Exploration in π Calculation Using Various Methods Implemented in Python

Kunyi Li

May 30, 2013

1 Introduction

The mathematical constant π, found ubiquitously in many applications, has been a subject of interest for mathematicians ever since the beginning of recorded history. In order to seek out better understanding of its nature, calculation of the its decimal digits has been a perennial favored activity of many mathematicians throughout history. The length of digit of π capable of being obtained has always been commensurate with the technological advance in mathematics at the time. First, the ancients Egyptians and Hebrew sufficed with the estimate of roughly 3 based on their intuition. Later, discoveries by Archimedes in Greece and Liu Hui in China led to more precise estimation of π using calculations based on regular polygons inscribed and circumscribed in a circle. However, it was not until the discovery of the infinite series that mathematicians began to make major strides in the calculating the digits of π. As we enter the modern era, the advent of digital computers allows us to obtain precision of π of astronomical proportions.

Even though it is said that 39 digits of π is sufficient to calculate the circumference of the entire measurable universe accurate to the width of one hydrogen atom,[1] π calculation is still beloved by hobbyists and computer benchmarkers. In this project, we will explore three π algorithms based on various types of infinite series and iterative techniques, in order to calculate π to as many digits as possible while optimizing for speed of calculation. In addition, we will make some observations about the distribution of digits in π using the results we calculate.

2 Methods and Implementation

In this project, we will implement the following three methods: arctan or Machin's Method, Chudnovsky's Algorithm and Brent-Salamin's Algorithm. The first method is chosen for its historical significance, since its most basic version was used by its discoverer and contemporaries in the 18th century in

[1] Arndt, Jorg; Haenel, Christoph (2006). *Pi Unleashed.* (Berlin:Springer-Verlag), 17.

calculation by hand. Next, the Chudnovsky's Algorithm, quite similar in nature to the algorithm first found by the legendary Ramanujan, has been a popular method adopted by record breakers everywhere in the world in recent years. Lastly, the Brent-Salamin Formula, which uses the arithmetic-geometric mean to compute π, is known for its rapid convergence.

All of these methods are implemented in Python 2.7, with the help of two packages of arbitrary precision mathematics, *gmpy2* and *mpmath.* In the implementation of the first two methods, the result obtained is π multiplied by 10^d where d is the number of digits, hence giving us an integer value that begins with 31415.... This is achieved by multiplying the appropriate initialization by a factor of 10^d. Theoretically this method can be implemented with Python's native *int* type, which accepts computation of arbitrarily large integers. But using *gmpy2*'s *mpz* number type, the exact same algorithm can be calculated much faster.[2] In addition, *mpmath* is used in the implementation of Brent-Salamin's Formula, which returns the decimal value 3.1415... of π using its *mpf* number type that allows decimals of arbitrary precision. Lastly, Python's built *time* module is used to measure the run-time of each algorithm.

In the following subsections I will describe each method and their respective implementation. For details please see the actual code.

2.1 Machin-like Formulae

First discovered by English mathematician John Machin in 1706, its original form appeared as the following:[3]

$$\frac{\pi}{4} = \arctan(1) = 4\arctan\frac{1}{5} - \arctan\frac{1}{239}$$

The derivation of this formula relies on the following property of the arctan function:

$$\arctan(x) + \arctan(y) = \arctan(\frac{x+y}{1-xy})$$

There are many of such Machin-like formula with up to 6 different arctan terms that allows for faster rate of convergence. In this project we will implement one additional Machin-like formula for comparison, with credit due to Gauss and Euler:

$$\frac{\pi}{4} = 12\arctan\frac{1}{18} + 8\arctan\frac{1}{57} - 5\arctan\frac{1}{239}$$

To actually compute digits of π, these formulae are used in conjunction with the Taylor series expansion for arctan, here written as $\arctan(1/x)$:

$$\arctan\frac{1}{x} = \frac{1}{x} - \frac{1}{3x^3} + \frac{1}{5x^5} - \frac{1}{7x^7} + ...$$

[2]According to *gmpy2*'s documentation. The *mpz* type becomes significantly better in performance than the built-in *int/long* type when an integer exceeds "as low as 20 to 40 digits". https://gmpy2.readthedocs.org/en/latest/overview.html

[3]Wolfram MathWorld, *Machin-Like Formulas,* http://mathworld.wolfram.com/Machin-LikeFormulas.html

