

Fast Calculation of π

Eric P. Nichols
CSCI B503 – Algorithms Design and Analysis
Prof. Purdom

April 21, 2004

Abstract

This report describes my implementation of the Schönage variant of the Gauss-AGM (arithmetic-geometric mean) algorithm to quickly compute an accurate approximation of π , as described in [1]. The algorithm was run using less than 60 seconds of CPU time on a node of the Steel cluster at Indiana University (where each node is a 500MHz UltraSPARC2 processor with 1GB of memory.) The algorithm was coded in the C programming language, with the GNU MultiPrecision Library (GMP). The algorithm calculates a floating-point approximation to π and then outputs integers a, b, c , and d such that $a/b < \pi < c/d$. These integers are computed using the known error bounds for the algorithm. The results are equivalent to an accuracy of 357,654 places of π after the decimal. Please note that I follow the excellent presentations given in [1] and [7] for the derivation of many of the formulas that follow.

Contents

1	Introduction	3
2	The Arithmetic-Geometric Mean (AGM)	3
3	An AGM Algorithm for Calculating π	4
3.1	A Formula for π	4
3.2	The Basic AGM Algorithm for π	4
3.3	The Schönage Variant	5
4	Algorithm Efficiency	6
4.1	Number of Loop Iterations	6
4.2	Time Complexity	7
4.2.1	Floating Point Multiplication in the Inner Loop	7
4.2.2	Floating Point Square Root in the Inner Loop	8
4.2.3	Radix Conversion for Final Output	8
4.2.4	Theoretical Results	8
4.2.5	Empirical Results	9
5	Error Bounds	12
A	Appendix: Derivation of the AGM Formula for π	14
A.1	The Fundamental Relationship between AGM and an Elliptic Integral	14
A.2	Elliptic Integrals	15
A.3	Legendre's Relation	16
A.4	The Formula for π	16
A.5	Derivation of Error Formulas Based on Number of Iterations . . .	16
B	Appendix: Floating-Point Calculation Details	17
B.1	Error at Each Iteration	18
B.2	Error in the Final Result	19
C	Appendix: The First 2800 Digits of π	20
D	Appendix: Source Code	21

1 Introduction

Until the 1970s, all the methods in common use for approximating the value π used algorithms yielding a number of digits linear in the number of computations performed. However, in 1976 Salamin [7] and Brent [3] independently published articles giving a new algorithm for computing π such that the number of digits is quadratic in the number of computations; i.e. the number of digits approximately doubles on each iteration of the algorithm. Surprisingly, this algorithm was likely known by Gauss, whose work on the Arithmetic-Geometric Mean (AGM) and elliptic integrals provides the basis for the calculation. However, his work was ignored or forgotten as a method for fast approximations to π until this 1976 “rediscovery” [1].

Below, I describe the AGM and its properties. Following the method of [7] a formula for π in terms of the AGM is described and the basic algorithm is made explicit. Finally, a simplification to the algorithm to increase speed is described.

2 The Arithmetic-Geometric Mean (AGM)

The AGM is a certain type of mean of two numbers. It differs from other means such as the arithmetic or the geometric mean in that the rule describing it is in the form of a limit of an iterative process rather than a finite sum. The iteration consists of computing the arithmetic mean of the lower and upper numbers, $(a_k + b_k)/2$, as well as the geometric mean between these two numbers, $\sqrt{a_k \cdot b_k}$. That is, to compute $\text{AGM}(a, b)$ one has the initial conditions:

$$\begin{aligned}a_0 &:= a \\ b_0 &:= b\end{aligned}$$

where $a < b$. We also have the iteration rule:

$$\begin{aligned}a_{k+1} &:= \frac{a_k + b_k}{2} \\ b_{k+1} &:= \sqrt{a_k \cdot b_k}\end{aligned}$$

so that

$$\text{AGM}(a, b) = \lim_{k \rightarrow \infty} a_k = \lim_{k \rightarrow \infty} b_k \tag{1}$$

It follows from the definitions that after the first iteration, $\{a_k\}$ is monotone increasing and $\{b_k\}$ is monotone decreasing. We also define an auxiliary sequence $\{c_k\}$:

$$c_{k+1} := \frac{1}{2}(a_k - b_k) \tag{2}$$

$$c_{k+1}^2 := a_{k+1}^2 - b_{k+1}^2 = (a_{k+1} - a_k)^2 \tag{3}$$

This last form is more useful for calculations, so c_k is never calculated explicitly in the algorithm that follows; c_k^2 is always used instead.

3 An AGM Algorithm for Calculating π

Quite surprisingly, the value of the AGM for a particular choice of a and b can relate to the value of π . The proof of this fact relies on results from elliptic integral theory and is presented in appendix A.

3.1 A Formula for π

Gauss's formula for π in terms of the AGM follows:

$$\pi = \frac{2 \operatorname{AGM}^2\left(1, \frac{1}{\sqrt{2}}\right)}{\frac{1}{2} - \sum_{k=1}^{\infty} 2^k c_k^2} \quad (4)$$

Here c_k is as defined in the description of the AGM above.

The error after the Nth iteration is given by:

$$\pi - p_N \leq \frac{\pi^2 2^{N+4} e^{-\pi 2^{N+1}}}{\operatorname{AGM}^2(1, 1/\sqrt{2})} \quad (5)$$

and after the next step,

$$\pi - p_{N+1} \leq \frac{(\pi - p_N)^2}{2^{N+1} \pi^2} \quad (6)$$

The error formulas will be derived in Appendix A.

