18.335 FINAL PROJECT: A QUEST FOR PI

DRAGOS VELICANU

Abstract. In this paper I give a brief overview of humanity's quest for uncovering the first transcendental number from its discovery in ancient times to the modern computer era. I compare some of the algorithms people have used to obtain more digits of π and cover two of the leading algorithms that have set current records for digits, the Chudnovsky Algorithm and the Arithmetic Geometric Mean algorithm. Both these algorithms I implement in Julia and use to extract 10 million digits of π.

1. Introduction. The idea of π as the ratio of the circumference to the diameter of a circle has been around for over 4000 years [1]. While various people or groups had ideas of what π should equal to, some even decreed $\pi = 3$ [1] , the first algorithm for computing π came from Archimedes by computing the perimeters of circumscribed and inscribed polygons [2] and thus the first 3 digits, 3.141... were correctly computed over 2000 years ago [6]. These polygonal algorithms were used to set the records for digits of pi up to the 17th century, with the Persian astronomer Jamshīd al-Kāshī producing 16 digits in 1424 by using a polygon with 3×2^{28} sides [3] up to 1630 when Christoph Grienberger arrived at 38 digits using a similar algorithm.

The next revolution for computing π came from infinite series, especially the development of calculus, provided novel ways to algebraically compute π to greater accuracies. For example the Taylor expansion of $\arctan(x)$:

$$\arctan(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - ...$$
$$\frac{\pi}{4} = \arctan(1) = 1 - \frac{1}{3} + \frac{1}{5} - ...$$

can be used to compute π and was used by mathematician Abraham Sharp to compute 71 digits in 1699 and break the record previously held by the polygonal algorithms [1]. This series however converges far too slowly to be of use in contemporary times, since half a million terms are required just to get the first five correct digits [4]. Today there are much faster algorithms for computing π than using the arctan series, but a similar arctan series that converges faster was used by George Reitwiesner and John von Neumann on the first computer, ENIAC, to compute over 2000 digits of π setting the then current record [5].

2. Modern Algorithms - Chudnovsky algorithm. The Chudnovsky Algorithm published by the Chudnovsky brothers is one of the current fastest algorithms for computing π and also the algorithm that holds the current world record of 10 trillion digits. Each term in the following series gives 14 new digits of π:

$$\frac{1}{\pi} = \frac{12}{640,320^{3/2}} \sum_{k=0}^{\infty} \frac{(-1)^k (6k)! (13,591,409 + 545,140,134k)}{(3k)! (k!)^3 640,320^{3k}} \tag{2.1}$$

One of the defining features of Chudnovsky's algorithm is that it does only integer operations at each step. Thus each step can be verified for errors by using prime moduli, and the probability of any local error can be reduced to below 10^{-290} [9] . One way to understand this is, if trillion digit numbers are being multiplied, if the last

300 digits are right, the full multiplication is also right with pretty high probability, and we can get the last 300 digits by doing the multiplications mod some prime number. Explicitly:

$$A + B \bmod p = [A \bmod p + B \bmod p] \bmod p$$
$$A - B \bmod p = [A \bmod p - B \bmod p] \bmod p$$
$$A \times B \bmod p = [A \bmod p \times B \bmod p] \bmod p$$

Can be used for very large arithmetic operations to check their validity in a much shorter time than it takes to carry them out.

In my implementation of Chudnovsky's algorithm I use Julia's arbitrary precision functionality in order to compute the large factorials and products using BigInt. For the division and final step I use BigFloat with precision set to the number of digits I want to compute π to. To verify the computed values of π I can just search for the last 100 digits of my computation in a larger computation (like what I compute from the AGM method or from downloading digits of π online, which at the moment up to 1 billion digits of π are available for download which is half a gigabyte compressed and 1.36GB pure text [7].

