Design of mathlib

Moore Macauley
University of California, Santa Cruz

October 6, 2022

1 Goals

Implement sin, cos, arcsin, arccos, arctan, and log without the use of any math.h functions, and test them against math.h functions. Sin, cos, and arcsin must be implemented using the taylor series, arccos and arctan can be derived from arcsin, and log must be implemented with Newton's method. The epsilon should be set to 10^{-10} for all of these functions.

2 makefile for mathlib-test

I intend to keep the Makefile the same. As of now, when run with no arguments or with all, it will compile mathlib-test.c, which is all it needs to do. When the clean argument is passed, it will remove the object and compiled c files, which are the only files I plan to make during runtime. As such, the file fits my needs as is, so I see no need to change it.

3 mathlib-test Design Overview and Pseudocode

For mathlib-test I will need to compare my functions to math.h, so I will start by including it. I will then need to parse command line prompts used for testing. To do this, I will use the argc and argv arguments along with getopt() to iterate over all of the arguments. When this happens, I will use a switch to determine what case has been imputed. When a case has been imputed, I will save it into the corrisponding location in an array containing all of the commands. By assigning each of the arguments their own location in the array, I ensure that when an argument is called multiple times, the tests will only be run once. I will then iterate over this command array in a for loop, which contains a switch that runs the tests. Once again, to ensure that all tests only run once, if one of the arguments was a, the maximum value of the for loop will be set to 1, ensuring that it only runs once.

When the switch is used and I have arrived at the necessary test to be run, I will start by printing the header containing title, function being tested, library, and difference, followed by another line full of '-'s, as is shown in the example's first five lines of tests being run. I will then construct a for loop, starting at the initial range for the requested process, ending at the end of the range for the function, and stepping based on the required step. All of the break commands inside the switch will be inside an if statement that tests the condition of fall through. If it is true, then a has been called, which means all of the tests should be run. I will achieve this simply by allowing the program to fall through all of the switch statements until reaching default. This is why this for loop must only be run once if a is called.

Pseudocode:

```
Include math.h
Include mathlib.h
Include unistd.h for command parsing
Get arguments argc and argv
Create the array commands
Create variable number of args, and set it to 7

Iterate over argv with a while loop and using getopt, saving the argument into the variable arg
    Switch arg
    If case is a
        Save a into commands[0]
        Break
    If case is s
        Save s into commands[1]
        Increace number of args by 1
        Break
    If case is c
        Save c into commands[2]
        Increace number of args by 1
        Break
    If case is S
        Save S into commands[3]
        Break
    If case is C
        Save C into commands[4]
        Break
    If case is T
        Save T into commands[5]
        Break
    If case is l
        Save l into commands[6]
        Break

If commands[0] is a
    Set number of args to 1

Iterate over commands with a for loop, with number of args as the max value

    Save the value of the current position of commands to current argument.

    Switch current argument
    If case is a
        Set fall through to true
```

```
If case is s
     Print the header
     For an initial value of i = 0.05pi, a maximum value of under 2pi,
     and a step of 0.05pi
        print i, my sin(x) calculation where x = i, math.h's sin value,
        and the difference between the two in the given format
    If fall through is false
        Break

If case is c
    Print the header
    For an initial value of i = 0.05pi, a maximum value of under 2pi,
    and a step of 0.05pi
        Print i, my cos(x) calculation for when x = i, math.h's cos value,
        and the difference between the two in the given format
    If fall through is false
        Break

If case is S
    Print the header
    For an initial value of i = -1, a maximum value of under 1,
    and a step of 0.05
        Print i, my arcsin(x) calculation where x = i, math.h's arcsin
        value, and the difference between the two in the given format
    If fall through is false
        Break

If case is C
    Print the header
    For an initial value of i = -1, a maximum value of under 1,
    and a step of 0.05
        Print i, my arccos(x) calculation for when x = 1, math.h's arccos
        value, and the difference between the two in the given format
    If fall through is false
        Break

If case is T
    Print the header
    For an initial value of 1, a maximum value of under 10,
    and a step of 0.05
        Print i, my arctan(x) calculation for where x = i, math.h's arctan
        value, and the difference between the two in the given format
    If fall through is false
        Break
```

```
If case is l
    Print the header
    For an initial value of 1, a maximum value of under 10,
    and a step of 0.05
        Print i, my log(x) calculation when x = i, math.h's log value,
        and the difference between the two in the given format
     If fall through is false
        Break

If case is default
    Break
```

4 mathlib Design Overview and Pseudocode

Above all of these functions, mathlib.h is included.

4.1 Log

To compute log, I will use Newton's method. To do this, the value to be logarithmized will be passed in as an argument and saved as y. Then, the variables x and exit case will be defined as 1. From here, we iterate in a while loop until exit case is less than epsilon. Using the provided Exp(a) function, which calculates e^a, I will calculate

$$x_i = x_{i-1} + \frac{y - e^x_{i-1}}{e^x_{i-1}}$$

which determines the next x value in Newton's method. Then, I save exit case as $e^{currentx} - y$. If this value is less than epsilon, that means that there was a sufficiently small difference between the previous x value and the current one. As such, once this happens, the while loop ends, and we return x.

