Practical Implementation of π Algorithms.

By Henrik Vestermark (hve@hvks.com)

Abstract

This paper examines the practical implementation of algorithms for computing π at arbitrary precision using the author's float_precision C++ library. Four families of algorithms are covered, each with C++ source code and performance benchmarks.

Chapter 1 presents the Newton iteration and a ninth-order fixed-point method derived from the truncated arcsin series, both with dynamic precision scheduling. Chapter 2 covers the Ramanujan, Chudnovsky, and Borwein infinite series, each implemented both sequentially and with binary splitting. The binary splitting method reduces the cost of the Ramanujan and Chudnovsky series from $O(p \cdot M(p))$ to $O(M(p) \cdot \log^2(p))$ and is the dominant method in practice, underlying the 2026 record computation of 314 trillion digits of π. Chapter 3 covers AGM-based iterative methods, including Brent-Salamin, Gauss-Legendre, and several Borwein variants of cubic, quartic, quintic, and nonic order. Benchmarks show that Brent-Salamin outperforms all higher-order Borwein variants despite their superior convergence rates. Chapter 4 presents the spigot algorithm family as a self-contained integer-arithmetic alternative suitable for modest precision, without requiring an arbitrary-precision library.

Nine appendices provide rigorous error-bound analyses of the principal implementations, deriving guard-digit requirements for each series and confirming that the unified formula $g = \lceil \log_{10}(p) \rceil + 2$ is sufficient for all four sequential series. The Chudnovsky binary splitting method is the recommended algorithm for high-precision computation of π at all levels of precision.

Introduction:

We have more or less always had the constant π readily available in our handheld calculators, Excel spreadsheets, or as a hardware instruction in the CPU.

However, it is usually only available at the Calculator's native precision or the IEEE 754 standard, which provides access to the 64-bit floating-point constant of π with approx. 15-digit accuracy.

However, if we want to venture outside this limited range of precision, we are on our own, meaning we have to resort to any of the many available formulas for finding π with higher precision.

Practical implementation of π Algorithms

In this paper, we examine the various methods available to us and, as usual, we explore the practical implementation of calculating π to higher precision than the IEEE 754 floating-point standard provides.

Before we do that, we will first establish a number of popular algorithms for finding π, using the standard IEEE754 floating-point calculation as an example, and then later show the results when these algorithms are applied using arbitrary-precision arithmetic.

The paper is divided into four sections. Section 1 calculates π using a classic Newton iteration, and in Section 2, we look at the infinite series of Ramanujan, Chudnovsky, and the Borwein brothers. In section 3, we looked at higher-order iteration to calculate π, which has been dominated by the Borwein brothers and their research, and finally, in section 4, we introduce the bounded spigot algorithm as an alternative way of generating π, which many times does not require us to resort to arbitrary precision arithmetic. As always, we list C++ source code for the practical implementation of these algorithms.

This paper is part of a series of arbitrary-precision papers that describe methods, implementation details, and optimization techniques. These papers can be found on my website at www.hvks.com/Numerical/papers.html and are listed below:

1. Fast arbitrary precision integer multiplication. HVE Fast arbitrary precision integer multiplication
2. Fast Computation of Math Constants in arbitrary precision. HVE Fast Gamma, Beta, Error, and Zeta functions for arbitrary precision.
3. Fast Gamma, Beta, Error, and Zeta functions for arbitrary precision. HVE Fast Gamma, Beta, Error, and Zeta functions for arbitrary precision.
4. Fast Square Root and inverse calculation for arbitrary precision math. HVE Fast Square Root & inverse calculation for arbitrary precision
5. Fast Exponential calculation for arbitrary precision math. HVE Fast Exp() calculation for arbitrary precision
6. Fast logarithm calculation for arbitrary precision math. HVE Fast Log() calculation for arbitrary precision
7. Practical implementation of Spigot Algorithms for Transcendental Constants. Practical implementation of Spigot Algorithms for transcendental constants
8. Practical implementation of π algorithms. HVE Practical implementation of PI Algorithms

