CSC 290E
Pi Project
9-28-2010

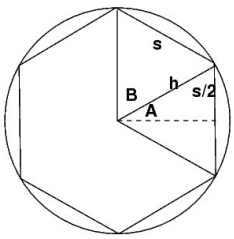
MATLAB approximating pi

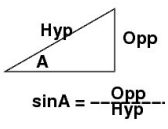
Introduction:

This project involved approximating the value of pi using MATLAB and testing to find out which method works best based on accuracy and processing speed. Four different approaches were used to approximate pi. The first was an Archimedes approximation, which involves approximating pi based on the perimeter of a regular N-gon, where each vertex is .5 units away from the center. Considering a regular N-gon becomes more circular as N approaches infinity, the perimeter of the N-gon essentially becomes the circumference of a circle. Circumference = 2*pi*rad, so by assuming rad = .5 we can approximate the value of pi. The next two approaches were the Leibniz and Wallis approximations. The Leibniz approximation is an infinite series of the form $\sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = \frac{\pi}{4}$. (Wikipedia.com). By multiplying the value of the series by four we can approximate the value of pi. The Wallis product states, $\frac{2}{\pi} = \prod_{n=1}^{\infty} \left(\frac{(2n-1)(2n+1)}{(2n)(2n)} \right)$ (midnightwiki.com). Two times the Wallis product yields an approximation of pi. The fourth approach involved a Monte Carlo Simulation of throwing darts at a board. Here we simulate random darts being thrown at a 1 x 1 square containing ¼ of a circle with radius one. We can calculate the distance of the random (x, y) coordinate of each dart to find whether it "stuck" inside or outside the circle. The ratio of darts inside the circle to total darts thrown approximates pi.

The Functions:

My first function, arch_pi, approximates pi using Archimedes' method. The function prompts the user for the number of sides of the N-gon. The function then uses trigonometry to calculate internal angles, side lengths, and finally the perimeter of the N-gon, approximating pi. Arch_pi also has the ability to take a 1 x N vector as input, and return a 1 x N vector of pi approximations.





(Archimedes approach according to the CSC290E blackboard page)

My functions for the Leibniz and Wallis approximations, leib_pi and wallis_pi respectively, have a different structure than arch_pi. Both leib_pi and wallis_pi

prompt the user for the number of terms (N), and calculates the value of the series to that number of terms. The calculation is achieved by using a 'for' loop to loop the general term over the range 1 to N.

The Monte Carlo approximation, monte_carlo_pi, has a unique algorithm. It prompts the user for the number of darts in the simulation, N. Variables x and y are assigned to random real numbers from 0 to 1. The function then uses a 'for' loop to run the Monte Carlo simulation using N darts and counts how many darts land in the circle. The ratio of darts in the circle to total darts is then used to approximate pi.

The function nests this first loop in another loop, which runs the simulation 1000 times. The approximated values of pi are assigned to one long vector. The vector is sorted, and the first 100 and last 100 terms are discarded to eliminate outlying values that will skew the results. The function returns the arithmetic mean of the remaining values. The function also calculates the standard deviation of the results. If the user assigns two variables to monte_carlo_pi(N), the first variable will return the approximated value pi and the second will return the standard deviation of the 800 middle trial values. The function also resets the seed of the random number stream based on the MATLAB clock (I found the code for this while navigating through 'help rand' and similar topics). By resetting the seed every time the function is open, it changes the sequence of random numbers generated.

How I Compared

To test and compare the functions, I wrote scripts to loop each function through a given range of input values, gather data such as error and CPU time, and plot the results. I compared Archimedes approximations of pi using polygons with 1 to 10^5 sides, Leibniz approximations with 1 to 10^5 terms, Wallis approximations with 1 to 10^5 terms, and Monte Carlo simulations with 1 to 10^4 darts.
I then wrote a script, pi_driver to launch all the plotting scripts.

