Assignment 2
A Small Numerical Library

Prof. Darrell Long
CSE 13S – Winter 2020

Due: January 26th at 11:59 pm

1 Introduction

Just look at the graceful way that he lectures, one hand waves while the other conjectures.

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 $f(a)$:

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

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:

$$\begin{aligned} f(x) = f(a) + (x-a)f'(a) + \frac{1}{2}(x-a)^2 f''(a) + \frac{1}{6}f^{(3)}(a)(x-a)^3 \\ + \frac{1}{24}f^{(4)}(a)(x-a)^4 + \frac{1}{120}f^{(5)}(a)(x-a)^5 + \frac{1}{720}f^{(6)}(a)(x-a)^6 \\ + \frac{f^{(7)}(a)(x-a)^7}{5040} + \frac{f^{(8)}(a)(x-a)^8}{40320} + \frac{f^{(9)}(a)(x-a)^9}{362880} + O\left((x-a)^{10}\right). \end{aligned}$$

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:

$$e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \frac{x^5}{120} + \frac{x^6}{720} + \frac{x^7}{5040} + \frac{x^8}{40320} + \frac{x^9}{362880} + O(x^{10}).$$

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

$$e^x = e^a + e^a(x-a) + \frac{1}{2}e^a(x-a)^2 + \frac{1}{6}e^a(x-a)^3 + \frac{1}{24}e^a(x-a)^4 + \frac{1}{120}e^a(x-a)^5 + \frac{1}{720}e^a(x-a)^6 + \frac{e^a(x-a)^7}{5040} + \frac{e^a(x-a)^8}{40320} + \frac{e^a(x-a)^9}{362880} + \frac{e^a(x-a)^{10}}{3628800} + O((x-a)^{11}).$$

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. Our goal is for you to have some idea of what must be done to implement functions that you use all of the time.

You will be writing and implementing `sin`, `cos`, `tan`, and `exp` using the Taylor series approximations for `exp` and Padé approximants for `sin`, `cos`, and `tan`. You will then run these functions and compare them to the standard library `math.h` implementations and output the results into a table similar to what is shown in Figures 1 and 2.

```
x           Sin          Library      Difference
-           ---          -------      ----------
-6.2832     0.00000000   0.00000000  -0.0000000000
-6.0868     0.19509032   0.19509032  -0.0000000000
```

Figure 1: Example of program output for sine.

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 sine and cosine from -2π to 2π with steps of $\pi/16$. Tangent will be tested from $-(\pi/2 - 0.001)$ to $\pi/2 - 0.001$ with steps of $\pi/16$. For the exponential function, e^x will be from 0 to 10 with steps of 0.1.

```
x         Cos          Library      Difference
-         ---          -------      ----------
-6.2832   1.00000000   1.00000000   0.0000000000
-6.0868   0.98078528   0.98078528   0.0000000000
```

Figure 2: Example of program output for cosine.

Each implementation will be a *separate function.* You must name the functions `Exp`, `Sin`, `Cos`, and `Tan`. Since the `math` library uses `exp`, `sin`, `cos`, and `tan`, you will not be able to use the same names. To use `math.h` in your program you must be sure to link the math library at compile time using the `-lm` flag.

