Assignment 2
A Small Numerical Library

Prof. Darrell Long
CSE 13S – Spring 2021

Due: April 18th at 11:59 pm

1 Introduction

> *Let us change our traditional attitude to the construction of programs. Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.*
>
> —Donald Knuth

As we know, computers are simple machines that carry out a sequence of very simple steps, albeit very quickly. Unless you have a special-purpose processor, a computer can only compute *addition*, *subtraction*, *multiplication*, and *division*. If you think about it, you will see that the functions that might interest you when dealing with real or complex numbers can be built up from those four operations. We use many of these functions in nearly every program that we write, so we ought to understand how they are created.

If you recall from your Calculus class, with some conditions a function $f(x)$ can be represented by its Taylor series expansion near some point $f(a)$:

$$f(x) = f(a) + \sum_{k=1}^{\infty} \frac{f^{(k)}(a)}{k!}(x-a)^k.$$

Note: when you see Σ, you should generally think of a `for` loop.

If you have forgotten (or never taken) Calculus, do not despair. Attend a laboratory section for review: the concepts required for this assignment are just derivatives.

Since we cannot compute an infinite series, we must be content to calculate a finite number of terms. In general, the more terms that we compute, the more accurate our approximation. For example, if we expand to 10 terms we get:

$$\begin{aligned} f(x) = f(a) &+ \frac{f^{(1)}(a)}{1!}(x-a)^1 + \frac{f^{(2)}(a)}{2!}(x-a)^2 + \frac{f^{(3)}(a)}{3!}(x-a)^3 + \frac{f^{(4)}(a)}{4!}(x-a)^4 \\ &+ \frac{f^{(5)}(a)}{5!}(x-a)^5 + \frac{f^{(6)}(a)}{6!}(x-a)^6 + \frac{f^{(7)}(a)}{7!}(x-a)^7 + \frac{f^{(8)}(a)}{8!}(x-a)^8 \\ &+ \frac{f^{(9)}(a)}{9!}(x-a)^9 + O((x-a)^{10}). \end{aligned}$$

© 2021 Darrell Long

Note: $k! = k(k-1)(k-2) \times \ldots \times 1$, and by definition, $0! = 1$.

Taylor series, named after Brook Taylor, requires that we pick a point a where we will center the approximation. In the case $a = 0$, then it is called a *Maclaurin series*). Often we choose 0, but the closer to the value of x the better we will approximate the function. For example, let's consider e^x centered around 0:

$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!} = 1 + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \frac{x^6}{6!} + \frac{x^7}{7!} + \frac{x^8}{8!} + \frac{x^9}{9!} + \ldots$$

This is one of the simplest series when centered at 0, since $e^0 = 1$. Consider the general case:

$$\begin{aligned} e^x = e^a &+ \frac{e^a}{1!}(x-a)^1 + \frac{e^a}{2!}(x-a)^2 + \frac{e^a}{3!}(x-a)^3 + \frac{e^a}{4!}(x-a)^4 + \frac{e^a}{5!}(x-a)^5 \\ &+ \frac{e^a}{6!}(x-a)^6 + \frac{e^a}{7!}(x-a)^7 + \frac{e^a}{8!}(x-a)^8 + \frac{e^a}{9!}(x-a)^9 + \frac{e^a}{10!}(x-a)^{10} + O((x-a)^{11}). \end{aligned}$$

Since $\frac{d}{dx}e^x = e^x$ the exponential function does not drop out as it does for $a = 0$, leaving us with the original problem. If we knew e^a for $a \approx x$ then we could use a small number of terms. However, we do *not* know it and so we must use $a = 0$.

What is the $O((x-a)^{11})$ term? That is the *error term* that is "on the order of" the value in parentheses. This is different from the *big-O* that we will discuss with regard to algorithm analysis.

2 Your Task

> *Programming is one of the most difficult branches of applied mathematics; the poorer mathematicians had better remain pure mathematicians.*
>
> —Edsger Dijkstra

For this assignment, you will be creating a small numerical library and a corresponding *test harness*. Our goal is for you to have some idea of what must be done to implement functions that you use all the time.

