos(\pi / 2 - \theta)$.

`arcTan:`
`arcTan` is also simple to calculate once `arcSin` is calculated. In terms of `arcSin`, it is:
`arcSin(x /` $\sqrt{x^2 + 1}$`)`. This equality can be proven through a right triangle with sides 1, x, and $\sqrt{x^2 + 1}$.

`Log:`
For this assignment, `Log` is calculated using Newton-Raphson's method. It is a root-finding algorithm that gets more accurate with successive guesses. The next best guess is defined as:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

where x_n is a guess for a root of f and x_{n+1} is the successive guess. The function f will be defined as the inverse of log(x) such that log(x) is a root of f. This gives us:

$$x_{k+1} = x_k + \frac{y - e^{x_k}}{e^{x_k}}.$$

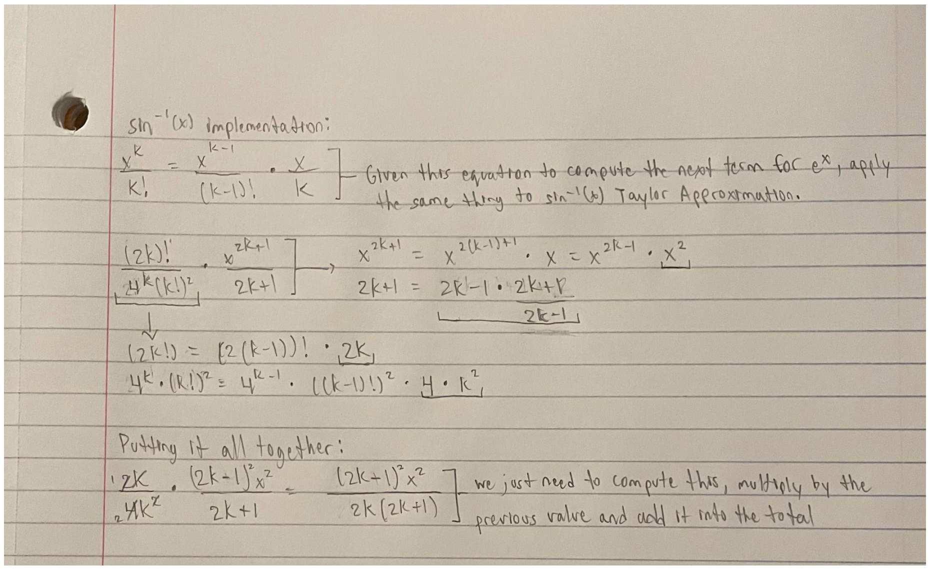
The `Log` function will run an infinite loop that makes better guesses until the difference between the guess and the successive guess is less than the epsilon constant, 10×10^{-10}.

DESIGN UPDATE 1:

My initial pseudocode for `arcSin` was faulty because I attempted to compute the Taylor approximation as it was presented to me instead of computing the next term and multiplying it by the previous term, as the assignment document suggested. Using this equation provided in the assignment document as a basis:

$$\frac{x^k}{k!} = \frac{x^{k-1}}{(k-1)!} \times \frac{x}{k}.$$

I calculated the next term for the $\sin^{-1}$ Taylor series approximation:



$\sin^{-1}(x)$ implementation:

$\frac{x^k}{k!} = \frac{x^{k-1}}{(k-1)!} \cdot \frac{x}{k}$] Given this equation to compute the next term for e^x, apply the same thing to $\sin^{-1}(x)$ Taylor Approximation.

$\underbrace{\frac{(2k)!}{4^k(k!)^2}} \cdot \frac{x^{2k+1}}{2k+1}$] $\rightarrow$ $x^{2k+1} = x^{2(k-1)+1} \cdot x = x^{2k-1} \cdot \underbrace{x^2}$

$2k+1 = \underbrace{2k-1 \cdot \frac{2k+1}{2k-1}}$

$(2k!) = (2(k-1))! \cdot \underbrace{2k}$

$4^k \cdot (k!)^2 = 4^{k-1} \cdot ((k-1)!)^2 \cdot \underbrace{4 \cdot k^2}$

Putting it all together:

$\frac{2k}{4k^2} \cdot \frac{(2k-1)^2 x^2}{2k+1} = \frac{(2k+1)^2 x^2}{2k(2k+1)}$] we just need to compute this, multiply by the previous value and add it into the total