The following is an example function 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(double x)`.

```
#define EPSILON 0.00001
static inline double abs(double x) { return x < 0 ? -x : x; }

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

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

2.1 Sine and Cosine

The *domain* of sin and cos is $[-2\pi, 2\pi]$, and so centering them around 0 makes sense. Since the domain is restricted, you should reduce any parameter to $[-2\pi, 2\pi]$ (making your approximation better). The Taylor series for $\sin(x)$ centered about 0 is:

$$\sin(x) = x - \frac{x^3}{6} + \frac{x^5}{120} - \frac{x^7}{5040} + \frac{x^9}{362880} - \frac{x^{11}}{39916800} + \frac{x^{13}}{6227020800} + O\left(x^{14}\right)$$

and the corresponding series for $\cos(x)$ is:

$$\cos(x) = 1 - \frac{x^2}{2} + \frac{x^4}{24} - \frac{x^6}{720} + \frac{x^8}{40320} - \frac{x^{10}}{3628800} + \frac{x^{12}}{479001600} + O\left(x^{14}\right).$$

We can use what is called a *Padé Approximant.* It's beyond the scope of this course to go into computing them, but essentially it is the ratio of two polynomials that conform to certain properties. It is often easier to compute and more accurate than a truncated series. The Padé approximant for a 14 term series for $\sin(x)$ centered around 0 is:

$$\sin(x) \approx \frac{-479249x^7 + 52785432x^5 - 1640635920x^3 + 11511339840x}{7\left(2623x^6 + 453960x^4 + 39702960x^2 + 1644477120\right)}.$$

It is a lot easier to square a number than to raise it to a power, so we can simplify it by putting the formula into *Horner normal form*, by factoring out x as much as possible:

$$\sin(x) \approx \frac{x\left(\left(x^2\left(52785432 - 479249x^2\right) - 1640635920\right)x^2 + 11511339840\right)}{\left(\left(18361x^2 + 3177720\right)x^2 + 277920720\right)x^2 + 11511339840}.$$

Why Horner normal form? It has the fewest multiplications.

If you want to be clever you can compute x^2 once, store it in a variable, using that result again and save a few instructions. Does this matter? A good compiler will recognize the common subexpression and do it for you behind the scenes, but numerical code tends to be heavily used so every little bit helps.

Consider the corresponding approximant for $\cos(x)$ centered around 0 written in Horner normal form:

$$\cos(x) \approx \frac{\left(x^2\left(1075032 - 14615x^2\right) - 18471600\right)x^2 + 39251520}{\left(\left(127x^2 + 16632\right)x^2 + 1154160\right)x^2 + 39251520}.$$

Restricting the domain is important. As we move away from the center of the series, our approximation gets worse. Consider Figure 3, and you can see that when we get much beyond $[2\pi, 2\pi]$ the graphs diverge not just a little, but wildly.

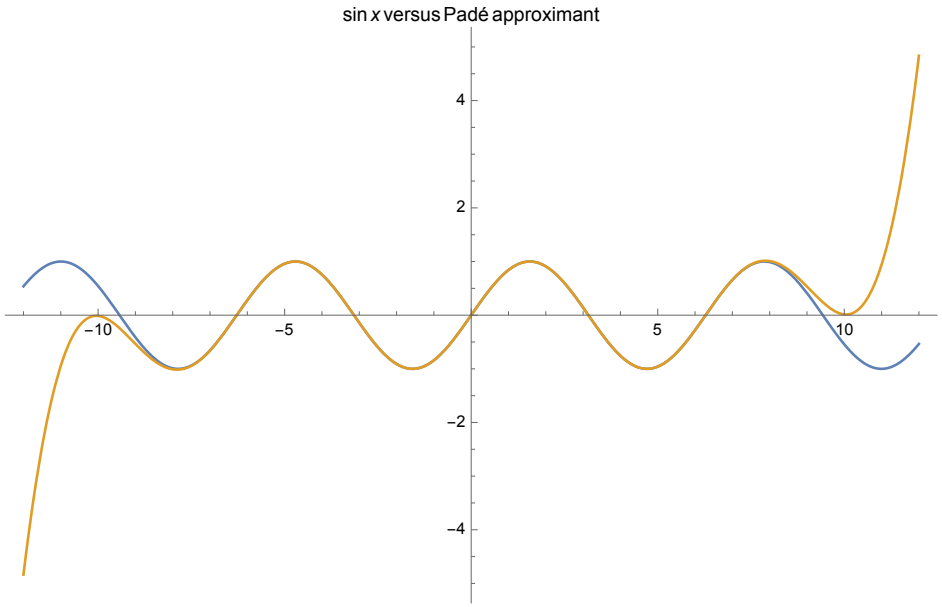

Figure 3: Comparing $\sin(x)$ with an order 10 Padé approximant.

2.2 Tangent

You will recall that $\tan(x) = \sin(x)/\cos(x)$ but that it is undefined when $\cos(x) = 0$, that is when x is a multiple of $\pi/2$. You *could* just do the division:

```
1 double tan(double x) { return sin(x) / cos(x); }
```

While doing the division is correct in the *real numbers*, remember that we are working with *approximations* and so you will be more accurate with a formula for directly computing your approximation for $\tan(x)$. A Padé approximant for $\tan(x)$ is:

$$\tan(x) \approx \frac{x\,(x^8 - 990x^6 + 135135x^4 - 4729725x^2 + 34459425)}{45\,(x^8 - 308x^6 + 21021x^4 - 360360x^2 + 765765)}.$$

You will want to rewrite it in Horner normal form before using it.

2.3 e^x

The other important function is e^x called the *exponential* function. While you might hope for a Padé approximant, you are bound for disappointment. The domain of e^x is unrestricted so we will *not* be able to use any fixed formula. As we get further away from where we center our series, the more terms that will need to get good results. Fortunately, the series for e^x converges very quickly.

Let's consider how well this works. In Figure 4, we will use our expansion to 10 terms and plot for $e^0, \ldots, e^{10}$. After about 7, the approximation starts to diverge significantly. What this tells us is that 10 terms is not enough.

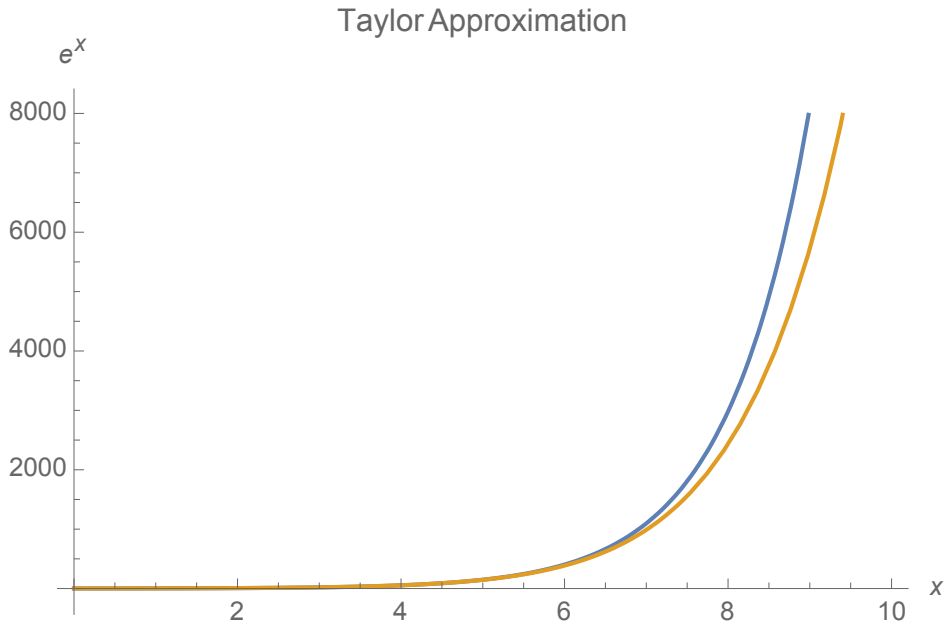

Figure 4: Comparing e^x with its Taylor approximation centered at zero.

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:

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

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

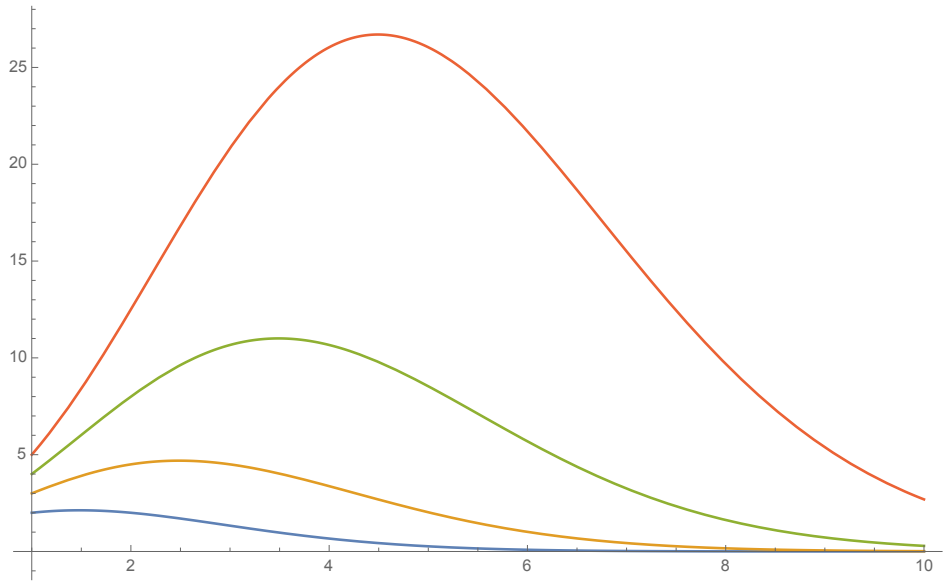

Figure 5: Comparing $x^n/n!$ for $x = 2,3,4,5$.

just have to compute the next term and multiply it by the previous term. At each step we just need to compute x/n, starting with $n = 0! = 1$ and multiply it by the previous value and add it into the total. It turns into a simple `for` or `while` loop.

We can use an ϵ (epsilon) to halt the computation since $|x^n| < n!$ for a sufficiently large n. Consider Figure 5: briefly, x^n dominates but is quickly overwhelmed by $n!$ and so the ratio rapidly approaches zero. For this assignment, $\epsilon = 10^{-9}$ will be sufficient.

Pre-lab Part 1

1. Write pseudo-code for approximating e^x with either a *for* or *while* loop.
2. Write pseudo-code for printing the output for e^x.

2.4 Command-line Arguments

Your program will use command-line arguments to select which one of your implemented functions to run. You will use `getopt()` to parse command-line arguments, which is included under `getopt.h`. In most **C** programs, you will see two parameters in the `main()` function: `int argc` and `char **argv`. The parameter `argc` refers to the argument counter, or how many arguments were supplied on the command-line. Arguments are delimited by white space, which includes spaces and tabs. The parameter `argv` refers to the argument values, and is interpreted as an array of strings, where `argv[0]` corresponds to the first argument on the command-line: the executable itself.

Try this code, and make sure that you understand it:

```
1 #include <stdio.h>
2
```

```c
int main(int argc, char **argv) {
  for (int i = 0; i < argc; i += 1) {
    printf("argv[%d] = %s\n", i, argv[i]);
  }
  return 0;
}
```

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.

As an example, the following 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 argument to be supplied following it. An error is thrown by `getopt()` if an argument for a flag requiring one is not supplied.

```c
#include <getopt.h>
#include <stdio.h>
#include <stdbool.h>

#define OPTIONS "pi:"

int main(int argc, char **argv) {
  int c = 0;
  bool print = false;
  char *infile = NULL;

  while ((c = getopt(argc, argv, OPTIONS)) != -1) {
    switch (c) {
    case 'p': // Print option.
      print = true;
      break;
    case 'i': // Input file option.
      infile = optarg;  // A pointer to the next element in argv.
      break;
    }
  }

  if (argc == 1) {
    puts("Error: no arguments supplied!");
    return -1;
  }

  if (!infile) {
    puts("Error: input file required!");
    return -1;
```

```
31    }
32
33    if (print) {
34      puts(infile);
35    }
36
37    return 0;
38  }
```

What this program does is check for the 'p' option to print and the 'i' option to specify some input file name. The `getopt()` defined variable `optarg` points at the following `argv`-element, which in this case should be the input file name. The condition `argc == 1` checks if no other command-line arguments were supplied other than the executable itself and errors out if so. The program also errors out in the event that an input file name was not supplied. The supplied input file name is only printed if the print option was specified on the command-line.

Example usage of program to specify input file and print its name (assume the executable name is `filename`):

```
1 ./filename -p -i <input file name>
```

Your program *must* support the following command-line options:

1. `-s` : runs `sin` tests
2. `-c` : runs `cos` tests
3. `-t` : runs `tan` tests
4. `-e` : runs `exp` tests
5. `-a` : runs *all* tests

These options are mutually exclusive; only one may be chosen at a time. Is a `bool` the best choice, or would an `enum` be better?

Pre-lab Part 2

1. What does `getopt()` return? Hint: check the man page.
2. Is a `bool` or an `enum` the best choice? Explain why.
3. Provide pseudo-code for your main function. Assume you have helper functions available to you.

3 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. `math.c`: This is your main program that determines whether `exp`, `sin`, `cos`, or `tan` will be run as well as contain your implementations. `getopt()` must be used to handle command-line arguments dictating which tests to run. The `getopt()` options you must support are '`-s`' to run `sin` tests, '`-c`' to run `cos` tests, '`-t`' to run `tan` tests, '`e`' to run `exp` tests, and '`-a`' to run *all* tests. Your output for each function must match the form shown in Figures 1 and 2. Your compiled program must be called `math`.

2. `Makefile`: This is a file that will allow the grader to type `make` to compile your program. Typing `make` must build your program and running `./math` alone as well as with option flags must run your program.

   - `CFLAGS=-Wall -Wextra -Werror -Wpedantic` must be included.
   - `CC=clang` must be specified.
   - `make clean` must remove all files that are compiler generated.
   - `make infer` must run `infer` on your program. Complaints generated by `infer` should either be fixed or explained in your `README`.
   - `make` should build your program, as should `make all`.

3. `README.md`: This must be in *markdown*. This must describe how to use your program and `Makefile`. This is also where you put any explanations for complaints generated by `infer`.

4. `DESIGN.pdf`: This *must* be a PDF. The design document should contain answers to the pre-lab in the beginning and describe your design for your program with enough detail 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 pseudo-code. For this assignment pay extra attention to how you describe your representation of a Taylor series approximation in **C**. You *must* push the DESIGN.pdf before you push *any* code.

5. `WRITEUP.pdf`: This *must* be a PDF. Your WRITEUP should be a discussion of the results for your experiments. 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 `math.h` version. Graphs can be especially useful in showing the differences and backing up your arguments.

4 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*.

5 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
  - Chapter 7 §7.2
  - Appendix B §B4