You will be writing and implementing $\sin^{-1}$ (arcsin), $\cos^{-1}$ (arccos), $\tan^{-1}$ (arctan), and $\log$ functions. For $\arcsin$, $\arccos$, and $\arctan$ you can choose to use a Taylor series approximation or the inverse method. You will use Newton's method to approximate $\log$. You will then compare your implemented functions to the corresponding implementations in the standard library `<math.h>` with a test harness and output the results into a table similar to what is shown in Figures 1 and 2.

```
$ ./mathlib-test -s | head -n 5
  x             arcSin          Library        Difference
  -             ------          -------        ----------
 -1.0000       -1.57079633     -1.57079633    -0.0000000000
 -0.9000       -1.11976951     -1.11976951     0.0000000000
 -0.8000       -0.92729522     -0.92729522    -0.0000000000
```

Figure 1: First five lines of program output for $\arcsin$.

© 2021 Darrell Long

```
$ ./mathlib-test -c | head -n 5
  x            arcCos           Library          Difference
  -            ------           -------          ----------
 -1.0000       3.14159265       3.14159265      -0.0000000000
 -0.9000       2.69056584       2.69056584       0.0000000000
 -0.8000       2.49809154       2.49809154      -0.0000000000
```

Figure 2: First five lines of program output for arccos.

From left to right, the columns represent the input parameter, your program's $\arcsin$/$\arccos$ value for the input parameter, the library's value for the input parameter and lastly, the difference between your value and the library's value.

You will test $\arcsin$ and $\arccos$ in the range $[-1,1)$ with steps of 0.1, while $\arctan$ and $\log$ will be tested in the range $[1,10)$ with steps of 0.1.

Each implementation will be a *separate function*. You must name the functions `arcSin()`, `arcCos()`, `arcTan()`, and `Log()`. Since the `<math.h>` library uses `asin`, `acos`, `atan`, and `log`, you will not be able to use the same names. You may use the `sin()` and `cos()` functions from `<math.h>`. You may not use any other functions from `<math.h>` in any of your functions. You may only use them in your `printf()` functions. However, you should use constants such as `M_PI` from `<math.h>`. *Do not* define π yourself.

2.1 Sin^{-1} and Cos^{-1}

The *domain* of $\sin^{-1}$ and $\cos^{-1}$ is $[-1,1]$, and so centering them around 0 makes sense. The Taylor series for $\arcsin(x)$ centered about 0 is:

$$\arcsin(x) = \sum_{k=0}^{\infty} \frac{(2k)!}{2^{2k}(k!)^2} \frac{x^{2k+1}}{2k+1}, \quad |x| \leq 1.$$

If we expand a few terms, then we get:

$$\arcsin(x) = x + (\frac{1}{2})\frac{x^3}{3} + (\frac{1\times 3}{2\times 4})\frac{x^5}{5} + (\frac{1\times 3\times 5}{2\times 4\times 6})\frac{x^7}{7} + O(x^9).$$

The series for $\arccos(x)$ centered about 0 is:

$$\arccos(x) = \frac{\pi}{2} - \sum_{k=0}^{\infty} \frac{(2k)!}{2^{2k}(k!)^2} \frac{x^{2k+1}}{2k+1}.$$

So you can implement it as,

$$\arccos(x) = \frac{\pi}{2} - \arcsin(x).$$

You can implement the Taylor series expansion or the inverse method when writing the functions for $\arcsin$ and $\arccos$.

© 2021 Darrell Long

2.2 $\mathbf{Tan}^{-1}$

The Taylor series expansion for $\arctan$ is also complex and is as follows:

$$\arctan(x) = \sum_{k=0}^{\infty} \frac{2^{2k}(k!)^2}{(2k+1)!} \frac{x^{2k+1}}{(1+x^2)^{k+1}}.$$

To make things simpler, you can also calculate $\arctan$ using the $\arcsin$ or $\arccos$ functions

$$\arctan(x) = \arcsin\left(\frac{x}{\sqrt{x^2+1}}\right) = \arccos\left(\frac{1}{\sqrt{x^2+1}}\right), \quad x > 0.$$

You can implement the Taylor series expansion or the inverse method when writing $\arctan$.