With this new equation, I modified my pseudocode for `arcSin` to the following:

```
arcSin(double x):
    define a current variable and set it to x
    define a sum variable and set it to current
    Define a previous variable and set it to 0.0
    for an double k = 0.0; as long as absolute value of previous - current greater than
    EPSILON; increment k:
        previous = current;
        current *= Pow((2k -1), 2) * Pow(x, 2)
        current /= 2k * (2k + 1)
        sum += current
    return sum
```

There are a couple of differences worth mentioning here. The first is that I split what was the `term` variable into `current` and `previous`, which represent the current and previous terms. I set `current` to `x` and `previous` to 0 so that the function can enter the loop. I set `sum` equal to `current` since `current`, which is `x` before the function enters the loop, is the first term of the $\sin^{-1}$ Taylor series approximation.

The new check for the loop is that the absolute value of `previous - current` term is less than the epsilon constant. Inside the loop, `previous` is set to `current` to update `previous`, and then the next term is calculated and added to the `sum`.

Since `Fact` is no longer needed for this function, I decided to remove it.

DESIGN UPDATE 2:

The first issue I had with my program was the output of the arcTan function. It relies on arcSin, and that function used the Taylor Series to approximate, which converges very slowly for arcSin at -1 and 1, and converges slowly for arcTan when x ≥ 1. I decided to use Newton's method instead.

After I implemented Newton's method, I realized that my approximations for arcSin and arcCos at -1 and 1 were still slightly inaccurate (they were only accurate to two decimal places). I decided to use implement the advice that Professor Long gave about these boundary cases.

For arcSin:

If x is 1, $sin^{-1}(x) = cos^{-1}(\sqrt{1 - x^2})$.

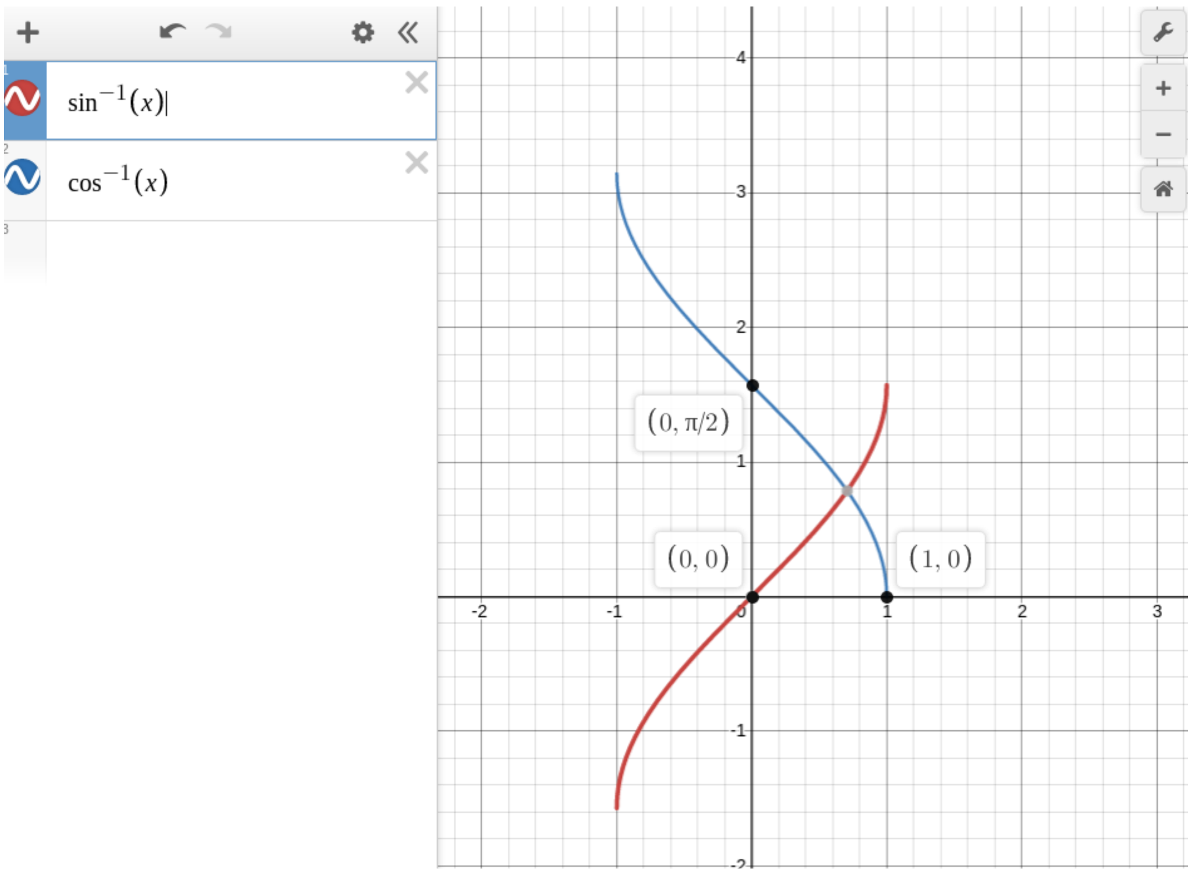
If x is -1, $sin^{-1}(x) = - cos^{-1}(\sqrt{1 - x^2})$.