9. Fast Trigonometric function for arbitrary precision. HVE Fast Trigonometric calculation for arbitrary precision

10. Fast Hyperbolic functions for arbitrary precision. HVE Fast Hyperbolic calculation for arbitrary precision

11. Fast conversion from arbitrary precision number to a string. HVE Fast conversion from arbitrary precision to string

12. Fast conversion from a decimal string to an arbitrary precision number. HVE Fast conversion from string to arbitrary precision

13. Fast Computation of Stirling's numbers in arbitrary precision. HVE Fast Computation of Stirling numbers in arbitrary precision

14. Fast Prime computation in arbitrary precision. HVE Fast Prime Computation in arbitrary precision

15. Fast PRNG in arbitrary precision. HVE Fast PRNG in arbitrary precision

16. Fast Fibonacci sequence in arbitrary precision. HVE Fast Fibonacci in arbitrary precision

Change log

For version 13 April 2026. Sections 1-3 were revised extensively, and Appendices B to I were added, which provide rigorous bounds on the error of the various methods presented here.

For Version 1 January 2025. I corrected some typos.

For version February 7, 2023. Correcting grammars.

For the January 17, 2023 version, we have added the relative cost of the binary splitting method to the Appendix.

For version August 2022. We have:

- Expanded the general introduction about the Author's arbitrary precision library packages.
- In general, all performance charts have been updated to reflect the newer binary version of the author's arbitrary precision libraries.
- Added the binary splitting method for the Ramanujan, Chudnovsky, Borwein25, and Borwein50 infinite series.
- Added the Borwein 1993 Infinite series, producing approx. 50 digits per iteration.
- The Gosper spigot algorithm has been added.

- Added an Appendix where the Binary splitting method is derived for Ramanujan, Chudnovsky, and Borwein

Practical implementation of π Algorithms

Contents

Practical Implementation of π Algorithms.....1
Practical implementation of π Algorithms.....1
Abstract.....1
Introduction:.....1
Change log.....3
The Arbitrary precision library.....9
Internal format for float_precision variables.....10
Normalized numbers.....10
A Historical Perspective on the Computation of π.....11
Ancient Approximations.....11
The Series Era.....12
The Age of Computers and the Quadratic Barrier.....12
Higher Order Methods and the Borwein Brothers.....13
The Chudnovsky Algorithm and Binary Splitting.....14
The Modern Era and World Records.....14
Introduction.....15
Chapter 1: Newton and 9th-order fixed-point Methods for π.....15
Chapter 2: Infinite Series for π.....16
Chapter 3: Higher Order Algorithms for π.....16
Chapter 4: The Spigot Algorithm.....17
1. Newton's & the 9th order fixed-point method for calculating π.....18
Newton (Cubic convergence).....20
Algorithm 1.1 Standard Newton.....20
Fixed-point 9th order convergence.....21
Algorithm 1.2 ninth-order fixed-point method.....21
Improvement of the Newton method?.....22
Algorithm 1.3 Newton Standard with Dynamic Precision.....22
Algorithm 1.4 Newton ninth-order convergence using dynamic precision.....22
Performance.....23
Recommendation for Newton's methods for generating π.....24
2. Infinite series for π.....24
Chudnovsky brothers.....27
Algorithm 2.2 Chudnovsky Infinite series.....29
Borwein Brothers.....30
Algorithm 2.3 Borwein 25.....32
Borwein 50.....33
Algorithm 2.4 Borwein 50.....36
The Binary splitting method for Ramanujan and Chudnovsky series.....37
Binary Splitting of the Ramanujan Infinite Series.....37
Binary splitting of the Ramanujan infinite series.....39
Algorithm 2.5 Binary splitting for Ramanujan π.....40
Binary splitting of the Chudnovsky infinite series.....41
Algorithm 2.6 Binary splitting for Chudnovsky π.....42
Binary splitting of the Borwein25 infinite series.....43
Algorithm 2.7 Binary splitting for Borwein25 π.....44