2.3 e^x

Fortunately, we have a nice series for e^x and it happens to converge very quickly. In Figure 3, we use our expansion to 10 terms and plot for $e^0, \ldots, e^{10}$. We see that the approximation starts to diverge significantly around $x = 7$. What this tells us is that 10 terms are insufficient for an accurate approximation, and more terms are needed.

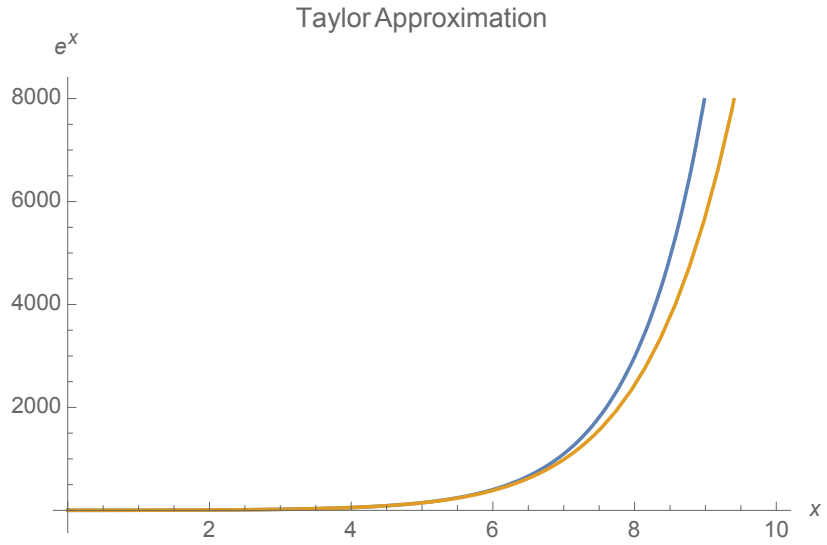

Figure 3: Comparing e^x with its Taylor approximation centered at zero.

If we are naïve about computing the terms of the series we can quickly get into trouble — the values of $k!$ get large *very quickly*. We can do better if we observe that:

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

At first, that looks like a recursive definition (and in fact, you could write it that way, but it would be wasteful). As we progress through the computation, assume that we know the previous result. We then just have to compute the next term and multiply it by the previous term. At each step we just need to compute

© 2021 Darrell Long

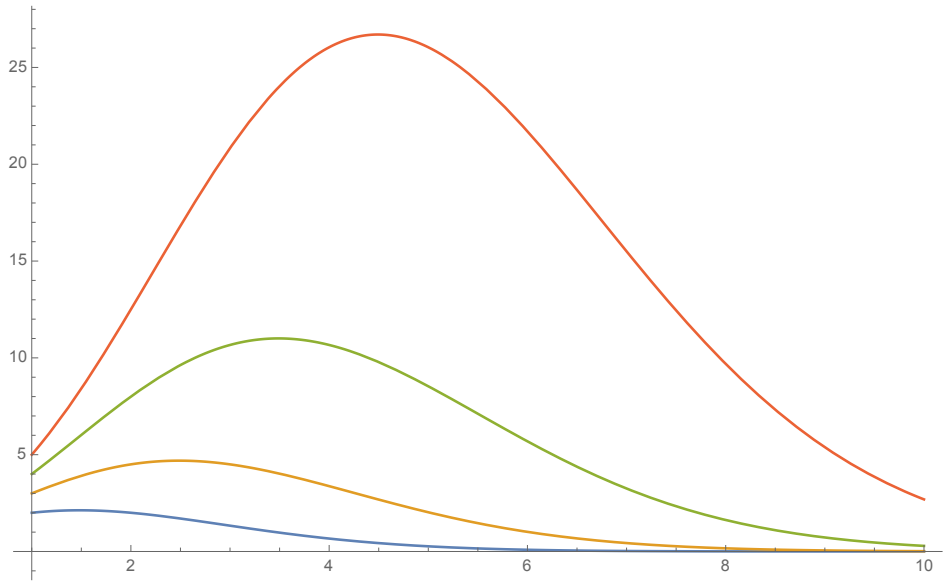

Figure 4: Comparing $\frac{x^k}{k!}$ for $x = 2, 3, 4, 5$.