There are several pitfalls one should be wary of when trying to implement a similar algorithm in Julia. The exponentiation of the constants or factorials should not be of normal float type as arbitrary precision will be lost and the resulting expression for π will be correct up to $O(\varepsilon_{machine})$ which is about 14-15 correct digits of π independent of the number of iterations, even though the BigFloat result will display as many as is asked of it, common code like $640320^{3.0}$ will not be good here. It's best to keep all factorials of BigInt type and all exponentiation of integers of integer or BigInt type. One should also be careful all integers in danger of overflowing should be BigInt as Julia won't throw an error when that happens, the overflowed int will still be an int but with nonsense value, which won't stop the iteration or even change the first correct terms of π but all future terms will be wrong. When this happens π will mysteriously be correct up for the first few iterations of Chudnovsky's algorithm but will give wrong digits of π after a certain point.

2.1. Chudnovsky Complexity and Performance. The Chudnovsky algorithm can be used to compute up to 100,000 digits of π in a straightforward Julia implementation in under five minutes. The Julia source code can be found here [11] . The current implementation computes the full values of all factorials at each step using the factorial of integers of arbitrarily large size. Since the Chudnovsky algorithm needs 3 separate factorials to be evaluated at each step, a $(6k)!$, $(3k)!$ and a $(k!)^3$ and then a product and division of these giant numbers. The computation grows very rapidly as for $k = 10000$ the biggest integer has 260634 digits that need to be multiplied and divided to get 14 more digits of π . Since this the most time consuming part of the algorithm, I investigated the time and space complexity of calling factorial on successively larger integers in full precision, summarized in Figure 2.1 below.

We see that Julia can compute factorials of integers as large as 10 million, however to compute one such factorial requires on the order of 1 second and 1GB of RAM . As 10 million factorial would be required in computing π to over 23 million digits, each extra 14 digits taking a second of CPU time makes computing π with this method extremely long on a single core machine into the millions of digits. To get some intuition of what is demanding in this algorithm at large values I computed the last term to get the 10th million digit of π and summarized the results below in Table 2.1.

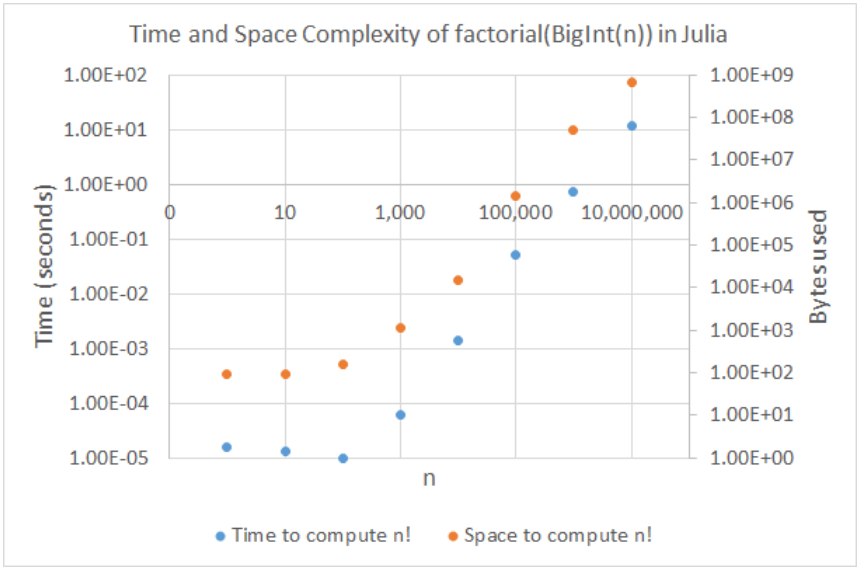

FIG. 2.1. *Time and Memory required for computing N! in Julia*

$k = 713287$	time (s)	space (GB)
$(6k)!$	4.6	0.27
$(3k)!$	2.0	0
$(k!)^3$	1.2	0.12
$c \times k$	6.6e-6	0.08
c^{3k}	0.5	0.03
$(a \times b) \,/\, (c \times d \times e)$	5.8	0.38
Total	14.3	0.89

TABLE 2.1

Time and space breakdown of computing a single term of the Chudnovsky series in Julia near the 10 million-th digit of π.

