Analysis – A Little Slice of π

Student name: *Serjo Barron*

Course: *CSE13S* – Professor: *Prof. Long*
Due date: *10/10/21*

1 Introduction

There are plenty of useful and convenient things in the C language. One of them are the various libraries and functions included in said libraries. One of them, specifically <math.h> is quite convenient as it has mathematical constants such as π, Trigonometric functions like sine, tangent, and cosine, along with other functions such as square root. The point here is, just like calculators we take these functions for granted and don't think about what goes on behind sqrt(), cos(), tan(), etc.

This paper will focus on some of the mathematical constants and functions included inside <math.h>, and compare the effectiveness in both the various ways to go about computing a constant such as π, and compare the accuracy of mimicked functions inside the <math.h> library to the <math.h> value.

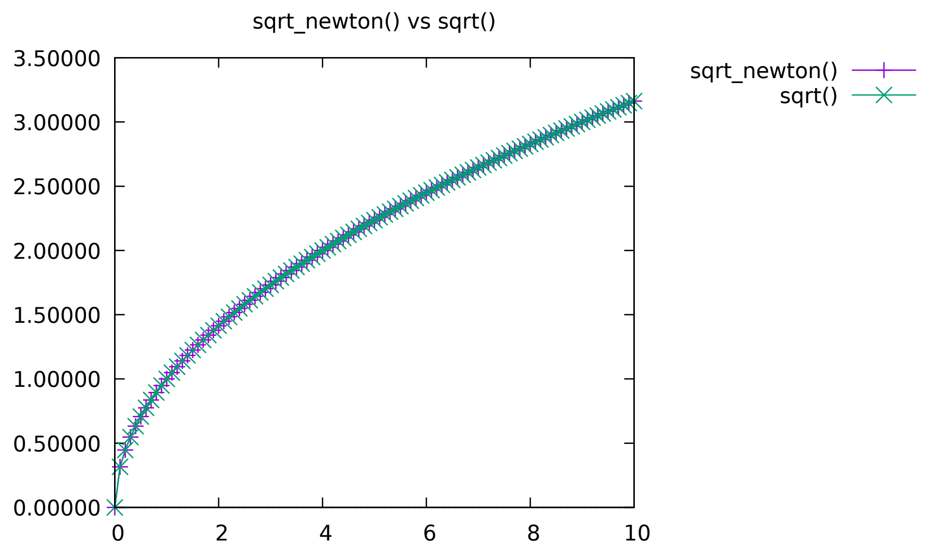
2 Methods

The data that will be used in this analytic paper has been produced and made in the programming C language. Nearly all of the numbers dealt with are double type numbers, with a set precision of 16.15. Though it should be noted that when graphed (using a tool called gnuplot), the points shown will not be entirely represented due to the original numbers gathered having a length of 15 decimal places, and the graphs being tuned to represent only 5 decimal places on the y-axis.

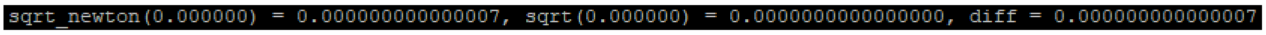
The purpose of the graphs that will be shown in the analysis portion of this paper are as visible representations of how each computation converges to a certain constant or number. Though of course, the findings will be supplemented with the necessary images of the printed results from the VM terminal.

3 Results/Analysis

3.1 Newton Raphson's Method vs sqrt():



In the graph above the x-axis represents the number being square rooted, while the y-axis represents the result of each function. Now examining the behavior of the graph, both functions sqrt_newton() and sqrt() behave quite similarly to the mathematical constant e. At first glance, there is little to no difference between the two functions since the lines as overlap.