3.2 The Basic AGM Algorithm for π

As given in [1], the formula for π can be converted in a straightforward fashion to the following algorithm, where t is a temporary variable introduced to store a copy of a_k so that the memory allocated for a_k can be set to the calculated value of a_{k+1} immediately, even though a_k is used in the computation of b_{k+1} and c_{k+1}^2 later on. The algorithm begins with initialization:

$$\begin{aligned} a_0 &:= 1 \\ b_0 &:= 1/\sqrt{2} \\ s_0 &:= 1/2 \end{aligned}$$

Then let k go from 0 to $N-1$ and iterate:

$$\begin{aligned} t &:= a_k \\ a_{k+1} &:= (a_k + b_k)/2 \\ b_{k+1} &:= \sqrt{t \cdot b_k} \\ c_{k+1}^2 &:= (a_{k+1} - t)^2 \\ s_{k+1} &:= s_k - 2^{k+1} c_{k+1}^2 \end{aligned}$$

Finally, calculate the approximation

$$\pi_N = \frac{(a_N + b_N)^2}{2 s_N}$$

3.3 The Schönhage Variant

Because multiplication of two high-precision floating point numbers is a relatively slow operation, Schönhage came up with a simplification to the algorithm above that improves efficiency by converting the multiplication of t and b_k to a more complicated expression only involving sums and powers of 2. The key simplification comes from observing:

$$a_k b_k = 2 \left(a_{k+1}^2 - \frac{1}{4} (a_k^2 + b_k^2) \right) \quad (7)$$

New variables A and B are introduced to store the temporary values a_k^2 and b_k^2 . Initialization step:

$$\begin{aligned} a_0 &:= 1 \\ A_0 &:= 1 \\ B_0 &:= 1/2 \\ s_0 &:= 1/2 \end{aligned}$$

The iteration step is transformed to the following:

$$\begin{aligned} t &:= (A_k + B_k)/2 \\ b_k &:= \sqrt{B_k} \\ a_{k+1} &:= (a_k + b_k)/2 \\ A_{k+1} &:= a_{k+1}^2 \\ B_{k+1} &:= 2(A_{k+1} - t) \\ s_{k+1} &:= s_k - 2^k (B_{k+1} - A_{k+1}) \end{aligned}$$

Now, the approximation to π is calculated with:

$$\pi_N = \frac{(A_N + B_N)}{s_N}$$

Note that this change also speeds things up by moving the initial square root calculation from the initialization step ($b_0 := \sqrt{2}$) to the main loop and by eliminating the squaring operation in the final approximation step. This optimized version is the algorithm used in my implementation. Also, I believe there are typos in the algorithm as presented in [1]; the algorithm above and implemented in my code has been corrected.

Please refer to the following table for a list of the first few results for small numbers of iterations using this algorithm. The first 2800 digits of π are given in Appendix C for reference.

Iteration (N)	π_N
1	3.14...
2	3.1415926...
3	3.141592653589793238...

4 Algorithm Efficiency

4.1 Number of Loop Iterations

Using *Mathematica* to compute with formula (5) given above,

$$\pi - p_N \leq \frac{\pi^2 2^{N+4} e^{-\pi 2^{N+1}}}{\text{AGM}^2(1, 1/\sqrt{2})}$$

with an approximation of the AGM in this formula to 10 digits ($\text{AGM}(1, 1/\sqrt{2}) \approx 0.8472130848$) shows that after 17 iterations the error will be less than or equal to $2.5 * 10^{-357656}$. In other words, the first 357,655 digits after the decimal place will be accurate. (However, due to the method used to convert from floating point to a rational range expressed in lowest terms, this number must be reduced by 1 digit; see below for details. By the other error formula (6) presented earlier:

$$\pi - p_{N+1} \leq \frac{(\pi - p_N)^2}{2^{N+1}\pi^2}$$

we see that the error decreases exponentially in the number of iterations of the algorithm. As the number of digits correct is approximately doubling at each iteration, we see that 19 iterations would be sufficient to compute over 1 million digits of π , as the following table based on formula (5) illustrates:

Iteration (N)	Number of Digits Correct
1	3
2	8
3	19
4	41
5	84
6	171
7	345
8	694
9	1,392
10	2,789
11	5,583
12	11,171
13	22,348
14	44,702
15	89,409
16	178,825
17	357,656
18	715,319
19	1,430,644
20	2,861,296

Experimenting with the algorithm points out an interesting phenomenon: although the algorithm is guaranteed to give results to an accuracy specified

by the formulas and table above, it is possible to get more accurate digits during a final iteration step by performing calculations to greater precision than necessary. For example, if I calculate the number of bits required in calculations for 17 iterations and run the algorithm, I should get 357,656 digits correct. However, if I increase the number of bits by a small amount and run an 18th iteration, the algorithm’s results agree with π to between 357,656 and 715,319 places; increasing the precision gradually increases the number of digits in the 18th iteration until it reaches 715,319 and plateaus.

If running 18 iterations to enough precision to get all 715,319 digits correct takes too much processor time while running 17 iterations allows extra time remaining, this suggests a mechanism to “squeeze” some final digits out of the algorithm on the final pass through the loop. Unfortunately, I have no idea how to prove the error bound for this modified algorithm with the “wrong” amount of precision on the final pass. The given proof is only valid for numbers of digits as given in the table above.

4.2 Time Complexity

A number of experiments and analyses were carried out to investigate the time complexity of the implemented algorithm. We begin by discussing the parts of the algorithm that contribute significantly to the running time and then give empirical and theoretical results, with a goal of describing the running time as a function of the number of digits of π to compute.

The following sections focus on the square root operation in the inner loop of the algorithm, the multiplication to compute a^2 , and the final step of radix conversion from the internal base 2 representation to output the results as a base 10 fraction. The other steps of the algorithm involve additions, subtractions, and multiplications by powers of 2, all of which are insignificant in time complexity compared to the steps highlighted below.