For arcCos:

If x is 1, $cos^{-1}(x) = sin^{-1}(\sqrt{1 - x^2})$.

If x is -1, $cos^{-1}(x) = \pi - sin^{-1}(\sqrt{1 - x^2})$.

To get the equations for when x = -1, I looked at the graphs of arcsin(x) and arccos(x) as our Professor also suggested:



With these equations, my pseudocode relating to arcSin and arcCos changes:

```
arcSinHelper(double x):
    define guess variable and set to 1
    while true:
        define guess_update = guess + (x - Exp(guess)) / Exp(guess)
        if absolute value of root - guess less than EPSILON:
            break
        set guess to guess_update to update guess
    return guess

arcSin(double x):
    check that x is between -1 and 1
    if x equals 1:
        return arcCos(Sqrt(1 - Pow(x, 2)))
    if x equals -1:
        return -arcCos(Sqrt(1 - Pow(x, 2)))
    normally calculate arcSin with arcSinHelper

arcCos(double x):
    if x equals 1:
        return arcSin(Sqrt(1 - Pow(x, 2)))
    if x equals -1:
        return pi - arcSin(Sqrt(1 - Pow(x, 2)))
    return pi/2 - arcSinHelper(x)
```

I have now split `arcSin` into `arcSinHelper` and `arcSin`. `arcSinHelper` looks very similar to `Log`, and that is because they both are using Newton's Method. Now, `arcSin` checks for cases -1 and 1 and returns the respective trig identity. If x are neither 1 nor -1, then `arcSin` calls `arcSinHelper`.

`arcCos` follows the same pattern as `arcSin`. It checks for cases -1 and 1 and returns the respective trig identity; otherwise, it returns $\pi / 2$ − `arcSin(x)`, its normal calculation.

DESIGN UPDATE 3:

I was unclear on whether I should include a design for testing the function implementations since I initially thought it was just printing, but there was a little logic that went into displaying the functions in the terminal.

The top-level design for mathlib-test.c is as follows:

```
main:
    define boolean flags for -a, -s, -c, -t, and -l and set them to false
    while command-line arguments are still being parsed:
        switch (current char) {
            case a:
                set a flag to true
            case s:
                set s flag to true
            case c:
                set c flag to true
            case t:
                set t flag to true
            case l:
                set l flag to true
            default:
```

```
            print error statement
            end function

    if a flag is true:
        print all functions
    if s flag is true:
        print arcSin
    if c flag is true:
        print arcCos
    if t flag is true:
        print arcTan
    if l flag is true:
        print Log

    end function
```

The printing is done in the `main` function. However, before printing, the program first defines boolean flags for each command-line argument so that it knows which ones to print. Then it enters a while loop that parses through the command-line arguments. Inside the while loop, there is a switch case to match the current character with the case and set the respective boolean flag to true. If the user enters an invalid argument then the default case is entered, in which case an error message is printed and the program is terminated.

After the arguments have been parsed through, the program exits the while loop and goes through a series of if statements to print the function (implementation, library, and their difference) whose boolean flags are true. The function then returns and the program is terminated.

After making these changes, I realized that my `Pow` function was also unnecessary, as I only used it twice and it was for raising x to the power of two, which I can simply do by multiplying x by itself.