Binary splitting of the Borwein50 infinite series....45
Algorithm 2.8 Binary splitting for Borwein50 π....46
Speed Comparison of the Infinite Series....48
Recommendation for the Infinite Series....50
3. Higher-order algorithm for π....50
Brent-Salamin method for π....51
Algorithm 3.1 Brent-Salamin....52
Gauss-Legendre method for π....52
Algorithm 3.2 Gauss-Legendre....53
Borwein Brothers Algorithms for PI....53
Borwein Quadratic Algorithm 2.1 from 1987....54
Algorithm 3.3 Borwein Quadratic 2.1....54
Borwein Quadratic 1984....55
Algorithm 3.4 Borwein Quadratic 1984....56
Borwein Quadratic algorithm 1985....56
Algorithm 3.5 Borwein Quadratic 1985....57
Borwein Cubic 1991....58
Algorithm 3.6 Borwein Cubic 1991....58
Borwein Quintic (fifth order) Algorithm....59
Algorithm 3.7 Borwein quintic (fifth order convergence)....60
Borwein Nonic (ninth order) algorithm....61
Algorithm 3.8 Borwein Nonic (9th order convergence)....62
Performance of higher-order methods for π....63
Recommendation for higher-order π....64
4. Spigot Algorithm....65
Algorithm 4.1 Gibbons spigot....68
Algorithm 4.2 64-bit Spigot....70
Gosper formula for π....72
Algorithm 4.3 Gosper 64-bit....75
Reference....79
Appendix A....80
The Binary Recursion algorithm for the three-variable splitting....80
Deriving the Binary splitting method for Ramanujan π....80
Deriving the Binary splitting method for Chudnovsky π....82
Deriving the Binary splitting method for Borwein25 π....84
Deriving the Binary splitting method for Borwein50 π....85
Appendix B....88
Derivation of the Higher-Order Sine Iterations for π....88
Setting Up the Error Equation....88
Constructing Higher-Order Iterations....88
The Connection to the arcsin Series....89
Horner Form and Single sine Evaluation....90
Is There a Practical Benefit Beyond Ninth Order?....91
Appendix C....92
Guard Digit Analysis for the Ramanujan Series....92
The Series and Its Convergence Rate....92

Structure of the Denominator Sum....92
Sources of Rounding Error....93
Combined Error in the Denominator Sum....94
Deriving the Guard Digit Requirement....94
Numerical Values....94
Summary of the Error Budget....95
Appendix D....96
Guard Digit Analysis for the Chudnovsky Series....96
The Series and Its Convergence Rate....96
Structure of the Denominator Sum....96
Sources of Rounding Error....97
Combined Error in the Denominator Sum....97
Numerical Values....98
Comparison with the Ramanujan Series....98
Summary of the Error Budget....99
Appendix E....99
Guard Digit Analysis for the Borwein 25-Digit Series....99
The Series and Its Convergence Rate....100
Numerical Values of the Constants....100
Structure of the Denominator Sum....101
Sources of Rounding Error....101
Combined Error in the Denominator Sum....101
Deriving the Guard Digit Requirement....102
Numerical Values....102
Comparison Across All Three Series....102
Appendix F....103
Guard Digit Analysis for the Borwein 50-Digit Series....103
The Series and Its Convergence Rate....103
Structure of the Denominator Sum....104
Sources of Rounding Error....104
Combined Error in the Denominator Sum....105
Deriving the Guard Digit Requirement....105
Unified Comparison Across All Four Series....105
Appendix G....106
Guard Digit Requirement for the Borwein 25 Binary Splitting Implementation....106
Error Accumulation in the Recursion Tree....106
Deriving the Guard Digit Requirement....107
Comparison with Pure Integer Binary Splitting....107
Appendix H....108
Guard Digit Requirement for the Borwein 50 Binary Splitting Implementation....108
Error Accumulation in the Recursion Tree....108
Deriving the Guard Digit Requirement....108
Comparison of Borwein 25 and Borwein 50 Binary Splitting....109
Guard Digit Analysis for the Brent-Salamin Algorithm....109
Appendix I....109
Iteration Count....109

Practical implementation of π Algorithms

Operations Per Iteration.. 109
Error Accumulation in the Partial Sum...110
Error Accumulation in asq.. 110
Total Error in π.. 110
Deriving the Guard Digit Requirement...110
Verification of the Implementation Formula..111

Practical implementation of π Algorithms

The Arbitrary precision library

If you are already familiar with the arbitrary precision library, you can skip this section.