4.2.1 Floating Point Multiplication in the Inner Loop

The GMP documentation goes into great detail about the types of algorithms used to multiply two numbers of high precision, but unfortunately because it can choose which algorithm to use based on some GMP-specific parameters, it is unclear what the exact time complexity is. The documentation mentions running times of $O(D^{1.333})$, $O(D^{1.4})$, and $O(D^{1.465})$, or more generally $O(D^{k/(k-1)})$ and $O(D^{k/(k-2)})$. So, for our purposes we can take the time complexity to be about $O(D^{1.4})$ to keep things simple.

Note that in the documentation the time complexity refers to the number of bits of precision, but because this is related linearly to the number of digits, we can consider D above to be the number of digits.

Now, this multiplication happens N times (the number of iterations) because it is in the inner loop. N is functionally related to the number of digits D via equation (5) above. We can rewrite this in the manner of [7] as follows:

$$D > (\pi/\log(10))2^{N+1} - N \log_{10}(2) - 2 \log_{10}(4\pi/AGM) \quad (8)$$

Ideally, we would now solve for N as a function of D to determine how many times the loop is executed for a particular number of digits. However, this involves non-trivial algebra; perhaps the method of asymptotic iteration could provide a big- O form of the solution. Instead, I will present an argument for the big- O solution based on equation (6) above. That equation shows that the number of accurate digits at least doubles after each step. So, $D = \Omega(2^N)$, or

$$N = O(\lg D) \tag{9}$$

Multiplying the time complexity for a single multiply operation by the number of iterations N yields an approximate time complexity for the multiply operations:

$$T = O(D^{1.4} \lg D) \tag{10}$$

4.2.2 Floating Point Square Root in the Inner Loop

The GMP documentation states that the square root is performed via Karatsuba’s method, which does a small number of multiplication operations to obtain the square root. Thus, the time complexity here is essentially the same as that above, multiplied by a function giving the number of multiplications required for the square root algorithm; unfortunately, there isn’t enough information to determine what this is. However, I get the impression that the number of multiplications is very small. The empirical results do show the time for square root operations apparently growing a bit faster with D than the time for the a^2 multiplications.

4.2.3 Radix Conversion for Final Output

Somewhat suprisingly, I found empirically that for large D , the base 2 to base 10 conversion in GMP takes longer than all the square root operations performed during the calculations. Turning again to the GMP documentation we find a “sub-quadratic” divide-and-conquer algorithm used for large numbers, with a quadratic base-case algorithm used for smaller pieces. However, after several paragraphs of discussion, it says “`mpf_get_str` doesn’t currently use the algorithm described here... This is $O(N^2)$ and is certainly not optimal.” (`mpf_get_str` is the function used to perform the radix conversion.) While it’s unclear why the better algorithm was described at all if not used, the quadratic nature of the time complexity seems in line with the empirical results. For $N > 17$ the radix conversion is the most significant component of the algorithm.

4.2.4 Theoretical Results

Due to the inefficiency of the radix conversion, the overall time complexity of the algorithm is:

$$T = O(N^2) \tag{11}$$

However, if radix conversion could be improved enough the dominating factor could become the square root calculations, with time complexity slightly greater than the following (the uncertainty is due to the unknowns in the square root and multiplication algorithm documentation):

$$T = O(D^{1.4} \lg D) \tag{12}$$

where $D^{1.4}$ should be replaced by the correct form based on which multiplication algorithm is being used in a particular run of the algorithm. Brent [4] gets around the problem by giving the time complexity as $O(M(n) \log n)$, where n is the number of digits and $M(n)$ is the number of operations needed to perform multiplication of numbers of size n .

4.2.5 Empirical Results

The table below gives results for timing the algorithm (and subportions of the algorithm) for different numbers of iterations of the main loop, with the associated increase in floating-point precision required for the given number of iterations. The number of bits of precision to use were calculated by multiplying the number of digits by $\log_2(10)$ and adding the number of “guard digit” bits necessary.

Note: for convenience, the following times are reported as run on the Shark cluster. The SQRT column gives the total time (in milliseconds) used by all square root calculations in the inner loop. The RADIX column gives the time used to convert from base 2 to base 10 and record the resulting integer (the very small times required to compute the four integers a, b, c , and d after radix conversion were also included here, but this extra time was insignificant.)

Iterations (N)	Digits	Bits of Precision	Time (ms)	SQRT	RADIX
1	3	9	1	-	-
2	8	40	1	-	-
3	19	84	1	-	-
4	41	166	1	-	-
5	84	319	1	-	-
6	171	619	1	-	-
7	345	1,209	1	-	-
8	694	2,381	2	-	-
9	1,392	4,714	3	-	-
10	2,789	9,370	7	-	-
11	5,583	18,667	19	-	-
12	11,171	37,247	57	24	9
13	22,348	74,394	182	78	31
14	44,702	148,672	559	280	102
15	89,409	297,205	1,678	855	373
16	178,825	594,260	5,152	2496	1454
17	357,656	1,188,346	17,032	7123	7069
18	715,319	2,376,499	56,345	19584	29687
19	1,430,644	4,752,781	191,313	52440	121062
20	2,861,296	9,505,330	segfault	segfault	segfault

A graph follows to illustrate some of this data; specifically, the total time is plotted with the number of digits. I also performed a nonlinear regression with *Mathematica* using the data from $N = 9$ through $N = 19$ (early values were ignored because there must be factors such as start-up costs preventing the total time from dropping below 1 ms.) I used a model based on the $O(D^{1.4} \lg D)$ result above to illustrate how the running time is affected by the radix conversion. The regression model derived is:

$$T = -2104 + 0.000032183 D^{1.4} \ln(D) \quad (13)$$

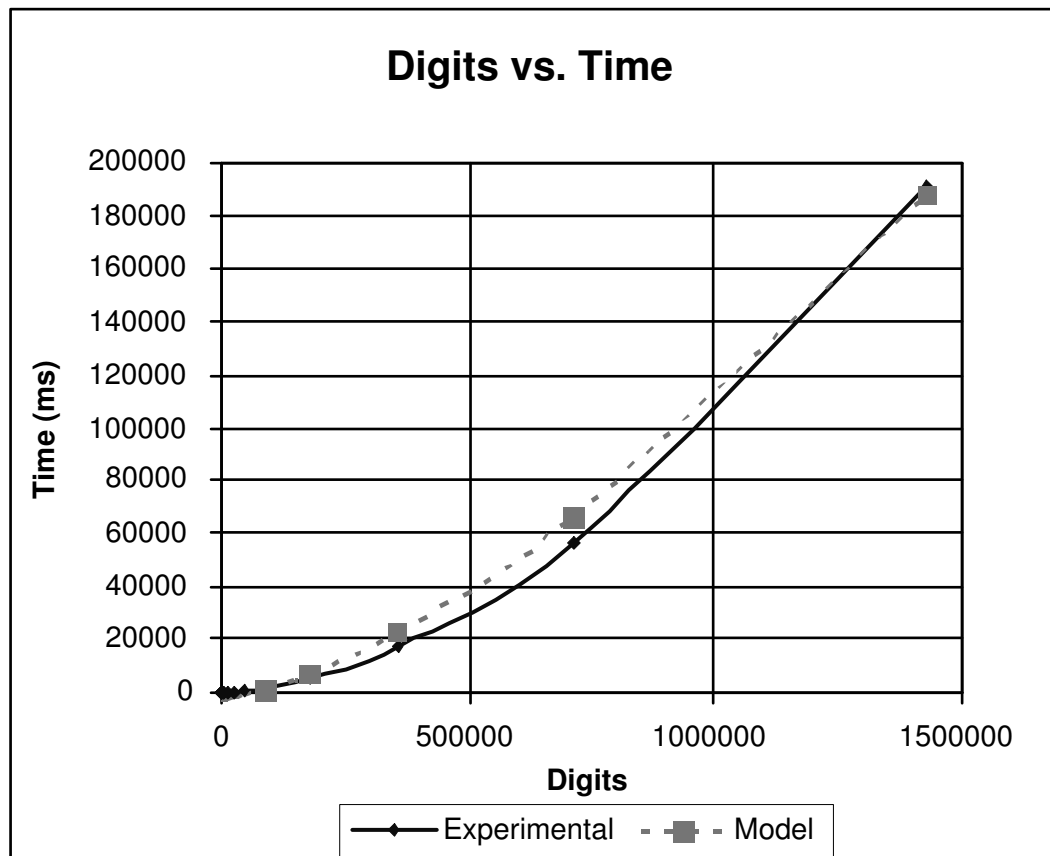


Figure 1: Effect of number of digits on runtime

5 Error Bounds

Although floating point arithmetic is used throughout the algorithm, derivation of rational bounds for π is straightforward due to the natural correspondence between fixed-precision floating point numbers and rationals. Consider the approximation p_i to π produced by the algorithm above where we know that p_i agrees with π to n bits of precision. That is,

$$p_i - 2^{-n} < \pi < p_i + 2^{-n} \tag{14}$$

In order to convert from the internal binary representation to decimal, we calculate $D = \text{Floor}(n/(\lg 10))$ and convert to a decimal representation using D digits. Note that here we are losing a small amount of precision due to the base conversion; this is taken into account by the floor function. Now we have

$$p_i - 10^{-D} < \pi < p_i + 10^{-D} \tag{15}$$

Finally, we can compute integers a, b, c , and d such that the following holds:

$$\frac{a}{b} < \pi < \frac{c}{d} \tag{16}$$

We do so simply by writing the left hand and right hand sides of the inequality as rationals such that the denominator is a power of 10 large enough to write the numerator as an integer. We do this by setting $b = d = 10^D$. a and c can be calculated simply by writing down the digits of p_i without the decimal place and subtracting or adding 1, respectively. For example, the approximation 3.14 with $D = 2$ would result in $a = 313, b = 315$, and $c = d = 100$. As a final step, we decrement a until it is relatively prime to b ; this will happen once the final digit becomes 1, 3, 7, or 9 because 10^D factors into powers of 2 and 5. We also increment c until its final digit is acceptable. This has the effect of decreasing our accurate number of digits by no more than one; this is why I report 357,654 places of accuracy instead of the 357,655 given by the error formula. Also note that this technique was used instead of reducing to lowest terms because reducing to lowest terms was hurting performance substantially.

References

- [1] Jörg Arndt, Christoph Haenel. *π Unleashed*. Springer-Verlag, Berlin Heidelberg, 2001.
- [2] J. Borwein, P. Borwein. *Pi and the AGM*. John Wiley & Sons, 1987.
- [3] R. P. Brent. “Fast Multiple-Precision Evaluation of Elementary Functions”. *Journal of the Association for Computing Machinery*, Vol. 23, No. 2 (Apr., 1976).
- [4] R. P. Brent. “Unrestricted Algorithms for Elementary and Special Functions”. *Information Processing 80*, 1980, 613-619.
- [5] Yasumasa Kanada. “Vectorization of Multiple-Precision Arithmetic Program and 201,326,000 Decimal Digits of π Calculation”. Reprinted in *Pi: A Sourcebook*, Ed. Berggren, Borwein. Springer-Verlag, 1999.
- [6] D. J. Newman. “A Simplified Version of the Fast Algorithms of Brent and Salamin”. *Mathematics of Computation*, Vol. 44, No. 169 (Jan., 1985), 207-210.
- [7] Eugene Salamin. “Computation of π Using Arithmetic-Geometric Mean”. *Mathematics of Computation*, Vol. 30, No. 135 (Jul., 1976), 565-570.

A Appendix: Derivation of the AGM Formula for π

While Gauss's original motivation for relating the AGM algorithm to the value of π is complicated, a relatively straightforward proof shows how the AGM algorithm relates to a certain class of integrals (the elliptic integrals). Then elliptic integral theory can relate the value of a combination of particular elliptic integrals to π using a formula known as Legendre's Relation. Borwein and Borwein [2] provide both the original motivation and the proofs that follow for relating elliptic integrals to the AGM, while I follow Salamin's introduction [7] to the relevant elliptic integral theory and the formula for π .