Some performance improvements can be made to the algorithm by noting that subsequent factorial terms in the series in Eq. 2.1 can be computed from the previous terms with a fixed amount of arbitrary precision operations by the following formula.

$$(6\,(k+1))!_{\text{new}} = (6k)!_{\text{old}} \;\times (6k+1) \times (6k+2) \times \ldots \times (6k+6)$$

An even bigger performance improvement can be made by parallelizing the terms in the sum since each term can be calculated independently from the rest and all the results can be gathered at the end as long as enough precision was used to do all independent operations, making this a perfect candidate for map-reduce. Due to this and the 18.337 class I am concurrently taking I couldn't resist trying to see how much this computation could be sped up by using Julia's built in parallel libraries. To compare I compute the first 100,000 digits of π using the straightforward implementation of the algorithm, using the factorial performance tweaks, and using the tweaks and all cores of the machine. In Table 2.2 we see using the more efficient factorial algorithm

100,000 digits of π	time (seconds)
Basic Chudnovsky algorithm	207
Optimized on 1 process	151
Optimized on 2 process	56
Optimized on 4 process	28
Optimized on 8 process	27

TABLE 2.2
Parralelization and optimization of Chudnovsky's Algorithm

improved the performance by 25% and expanding the sum to use all the cores available on the machine (my laptop has 2 cores and 4 logical processors at 2.8 GHz) sped up the computation by another factor of 5. A bit of manual load balancing had to be implemented as Julia's default @parallel macro splits up terms sequentially and in this computation the hardest terms are always at the end, for large k , so shuffling the terms was necessary so that one processor doesn't get all the hardest terms.

3. Modern Algorithms - Gauss–Legendre algorithm. The Gauss-Legendre algorithm, also known as the Arithmetic-Geometric mean algorithm, or AGM, is an iterative method for computing digits of π where each successive iteration doubles the number of correct digits from before. Due to the extremely fast rate of convergence, very few iterations are required to get record breaking digits of π. For example one would need only 45 iterations to beat the current record of 10 trillion digits of π , although the current record was set by using the Chudnovsky algorithm. The algorithm was originally discovered by Gauss and Legendre in the early 1800's but wasn't used to compute digits of π for over a century until it was re-discovered by Eugene Salamin and Richard Brent in 1976 [8]. The algorithm is as follows:

$$
\begin{aligned}
a_0 &= 1 \\
b_0 &= \frac{1}{\sqrt{2}} \\
s_0 &= \frac{1}{2} \\
&\text{Loop} \\
a_i &= \frac{a_{i-1} + b_{i-1}}{2} \\
b_i &= \sqrt{a_{i-1} b_{i-1}} \\
c_i &= a_i^2 - b_i^2 \\
s_i &= s_{i-1} - 2^i c_i \\
\pi_i &= \frac{2a_i^2}{s_i}
\end{aligned}
$$

With π_i being the approximation of π at the i^{th} iteration.

This algorithm is incredibly fast and powerful, with the most straight forward implementation in Julia able to produce 100,000 digits of π in one second, 30 times faster than Chudnovsky's implementation for computing the same number of digits. In fact I have used this method to fully calculate 10 million digits of π in 273 seconds.

Despite it's stunning simplicity, there are some pitfalls one can run into when

trying to implement it in a language like Julia. In order for this algorithm to converge to π it must be properly seeded by $1/\sqrt{2}$ accurate to as many digits as the final π computation. Accidentally seeding the series with the single or double precision value for $1/\sqrt{2}$ will still show a series that each iteration doubles the converged digits, but the value it converges to will no longer be π ! Additionally the AGM algorithms are inherently floating point in nature so error accumulations due to round off aren't self correcting, so the only way to verify the AGM algorithm is by recomputing π on different hardware or by a different method and comparing digits [9]. The AGM method is also not naturally parallelizible as each iteration needs the previous iteration to continue and access to all digits to full precision, so while it vastly outperforms the Chudnovsky algorithm on a laptop, it doesn't scale as easily to supercomputers.