Results

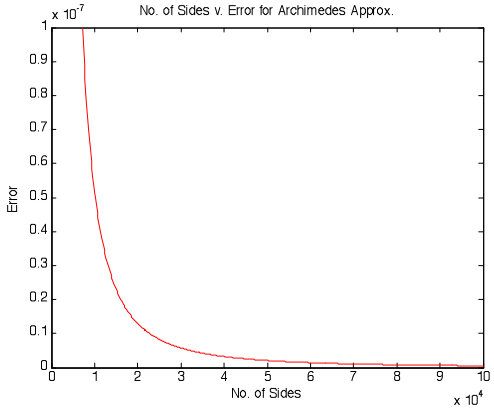


Figure 1, No. of Sides v. Error, Archimedes

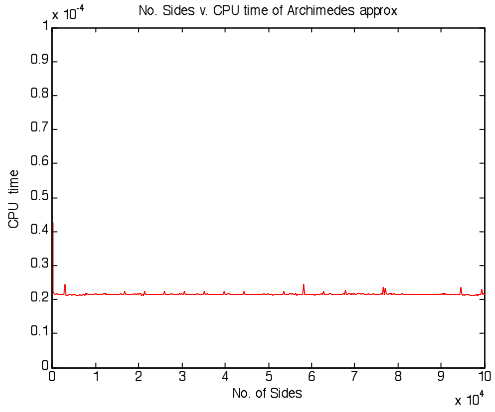


Figure 2, No. of Sides v. CPU time, Archimedes

According to figure 1, an Archimedes approximation of pi using a polygon with more than 10^4 sides has an error of less than 10^{-7}. In other words, a 10^4-gon approximates pi correctly to 7 decimal places. The

data also indicates that an inverse relationship exists between the number of polygon sides and the error of approximation.

According to figure 2, it appears that the arch_pi function operates at a somewhat constant CPU time of approximately $2*10^{-5}$ seconds. The plotted value of $2*10^{-5}$ is rather peculiar, however. When I launched arch_pi with various input values on the range 10,000 to 100,000 in the MATLAB command window and measured time elapsed, the time elapsed averaged at about $1.4*10^{-4}$ seconds. CB suggested that MATLAB may be recognizing the existence of a loop and changing the processing algorithm behind our back.

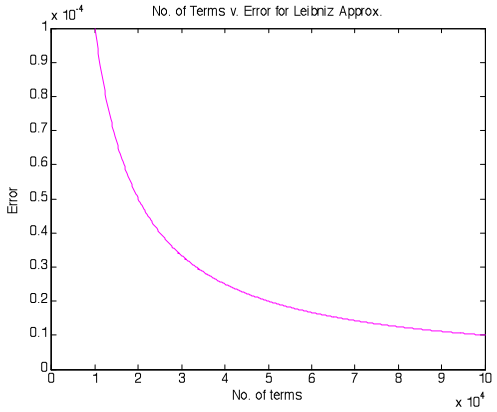


Figure 3, No. of Terms v. Error, Leibniz

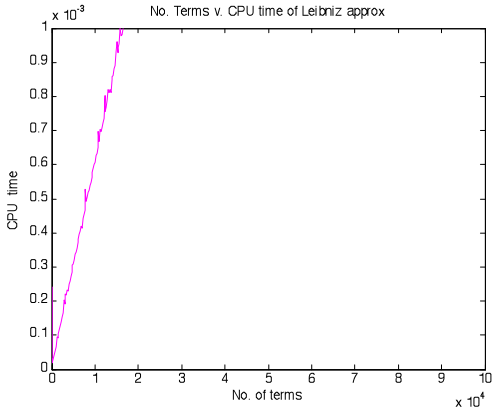


Figure 4, No. of Terms v. CPU time, Leibniz

Figure 3 shows that the Leibniz function approximated pi correctly to 4 decimal places using 10^4 to 10^5 terms. It also indicated an inverse relationship between number of terms and error.

Figure 4 illustrates that the CPU time of the function increases with a near constant rate of approximately $(.6*10^{-3}$ seconds)/(10000 terms), or $6*10^{-5}$ seconds per 1000 terms.

