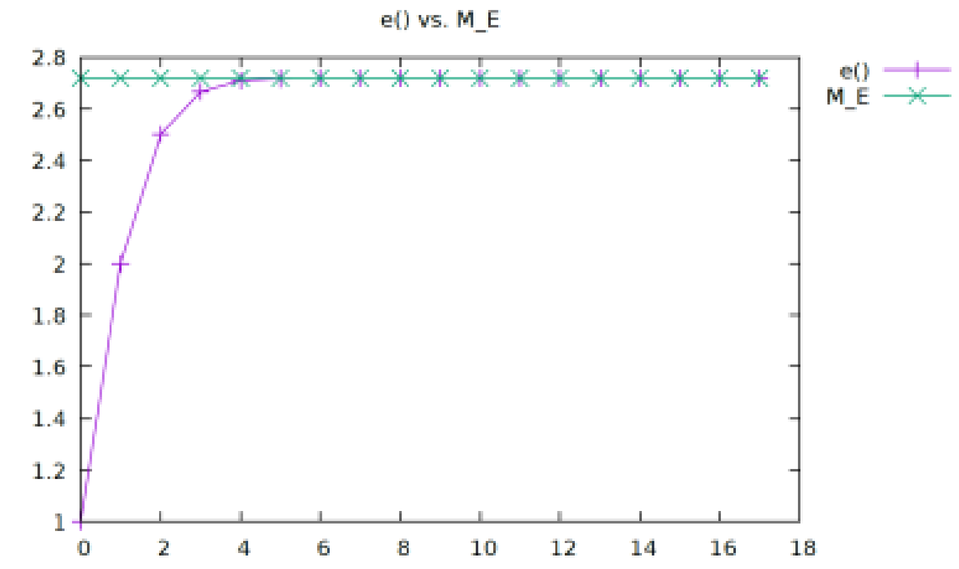
Dmitrius Agoston
Prof. Darrell Long
10 October 2021

CSE 13S Fall 2021
Assignment 2: A Little Slice of Pi
Writeup

The purpose of this writeup is to document and analyze my findings after comparing my code's results to the default <math.h> library.

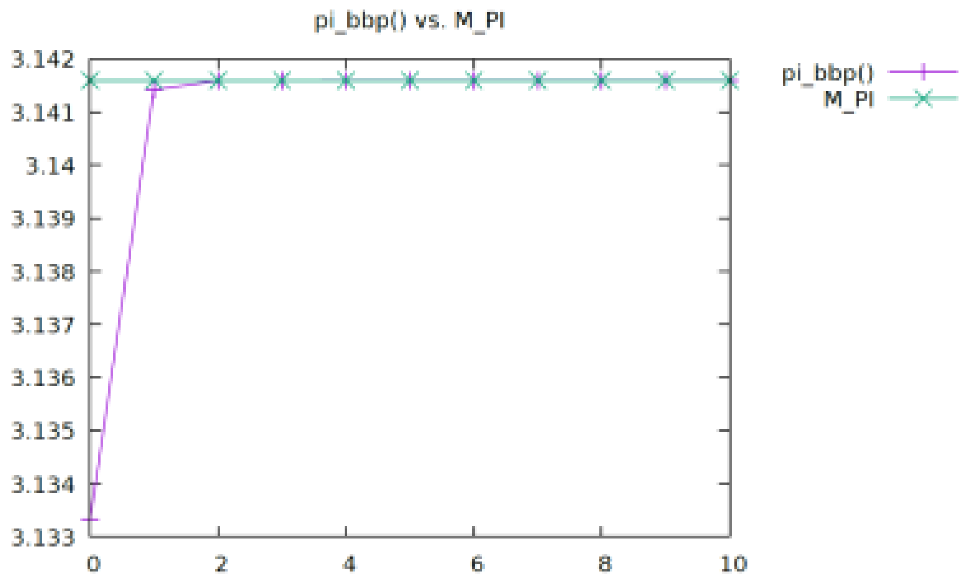
e():

The first function that I analyzed was the e() function. The purpose of e() was to return an approximation of the number e through the taylor series. After running the test harness for e() the value that was returned was 2.718281828459046. When compared to the default math library's value of e at 15 places after the decimal, there is no difference between the two numbers. The e function was able to accurately simulate the approximation of the value e and arrive at the same result as the math library. As seen in the graph the function e() ends up converging and arriving at the same value after 17 terms.



pi_bbp():

