Assignment 2
A Small Numerical Library

Prof. Darrell Long
CSE 13S – Winter 2021

Due: January 24th 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. Go to 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:

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

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:

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

Since $\frac{d}{dx}e^x = e^x$ the exponential function does not drop out as it does for $a = 0$, leaving us with our 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, cos, tan, e^x, and log. You will use Taylor series approximation for sin, cos, tan, and e^x, and 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 show in Figures 1 and 2.

```
$ ./mathlib-test -s | head -n 5
  x            Sin              Library        Difference
  -            ---              -------        ----------
 -6.2832      -0.00000000       0.00000000    -0.0000000000
 -6.1832       0.09983342       0.09983342     0.0000000000
 -6.0832       0.19866933       0.19866933    -0.0000000000
```

Figure 1: First five lines of program output for sin.

```
$ ./mathlib-test -c | head -n 5
  x            Cos              Library        Difference
  -            ---              -------        ----------
 -6.2832       1.00000000       1.00000000    -0.0000000000
 -6.1832       0.99500417       0.99500417    -0.0000000000
 -6.0832       0.98006658       0.98006658     0.0000000000
```

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

From left to right, the columns represent the input number, your program's cosine value from the input number, the actual math library's value from the input number and lastly, the difference between your value and the library's value.

You will test sin and cos in the range $[-2\pi, 2\pi)$ with steps of 0.1, while tan will be tested in the range $[-\pi/3, \pi/3)$ with steps of 0.1. e^x and log will be tested in the range $[1, 10)$ with steps of 0.1. Since sin and cos are valid over the real numbers, you need to accept any valid `double`—which means you will need to renormalize the input $-2\pi \leq x \leq 2\pi$. You also want to do this so that your Taylor series converges rapidly.

Each implementation will be a *separate function*. You must name the functions `Sin`, `Cos`, `Tan`, `Exp`, and `Log`. Since the `math` library uses `sin`, `cos`, `tan`, `exp`, and `log`, you will not be able to use the same names. You may not use any of the 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 π from `<math.h>`. *Do not* define π yourself.

2.1 Sine and Cosine

The *domain* of sin and cos is $[-2\pi, 2\pi)$, and so centering them around 0 makes sense. The Taylor series for $\sin(x)$ centered about 0 is:

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

If we expand a few terms, then we get:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \frac{x^{11}}{11!} + \frac{x^{13}}{13!} + O(x^{14}).$$

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

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

If we expand a few terms, then we get:

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \frac{x^{10}}{10!} + \frac{x^{12}}{12!} + O(x^{14}).$$

2.2 Tangent

Unlike sin and cos, tan does not have a simple Taylor series expansion:

$$\tan(x) = \sum_{n=1}^{\infty} \frac{(-1)^{n-1} 2^{2n} (2^{2n}-1) B_{2n}}{(2n)!} x^{2n-1}, \quad |x| < \frac{\pi}{2}$$

where B_n denotes the n^{th} Bernoulli number. Suffice to say that calculating Bernoulli numbers lies beyond the scope of this assignment. You will instead recall that

$$\tan(x) = \frac{\sin(x)}{\cos(x)}$$

and is undefined when $\cos(x) = 0$, that is, when x is a multiple of $\frac{\pi}{2}$. To implement tan, you will simply take the ratio of sin and cos.

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.