$\frac{x}{k}$, starting with $k = 0!$ (remember $0! = 1$) and multiply it by the previous value and add it into the total. It turns into a simple `for` or `while` loop.

Conceptually, what you need to think about is:

```
new = previous * current;
previous = current;
```

We can use an ϵ (epsilon) to halt the computation since $|x^k| < k!$ for a sufficiently large k. Consider Figure 4: briefly, x^k dominates but is quickly overwhelmed by $k!$ and so the ratio rapidly approaches zero. You should set $\epsilon = 10^{-10}$ for this assignment.

For this assignment, the Exp(x) function is provided to you and you must use it in order to write your Log function.

2.4 Log

To compute $\log$, you will use Newton's method, also called the Newton-Raphson method, by computing the inverse of e^x. It is an iterative algorithm to approximate roots of real-valued functions, *i.e.*, solving $f(x) = 0$. Each iteration of Newton's method produces successively better approximations.

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

Your function `Log()` should behave the same as `log()` from `<math.h>`: compute $\ln(x)$. Here we are

© 2021 Darrell Long

finding the root of $f(x) = y - e^x$. Since e^x is the inverse of $\ln$, *i.e.* $\ln(e^y) = x$. This gives us:

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

Each guess x_{k+1} gives a successive improvement over the previous guess x_k. The function begins with an initial guess $x_0 = 1.0$ that it uses to compute better approximations. `Log()` is sufficiently calculated once the value converges, *i.e.* the difference between consecutive approximations is sufficiently small. This occurs when when $e^{x_i} - y$ is small.

In order to implement this function, you will have to use the given `Exp()` function. Using the `exp()` and `pow()` functions from `<math.h>` is strictly prohibited.

3 Command-line Options

> *GUIs tend to impose a large overhead on every single piece of software, even the smallest, and this overhead completely changes the programming environment. Small utility programs are no longer worth writing. Their functions, instead, tend to get swallowed up into omnibus software packages.*
>
> —Neal Stephenson, *In the Beginning…Was the Command Line*

Your test harness will determine which implemented functions to run through the use of *command-line options*. In most **C** programs, the `main()` function has two parameters: `int argc` and `char **argv`. A command, such as `./hello arg1 arg2`, is split into an array of strings referred to as arguments. The parameter `argv` is this array of strings. The parameter `argc` serves as the argument counter, the value in which is the number of arguments supplied. Try this code, and make sure that you understand it:

Trying out command-line arguments.

```
1 #include <stdio.h>
2
3 int main(int argc, char **argv) {
4     for (int i = 0; i < argc; i += 1) {
5         printf("argv[%d] = %s\n", i, argv[i]);
6     }
7     return 0;
8 }
```

A command-line option is an argument, usually prefixed with a hyphen, that modifies the behavior of a command or program. They are typically parsed using the `getopt()` function. *Do not* attempt to parse the command-line arguments yourself. Instead, use the `getopt()` function. Command-line options must be defined in order for `getopt()` to parse them. These options are defined in a string, where each character in the string corresponds to an option character that can be specified on the on the command-line. Upon running the executable, `getopt()` scans through the command-line arguments, checking for option characters.

© 2021 Darrell Long

Parsing options with `getopt()`.

```c
#include <stdio.h>
#include <unistd.h> // For getopt().

#define OPTIONS "pi:"

int main(int argc, char **argv) {
    int opt = 0;
    while ((opt = getopt(argc, argv, OPTIONS)) != -1) {
        switch (opt) {
        case 'p':
            printf("-p option.\n");
            break;
        case 'i':
            printf("-i option: %s is parameter.\n", optarg);
            break;
        }
    }
    return 0;
}
```

This example program supports two command-line options, '`p`' and '`i`'. Note that the option character '`i`' in the defined option string `OPTIONS` has a colon following it. The colon signifies that, when the '`i`' option is enabled on the command-line using '`-i`', `getopt()` is looking for an parameter to be supplied following it. An error is thrown by `getopt()` if an argument for a flag requiring one is not supplied.

