Karthi Sankar
kasankar@ucsc.edu
10/10/21

CSE 13S Assignment 2 Design Document

Purpose:

The purpose of this program is to write functions that approximate the values of e, pi, and the square root of a range of numbers, then compare our approximation to the values of C's math library. One of these functions approximates the value of the constant e, another will approximate the square root function, while all the other functions approximate the value of pi. Then, we will compare our calculated values with the math library's values through graphs and analyze them. We will analyze how well our function worked, how close they got, and what that tells us about approximations with floating point numbers.

Structure:

The structure of the program is a test harness file, mathlib-test.c, will be used to run and print the values of our functions versus the math-library's value. Within the file, we will define command-line options in order to run each individual function and the results. We will be able to run the function in the test harness because we include the file, mathlib.h, a header file containing all the function prototypes we will need for this assignment. The files we will create are e.c, euler.c, madhava.c, bbp.c, viete.c, and newton.c.

Description:

(all pseudocode is in semi-Python/English)
Note: Each file includes a term() function which returns a static variable counting the number of terms/iterations it took to get the desired approximation.

e.c:

This function takes no parameters and returns the summation that results in an approximation of e. For this code, I basically wanted to represent the summation as a

for loop, as I did for all of the other functions. So the first two terms in the summation are just 1/k as there is no factorial yet. Then for every term for k greater than 2, I save the last term in a variable called prev_term. Then the variable term is set to 1/k (the newest k being used here), and then multiplying the previous term by the newest term has the same effect as a factorial, accumulating the multiplication of all the k's in the denominator. Also, at each iteration, I add term to the sum, resulting in the final summation at the end. The for loop iterates until the latest term is less than epsilon (so it is essentially 0), and the sum is returned.

```
float e ():
   float term = 1.0
   float sum = term   #sum = 1
   for (float k = 1.0: abs(term) > EPSILON: k += 1.0):
      # continues until latest term is less than EPSILON
      if k is equal to 1 or 2:
         term = 1 / k
         add term to sum
      #end
      if k is greater than 2:
         set prev_term to term     #save the last term
         term = 1 / k
         term = prev_term mutiplied by term   #update latest term
         add term to sum
      #end
   #end
   return sum
#end
```

euler.c:

This function takes no parameters and returns the summation that results in an approximation of pi. For the code, I again use a for loop that iterates until the latest term is less than epsilon. For each iteration, because there is no need for the saving of previous terms, I just calculate 1/k^2 and add it to the sum. Then, in order to get pi (approximately), the sum must be multiplied by 6 and "square rooted" (using sqrt_newton created in newton.c). The sum that is returned is the approximation of pi.

```
float pi_euler():
    float term = 1.0
    float sum = 0.0
    for (float k = 1.0: abs(term) > EPSILON: k += 1.0):
        # continues until latest term is less than EPSILON
        term = 1 / (k * k)  #1 over k^2
        add term to sum
        #end
    multiply term by 6
    take the square root of the sum
    return sum
#end
```

madhava.c:

This function takes no parameters and returns the summation that results in an approximation of pi. I have variables for the term, sum, numerator, and denominator. The numerator and denominator are the parts of the fraction in each term. The for loop again iterates until the latest term is less than epsilon. When k is equal to one, the numerator of the first term is 1/-3. Then, with each future iteration, I save the last term in prev_numer, set numerator to (1/-3) (for each negative power of -3), and multiply the prev_numer and numerator together, which will accumulate the exponent of -3 in the numerator of each term. The denominator is the same with every iteration, so after the numerator is determined, each term is calculated with numerator and denominator and added to the sum. Finally, to get to pi, the sum must be multiplied by the square root of 12, and then returned.

```
float pi_madhava():
    float term = 0.0
    float sum = 1.0
    float numerator = 0.0
    float denominator = 0.0
    for (float k = 1.0: absolute(term) > EPSILON: k += 1.0):
        # continues until latest term is less than EPSILON
        if k is equal to 1:
            numerator = (1.0/-3.0)
        #end
        if k is greater than 1:
            set prev_numer to term    #save the last numerator
            numerator = (1.0/(-3.0))
            numerator = prev_numer mutiplied by numerator   #update latest numer.
        #end
        float denominator = (2*k) + 1
        term = numerator/denominator  # (-3)^(-k)/(2k+1)
        add term to sum
    #end
    mutiply sum by the square root of 12
    return sum
#end
```

bbp.c:

This function takes no parameters and returns the summation that results in an approximation of pi. I use variables for the term, sum, and sixteen_exp. The variable sixteen_exp represents the sixteen to the negative power at the beginning of each term. The large fractional part of the term is the same each time, so I use a numerator and denominator variable that will change values with each k. For sixteen_exp, if k is 0, it is 1, and if k is 1, it will be (1/16). Then, when k is greater than one, I save each prev_sixteen_exp, set sixteen_exp to (1/16), and multiply prev_sixteen_exp and sixteen_exp, updating the exponential value of each term with each iteration. Then each term is put together by multiplying the exponential part with the fraction, and the term is added to the sum. Finally the sum is returned.

```
float pi_bbp():
   float term = 0.0
   float sum = 1.0
   float sixteen_exp = 0.0   #exponential term separate from fraction
   # 16^(-k) is mutiplied to the large fraction
   for (float k = 0.0: absolute(term) > EPSILON: k += 1.0):
      # continues until latest term is less than EPSILON
      float numerator = (k * ((120.0 * k) + 151.0) + 47.0)
      float denominator = (k * (k * (k * ((512.0 * k) + 1024) + 712) + 194) + 15)
      if k is equal to 0:
         sixteen_exp = 1
      #end
      if k is equal to 1:
         sixteen_exp = (1.0/16.0)
      #end
      if k is greater than 1:
         set prev_sixteen_exp to sixteen_exp  #save the last term
         sixteen_exp = (1.0/16.0)
         sixteen_exp = prev_sixteen_exp mutiplied by sixteen_exp  #update latest term
      #end
      # put exp. term + fraction together (mutiply)
      term = sixteen_exp * (numerator/denominator)
      add term to sum
   #end
   return sum
#end
```

newton.c:

This function takes 1 parameter, x, and returns the approximated square root of the given parameter. For this code, I used the Python code of Professor Long as my pseudocode. The z represents the previous term, while y represents the current term (both used in the loop). The loop ends when the difference between the previous and current term is less than epsilon. First, z = y, which means the current term is stored as the previous term. Then the formula from the assignment is implemented, using both y, z, and the given parameter x, with the result being stored in y. As the loop iterates, the result will converge to the approximated square root of the given parameter. When the while loop ends, the variable y, being the approximation, is returned.

```python
def sqrt(x):
    z = 0.0
    y = 1.0
    while abs(y - z) > epsilon:
        z = y
        y = 0.5 * (z + x / z)
    return y
```

viete.c:

Note: the variable product should be one in the pseudocode

This function takes no parameters and returns the product of the series that results in an approximation of pi. The variables factor, product, and numerator are used. The numerator is updated with each iteration; at first, it is set to the square root of 2, the numerator of the first term, when k is equal to 1. The for loop iterates until the difference between one and the latest term is less than epsilon. If k is greater than one, the variable prev_numer is set to the numerator, numerator is set to 2 (which must be added with each iteration), and then, the numerator is set to the square root of the previous numerator plus the numerator (which is two). This serves to update the numerator with each iteration, adding two and taking the square root. Then, each factor is calculated by putting the numerator over 2. Then the factor is multiplied to the product, accumulating the factors of the series. Finally, 2.0/product is returned because the series itself results in 2/the approximation of pi.

```
float viete() :
   float factor = 0.0
   float product = 0.0
   float numerator = square root of 2      #initial term when k = 1
   for (double k = 1.0; (1 - absolute(factor)) > EPSILON; k += 1.0)
   # continues until difference between latest term and 1 is less than EPSILON
      if k is greater than 1:
         prev_numer is set to numerator   #save latest numerator
         numerator = 2                    #add 2 with every iteration
         numerator = square root of previous numerator + numerator
         # updated numer equals last numerator + 2
      #end
      factor = numerator / 2.0
      multiply product by factor  #accumulates the product
   #end
   return 2.0 / product
#end
```

mathlib-test.c:

This is the test harness file that is used to run all other files in the assignment. It functions by using a switch statement within the a while loop, inside the main function. The switch statement checks the input of the user, determining which command line option is entered in, which will determine what to print. At the beginning of the file, I initialize booleans for each of the different command line options in the program. Then, within the switch statement, whenever each character is encountered, I set the corresponding boolean to true. Then after the switch and while loops, I use an if statement for each boolean (for each character): if the boolean is true, print my function's approximation, the math library's value (for e, pi, or square root of number), and the difference between them. There is also a command line option for a help menu and one for stats, which will print out the number of terms/iteration it took to get each separate approximation.