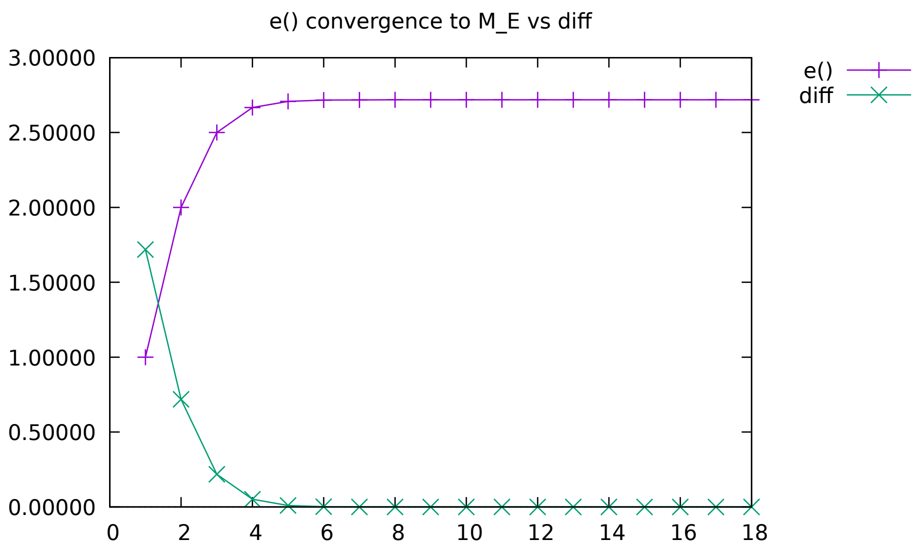
```
sqrt_newton(0.000000) = 0.000000000000007, sqrt(0.000000) = 0.0000000000000000, diff = 0.000000000000007
```

Though as shown in the image above, the only computed square root of x from range 0 to 10 inclusive with steps of 0.1, is sqrt_newton(0). This result is likely caused by the *Newton Iterate*, which is defined as:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

The value x_{k+1} serves as a successive guess to the previous improvement x_k, and after each guess the result (the square root of a number) eventually gets closer to the actual approximation. But since the formula above works in a way such that each guess gets closer to the next, taking the square root of 0 results in an infinite amount of iterations to reach an actual value of 0. That is to say that using Newton Raphson's method of the square root of 0 would be impossible since there is an infinite amount of decimal places to represent each successive guess.

3.2 The convergence of e:

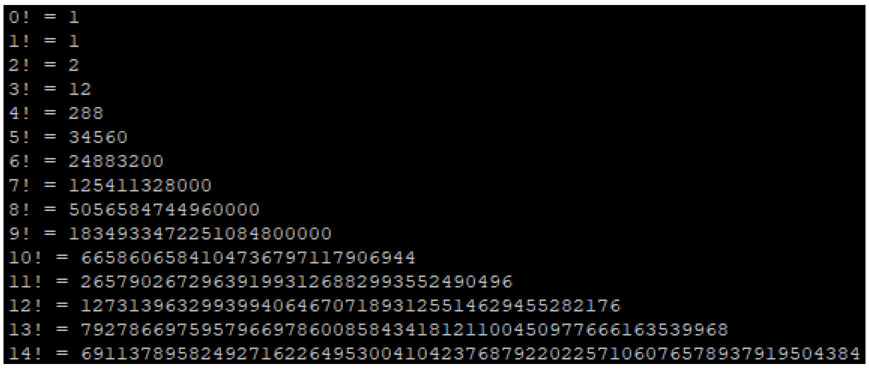


The graph above demonstrates the how the value of e changes (the y-axis) over a certain amount of terms (the x-axis). e() being the line that converges to the mathematical constant e, while the diff line converges to 0.

As visible by the graph, it does not require much terms for the line e() to converge to e. The reason behind this lies in the following formula:

$$e = \sum_{k=0}^{\infty} \frac{1}{k!}$$

The important part of this equation is $k!$, and as one could imagine k! can get very big very fast, as demonstrated in the image below:



```
0! = 1
1! = 1
2! = 2
3! = 12
4! = 288
5! = 34560
6! = 24883200
7! = 125411328000
8! = 5056584744960000
9! = 1834933472251084800000
10! = 6658606584104736797117906944
11! = 265790267296391993126882993552490496
12! = 127313963299399406467071893125514629455282176
13! = 792786697595796697860085843418121100450977666163539968
14! = 69113789582492716226495300410423768792202257106076578937919504384
```