A.1 The Fundamental Relationship between AGM and an Elliptic Integral

The simple proof relating the AGM algorithm to an elliptic integral rests on showing that the integral is invariant under a certain change of variables. As will be seen, this change of variables is exactly the same as the iteration in the AGM algorithm.

Following [2], define

$$T(a, b) := \frac{2}{\pi} \int_0^{\pi/2} \frac{d\theta}{\sqrt{a^2 \cos^2 \theta + b^2 \sin^2 \theta}} \quad (17)$$

Now, make the change of variables $t := b \tan \theta$, yielding

$$T(a, b) := \frac{1}{\pi} \int_{-\infty}^{\infty} \frac{dt}{\sqrt{(a^2 + t^2)(b^2 + t^2)}} \quad (18)$$

Next, make another change of variables $u := \frac{1}{2}(t - \frac{ab}{t})$, giving T in terms of itself with different arguments:

$$T(a, b) := T\left(\frac{a+b}{2}, \sqrt{ab}\right) \quad (19)$$

This is very reminiscent of the AGM iteration. In fact, if we take terms $\{a_k\}$ and $\{b_k\}$ from the AGM iteration, and use these as the arguments to T , it does not matter which term of the sequence we use for the arguments due to this invariance condition. More precisely, we can say that $T(a_n, b_n)$ is independent of n . In particular, we can write

$$T(a_0, b_0) = T(\text{AGM}(a_0, b_0), \text{AGM}(a_0, b_0)) \quad (20)$$

because the AGM is defined by taking the limit as n goes to ∞ . Finally, note that if a and b are equal as in the previous equation, the integral form (18) simplifies to

$$T(a, a) := \frac{1}{\pi\sqrt{2}} \int_{-\infty}^{\infty} \frac{dt}{\sqrt{a^2 + t^2}} = \frac{1}{a} \quad (21)$$

Combining (20) and (21), we get the desired relationship between the AGM and the elliptic integral T :

$$T(a_0, b_0) = \frac{1}{\text{AGM}(a_0, b_0)} \quad (22)$$

Note that Newman [6] gives a similar derivation of equation (22) that is a bit longer but perhaps easier to follow, based on explicit repeated changes of variables instead of the description of T 's independence of n .

A.2 Elliptic Integrals

The following functions are known as the complete elliptic integrals:

$$K(k) = \int_0^{\pi/2} \frac{dt}{\sqrt{1 - k^2 \sin^2 t}} \quad (23)$$

$$E(k) = \int_0^{\pi/2} \sqrt{1 - k^2 \sin^2 t} dt \quad (24)$$

If we define $k'^2 = 1 - k^2$, then the complimentary elliptic integrals are defined by $K'(k) = K(k')$ and $E'(k) = E(k')$. Also, the symmetric elliptic integrals (of which the function T in the previous section is an example) are defined as follows:

$$I(a, b) = \int_0^{\pi/2} \frac{dt}{\sqrt{a^2 \cos^2 t + b^2 \sin^2 t}} \quad (25)$$

$$J(a, b) = \int_0^{\pi/2} \sqrt{a^2 \cos^2 t + b^2 \sin^2 t} dt \quad (26)$$

The relations between the regular and symmetric elliptic integrals are simply:

$$I(a, b) = \frac{K'\left(\frac{b}{a}\right)}{a} \quad (27)$$

$$J(a, b) = aE'\left(\frac{b}{a}\right) \quad (28)$$

Now we can rewrite the T 's independence of n from the last section as follows:

$$I(a_n, b_n) = I(a_{n+1}, b_{n+1}) \quad (29)$$

Similar logic with J yields:

$$J(a_n, b_n) = 2J(a_{n+1}, b_{n+1}) - a_n b_n I(a_{n+1}, b_{n+1}) \quad (30)$$

We can now give a generalization of equation (22) in terms of I :

$$I(a_0, b_0) = \frac{\pi}{2 \text{AGM}(a_0, b_0)} \quad (31)$$

similarly, equation (30) results in:

$$J(a_0, b_0) = \left(a_0^2 - \frac{1}{2} \sum_{j=0}^{\infty} 2^j c_j^2 \right) I(a_0, b_0) \quad (32)$$

A.3 Legendre's Relation

Elliptic integral theory produces the following fundamental result, known as Legendre's Relation [7]:

$$K(k)E'(k) + K'(k)E(k) - K(k)K'(k) = \frac{\pi}{2} \quad (33)$$

If we define new variables a, b, a', b' such that $(b/a)^2 + (b'/a')^2 = 1$ then we can rewrite this in terms of the symmetric integrals I and J :

$$a^2 I(a, b) J(a', b') + a'^2 I(a', b') J(a, b) - a^2 a'^2 I(a, b) I(a', b') = \frac{\pi a a'}{2} \quad (34)$$

A.4 The Formula for π

Salamin's trick now is to obtain a formula for π by making particular choices for the variables in equation (34) and converting J integrals into I integrals and a summation by using equation (32). Finally, the I integrals are converted to *AGM* limits using (31). The choices $a_0 = a'_0 = 1, b_0 = k, b'_0 = k'$ result in a formula for π in terms of the *AGM*:

$$\pi = \frac{4 \operatorname{AGM}(1, k) \operatorname{AGM}(1, k')}{1 - \sum_{j=1}^{\infty} 2^j (c_j^2 + c_j'^2)} \quad (35)$$

Notice that this works for any choices of k and k' satisfying the constraints. For example, $k = k' = 1/\sqrt{2}$ is reasonable. Choosing $k = k'$ makes the two *AGM* calculations identical, eliminating the need to run the algorithm twice. This provides the main result:

$$\pi = \frac{4 \operatorname{AGM}^2(1, 1/\sqrt{2})}{1 - \sum_{j=1}^{\infty} 2^{j+1} c_j^2} \quad (36)$$

A.5 Derivation of Error Formulas Based on Number of Iterations

Salamin's proof of the error bounds (5) and (6) given above is somewhat lengthy and tedious. Rather than reproducing his work here, I will sketch the outline of his proof and refer the reader to [7] for the details.