3.1. AGM Complexity and Performance. The AGM algorithm can be used to compute up to 10,000,000 digits of π in a straightforward Julia implementation in under five minutes. The Julia source code can be found here [11] . The theoretical complexity of using the AGM to calculate n digits of π has been calculated by Richard Brent [12] as follows:

Let $M(n)$ be the complexity of performing a floating point multiplication with n digits of precision, which up to the state-of-the-art algorithms is $O(n \log n \log\log n)$for very large n. With some basic assumptions he derived that division up to n precision can be done in time $4M(n)$. Inverse square roots, like the $1/\sqrt{2}$ used to seed the AGM algorithm can be computed in $\frac{9}{2}M(n)$ time and regular square roots in $\frac{11}{2}M(n)$ time. While Brent calculates the complexity of many more operations of arbitrary precision in his paper, these are sufficient to calculate the complexity of the AGM iteration. Each step of the AGM iteration takes $\frac{15}{2}M(n)$ time, and since each iterations doubles the correct digits we need $\log_2 n$ iterations to achieve the desired n precision. Thus the total complexity of the Gauss-Legendre Method:

$$\text{time to evalute first } n \text{ digits of } \pi \approx \frac{15}{2} M(n) \log_2 n$$

This iteration is not self correcting, so it's not possible to start from lower precision and build up to more and more digits of π like with the Chudnovsky algorithm. Secondly since there are $\log_2 n$ iterations, $O(\log\log n)$ bits of accuracy are lost due to rounding errors even though the algorithm is stable.

The performance of the AGM algorithm and Chudnovsky algorithm can be compared directly in Julia for computing successively bigger digits of π . The results are shown in Figure 3.1. On a single core the AGM gets large number of digits much faster than the Chudnovsky algorithm since it only needs to do very few iterations to get very large precision.

4. Conclusion. Humans have come a long way from guessing or decreeing values of π, to being able to compute millions of digits in the blink of an eye. When one wonders why should anyone want to calculate so many digits of π when no more than 39 are needed for any conceivable calculation (like what's the volume of the universe to within one atom) , one needs only dabble in writing a program to compute a few in order to discover the great fun to be had in this adventure. The AGM and Chudnovsky Algorithms are two fantastic and elegant algorithms for unraveling this irrational number. Both methods work very nicely in Julia, and can be used to get millions of digits of π.

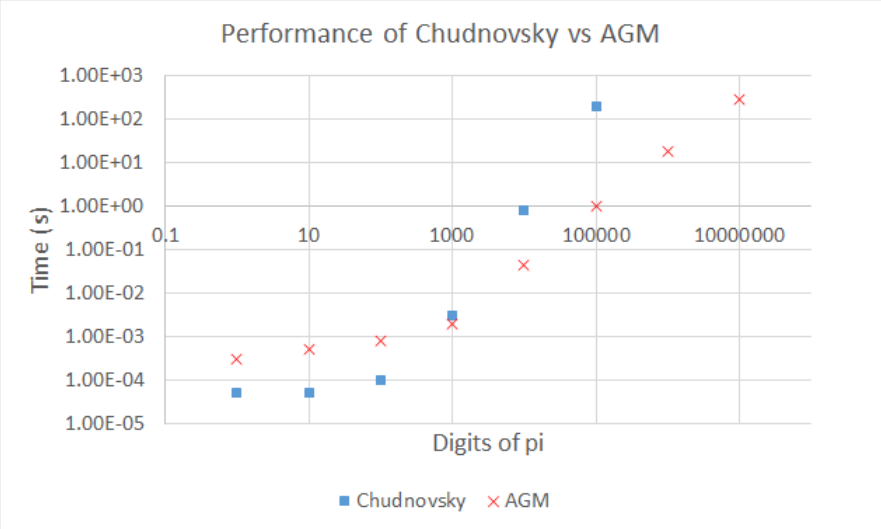

FIG. 3.1. *Performance plot of the Chudnovsky Algorithm and the Arithmetic Geometric Mean Algorithm for computing digits of π from 1 to 10 million digits. The Chudnovsky Algorithm outperforms the AGM up to computing a thousand digits of π , however after that the AGM is clearly much faster, able to reach digits inaccessible to the Chudnovsky Algorithm on a laptop. Both of these tests were run on a single core, so even Chudnovsky could catch up with sufficient parallelization.*

