Assignment 2 - pi

Angela Shen

January 2023

1 Introduction

The purpose of this assignment is to implement multiple functions that approximate the value of pi or e and use a library to test their accuracy and how quickly they converge.

The following pseudo code is a bit repetitive.

2 Terms

Most of the c files for this assignment require a function that can return how many terms is required to calculate e or pi. It is a static int that should increase by one each time a new term is calculated (should be implemented within the function's while or for loop). It should also be set to 1 or 0 (depending on the function) so that its value "resets" and does not continually increase with each run. This reset can also be done within the corresponding _terms() function in each c file.

3 E

To calculate e, four doubles need to be initialized. Double k_fact represents the value of k! and is initialized to 1.0, double k represents the value of k (increases with each iteration) and is initialized to 1.0, double approx_e represents the estimate of e and is initialized to 1.0, and double prev_e represents the previous estimate of pi and should be initialized to 0.0. Static int terms represents the number of terms that have been used to calculate pi, and is defined outside the function and set to 1 inside the function (since approx_pi being equal to 1.0 already counts is the first term)

To calculate e, a while loop should be created, with the conditional being that the absolute value between the previous e and the current approximation of e is greater than epsilon. The while loop should set the value of approx_e to prev_e, since the while loop will change the value of approx_e to be different. Then, add to approx_e the value of 1/k_fact. After that, the next value of k and k factorial need to be calculated. To calculate the new k, add 1 to the current value of k. To calculate the new factorial, multiply the current value of k_fact by k. Increase the value of terms by one since a new term was calculated. After the while loop terminates, return the value of approx_e.

4 Madhava

To calculate pi using Madhava's formula, the variables required are an double k, a double num, a double approx_pi, and a double prev_pi. Double k should be initialized to 1.0, num (represents the numerator, 3^{-k}) should be initialized to 1.0, approx_pi should be initialized to 0.0, and prev_pi should be initialized to 1.0 (or any value not equal to approx_pi, since it should be overridden by the first line of the while loop). Since no values of pi have been calculated, static int terms should be set equal to 0.

A while loop should be created, with the conditional being that the absolute value between the previous pi and the current approximation of pi is greater than epsilon. The first line should set the value of approx_pi to prev_pi. Then, add to approx_pi 1/num multiplied by 1 over 2 times k plus 1, which is the implementation of Madhava's formula. After that, the next value of k should be increased by one. The next value of num should be its current value times -3. The number of terms should also increase by 1. When the while loop terminates, it should return the approximate value of pi over square root 12. Using the sqrt function from newton.c, the value of square root 12 can be calculated and should be multiplied by the value of approx_pi to calculate the value of pi.

5 Euler

For Euler's formula, the variables required are a double k, a double approx_pi,and a double prev_pi. Double k is to 1.0, the double approx_pi is initialized to 0.0, and prev_pi is initialized to 1.0. No terms of pi have been calculated yet, so terms is set equal to 0.

To calculate pi, a while loop with a condition that it terminates when the absolute value between previous pi and the current approximation of pi is less than epsilon. The first line should set the value of prev_pi as the current value of approx_pi, to store the current estimate of pi before the rest of the loop changes its value. Then, add to the current value of approx_pi 1 over k times k. Then increase the value of k and terms by one. When the loop terminates, approx_pi should approximate the value of pi squared over 6, so multiply the value of approx_pi by 6 and then square root it and return the final result.

6 BBP

For the Bailey-Borwein-Plouffe formula, the variables required are a double k, a double pow_sixteen (which represents 16^{-k}, a double approx_pi,and a double prev_pi. Double k should be initialized to 0.0, the double approx_pi should be initialized to 1.0, the double prev_pi should be initialized to 0.0, and pow_sixteen should be initialized to 1.0. Since no values of pi have been calculated, terms should be set to 0.

Create a while loop with a condition that it terminates when the absolute value between previous pi and the current approximation of pi is less than epsilon. The first line sets the value of prev_pi as the current value of approx_pi. Then, add to the current value of approx_pi 1 over pow_sixteen multiplied by all of the following: $\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6}$, with k being equal to the double k. While Horner normal form might minimize the number of terms, I think it would be fairly easy to make and not easily find a syntactical error in the implementation. Then increase the value of k by one and multiply the current value of pow_sixteen by sixteen. When the loop terminates, approx_pi should be equal to an estimate of pi, which can be returned.

7 Viète

To calculate pi using Viète's formula, the variables required are a double num, a double approx_pi, and a double prev_pi. The double num should be initialized to square root of 2, since it represents the numerator, double approx_pi should be initialized to 1.0, and prev_pi should be initialized to 0.0. Since no values of pi have been calculated, terms is set equal to 0.

To calculate pi, a while loop with a condition that it terminates when the absolute value between previous pi and the current approximation of pi is less than epsilon. The first line should set the value of prev_pi as the current value of approx_pi, to store the current estimate of pi before the rest of the loop changes its value. Then, multiply the current value of approx_pi by num over 2. To change the value of the numerator in accordance with the nested radicals, set num equal to the square root of two plus the current value of num. When the loop terminates, approx_pi should be equal to 2 over pi. Set the value of approx_pi equal to 1 / approx_pi and then multiply by two to derive pi and return the value.

8 Newton

To calculate sqrt with the Newton-Raphson formula, create a function that accepts a long double x. First, initialize an int f that is equal to 1. Since the value of x is unknown and could be a very large number, its value should be scaled down to decrease the amount of iterations sqrt has to perform. The while loop should have a conditional of x ¡ 4, so that it terminates or does not run when x is a sufficiently small number. The while loop should divide the current value of x by 4 and assign it to itself while multiplying the value of f by 2 each time. The int f acts as a scaling factor to multiply the estimation of the square root to scale it by the same amount x has been scaled down.

Next, actually implement the formula. Initialize two doubles y and z as 1.0 and 0.0 respectively, which will be used as guesses for calculating the square root. Create a while loop that terminates when the difference between the absolute value of y and z is less than epsilon. Assign the current value of y to z and then use y and x to implement the formula, $y = (y + (x/y))/2$. When the while loop terminates, return the value of y times the scale factor int f.

9 Mathlib-test

Including the header mathlib.h in the c files described above allows for functions to be used across multiple c files. This is important for using the Newton square formula in newton.c in other formulas to calculate square root values. It also allows the results of the other c files to be compared with constants from math.h in mathlib-test.c.

To implement mathlib-test.c, getopt() is used to parse the arguments. Integer values representing booleans are used to determine which tests to run. The OPTIONS for getopt should be defined as aebmrvnsh, which are all the options that . Using a switch, parse the arguments given by the user and set the correct integers to "1" to represent a true value. If a is called, all test values (ebmrvn) should be set to 1. If h is called or no test options are called, all test values should be set to 0 and the help() function, which displays the synopsis and usage for mathlib-test. If s is called, any other tests that are called should include the term count.

10 Credit and Notes

My sqrt_newton() function was not based directly on any c code, but rather portable.py in the resources repository. Notably, most of the formulas are missing graphs. I unfortunately could not produce suitable graphs before the due date. Epsilon is defined as 1e-14.