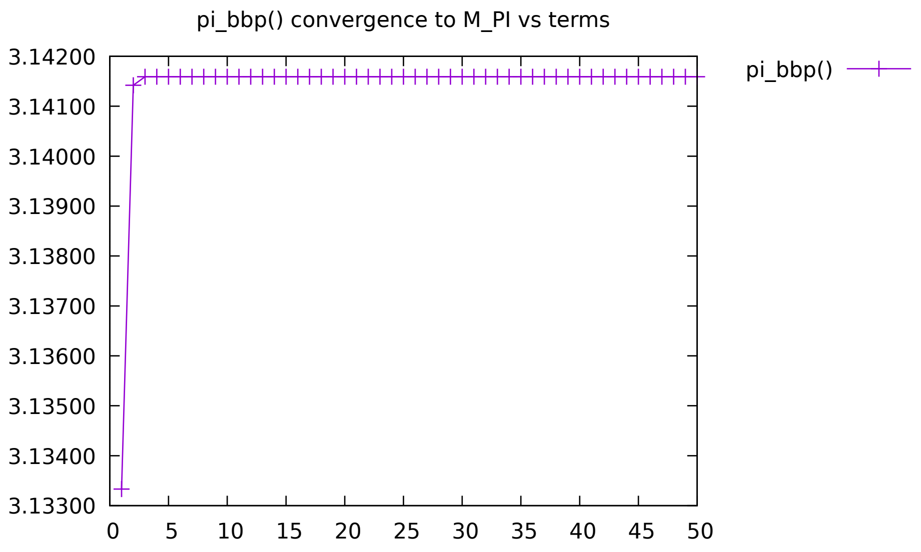
As shown at just 7! the fraction would be $\frac{1}{125,411,328,000}$, and as k gets bigger towards a number such as 18!, the fraction $\frac{1}{k!}$ will converge to 0 at a rapid rate.

If we were to look back at the e() convergence graph, the exponential growth of k! explains why the diff (difference between e() and M_E in the <math.h> library) reaches very close to 0 at just the 18th term.

Now, we will move on from the results of newton_sqrt() and e() functions, and focus on the various implementations of calculating the constant π.

It should also be noted that all the π computation functions have been graphed to 50 terms regardless of when they reach epsilon, the reason behind this is for a better visual representation of how each function converges to the constant π. The exception to this however is the pi_euler() function, which will be discussed in detail later.

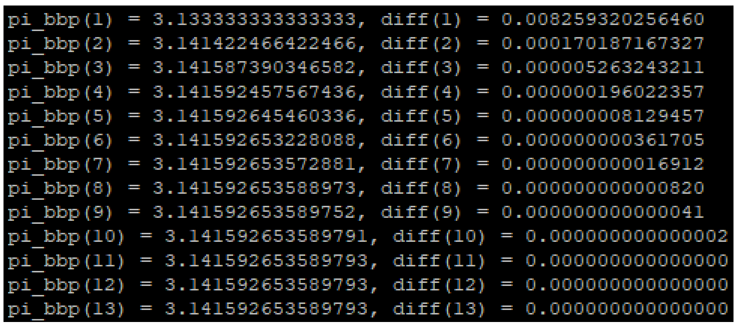
3.3 The Bailey-Borwein-Plouffe Formula:



The graph above illustrates how the value of π (the y-axis) changes over a certain amount of terms (the x-axis).

As shown in the graph, just after the first term the current calculated value is already at an approximation of $3.1\overline{3}$, which is fairly near to the constant π. Then around the 5th or so term, the graph has noticeably converged, and from there on is quite flat.