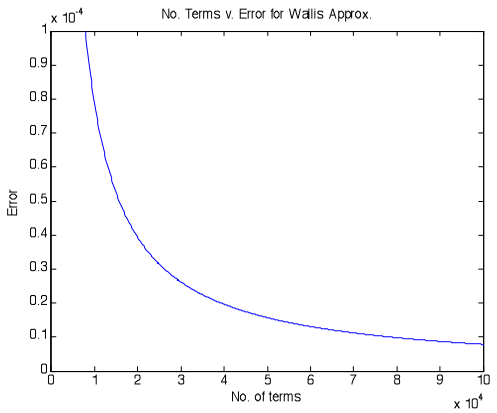


Figure 5, No. of Terms v. Error, Wallis

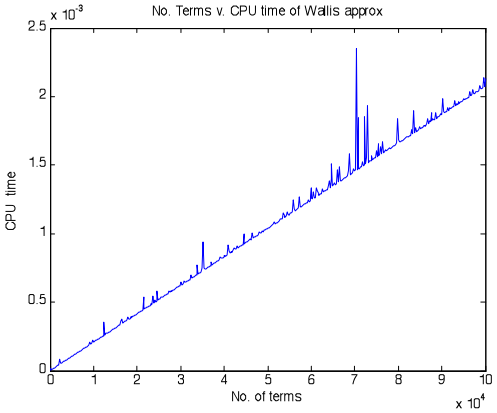


Figure 6, No. of Terms v. CPU time, Wallis

Figure 5 is very similar to figure 3 of the Leibniz function. Once again, pi was approximated correctly to 4 decimal places using 10^4 to 10^5. However, the Wallis function also approximates pi correctly to 5 decimal places using close to 10^5 terms.

According figure 6, the CPU time for the Wallis function is directly related to the number of terms and increases with an approximate rate of $2*10^{-5}$ seconds per 1000 terms.

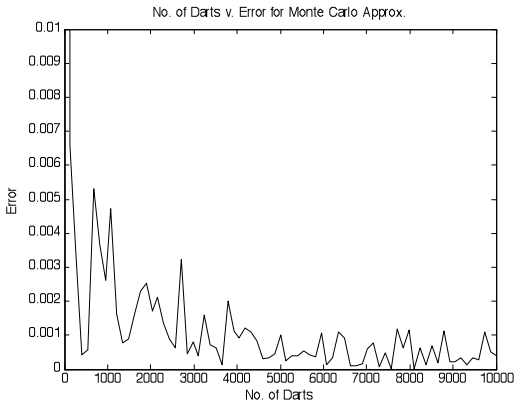


Figure 7, No. of Darts v. Error, Monte Carlo

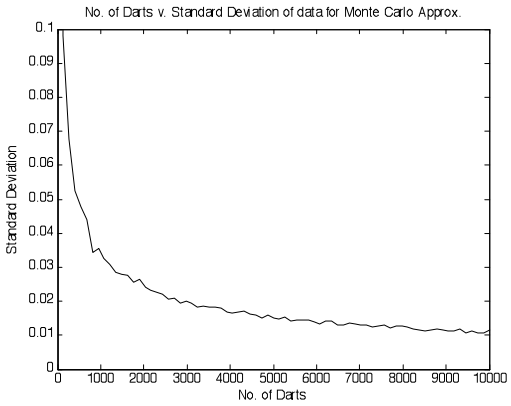


Figure 8, No. of Darts v. Standard Deviation

Figure 7, while appearing scattered, does relay some important information. As the number of darts approached 10000, the error lingered predominantly in a range of .001 and below. This indicates that, on average, a Monte Carlo simulation with 10000 darts will approximate pi within three decimal places of accuracy. If I had more knowledge of data fitting and MATLAB I would have tried to model the error with a best fit curve.

Because the Monte Carlo Simulation deals with random numbers, I also plotted darts v. the standard deviation of values (figure 8). We can see that as the number of darts increases, the standard deviation of the data decreases. This inverse relationship tells us that the Monte Carlo simulation increases in precision as the number of darts increases.

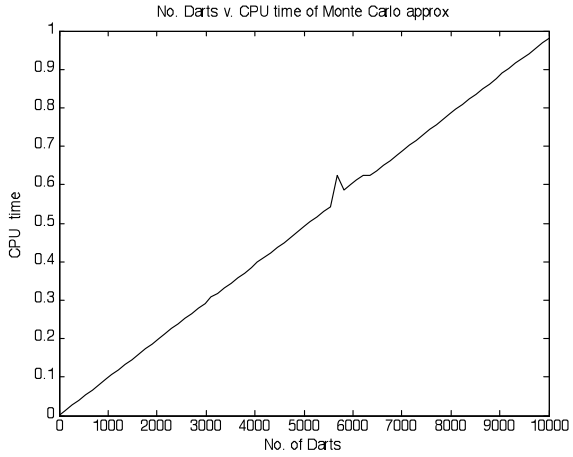


Figure 9, No. of Darts v. CPU time

Figure 9 indicates that the CPU time of the Monte Carlo function is directly related to the number of darts. CPU increases with an approximate rate of 1 second per 1000 darts, much slower than the other three functions.