4 Deliverables

> *Thinking doesn't guarantee that we won't make mistakes. But not thinking guarantees that we will.*
>
> —Leslie Lamport

You will need to turn in:

1. `mathlib.h`: You will be supplied this file and you *are not* allowed to modify it. The function prototypes for the math functions you are required to implement are as follows:
    - `double arcSin(double x);`
    - `double arcCos(double x);`
    - `double arcTan(double x);`
    - `double Log(double x);`

© 2021 Darrell Long

2. `mathlib.c`: This file will contain your math function implementations, as prototyped in `mathlib.h`. Your functions *must not* print or have any side effects.

3. `mathlib-test.c`: This file will contain the `main()` program and acts as a test harness for your math library. The code to parse command-line options and print out the results of testing your math functions belong here. The `getopt()` options that you *must* support are:

   - `-a`: to run all tests.
   - `-s`: to run $\arcsin$ tests.
   - `-c`: to run $\arccos$ tests.
   - `-t`: to run $\arctan$ tests.
   - `-l`: to run $\log$ tests.

   Note that these options are *not* mutually exclusive. You should be able to support any combination of these options. Each function is tested at most once. Specifying all tests to be run and a specific test does not result in that test running twice (*e.g.* `./mathlib-test -a -s`). The output for each function must match the format shown in Figures 1 and 2. Your compiled program must be called `mathlib-test`. To aid you with the print formatting, the print statement is given as follows:

   ```
   1 printf(" %7.4lf % 16.8lf % 16.8lf % 16.10lf\n", ...);
   ```

   The spaces in the print format statement are *intentional* and account for negative (but not positive) signs.

4. `Makefile`: This is a file that will allow the grader to type `make` to compile your program. Running `make` or `make all` must build your `mathlib-test` program. Running `make clean` must remove any compiler-generated files.

5. `README.md`: This must be in *Markdown*. This must describe how to build and run your program and list the command-line options it accepts and what they do.

6. `DESIGN.pdf`: This *must* be in PDF. The design document should contain answers to the pre-lab questions. It should also describe the purpose of your program and communicate the overall design of the program with enough detail such that a sufficiently knowledgeable programmer would be able to replicate your implementation. This does not mean copying your entire program in verbatim. You should instead describe how your program works with supporting pseudocode. **C** code is **not** considered pseudocode. You *must* push `DESIGN.pdf` before you push *any* code.

7. `WRITEUP.pdf`: This *must* be in PDF. `WRITEUP.pdf` should contain a discussion of the results for your tests. This means analyzing the differences in the output of your implementations versus those in the `<math.h>` library. Include possible reasons for the differences between your implementation and the standard library's.

   Graphs can be especially useful in showing the differences and backing up your arguments. In this document, graphs were effectively used to show the divergence of the Taylor series approximation

for e^x as seen in Figure 3. A visual representation of data makes it easier to get your point across. For this assignment, a plot can be used to show how close your approximation is to the value returned by the library function. An example script for using `gnuplot` to create your graphs has been provided to you in the `Resources` repository. `gnuplot` is a command-line graphing utility that allows you to visualize data easily and with a few commands you can make your writeup very pretty!

5 Submission

> *A man who procrastinates in his choosing will inevitably have his choice made for him by circumstance.*
>
> —Hunter S. Thompson, *The Proud Highway: Saga of a Desperate Southern Gentleman*, 1955–1967

To submit your assignment, refer back to `asgn0` for the steps on how to submit your assignment through `git`. Remember: *add, commit,* and *push*!

Your assignment is turned in *only* after you have pushed. If you forget to push, you have not turned in your assignment and you will get a *zero*. "I forgot to push" is not a valid excuse. It is *highly* recommended to commit and push your changes *often*.

6 Supplemental Readings

- *The C Programming Language* by Kernighan & Ritchie
  - Chapter 3 §3.4-3.7
  - Chapter 4 §4.1 & 4.2 & 4.5
  - Chapter 7 §7.2
  - Appendix B §B4
- *The Kollected Kode Vicious* by George V. Neville-Neil
  - Chapter 2 §2.6

© 2021 Darrell Long

C में प्रोग्रामिंग करना एक बंदर को चेनसॉ देने जैसा है।

© 2021 Darrell Long