Another algorithm that is worth mentioning when discussing π is the Spigot algorithm, which differs from all the other algorithms in that it is capable of computing a specific (hexadecicmal) digit of π without having to compute any of the previous digits before hand [13]. I did not get a chance to implement and test this class of algorithms in Julia so they won't be covered in any more depth here.

I was able to push the AGM to get 10 million correct digits of π , which are available on github [11] in addition to all the source code. Getting 100 million digits would have taken under an hour but Julia ran out of memory as even the @time commands were showing negative amounts of memory used due to the actual usage overflowing the integer bounds. Attaining even larger values is definitely within the scope of even novice programmers, since 1 billion digits would only take 10 hours, and 10 billion digits would take under two weeks, so it's possible to beat records set as late as 1995 on a normal computer although I was only able to reach the record set in 1983 of 10 million digits, and what fun it was.

REFERENCES

[1] Arndt, Jörg, and Christoph Haenel. Pi-unleashed. Berlin: Springer, 2001. 167,169,189,17. Print.
[2] Smith, David Eugene. History of Mathematics. New York: Dover, 1958. Print.
[3] O'Connor, John J.; Robertson, Edmund F. (1999), "Ghiyath al-Din Jamshid Mas'ud al-Kashi", MacTutor History of Mathematics archive
[4] Borwein, J. M.; Borwein, P. B.; Dilcher, K. (1989). "Pi, Euler Numbers, and Asymptotic Expansions". American Mathematical Monthly 96 (8): 681–687. doi:10.2307/2324715.
[5] Reitwiesner, George (1950). "An ENIAC Determination of pi and e to 2000 Decimal Places". Mathematical Tables and Other Aids to Computation 4 (29): 11–15. doi:10.2307/2002695.
[6] Bailey, David H. et al. The Quest for Pi. Mathematical Intelligencer, vol. 19, no. 1 (Jan. 1997), pg. 50–57
[7] Yee, Alexander; Kondo, Shigeru (2011), 10 Trillion Digits of Pi: A Case Study of summing

Hypergeometric Series to high precision on Multicore Systems, Technical Report, Computer Science Department, University of Illinois.

[8] E. Salamin (1976). "Computation of \pi using arithmetic-geometric mean". Math. Comp. 30 (135): 565–570. doi:10.2307/2005327. MR 0404124.

[9] Chudnovsky, David V.; Chudnovsky, Gregory V. (1989), "The Computation of Classical Constants", Proceedings of the National Academy of Sciences of the United States of America 86 (21): 8178–8182, doi:10.1073/pnas.86.21.8178, ISSN 0027-8424, JSTOR 34831, PMC 298242, PMID 16594075.

[10] "Download 1,000,000,000 Digits of PI." http://micronetsoftware.com/pi_day/ Free Pi. N.p., n.d. Web. 14 Dec. 2013.

[11] Velicanu, Dragos. "Public Velicanu / Pi_18335_final_project." GitHub. N.p., 15 Dec. 2013. Web. 15 Dec. 2013. ¡https://github.com/velicanu/pi_18335_final_project¿.

[12] Brent, Richard (1975), "Multiple-precision zero-finding methods and the complexity of elementary function evaluation", in Traub, J F, Analytic Computational Complexity (New York: Academic Press): 151–176, retrieved 8 September 2007

[13] Rabinowitz, Stanley; Wagon, Stan (1995). "A Spigot Algorithm for the Digits of Pi". American Mathematical Monthly 102 (3): 195–203. doi:10.2307/2975006.