According to the result, to obtain π accurate to n digits with the original Machin's formula, whose term evaluated farthest from the origin is $\frac{1}{5}$, it requires $\sim 1.4n$ number of terms. However, we can half the number of terms with this modified version of the arctan series discovered by Euler:[4]

$$\arctan\frac{1}{x} = \frac{x}{1+x^2} + \frac{2}{3}\frac{x}{(1+x^2)^2} + \frac{2\cdot 4}{3\cdot 5}\frac{x}{(1+x^2)^3} + \frac{2\cdot 4\cdot 6}{3\cdot 5\cdot 7}\frac{x}{(1+x^2)^4} + ...$$

As mentioned briefly above, to obtain π as an integer raised to 10^d where d is the number of desired digits, the first term is initialized as $(x \cdot 10^d)/(1+x^2)$. The terms that follow are derived using Python's built-in floor division operator $//$ such that $y_{n+1} = y_n//f = \lfloor y_n/f \rfloor$, where f is the appropriate factor to be divided. Then, the calculation of the arctan series breaks when $y_n < f$ such that $y_{n+1} = 0$. This method of calculation allows us to avoid the rounding problem with floating decimals and take advantage of Python's built *int* type or *gmpy2*'s optimized alternative *mpz.*

2.2 Chudnovsky/Ramanujan's Algorithm

Chudnovsky's algorithm, discovered by the Chudnovsky brothers in 1989, appears in the following form:

$$\frac{1}{\pi} = 12\sum_{k=0}^{\infty}\frac{(-1)^k(6k)!(13591409+545140134k)}{(3k)!(k!)^3 640320^{3k+3/2}}$$

In practice, each additional term of this algorithm provides approximately 14 additional digits of π. To implement this algorithm, we manipulate the above expression to compute two series a and b and use them in a final equation that gives the value of π. The initialization and final formula appears as following:

$$a_k = \frac{(-1)^k(6k)!}{(3k)!(k!)^3 640320^{3k}}$$

$$\frac{a_k}{a_{k-1}} = -\frac{24(6k-5)(2k-1)(6k-1)}{k^3 640320^3}$$

$$b_k = k \cdot a_k$$

$$\pi = \frac{426880\sqrt{10005}}{13591409a + 5451401346}$$

Similar to the above method, the first term of a and b are initialized as 10^d so that we can perform the calculation using Python's *int* and *gmpy2*'s *mpz*, and subsequent terms are derived using floor division. Meanwhile, b is calculating by summing each term of a multiplied by the index.

Furthermore, binary splitting is a technique commonly used for this type of calculation that can dramatically speed up the time of calculation. Division,

[4]Wolfram MathWorld, *Inverse Tangent,* http://mathworld.wolfram.com/InverseTangent.html

which is known to be a costly operation, is minimized in the binary splitting method, which allows series of the type $S(a,b) = \sum_{n=a}^{b} \frac{p_n}{q_n}$ to be calculated as $S(a,b) = \frac{P(a,b)}{Q(a,b)}$. Compared to the naive implementation, which computes $b-a$ separate divisions, binary splitting only performs one division in the end.[5] For the adaptation of this method please see the code implemented in Python 3.0 authored by Craig Wood.[6]

2.3 Brent-Salamin's Formula / Arithmetic-Geometric Mean Method

Based on the work of Gauss and Legendre in the 18th century, this formula computes π by repeatedly replacing two numbers by their arithmetic and geometric mean, returning in the two numbers converging to their arithmetic-geometric mean. This algorithm converges quadratically and takes around 3.5 terms to multiply the number of digits by 10.[7] The version adopted for this project calculates π using the following equation, where $M(a,b)$ is the converged arithmetic-geometric mean of the two initial values $a_0 = 1$ and $b_0 = \frac{1}{\sqrt{2}}$, and a_{n+1} is the arithmetic mean of a_n and b_n and b_{n+1} is the geometric mean of a_n and b_n :

$$\pi = \frac{4 \cdot M^2(1, \frac{1}{\sqrt{2}})}{1 - \sum_{n=1}^{\infty} 2^{n+1}(a_n^2 - b_n^2)}$$

$$a_{n+1} = \frac{a_n + b_n}{2}$$

$$b_{n+1} = \sqrt{a_n b_n}$$

The implementation of this method in Python is rather straightforward. First the code sets up the initial values and then enters a loop that calculates the two means and the series on the bottom of the fraction then swaps a and b with their respective means. The loop breaks when the difference between a and b falls below a threshold (i.e. have converged in terms of this given threshold). This threshold is calculated by $\frac{1}{10^d}$ where d is the number of digits of π desired. Since the previous fraction diminishes rapidly and goes below Python's built-in float point precision when we want $d > 16$, the package *mpmath* is used to accommodate float point decimal of very high precision.