Salamin begins by writing an approximation to π based on equation (35):

$$\pi_{nn'} = \frac{4 a_{n+1} a_{n'+1}}{1 - \sum_{j=1}^n 2^j c_j^2 - \sum_{j=1}^{n'} 2^j c_j'^2} \quad (37)$$

Another fraction $\bar{\pi}_{nn'}$ is defined such that its numerator is that of (35) but the denominator is from (37). Using these quantities, he proves the existence of two error values, $e_{nn'}$ and $\bar{e}_{nn'}$ such that the following equations are satisfied:

$$0 < \pi - \bar{\pi}_{nn'} < e_{nn'} \quad (38)$$

$$0 < \pi_{nn'} - \bar{\pi}_{nn'} < \bar{e}_{nn'} \quad (39)$$

$$\bar{e}_{nn'} < e_{nn'} \quad (40)$$

Combining these inequalities yields the desired intermediate result

$$|\pi - \pi_{nn'}| < e_{nn'} \quad (41)$$

An error bound is calculated for (38):

$$e_{nn'} = \frac{\pi^2}{2 \operatorname{agm} \operatorname{agm}'} \left(\sum_{n+2}^{\infty} 2^j a_j c_j + \sum_{n'+2}^{\infty} 2^j a'_j c'_j \right) \quad (42)$$

where $\operatorname{agm} = \operatorname{AGM}(1, k)$ and $\operatorname{agm}' = \operatorname{AGM}(1, k')$

The summations in this formula are simplified and it is converted to the following inequality:

$$e_{nn'} < \frac{2\pi^2}{\operatorname{agm} \operatorname{agm}'} \left(2^n c_{n+2} + 2^{n'} c'_{n'+2} \right) \quad (43)$$

The next step is the most involved: the c_{n+1} term is simplified by considering $\log c_n$ as “the solution to an inhomogeneous linear difference equation” based on the definition of c_n from the AGM algorithm. Solving for c_n gives:

$$\log c_n < -\pi(\operatorname{agm} / \operatorname{agm}') 2^{n-1} + \log 4 \quad (44)$$

The desired error bound for the algorithm’s approximation to π is found by substituting this expression for c_n into (43), yielding:

$$|\pi - \pi_{nn'}| < \frac{8\pi^2}{\operatorname{agm} \operatorname{agm}'} \left[2^n \exp \left(-\pi \frac{\operatorname{agm}}{\operatorname{agm}'} 2^{n+1} \right) + 2^{n'} \exp \left(-\pi \frac{\operatorname{agm}'}{\operatorname{agm}} 2^{n'+1} \right) \right] \quad (45)$$

If $k = k'$ as in the algorithm used to calculate π , this simplifies to the final results (5) and (6), completing the sketch of the proof.

B Appendix: Floating-Point Calculation Details

In order to guarantee that error is not introduced by floating-point rounding or truncation errors, I perform all arithmetic in the algorithm using extra precision. The following subsections calculate the bits of precision lost at each step in the computation. All intermediate computations are performed using the desired number of bits of accuracy plus the number that will be lost due to floating point arithmetic. This is straightforward because as I show below the number of bits lost is related to the square of the number of iterations of the algorithm, and the number of iterations is very relatively small.

B.1 Error at Each Iteration

During each iteration of the inner loop, several different variables are updated; the updates consist of operations including addition, division by 2 or 4 multiplication by positive integer powers of 2, squaring, subtraction, and square root. We consider each of these primitive operations separately and then choose which variable update results in the maximum loss of precision. In all cases we consider floating-point numbers x and y where each number is represented to accuracy ϵ . That is, the error in each number is of the form $x \pm \epsilon$. Also, note that all these operations are implemented in the GMP library so that they work at a user-specified precision. Precision is defined (by Brent) as follows: “We say that an algorithm has precision n if its result is computed with error $O(2^{-n})$ ” [4]. Thus each operation performed at precision n will introduce at most 1 bit of error *due to the operation itself*. The additional error quantified below is *due to the compounding of error* when executing an operation on a number that already has a certain amount of error. In the discussion that follows, we relate precision n to quantify ϵ as follows: $\epsilon = 2^{-n}$.

Addition. Because $(x \pm \epsilon) + (y \pm \epsilon) = x + y \pm 2\epsilon$, we see that our answer has at most $2\epsilon - \epsilon = \epsilon$ error more than was present in either individual argument.

Subtraction. Similarly, we have $(x \pm \epsilon) - (y \pm \epsilon) = x - y \pm 2\epsilon$, so that we still introduce at most ϵ new error.

Division by 2. $(x \pm \epsilon)/2 = x/2 \pm \epsilon/2$, so in this case, division does not introduce any new compound error; the interval of error in the original argument has shrunk by half, although there is still the potential to lose a bit of information due to the division operation itself.

Division by 4. The same argument as above holds; the result of the division is the value $x/4 \pm \epsilon/4$.

Multiplication by powers of 2. $2^i(x \pm \epsilon) = 2^i x \pm 2^i \epsilon$. Thus, multiplication by 2^i introduces at most about $2^i \epsilon$ of error, or i bits of error. (Fortunately, in this algorithm this is a small number of bits in relation to the high precision of computation).