To understand the C++ code and text, we have to highlight a few features of the arbitrary-precision library, whose class name is *float_precision*. Instead of declaring a variable with a float or double, you just replace the type name with *float_precision*. E.g.

```
float_precision f;  // Declare an arbitrary precision float with 20 decimal digits precision
```

You can add a few parameters to the declaration. The first is the optional initial value, and the second is the optional parameter specifying the floating-point precision. The native type of a *float* has a fixed size of 4 bytes, and a double has 8 bytes. Since this precision can be arbitrary, we can declare the desired precision as the number of decimal digits to use when dealing with the variable. E.g.

```
float_precision fp(4.5);  // Initialize it to 4.5 with default 20 digits precision
float_precision fp(6.5,10000); // Initialize it to 6.5 with a precision of 10,000 digits
```

The precision of a variable can be dynamic and change throughout the code, which is very handy for manipulating the variable. To change or set the precision, you can call the method .precision(), E.g

```
f.precision(100000);              // Change the precision to 100,000 digits
f.precision(fp.precision()-10);  // Lower the precision with 10 digits
f.precision(fp.precision()+20); // Increase precision with 20 digits
```

There is another method for manipulating the exponents of the variables. The method is called .exponent() and returns or sets the exponent as a power of two (as with our regular built-in types float and double). E.g.

```
f.exponent();           // Return the exponent as 2^e
f.exponent(0)           // Remove the exponent
f.exponen(16)           // Set the exponent to 2^16
```

There is a second way to manipulate the exponent, and that is the class method. .adjustExponent(). This method just adds the parameter to the internal variable that holds the exponent of the float_precision variable. E.g.

```
f.adjustExponent(+1); // Add 1 to the exponent, the same as multiplying the number by 2.
f.adjustExponent(-1);  // Subtract 1 from the exponent, the same as dividing the number by 2.
```

This allows very fast multiplication or division by any power of two.

The method .iszero() returns true if the float_precision number is zero, otherwise false.

There is an additional method(), but I will refer to the user manual for the arbitrary-precision math package for details.

All the normal operators and library calls that work with the built-in type float or double will also work with the float_precision type using the same name and calling parameters.

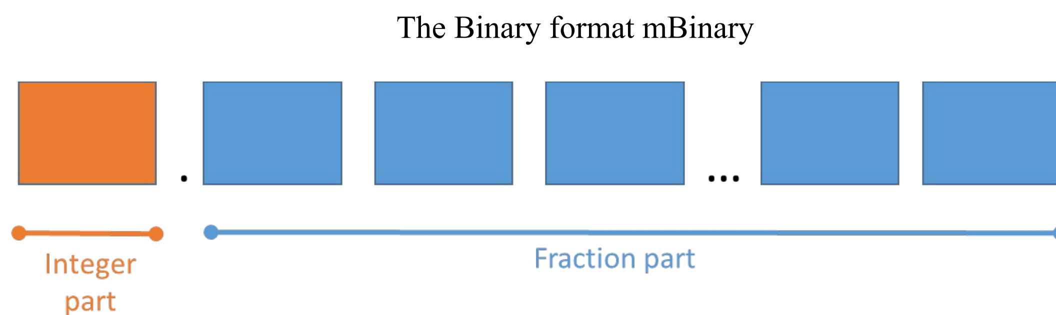
Internal format for float_precision variables

For the internal layout of the arbitrary precision number, we are using the STL vector library declared as:

```
vector<uintmax_t> mBinary;
```

uintmax_t is mostly a 64-bit quantity on most systems, so we use a vector of 64-bit unsigned integers to store our floating-point precision number.

The method .size() returns the number of internal vector entries needed to hold the number.



- The binary format consist of an unlimited number of 64bit unsigned integer blocks.
- One block in front of the period sign '.' (the integer part of the number)
- Zero or more blocks of fractions after the '.' (the fraction sign of the number)
- The binary number is stored in a STL vector class and defined
    - vector<uintmax_t> mBinary;
- There is always one entry in the mBinary vector.
- Size of vector is always >=1
- A Number is always stored normalized. E.g. the integer part is 1 or zero
- The sign, exponent, precision, rounding mode is stored in separate class fields.