[5] Haible, Bruno; Papanikolaou, Thomas. Fast Multiprecision Evaluation of Series of Rational Numbers, 4 http://www.ginac.de/CLN/binsplit.pdf

[6] http://www.craig-wood.com/nick/pub/pymath/pi_chudnovsky_bs_gmpy.py

[7] Gourdon, Xavier; Sebah, Pascal, *π and Its Computation Through the Ages*, http://numbers.computation.free.fr/Constants/Pi/piCompute.html

3 Results and Analysis

3.1 Performance

In my test run, all the algorithms and their various implementations were clocked for different levels of precision. First the two Machin-like formulae, implemented in both Python's *int* type and *gmpy2*'s *mpz* type, were timed for 10^d digits for d up to 6 or 1 million digits of π. For Chudnovsky's algorithm, the limit for number of digits was raised to 10 million for the regular version and 100 million for the binary splitting version. Finally, Brent-Salamin's Formula were limited to 100 million digits.

For this project, all the code was created and ran on my laptop with Intel Core i7 2.40Ghz Quad-core CPU and 8GB DDR3 RAM running Python 2.7 64bit on Windows 8 Home Premium. Note the code did not optimize for parallel processing hence only one of the cores of the i7 processor was used in the process. During most calculations CPU usage hovered around 20%, which is presumably the maximum utilization by Python on my machine for non-parallel code.

Digits	Machin_Int	Machin Mpz	GaussEuler_Int	GaussEuler_Mpz
100	0.001		0.001	
1K	0.002	0.000999928	0.002000093	0.000999928
10K	0.09800005	0.029000044	0.093999863	0.028000116
100K	8.888939	1.5989	8.696000099	1.569000006
1Mil	977.109	159.789	976.3409998	156.1230001

Table 1: Time (s) to Calculate π Using Machin-like Formulae

In the results for Machin-like formulae we note two key observations. First, the Gauss-Euler variation, which supposedly converges faster with one extra term, did perform better than Machin's variation but only by a negligible amount. Second, the major difference in performance is between the different data-type used, where *mpz* type took only $\frac{1}{6}$ the amount of time than its *int* counterpart using the exact same algorithm. However, it appears that to increase the number of digits by ten-fold the amount of time required multiplies by ~ 100. At this rate 10 million digits would require more than 4 hours.

Digits	Chudnovksy	Chudnovsky_BS	AGM
10K	0.004999876	0.001999855	0.004999876
100K	0.252000093	0.041000128	0.181999922
1Mil	26.81000018	0.666000128	2.855000019
10Mil	6020.742	10.91400003	44.75099993
100Mil		160.141	706.2420001

Table 2: Time (s) to Calculate π Using Chudnovsky and Brent-Salamin

From a glance, Chudnovsky and Brent-Salamin's algorithm are far superior at calculating π, at least in this implementation. Here the astounding thing to

notice is that even though the vanilla Chudnovsky implementation reached 1 million digits in only $\frac{1}{6}$ the amount of time the Machin-like formula took, the binary splitting technique took only $\frac{1}{40}$ the time of the former! In the amount of time the *Machin-mpz* method reached 1 million digits, it has calculated 100 million digits. On the other hand, Brent-Salamin's method did not disappoint, performing $\sim$ 20 times better than the vanilla Chudnovsky and only 4 times slower than binary splitting Chudnovsky.