Squaring. $(x \pm \epsilon)^2 = x^2 \pm 2x\epsilon \pm O(\epsilon^2)$, so the squaring process can compound error by as much as $2x\epsilon$. In this algorithm, the number being squared is a term of the a_k series, which is bounded between 1 and $\sqrt{2}$, so the error is at most about 3ϵ .

Square root. A Taylor series expansion of $f(x) = \sqrt{x}$ about a particular point x gives:

$$\sqrt{x \pm \epsilon} = \sqrt{x} \pm \frac{\epsilon}{2\sqrt{x}} \pm O(\epsilon^2)$$

Almost as in the division case, the error range is reduced by the square root operation (although here this is only true when $|x| > 1/4$). In this

algorithm, the square root may be used for arguments ranging between .8 and 1, so the error range does shrink.

By reviewing the algorithm and the results of operations listed above, we see that the step introducing the most error is the one including a multiplication by 2^{k+1} ; this is the calculation of s_{k+1} in the final step of the algorithm loop. We have a possibility of error due to subtraction, multiplication by the power of 2, and another subtraction. The first subtraction can contribute up to 2 bits of error, the multiplication can add $k + 1$ bits of error, and the other subtraction can add another 2 bits of error, so the maximum error bits in the k^{th} step is $2 + (k + 1) + 2 = k + 5$.

B.2 Error in the Final Result

If we do N iterations of the algorithm inner loop, the error for each step can be summed as follows:

$$\sum_{k=1}^N (k + 5) = 5N + \frac{N(N + 1)}{2} \quad (46)$$

This is the number of extra bits of precision required in intermediate calculations to guarantee the desired precision at the termination of the loop. Note that Kanada [5] implies that a much smaller amount of extra precision required, saying that only “20 to 30” guard digits must be used for his AGM calculation of over 200 million digits of π . This seems to imply that only about 1 digit of accuracy is lost for each iteration of the loop. Unfortunately, that article does not provide a proof.

I would like to express my thanks to Abhijit Mahabal for helping me get started with these error ideas. However, any mistakes are certainly mine, not his.

C Appendix: The First 2800 Digits of π

3.141592653589793238462643383279502884197169399375105820974944592
3078164062862089986280348253421170679821480865132823066470938446
0955058223172535940812848111745028410270193852110555964462294895
4930381964428810975665933446128475648233786783165271201909145648
5669234603486104543266482133936072602491412737245870066063155881
7488152092096282925409171536436789259036001133053054882046652138
4146951941511609433057270365759591953092186117381932611793105118
5480744623799627495673518857527248912279381830119491298336733624
4065664308602139494639522473719070217986094370277053921717629317
6752384674818467669405132000568127145263560827785771342757789609
1736371787214684409012249534301465495853710507922796892589235420
1995611212902196086403441815981362977477130996051870721134999999
8372978049951059731732816096318595024459455346908302642522308253
3446850352619311881710100031378387528865875332083814206171776691
4730359825349042875546873115956286388235378759375195778185778053
2171226806613001927876611195909216420198938095257201065485863278
8659361533818279682303019520353018529689957736225994138912497217
7528347913151557485724245415069595082953311686172785588907509838
1754637464939319255060400927701671139009848824012858361603563707
6601047101819429555961989467678374494482553797747268471040475346
4620804668425906949129331367702898915210475216205696602405803815
0193511253382430035587640247496473263914199272604269922796782354
7816360093417216412199245863150302861829745557067498385054945885
8692699569092721079750930295532116534498720275596023648066549911
9881834797753566369807426542527862551818417574672890977772793800
0816470600161452491921732172147723501414419735685481613611573525
5213347574184946843852332390739414333454776241686251898356948556
2099219222184272550254256887671790494601653466804988627232791786
0857843838279679766814541009538837863609506800642251252051173929
8489608412848862694560424196528502221066118630674427862203919494
5047123713786960956364371917287467764657573962413890865832645995
8133904780275900994657640789512694683983525957098258226205224894
0772671947826848260147699090264013639443745530506820349625245174
9399651431429809190659250937221696461515709858387410597885959772
9754989301617539284681382686838689427741559918559252459539594310
4997252468084598727364469584865383673622262609912460805124388439
0451244136549762780797715691435997700129616089441694868555848406
3534220722258284886481584560285060168427394522674676788952521385
2254995466672782398645659611635488623057745649803559363456817432
4112515076069479451096596094025228879710893145669136867228748940
5601015033086179286809208747609178249385890097149096759852613655
4978189312978482168299894872265880485756401427047755513237964145
1523746234364542858444795265867821051141354735739523113427166102
135969536231442952484937187110145765403590279934...