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 $\frac{x}{k}$, 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;
```

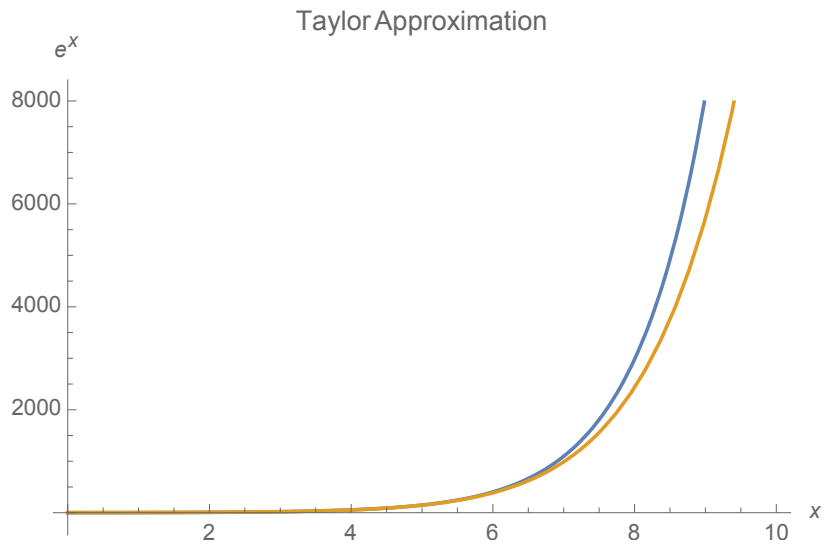

Figure 3: Comparing e^x with its Taylor approximation centered at zero.

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 $\epsilon = 10^{-14}$ for this assignment.

2.4 Log

To compute log, you will use Newton's method, also called the Newton-Raphson method. 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.

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

For example, consider computing *square roots* with Newton's method. That is, in order to solve for some $\sqrt{y}$, we are searching for a positive x such that $x^2 = y$. We can express this as finding the root of $f(x) = x^2 - y$, giving us:

$$x_{k+1} = \frac{1}{2}(x_k + \frac{y}{x_k}).$$

Each guess x_{k+1} gives a successive improvement over the previous guess x_k. The following is an example that implements Newton's method of computing square roots that doesn't conflict with `sqrt()` found in `<math.h>`. Note that the function is named `Sqrt()`. It begins with an initial guess $x_0 = 1.0$ that it uses to compute better approximations. The square root is said to be calculated once the value converges, *i.e.* the difference between consecutive approximations is small.

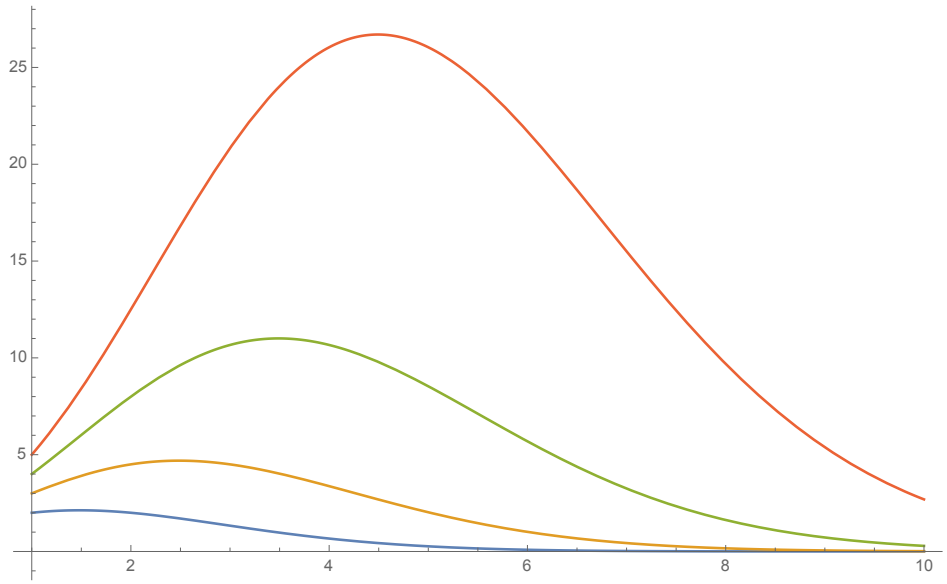

Figure 4: Comparing $\frac{x^k}{k!}$ for $x = 2, 3, 4, 5$.

Computing $\sqrt{x}$ using Newton's method.

```
#define EPSILON 1e-9

static inline double Abs(double x) {
  return x < 0 ? -x : x;
}

double Sqrt(double y) {
  double x_n = 1.0;
  double old = 0.0;
  while (Abs(x_n - old) > EPSILON) {
    old = x_n;
    x_n = 0.5 * (x_n + y / x_n);
  }
  return x_n;
}
```

Your function `Log()` should behave the same as `log()` from `<math.h>`: compute $\ln(x)$. The procedure is very much the same as it was for the square root example, the main difference being that $f(x) = y - e^x$, since e^x is the inverse of ln, *i.e.* $\ln(e^x) = x$. Another key difference is the value converges when $e^{x_i} - y$ is small, where x_i is initially 1.0 and is used to compute better approximations. In order to implement this function, you will have to use your `Exp()` function. Using the `exp()` function 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.

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 Sin(double x);`
    - `double Cos(double x);`
    - `double Tan(double x);`
    - `double Exp(double x);`
    - `double Log(double x);`

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 you *must* support are:

   - `-a`: to run all tests.
   - `-s`: to run sin tests.
   - `-c`: to run cos tests.
   - `-t`: to run tan tests.
   - `-e`: to run e^x 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 a 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 a 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.

5 Submission

> *We in science are spoiled by the success of mathematics. Mathematics is the study of problems so simple that they have good solutions.*
>
> —Whitfield Diffie

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 more that you read, the more things you will know. The more that you learn, the more places you'll go.*
>
> —Dr. Seuss

- *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