Which One is Best?

According to my results, the Archimedes approximation of pi is most accurate and practical.

Accuracy

The Archimedes approximation had the lowest calculated error of all the functions. An Archimedes approximation of pi using a 10000-gon had seven degrees of precision, while 10000 terms of the other functions only had up to 4 degrees of precision.

Practicality

The Archimedes approximation of pi offered several advantages related to CPU speed. Most importantly, as the number of sides increased, the CPU time remained nearly constant. Each other function showed a direct relationship between the number of terms and the CPU time. Therefore, one can approximate pi using my Archimedes function with nearly any number of sides without worrying about waiting around for MATLAB to process the result. Even though other functions, such as Leib_pi, were faster at calculating, say, 3 terms, they fell far behind when processing very large numbers of terms (which we use to get a good approximation).

The reason for the lack of a direct relationship between terms and CPU time for the Archimedes approximation is its lack of a loop. The Archimedes function only executes a few calculations in sequence and displays a result. The other three pi-approximating functions involve 'for' loops that drastically increase CPU time as the number of terms, and therefore the number of loop iterations, becomes very large. My Monte Carlo simulation, for example, contains two large loops that cause the function to operate at the speed of 1 sec/1000 darts.

The lack of a 'for' loop in the Archimedes function leads to another advantage. In MATLAB, a function truncates after 2147483647 iterations of a loop. This further limits the functionality of the other three pi-approximating functions, which will reach a term limit sooner than the Archimedes function will.

In short, the Archimedes function is the simplest in design. In engineering, we strive to reach a goal as efficiently as possible. According to my results, of the Archimedes, Leibniz, Wallis, and Monte Carlo approaches, the Archimedes approach can be most efficiently written as a practical MATLAB function.

Extra Credit

For extra credit, I also included a pi-approximating function that uses the Euler approximation of pi. This approximation is represented by the notation $\frac{\pi^2}{6} = \sum_{j=1}^{\infty} \frac{1}{j^2}$. The code for this function is very similar to that of leib_pi. The plots of No. terms v. speed and No. terms v. CPU time appear as follows.