There are other internal class variables, such as the sign, exponent, precision, and rounding mode, but they are not important for understanding the code segments.

Normalized numbers

A float_precision variable is always stored as a normalized number with a one in the integer portion of the number. The only exception is zero, which is stored as zero. Furthermore, a normalized number has no trailing zeros.

For more details, see [1].

A Historical Perspective on the Computation of π

The number π, defined as the ratio of a circle's circumference to its diameter, has occupied mathematicians for nearly four thousand years. What began as a practical engineering approximation evolved into one of the deepest problems in the history of mathematics, ultimately driving the development of entirely new branches of analysis, number theory, and computational algorithms. This section traces that evolution, from the earliest known approximations to the modern algorithms capable of computing π to trillions of decimal digits.

Ancient Approximations

The earliest recorded approximations of π date to around 1900 BC. The Babylonians used the value 25/8 = 3.125, while the Rhind Mathematical Papyrus from ancient Egypt, dated to approximately 1650 BC, implies a value of $(16/9)^2 \approx 3.1605$. Both values are accurate to within one percent, which was entirely sufficient for the architectural and engineering purposes of the time.

The first rigorous mathematical treatment appears in the work of Archimedes of Syracuse around 250 BC. Rather than measuring, Archimedes established bounds by inscribing and circumscribing regular polygons around a circle. Starting with hexagons and doubling the number of sides repeatedly up to 96-gons, he proved that $3 + 10/71 < \pi < 3 + 1/7$, giving a lower bound of approximately 3.1408 and an upper bound of approximately 3.1429. This polygon method was not merely a numerical trick: it was a rigorous proof, and the approach of successively refining geometric approximations would remain the dominant method for nearly two thousand years.

In China, Liu Hui independently developed a similar polygon method around 263 AD, reaching 3.14159 using a 3072-gon. The most impressive ancient result came from Zu Chongzhi around 480 AD, who obtained the rational approximation $355/113 \approx 3.14159265$, correct to six decimal places. This remarkable result stood as the most accurate known value of π for nearly a thousand years, and the approximation 355/113 remains useful today, as it is accurate to 6 decimal places and expressed as a simple fraction.

Practical implementation of π Algorithms

The Series Era

The invention of calculus in the late seventeenth century opened an entirely new approach to computing π. Rather than geometric approximations, it became possible to express π as the limit of an infinite series.

The first such formula was the Leibniz-Gregory series, discovered independently by James Gregory in 1671 and Gottfried Leibniz in 1674. It states that $\pi/4 = 1 - 1/3 + 1/5 - 1/7 + \cdots$, which follows directly from the Taylor series expansion of arctan(x) evaluated at $x = 1$. While elegant, this series converges far too slowly for practical computational use; obtaining even 10 correct digits requires millions of terms. The key insight, however, was that *arctan* identities could be used to shift the argument away from 1 toward smaller values where convergence is far faster.

John Machin exploited this in 1706 with the formula $\pi/4 = 4\cdot\arctan(1/5) - \arctan(1/239)$. By evaluating two arctangent series at small arguments, the series converged much more rapidly, and Machin was able to compute 100 decimal digits of π by hand using this formula. It was the first time π had been computed beyond a few dozen digits, and variations of Machin-type formulas remained the basis for hand computation and early computer computation for the next two and a half centuries.

The nineteenth century saw a steady accumulation of digits through Machin-type formulas, culminating in William Shanks computing 707 digits in 1873, a feat that took him many years of hand calculation. It was later discovered, in 1944, that Shanks had made an error at digit 528, rendering all subsequent digits incorrect. This episode illustrated both the remarkable human effort invested in computing π and the fundamental fragility of such long manual calculations.

The fundamental limitation of all arctangent-based methods is computational complexity. To compute p decimal digits of π, the series must be evaluated to sufficient depth, and each term requires arithmetic on numbers of increasing size. The overall cost scales as $O(p^2)$, meaning that doubling the number of digits requires roughly four times the work. This quadratic barrier would not be broken until the late twentieth century.