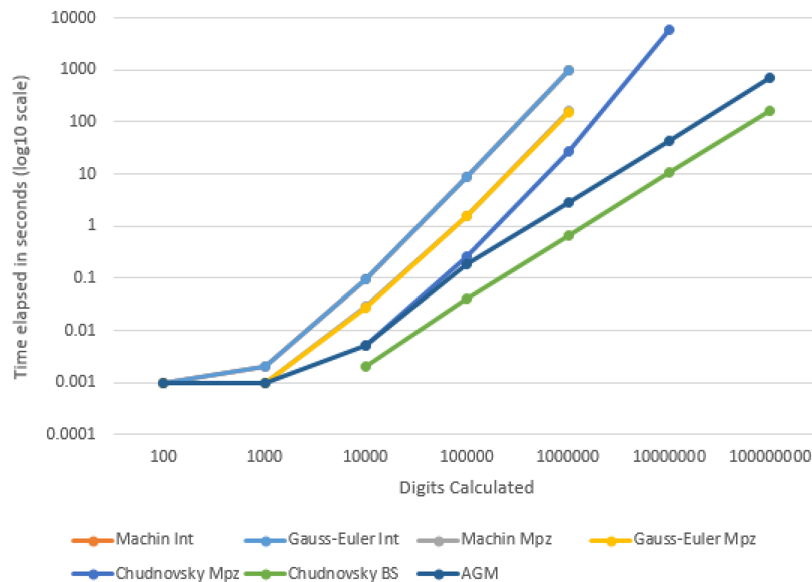

Figure 1: Time (in log10 scale) to Calculate π Using Various method

3.2 Distribution of Digits In π of Various Precision

Using results obtained from some of the calculations, the π variable was passed into a string and then the *count* function was used to obtain the frequency of each digit in the variable. The result was then divided by the precision to obtain the percent distribution. In the results displayed in Table-3 on the next page, we can observe that the distribution of the digits of π is quite gives each digit from 0 to 9 almost the same amount of frequency, with the standard deviation from the mean of 0.1 decreasing as we move up to higher precision and hence the digits becoming more evenly distributed.

Digit	100K	1Mil	10Mil
0	0.099319007	0.1000909	0.10006429
1	0.099699003	0.0997329	0.09998999
2	0.100928991	0.0998799	0.10002779
3	0.101408986	0.1001069	0.10004179
4	0.098079019	0.0995399	0.10000259
5	0.099949001	0.0996159	0.09996959
6	0.099809002	0.1002089	0.09995539
7	0.099749003	0.1001799	0.10007159
8	0.100858991	0.1002969	0.09995269
9	0.100198998	0.1003479	0.09992429
Mean	0.1	0.1	0.1
Std.Dev	0.000892879	0.000273837	4.77169E-05

Table 3: Percent Frequency of Each Digit from 0 to 9 in π of Various Precision

To affirm that the digits of π are normally distributed, a χ-square test was performed on the data for each precision, with expected value for each digit equal $\frac{1}{10} \cdot d$ where d is the number of digits. The $\chi(p)$ value for 100K, 1Mil and 10Mil digits are, 0.5369, 0.5853, 0.9863, respectively. Since the values are much higher than any commonly used statistical threshold (0.05 or 0.01), we can conclude that the digits of π are normally distributed for these precisions. Although it has not be proven π is normal,[8] our experimental result and those of others seem to suggest it is.

4 Potential for Future Studies

Although the above algorithms for π calculation seem to have performed satisfactorily for this project's purposes, I believe it is possible to enhance the performance of specific algorithms even further with other techniques. First, the implementation in Python is quite limited since it does not include any attempt to optimize memory usage. Presumably, by using a lower level language we can speed up the calculation time. In addition, the implementation of this project only used about $\sim 20\%$ of CPU at peak performance. To use up more CPU, multicore parallel processing must be used. for both Chudnovsky and Machin, we can split the series into n parts where n is the number of parallel processes that can be supported. Then we can evaluate the series at n different intervals simultaneously, effective cutting down the obtaining terms and summing them part of the calculation to $1/n$ the amount of time the serially processed code takes. We can observe the effect of the aforementioned optimization by using the free software *y-cruncher*,[9] which contains a multicore implementation of Chudnovsky's algorithm (assuming binary splitting) in C++. In my run of

[8] Stan Wagon, *Is π normal?*, Friends of Pi, http://pi314.at/math/normal.html. [Original Source: The Math. Intell. 7, 65, (1985)]

[9] http://www.numberworld.org/y-cruncher/

y-cruncher to 100 million digits, it only took 24 seconds to get the result, which is $\frac{1}{6}$ the amount of time of my best result.

Furthermore, we can enhance the speed of calculation by using the GPU instead of CPU. The current world record for digits of π, as recent as March 15th, 2013 sits at 8×10^{15} digits, was accomplished using a few NVIDIA GPUs in parallel and took about 90 days.[10] This a potential field that is left to be explored.

[10] http://www.engadget.com/2013/03/15/researcher-breaks-pi-record-with-nvidia/