Pseudocode:

```
Get the value to be logarithmized as argument y
Set variables x and exit case to 1
While exit case is greater than epsilon
    x += ((y - e^x)/(e^x))
    exit case = e^x - y
Return x
```

4.2 Sin

For sin(x), the taylor series is

$$sin(x) = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!}$$

Based on this, we can calculate that the absolute value of the nth term in the taylor sequence is equal to

$$|a_n| = \prod_{k=1}^{2n+1} \frac{x}{k}$$

Thanks to this additional equation, if we know the value for $|a_{n-1}|$, we can calculate $|a_n|$ via

$$|a_n| = \frac{x}{2n} \times \frac{x}{2n+1} \times |a_{n-1}|$$

Then, to get from $|a_n|$ to a_n, all we need do is multiply it by $(-1)^n$ Therefore, to calculate sin(x) with the Taylor sequence, we should start by getting the argument x, before creating the variables current term, previous term, n, and total value. Previous term and total value should be set to x, as the 0th term of the Taylor sequence is x, while n should be set to 0. Next, we start a do-while loop, with the loop ending if current term is less than epsilon. Inside the loop, we first increment n, before multiplying previous term by $\frac{x}{2n}$ and $\frac{x}{2n+1}$ and saving that value to current term. We then determine if n is odd. If it is, we subtract current term from final value. If it is not, we add current term to final value. We then set previous term equal to current term. Once the loop ends, sin(x) should have been approximated using the taylor sequence, and we simply return final value.

While this strategy works well for positive numbers, it breaks down for negatives. Thankfully, however, sine is an odd function, which means that sin(-x) = -sin(x). As this is the case, all we need to do to find the sine of a negative number is to check to see if the input is negative, record that it is, then simply find the sine of the absolute value of that negative input. After that, once we reach the return statement, if we recorded that the input was negative, we return total value * -1.

```
Pseudocode:

Get the value to be plugged into sin as x
Set variable neg to 0
If x is less than 0
    Set neg to 1
    x *= -1

Set previous term and total value equal to x, and set n equal to 0.
Do:
    Increase n by 1
    Current value is equal to previous value multiplied by x/2n * x/(2n+1)
    If n mod 2 equals 1
        Subtract current value from total value
    Otherwise,
        Add current value to total value
    Set previous term to current term.
While current value is less than epsilon

If neg is true,
    Return total value * -1
Else return total value.
```

4.3 Cos

For cos(x), the taylor series is

$$cos(x) = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n}}{(2n)!}$$

Much the same as sin(x), we can then calculate that the absolute value of the nth term in the taylor sequence is equal to

$$|a_n| = \prod_{k=1}^{2n} \frac{x}{k}$$

Thanks to this additional equation, if we know the value for $|a_{n-1}|$, we can calculate $|a_n|$ just like sin(x) via

$$|a_n| = \frac{x}{2n-1} \times \frac{x}{2n} \times |a_{n-1}|$$

Then, to get from $|a_n|$ to a_n, all we need do is multiply it by $(-1)^n$ To calculate cos(x) with the taylor sequence, we follow much the same procedure as sin(x). We start by getting the argument x, before creating the variables current term, previous term, n, and total value. Previous term and total value should be set to 1, as the 0th term of the Taylor sequence is 1, the only substantial difference in procedure between calculating sin(x) and cos(x), while n should be set to 0. Next, we start a do-while loop, with the loop ending if the current value is less than epsilon. Inside the loop, we first increment n, before multiplying previous term by $\frac{x}{2n}$ and $\frac{x}{2n-1}$ and saving that value to current term. We then determine if n is odd. If it is, we subtract current term from final value. If it is not, we add current term to final value. N We then set previous term equal to current term. Once the loop ends, cos(x) should have been approximated using the taylor sequence, and we simply return final value.

While this strategy works well for positive numbers, it breaks down for negatives. Thankfully, however, cos is an even function, which means that cos(-x) = cos(x). As this is the case, all we need to do to find the cos of a negative number is find the cos of the absolute value of the input.

```
Pseudocode:

Get the value to be plugged into cos as x
If x is less than 0
    x *= -1

Set previous term and total value equal to 1, and set n equal to 0.
Do:
    Increase n by 1
    Current value is equal to previous value multiplied by x/(2n-1) * x/2n
    If n mod 2 equals 1
        Subtract current value from total value
    Otherwise,
        Add current value to total value
    Set previous term to current term.
While current value is less than epsilon
Return total value
```

4.4 Arcsin

For arcsin, now that we have already created functions for sin(x) and cos(x), arcsin(x) is relatively easy to calculate. We can use the iterative formula

$$z_n = z_{n-1} - \frac{sin(z_{n-1} - x)}{cos(z_{n-1})}$$

to calculate arcsin(x). In order to implement this via code, we should simply need to use a while loop that performs the above function until $|z_n - z_{n-1}|$ is less than epsilon.

```
Pseudocode:

Get argument x
create variable current term
create variable previous term and exit case, and set them to 1
while exit case is greater than epsilon:
    set sin of previous term to variable sin val
    set cos of previous term to variable cos vas
    set current term to previous term - (sin val - x) / cos val
    set exit case to current term - previous term
    if exit case is less than 0
        set exit case to exitcase * -1
    set previous term to current term

return current term
```

4.5 Arccos

Simply subtract my_arcsin(x) from $\frac{\pi}{2}$ to solve for arccos(x).

```
Pseudocode:

Calculate arcsin of x with the function
Return pi/2 - arcsin
```

4.6 Arctan

Just use the already completed my_arcsin and the provided square root function and use them to calculate arcsin($\frac{x}{\sqrt{x^2+1}}$) to create my_arctan(x).

```
Pseudocode:

Take the square root of x*x + 1
Divide x by that
Return the arcsin of the solution
```