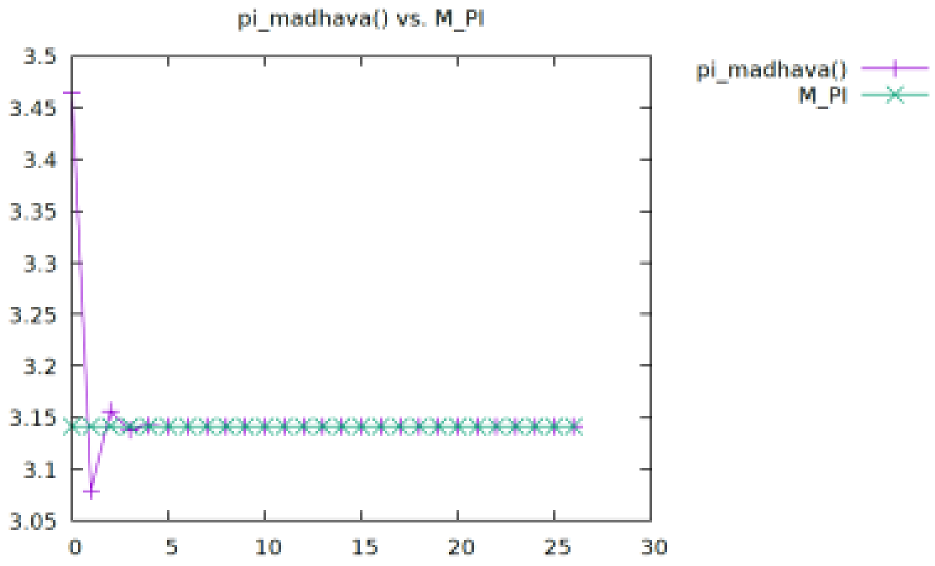
The next function analyzed was the pi_bbp() function. The purpose of pi_bbp() was to return an approximation of pi using the Bailey-Borwein-Plouffe formula. The value that was returned through the test harness for pi_bbp() was



3.141592653589793. Compared to the original value of pi used in the math library, it matches up perfectly with a difference of 0.0. The pi_bbp() function simulated the calculation of pi with great accuracy and ended up with the same approximation as the default math library. As seen in the graph, the function converges with the value of pi relatively fast and ends at the same result in 10 terms.

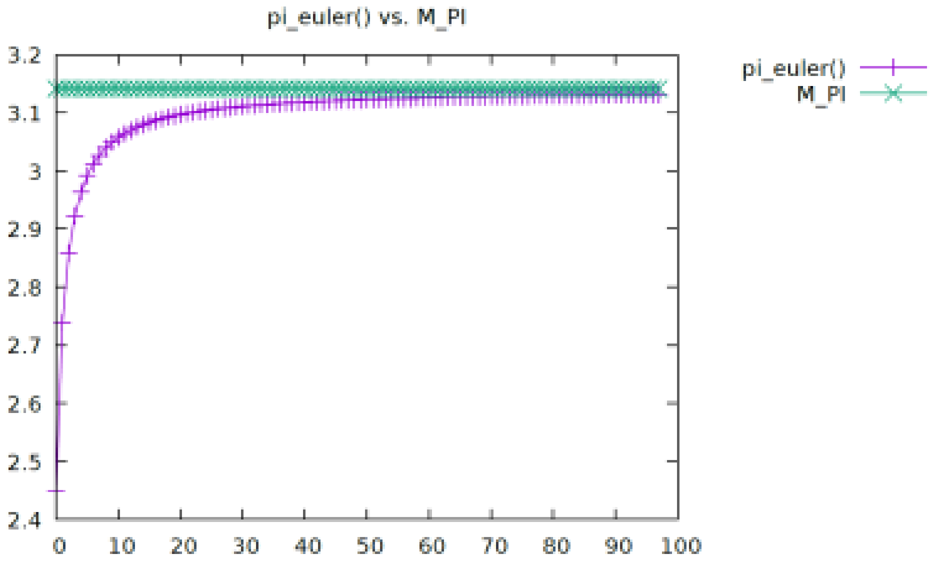
pi_madhava():

The next function that was analyzed was the pi_madhava() function. This function's purpose was to approximate the value of pi through a different method. pi_madhava() used the Madhava series to approximate its result which was 3.141592653589800. When compared to the math libraries value of pi, this approximation is slightly off. The difference between the two is 7.0x10^{-15}. The most likely reason for the difference in outputs would probably have to be because of a rounding error. After 26 terms the current approximation and the one prior were less than 1.0x10^{-14}, meaning the changes in the approximation were becoming menial, but still enough to be slightly different from the full desired value.



pi_euler():