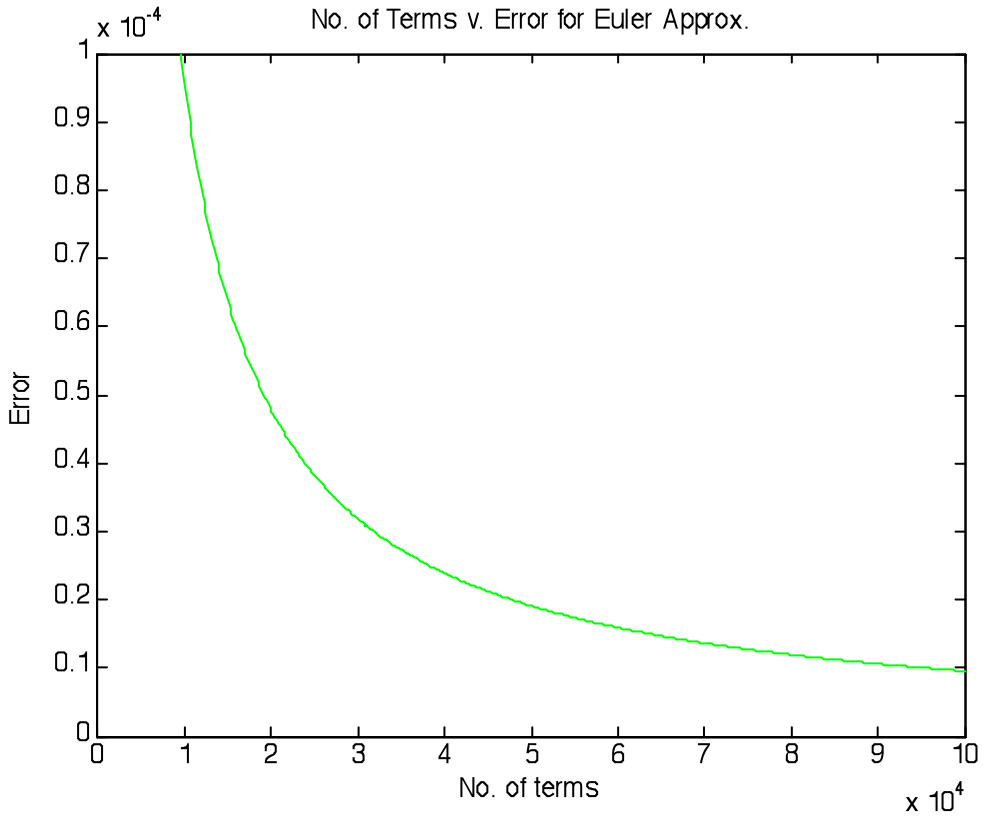


Figure 10

Figure 10 shows the number of terms v. error for this function for 1 to $1*10^5$ terms. Figure 10 looks almost identical to figure 3. The graph shows an inverse relationship between number of terms and error. It also shows that for a number of terms greater that $1*10^4$, eulerpi will approximate pi will an error of less than $1*10^{-4}$.

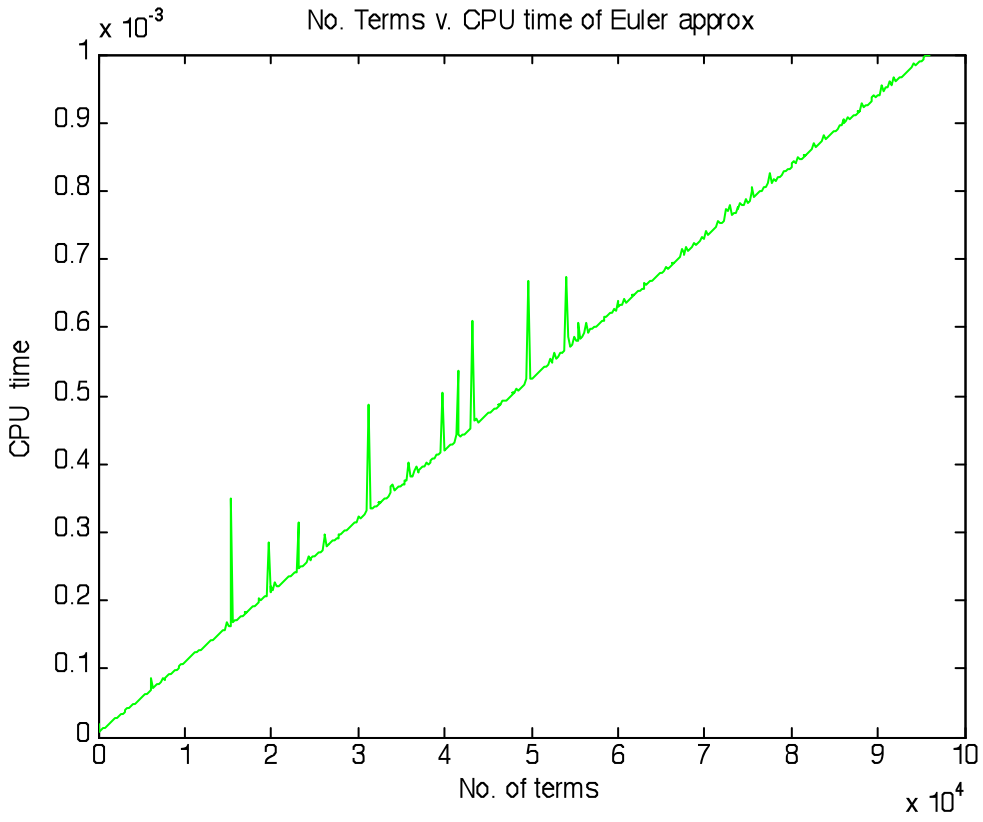


Figure 11

Figure 11 shows the graph of No. of terms v. CPU time for the Euler approximating function. CPU time increases with a seemingly constant slope of roughly $1*10^{-5}$seconds per 1000 terms.

The Euler approximation was similar in accuracy to both the Leibniz and Wallis approximations, and a little faster than both. Even still, it was much slower than arch_pi for large numbers of terms, further cementing arch_pi's place as the best overall pi approximating function.