Jacob McClure
jatmcclu@ucsc.edu

math.c

I will be writing a C program that contains the Sine, Cosine, Tangent, and Exp functions. Once these functions have been implemented successfully, they will be tested and compared to the equivalent functions in the math.h library, and printed in a series of organized tables.

BEFORE I BEGIN:

Exponential function plans:

1. Pseudocode for Exp() using a for or while loop:

```
double Exp(double x) {
  // num = 1.0;
  // sum = num;
  // for (double k = 1.0; |num| > EPSILON; k += 1.0) {
    // num = x / k * num;
    // sum = sum + num;
  }
  return sum;
}
```

2. Pseudocode for printing Exp():

```
void printExp(void) {
  // print table heading and the corresponding dashed lines
  for (double i = 0.0; i <= 10.0; i += 0.1) {
    // print i, Exp(i), exp(i), (exp(i) - Exp(i));
  }
}
```

Part 2:

1. The getopt() function returns the option specified by the user in the command line (for this program, it is s, c, t, e, or a). We use it for the terminating condition of our loop that contains the switch statement because it returns a -1 when there are no more options to process – so it's easy in this case for us to "use getopt != -1" as the condition in our while loop.
2. Using enum would be a feasible, but less efficient way to implement this in our program. Because we are dealing with five options that are all chars, it is easier (and more efficient) to just use getopt() with a bool to set up a switch statement that handles the five different cases. The switch statement allows us to skip over

the cases that aren't relevant and waste less time processing code that won't be used (as opposed to an if, else if, else if, etc. configuration); for this reason, using the enum strategy would be less efficient and bool is better.

3. Pseudocode for my main() function:

```
// the following line will go at the very top of the program:
#define OPTIONS "sctea"
...
int main(int argc, char **argv) {
  int c = 0;
  // Make sure that the user enters arguments correctly
  if (argc != 2) {
    // throw error message
  }

  while ((c = getopt(argc, argv, OPTIONS)) != -1) {
    switch (c) {
    case 's':
      printSin();
      break;
    case 'c':
      printCos();
      break;
    case 't':
      printTan();
      break;
    case 'e':
      printExp();
      break;
    case 'a':
      // call all of the print[Op]() functions
      break;
    }
  }
  return 0;
```

Explanation of each function (and supporting pseudocode):

main():
Function `main()` will return 0 if arguments are entered and processed without errors, or it will return -1 if the argument count is incorrect/something goes wrong. The main() function will use getopt() to let the user choose which option they are going to use as an argument in the command line (following math – e.g. "./math -s"). I will use a switch statement to efficiently provide 5 different cases for the 5 different options a user has. Once the option has been chosen and the function calls are done, the switch statement will break and the program will end.

Sine:
I will write a Sine function that calculates the sine of all doubles in the range of -2π to 2π with steps of π/16. I will use what I have learned in calculus classes and also Professor Long's lecture on approximations to write Sin(double x). The return value of this function will be a double that is calculated using the Padé Approximant of sin(x) in *Horner Normal Form.* The pseudocode for this function is:

```
double Sin(double x) {
  // num = numerator for Padé Approximant of sin(x)
  // den = denominator for Padé Approximant of sin(x)
  // return num/den
}
```

Cosine:
I will write a Cosine function called Cos that calculates the cosine of all doubles in the range of -2π to 2π with steps of π/16. This will be very similar to the Sin function (logically), except the numerator and denominator for the Padé Approximant will obviously be different. Again, I will do this using my knowledge from past courses covering calculus and my lecture notes from when Professor long taught us about Padé Approximants. The pseudocode for this function is:

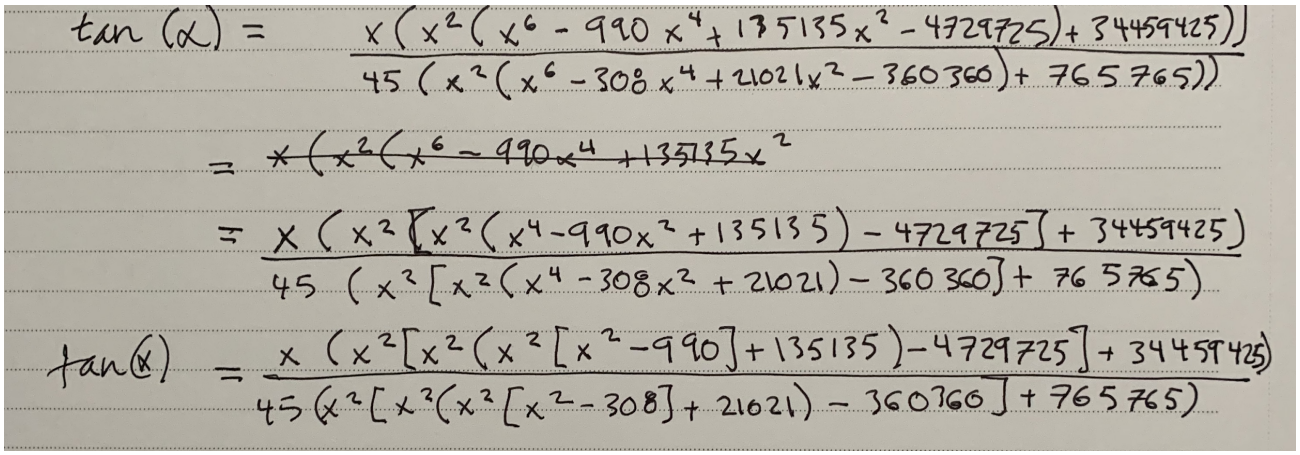
```
double Cos(double x) {
  // num = numerator for Padé Approximant of cos(x)
  // denom = denominator for Padé Approximant of cos(x)
  // return num/den
}
```

Tangent:

Next, I write a function that computes the tangent of all doubles in the range: −(π/2−0.001) to π/2−0.001 with steps of π/16. In order to do this, I will use the Padé Approximant formula given to us in the lab manual for this assignment. Before I can implement this function in my source code, I will have to put it in *Horner Normal Form* (see the work below my pseudocode for how I did the factorization). As with Sin() and Cos(), the numerator (num) will be the numerator from the *Horner Normal Form* of the Padé Approximant of tan(x), and the denominator (den) will be the denominator from the *Horner Normal Form* of the Padé Approximant of tan(x). The pseudocode for this function is:

```
double tan(double x) {
  // num = numerator for Padé Approximant of tan(x)
  // denom = denominator for Padé Approximant of tan(x)
  // return num/denom
}
```

Horner Normal Form for tangent derived in the following image:



Exp():

This function will essentially teach the computer how to calculate e^x (given a value for x) without using the math.h library's built-in exp() function. In order to do this, I will be using the example code that Professor Long gave us as a starting point. I will then improve upon it to minimize the differences in the output from my Exp() function when compared to the math.h library's exp() function. That is, I'll make it more and more precise until I have the difference at 0.00000.

printExp(), prinSin(), printCos(), printTan():

The next thing I will be doing is writing helper functions that can be called from main based on which command line arguments the user has entered. The options are: s, c, t,

e, and a; -s will call printSin(), -c will call printCos(), -t will call printTan(), -e will call printExp(), and -a will call all of the latter helper functions. Each of the helper functions will have the same exact structure, but will be calling on a different computational function, so I will simply show the pseudocode for two of them (since they're all going to be virtually identical). This will illustrate how the functions work and also how they are extremely similar.

Pseudocode for printSin() is:

```
void printSin(void) {
  // print table heading and the corresponding dashed lines
  // for (double i = -2.0 * π; i <= 2.0 * π; i += π/ 16.0) {
    // print i, Sin(i), sin(i), (sin(i) - Sin(i)));
  }
}
```

Pseudocode for printCos() is:

```
void printCos(void) {
  // print table heading and the dashed lines to correspond with
it
  // for (double i = -2.0 * π; i <= 2.0 * π; i += π/ 16.0) {
    // print i, Cos(i), cos(i), (cos(i) - Cos(i)));
  }
}
```

As you can see, the print[Op]() helper functions are all virtually the same, but with different function calls. Of course, there will be different bounds for the for loop in printTan() and printExp().