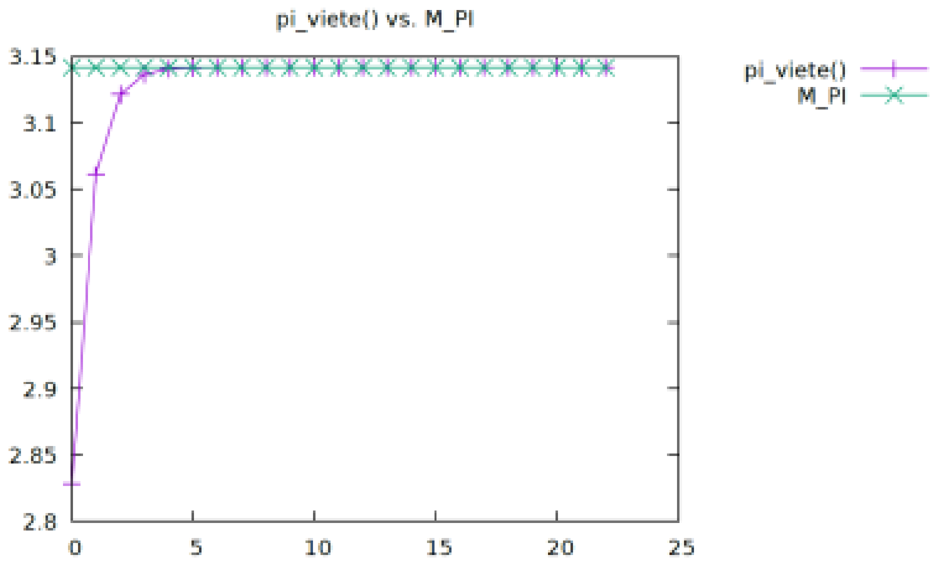
The pi_euler() function was analyzed next. The purpose of pi_euler() was to approximate the value of pi through Euler's solution. When the test was run, the function returned a value of 3.141592557608133. Comparing this number to the pi approximation that the math library uses, there is a



pretty noticeable difference. The exact difference between the two is 9.5981660x10^{-8}, which is much more significant when compared to pi_madhava()'s difference. A possible reason for the difference in outputs could be that the series treats the number pi as a limit and slowly gets closer to it, over numerous terms. To reach the number that it did, it already had to go through 9,948,879 terms before the margin of error between the last approximation and the one before was less than 10^{-14}. If given more terms to go through it might get closer, but the changes end up becoming negligible.

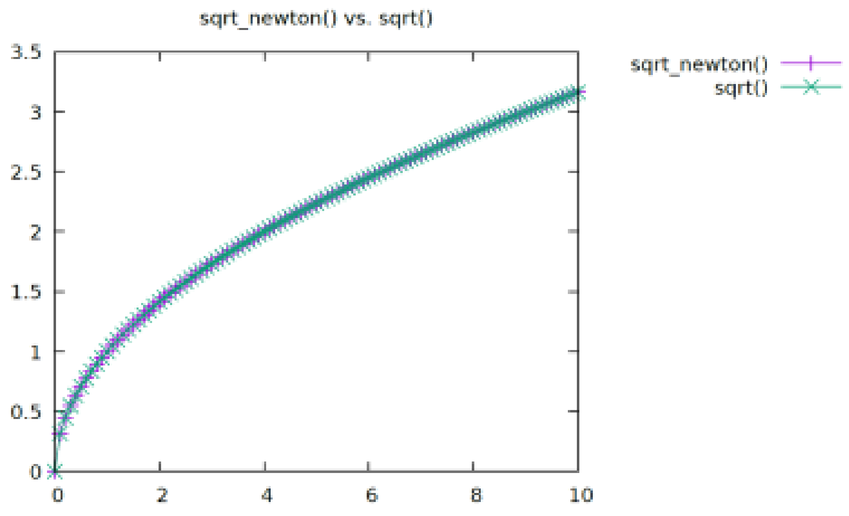
pi_viete():

The pi_viete() function was the last function tested that was meant to approximate pi. pi_viete() approximates the value of pi with Viete's formula. When analyzing the test harness results, pi_viete() returned a value of 3.141592653589789. When compared against the default value of pi, this number was not far off. The difference between the two was 4.0x10^{-15}. Much like pi_madhava, the difference is incredibly small between the two. The only reason I can think of this error occurring is from a rounding error, much like with pi_madhava(). Especially when analyzing the graph and seeing how quickly it started to converge. Also how it only took 23 terms to arrive at the approximation that it did.



sqrt_newton():

The last function tested in the harness was the sqrt_newton(). This function's purpose was to return the square root of a number given to it through the Newton-Raphson method. When given the values from 0.0 to 10.0 with a 0.1 step between each value, the return values matched those of the same



values square rooted with the default math library. Each value is able to be approximated with around 5-7 terms each, which is relatively quick compared to the other functions. The only outlier is the square root of 0 which returns $7.0\text{x}10^{-15}$, when the value should be 0. It also is important to mention that it takes 211 terms to reach this answer. I think that the reason for this is that the variable holding the approximation never becomes greater than 0 and after 211 iterations an error is thrown giving the odd result that is returned instead of 0 being returned.