D Appendix: Source Code

```
/******  
*main.c  
*Eric P. Nichols  
*April 20, 2004  
*  
*Description:  
* Calculates digits of Pi quickly (target time = 60 seconds).  
* The results are written to files a.txt, b.txt, c.txt, and d.txt  
* so that  $a/b < \pi < c/d$ .  
*  
* Uses the GNU MultiPrecision Library (GMP) for floating point calculations.  
*  
*****/  
  
#include <stdio.h>  
#include <sys/time.h>  
#include <gmp.h>  
  
// These defines control the number of iterations, the  
// number of bits of precision used in all floating point  
// calculations, and the number of digits of the a and c  
// values to compute in the last step.  
//  
// For the most digits possible on Steel in less than a minute  
// using this code, the following values are used:  
// N = 17, BITS = 1188342, and DIGITS = 357655.  
  
#define N 17  
#define BITS 1188342  
#define DIGITS 357655  
  
int main() {  
  
    unsigned long k; // Number of iterations.  
    int finished;  
  
    mpf_t a, A, b, B, s, t, t2, pi; // Algorithm variables; t and t2 are temps.  
    mp_exp_t exponent; // Used to retrieve the exponent of a GMP float.  
    mpz_t intPi, intA, intB, intC; // Integers calculated in final step.  
    mpz_t mod; // Modulus used to make lowest-terms.  
    FILE *pifile; // An output file handle for writing a, b, c, d.  
    char pi_digits[DIGITS+2]; // A string used in pi digit radix conversion.  
    hrtime_t time_start, time_end; // Measure the performance in nanoseconds.
```

```

// Start the clock.
time_start = gethrtime();

// Initialize the GMP variables.
mpf_init2(a, BITS);
mpf_init2(A, BITS);
mpf_init2(b, BITS);
mpf_init2(B, BITS);
mpf_init2(s, BITS);
mpf_init2(t, BITS);
mpf_init2(t2, BITS);
mpf_init2(pi, BITS);

mpz_init(intPi);
mpz_init(intA);
mpz_init(intB);
mpz_init(intC);
mpz_init(mod);

// Set initial values as specified in the algorithm for a, A, B, and s.
mpf_set_ui(a, 1);
mpf_set_ui(A, 1);
mpf_set_d(B, 0.5);
mpf_set_d(s, 0.5);

printf("\nPi (doing %d iterations):\n", N);

// Loop N times.
for (k = 1; k <= N; k++) {
    // Update screen.
    printf("\n\nIteration #%d\t", k);

    // t = (A+B)/4
    mpf_add(t, A, B);
    mpf_div_2exp(t, t, 2);

    // b = sqrt(B)
    mpf_sqrt(b, B);

    // a = (a+b)/2
    mpf_add(a, a, b);
    mpf_div_2exp(a, a, 1);

    // A = a^2

```

```

mpf_mul(A, a, a);

// B = (A-t)*2
mpf_sub(B, A, t);
mpf_mul_2exp(B, B, 1);

// s = s + (B-A) * 2^k
// calculate t2 = (B-A) * 2^k
mpf_sub(t2, B, A);
mpf_mul_2exp(t2, t2, k);

// s = s + t2;
mpf_add(s, s, t2);

// Output t2 so we can watch the progress.
printf("\nt2: ");
mpf_out_str(NULL, 10, 15, t2);
}

// Final computation step: calculate pi = (A+B) / s.
mpf_add(pi, A, B);
mpf_div(pi, pi, s);

// All that remains is conversion of results to lowest terms and output.

// Write the digits of pi to a string.
mpf_get_str(pi_digits, &exponent, 10, DIGITS, pi);

// Future improvement: Check for truncation of trailing 0s; this is an unfortunate
// behavior of the mpf_get_str function. At the current number of
// digits, the last digit is 5 so it's not a problem.

// Calculate a,b,c, and d as follows:
// a = pi * 10^357655 [- 1, iterated as necessary; see below]
// b = 10^357655
// c = pi * 10^357655 [+ 1...]
// d = b

// Calculate the integer a = pi_digits
mpz_set_str(intA, pi_digits, 10);

// Calculate the integer c = pi_digits = a
mpz_set(intC, intA);

// Calculate the integer b = d = 10^(DIGITS-1)

```

```

mpz_ui_pow_ui(intB, 10, DIGITS - 1);

// Now, we need results in lowest terms. Using the GMP functions to
// canonicalize takes around 30 seconds due to the time needed to
// set up, canonicalize, and retrieve the numerator and denominators.
// Thus, instead of using the GMP rationals, we simply expand the interval
// a slight amount until we reach a and c values that are relatively prime to
// b and d. This is easy because b=d is only divisible by powers of 2 and 5.
// Thus, we reduce a and increase c until each ends in 1, 3, 7, or 9.
// We also force the interval to expand by at least 1 numerator unit in both
// directions to ensure that we are providing upper and lower bounds.
do {
    // Subtract 1 from a.
    mpz_sub_ui(intA, intA, 1);

    // Assume we're done and it's neither even nor divisible by 5.
    finished = 1;

    // Test for an even number.
    mpz_mod_ui(mod, intA, 2);
    if (mpz_sgn(mod) == 0) {
        finished = 0; // It's even; we have to continue.
    } else {
        // Test for divisible by 5.
        mpz_mod_ui(mod, intA, 5);
        if (mpz_sgn(mod) == 0)
            finished = 0; // It's divisible by 5; we have to continue.
    }
} while(finished == 0);

// Now do the same for c.
do {
    // Add 1 to c.
    mpz_add_ui(intC, intC, 1);

    // Assume we're done and it's neither even nor divisible by 5.
    finished = 1;

    // Test for an even number.
    mpz_mod_ui(mod, intC, 2);
    if (mpz_sgn(mod) == 0) {
        finished = 0; // It's even; we have to continue.
    } else {
        // Test for divisible by 5.
        mpz_mod_ui(mod, intC, 5);
        if (mpz_sgn(mod) == 0)

```

```

        finished = 0; // It's divisible by 5; we have to continue.
    }
} while(finished == 0);

// Time usage stops here; the rest is just output to disk.
time_end = gethrtime();

// Display the time used.
printf("\n\nTime used = %lld milliseconds\n", (time_end - time_start) / 1000000);

// Write the bounds to disk in 4 different files.
pifile = fopen("a.txt", "w");
mpz_out_str(pifile, 10, intA);
fclose(pifile);
pifile = fopen("b.txt", "w");
mpz_out_str(pifile, 10, intB);
fclose(pifile);
pifile = fopen("c.txt", "w");
mpz_out_str(pifile, 10, intC);
fclose(pifile);
pifile = fopen("d.txt", "w");
mpz_out_str(pifile, 10, intB);
fclose(pifile);

// Finish.
printf("\nDone!\n\n");

// Clean up.
mpf_clear(a);
mpf_clear(A);
mpf_clear(b);
mpf_clear(B);
mpf_clear(s);
mpf_clear(t);
mpf_clear(t2);
mpf_clear(pi);
mpz_clear(intPi);
mpz_clear(intA);
mpz_clear(intB);
mpz_clear(intC);
mpz_clear(mod);

// Return.
return 0;
}

```