The Age of Computers and the Quadratic Barrier

The arrival of electronic computers in the 1950s dramatically accelerated the computation of π. Still, for the first two decades, the algorithms remained essentially the same arctangent-based series that Machin had used. In 1949, ENIAC computed 2037 digits in 70 hours using a Machin-type formula. By 1967, a CDC 6600 had reached 500,000 digits, still using arctangent series. Computers made arithmetic fast, but the algorithms themselves were not yet fundamentally better.

The conceptual breakthrough came in 1976, when Eugene Salamin and Richard Brent independently discovered an algorithm for π based on the arithmetic-geometric mean

(AGM). The AGM of two numbers a and b is computed by iterating the recurrences $a_{n+1} = (a_n + b_n)/2$ and $b_{n+1} = \sqrt{(a_n b_n)}$ simultaneously. The sequence converges quadratically, meaning each iteration roughly doubles the number of correct digits. Salamin and Brent showed how to express π in terms of the AGM limit, yielding an iteration that doubles the number of correct digits with each step. Starting from just a few digits of precision, only about 30 iterations are required to reach one billion correct digits. This was a fundamental change in the scaling behavior: the Brent-Salamin algorithm computes p digits in $O(M(p)\cdot\log p)$ operations, where $M(p)$ is the cost of multiplying two p-digit numbers.

An equivalent formulation was developed much earlier by Carl Friedrich Gauss, in a result known as the Gauss-Legendre algorithm. Gauss had studied the AGM in detail around 1800 and understood its connection to elliptic integrals. Still, the computational significance for computing π was not exploited until Salamin and Brent brought these threads together. The Gauss-Legendre and Brent-Salamin algorithms are closely related and have essentially identical performance in practice, both achieving the same quadratic convergence per iteration.

Higher Order Methods and the Borwein Brothers

Following Salamin and Brent's discovery, the brothers Jonathan and Peter Borwein made a series of contributions throughout the 1980s that significantly enriched the landscape of π algorithms. Where Brent-Salamin offers quadratic convergence, meaning the number of correct digits doubles per iteration, the Borwein brothers constructed algorithms with cubic, quartic, quintic, and even nonic (ninth-order) convergence. Their 1984 quartic algorithm quadruples the number of correct digits per step, while their 1991 nonic algorithm multiplies the number of correct digits by 9 per iteration.

These higher-order methods are mathematically elegant and, on first inspection, appear to offer dramatic improvements. In practice, however, the situation is more subtle. Each iteration of a higher-order algorithm requires a proportionally larger number of arithmetic operations, specifically additional square roots and multiplications. A cubic iteration requires two square roots per step instead of one, a quintic iteration requires three or four, and a nonic iteration requires five or six. The gain in convergence rate is therefore largely offset by the increased per-step cost, and benchmarking consistently shows that Brent-Salamin and Gauss-Legendre outperform all higher-order Borwein variants by a factor of roughly 2 to 2.5. This is a clear practical demonstration of a general principle in numerical computing: a higher theoretical convergence order does not guarantee better wall-clock performance once the full cost of each step is accounted for.

Practical implementation of π Algorithms

The Chudnovsky Algorithm and Binary Splitting

In 1987, David and Gregory Chudnovsky introduced a formula for π based on Ramanujan's earlier work on modular equations and hypergeometric series. The Chudnovsky formula is a rapidly converging series in which each additional term contributes approximately 14.18 decimal digits of accuracy. For comparison, the earlier Ramanujan series contributes roughly 8 digits per term. The Chudnovsky formula is the basis for nearly all modern world record computations of π.

The Chudnovsky series, taken alone, still has $O(p^2)$ computational cost due to the need to evaluate a large number of terms, each involving high-precision arithmetic. The key to making it competitive with AGM-based methods is a technique called binary splitting. Rather than evaluating the terms of the series sequentially at full precision, binary splitting recursively splits the summation range in half, evaluating each half at a lower intermediate precision and combining the results. This reduces the cost of evaluating the full series from $O(p^2)$ to $O(M(p)\cdot\log^2(p))$, matching the scaling of the AGM methods while benefiting from the Chudnovsky formula's very rapid per-term convergence. The crossover point at which Chudnovsky with binary splitting overtakes Brent-Salamin depends on implementation details and hardware. Still, for very high precision computations, the faster per-term convergence of Chudnovsky tends to give it an edge.