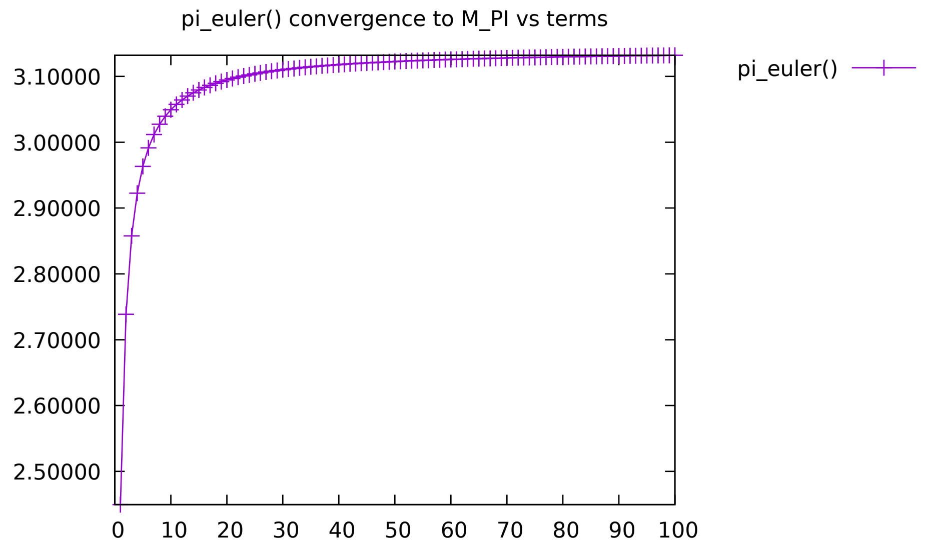
This being the case, pi_bbp is the fastest converging series out of the four implemented π functions.



```
pi_bbp(1) = 3.133333333333333, diff(1) = 0.008259320256460
pi_bbp(2) = 3.141422466422466, diff(2) = 0.000170187167327
pi_bbp(3) = 3.141587390346582, diff(3) = 0.000005263243211
pi_bbp(4) = 3.141592457567436, diff(4) = 0.000000196022357
pi_bbp(5) = 3.141592645460336, diff(5) = 0.000000008129457
pi_bbp(6) = 3.141592653228088, diff(6) = 0.000000000361705
pi_bbp(7) = 3.141592653572881, diff(7) = 0.000000000016912
pi_bbp(8) = 3.141592653588973, diff(8) = 0.000000000000820
pi_bbp(9) = 3.141592653589752, diff(9) = 0.000000000000041
pi_bbp(10) = 3.141592653589791, diff(10) = 0.000000000000002
pi_bbp(11) = 3.141592653589793, diff(11) = 0.000000000000000
pi_bbp(12) = 3.141592653589793, diff(12) = 0.000000000000000
pi_bbp(13) = 3.141592653589793, diff(13) = 0.000000000000000
```

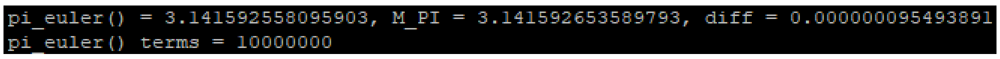
The values given above show the difference between the value π computed from pi_bbp() and M_PI available in the <math.h> library. And as shown, the difference approximates to 0 at just the 11th term demonstrating how rapidly the pi_bbp() function converges.

3.4 Euler's Solution:



The graph above demonstrates how the implementation of Leonhard Euler's formula converges to the constant π by representing the current value of π (the y-axis) over the number of iterated terms (the x-axis).

As warned earlier, the graph above is not entirely accurate since the pi_euler() function runs for a total of 10,000,000 terms until it reaches the error term epsilon. Though, for the sake of a readable and meaningful graph, and not having to plot 10,000,000 terms via gnuplot, the range of terms has been limited to the first 100. However, the current graph seen above will still suffice for the findings of using Euler's Solution to calculate π.

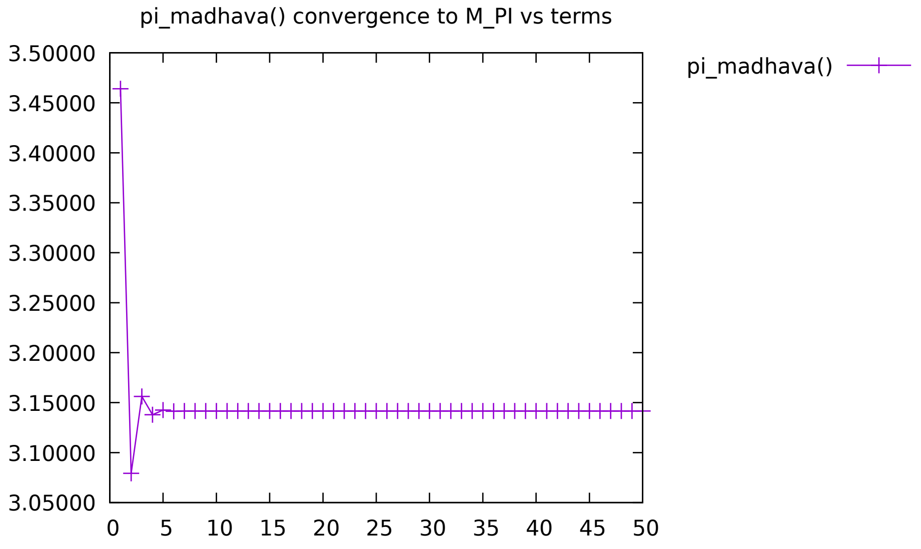


```
pi_euler() = 3.141592558095903, M_PI = 3.141592653589793, diff = 0.000000095493891
pi_euler() terms = 10000000
```

As mentioned and shown above the pi_euler() function does in fact run for a total of 10,000,000 terms until it hits the error term epsilon. Meaning that the implementation of Euler's Solution converges the slowest out of the π computational functions.

A reason behind the case lies in the formula, which involves harmonic numbers. And since the error term involves harmonic numbers, the corresponding graph will converge extremely slowly, and to us would be impractical to calculate since it would go on for infinity.

3.5 The Madhava Series:

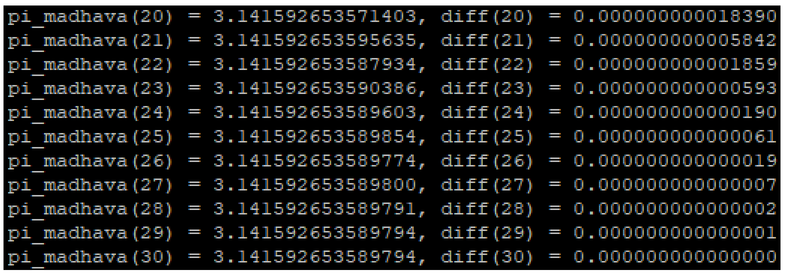


The graph above represents the implementation of The Madhava Series, with the y-axis representing the current value of π over its current term, the x-axis.

As visible by the graph, the first few terms has interesting behavior with π being approximately 3.46410 at term 0, then after it rapidly slopes down until it converge quite smoothly starting around term 5.

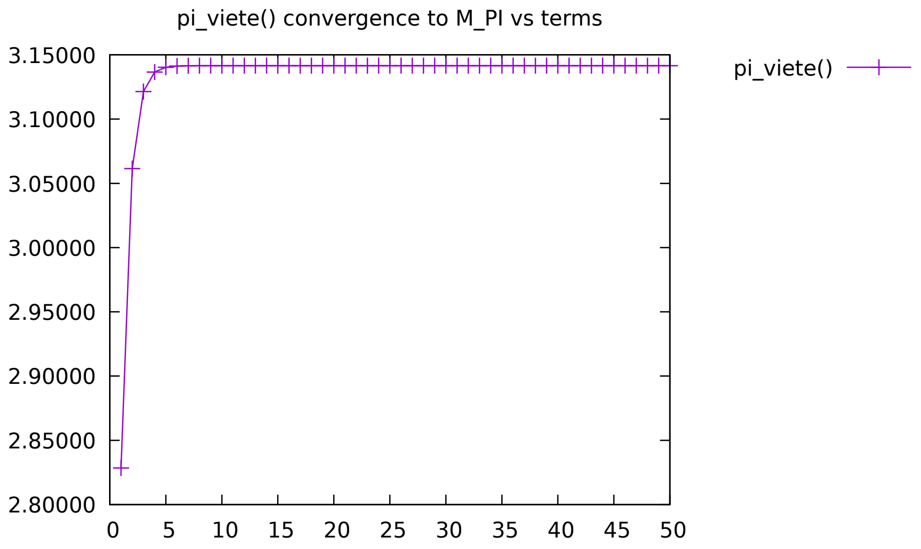
Similar to pi_bbp(), pi_madhava() also converges rapidly to the constant π, though if it were a battle of efficiency, pi_bbp() still pulls ahead by being the faster function and method of computing pi

As shown below, it takes approximately 27 terms for The Madhava Series to be on equal footing with The Bailey-Borwein-Plouffe Formula in terms of how quickly each method converges to π.



```
pi_madhava(20) = 3.141592653571403, diff(20) = 0.000000000018390
pi_madhava(21) = 3.141592653595635, diff(21) = 0.000000000005842
pi_madhava(22) = 3.141592653587934, diff(22) = 0.000000000001859
pi_madhava(23) = 3.141592653590386, diff(23) = 0.000000000000593
pi_madhava(24) = 3.141592653589603, diff(24) = 0.000000000000190
pi_madhava(25) = 3.141592653589854, diff(25) = 0.000000000000061
pi_madhava(26) = 3.141592653589774, diff(26) = 0.000000000000019
pi_madhava(27) = 3.141592653589800, diff(27) = 0.000000000000007
pi_madhava(28) = 3.141592653589791, diff(28) = 0.000000000000002
pi_madhava(29) = 3.141592653589794, diff(29) = 0.000000000000001
pi_madhava(30) = 3.141592653589794, diff(30) = 0.000000000000000
```

3.6 Viète's Formula:

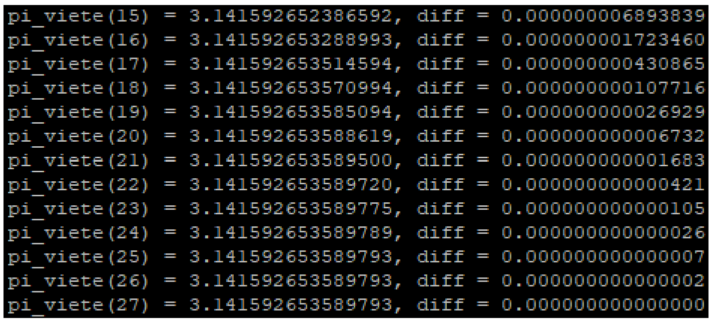


The graph above demonstrates the final implementation of computing π, with the y-axis representing the current value of π over the current term, the x-axis.

As shown by the graph, pi_viete() converges to π in a similar fashion to the pi_bbp() function. Though of course it does appear to converge slower as it takes about 5 terms for the graph to converge smoothly.

The implementation of Viète's Formula to compute π in fact has similar results to the previous method of calculating π covered, The Madhava Series. Though the graphs visually look very distinct, both implementations converge to π (or hit the error term epsilon), at a similar speed.

As shown in the image below, pi_viete() hits the error term epsilon at the 25th term and has the same difference to M_PI in the <math.h> library as pi_madhava() did on its 27th term.



```
pi_viete(15) = 3.141592652386592, diff = 0.000000006893839
pi_viete(16) = 3.141592653288993, diff = 0.000000001723460
pi_viete(17) = 3.141592653514594, diff = 0.000000000430865
pi_viete(18) = 3.141592653570994, diff = 0.000000000107716
pi_viete(19) = 3.141592653585094, diff = 0.000000000026929
pi_viete(20) = 3.141592653588619, diff = 0.000000000006732
pi_viete(21) = 3.141592653589500, diff = 0.000000000001683
pi_viete(22) = 3.141592653589720, diff = 0.000000000000421
pi_viete(23) = 3.141592653589775, diff = 0.000000000000105
pi_viete(24) = 3.141592653589789, diff = 0.000000000000026
pi_viete(25) = 3.141592653589793, diff = 0.000000000000007
pi_viete(26) = 3.141592653589793, diff = 0.000000000000002
pi_viete(27) = 3.141592653589793, diff = 0.000000000000000
```

That is to say, pi_viete() in fact beats the pi_madhava() implementation of calculating π by 2 terms, yet it still takes double the amount of terms when compared to the fastest converging implementation, the pi_bbp() function.

4 Conclusion

After analyzing the results and graphs, it has been made evident that the functions which mimic the <math.h> library do a fair job in calculating the value they were made to do (e, square root of a number, or π). Though to go back to the main purpose of conducting such experiments – the goal to better understand what goes behind M_PI, M_E, or sqrt(), the analysis has proven there are efficient methods of calculating such values.

The π computation functions were a great example of various methods to calculate the same result. With the 4 methods of computing π being euler, bbp, viete, and madhava. And we found that in terms of a list of quickest to slowest converging graphs, it goes as follows: bbp, viete, madhava, then euler. So in the end, we can with high certainty say that some methods of computing the constant π are far more efficient and practical than others.

That is to say, the functions implemented would still be no match to using the standard <math.h> library for conducting computations to the highest precision/accuracy. Though as hinted in the introduction we have implemented potential ways to calculate such a value, but it was using the basic mathematical operators addition, subtraction, multiplication, and division. Proving that computers do not come up with these numbers out of thin air, and that there are actual far more complicated methods of calculating them that are used behind the words #include <math.h>.