It is worth noting that spigot algorithms, of which the best known are due to Stanley Rabinowitz and Stan Wagon (1995) and the more efficient variant by Jeremy Gibbons (2004), take a fundamentally different approach. Rather than computing all p digits at once, spigot algorithms produce the decimal digits of π sequentially, one at a time, without requiring arbitrary precision arithmetic in principle. However, the bounded spigot algorithms that avoid arbitrary-precision arithmetic are inherently $O(p^2)$ overall and cannot be restructured to exploit binary splitting or dynamic precision scheduling. They remain an interesting computational curiosity but are not suitable as the foundation for a high-performance arbitrary precision library.

The Modern Era and World Records

From the 1990s onwards, the history of π computation became a story of hardware and software engineering as much as algorithmic innovation. The Japanese mathematician Yasumasa Kanada and his team set a long series of records through the 1990s and early 2000s using Brent-Salamin and Gauss-Legendre on supercomputers, reaching 206 billion digits in 1999. The software tool y-cruncher, developed by Alexander Yee, brought world-record computation within reach of consumer hardware by combining the Chudnovsky algorithm with binary splitting and highly optimized multi-threading. In 2022, a team at Google set the record for 100 trillion decimal digits of π using y-cruncher running on Google Cloud infrastructure, requiring approximately 157 days of computation. As of March 14, 2026, the record stands at 314 trillion digits.

Practical implementation of π Algorithms

The trajectory from Archimedes' 96-gon to 100 trillion digits spans nearly 2300 years. It reflects the entire arc of mathematical development: from geometric reasoning, through the invention of calculus and infinite series, through the discovery of AGM-based quadratic convergence, to modern implementations combining hypergeometric series, binary splitting, and fast multiplication algorithms such as Schönhage-Strassen and Harvey-Hörmann. The algorithms described in the remaining sections of this paper sit within this lineage, and understanding the historical context helps clarify why certain methods are preferred in practice and why the apparent promise of higher convergence order does not always translate into better computational performance.

Introduction

This paper examines various algorithms for computing π at arbitrary precision, with the practical aim of determining which methods are best suited for implementation in an arbitrary-precision library. The historical context established in the preceding section shows that the landscape of π algorithms is rich and spans centuries of mathematical development. Not every algorithm that is mathematically elegant or theoretically fast translates into an efficient implementation, and one of the central themes of this paper is the gap between convergence order on paper and actual computational performance in practice. The paper is therefore structured to follow the historical arc of algorithm development, beginning with the oldest approaches and progressing toward the methods that underpin modern record computations, with C++ implementations provided throughout, using the float_precision arbitrary-precision library.

The paper is divided into four chapters, each addressing a distinct family of algorithms. The following paragraphs outline what each chapter covers and what the reader should take away from it.

Chapter 1: Newton and 9th-order fixed-point Methods for π

The first chapter examines the Newton iteration approach to computing π. The foundational idea is that since $\sin(\pi) = 0$, one can apply Newton's root-finding iteration to $f(x) = \sin(x)$ to converge toward π. Starting from an initial estimate near 3, the basic iteration $x_{n+1} = x_n + \sin(x_n)$ achieves quadratic convergence, doubling the number of correct digits with each step. A key practical observation is that the $\cos(x)$ term in the full Newton update tends to 1 near π and can be dropped without affecting the convergence order, cutting the cost of each iteration roughly in half since evaluating $\sin(x)$ alone is cheaper than evaluating both trigonometric functions.

The chapter then explores whether higher-order root-finding schemes can improve on this. By constructing a ninth-order fixed-point iteration based on the truncated arcsin series applied to $\sin(x)$, it is possible to multiply the number of correct digits by nine per step rather than three (explained later on why it is not two). This is attractive in principle,

but the decisive limitation of all Newton-based or householder methods for computing π is that every iteration requires the evaluation of sin(x) at high precision. The trigonometric functions are themselves expensive to compute at arbitrary precision, requiring internal series evaluations, and this inner cost dominates the total runtime for large digit counts. As the performance comparison at the end of the chapter shows, even the ninth-order fixed-point method cannot match the scaling of algorithms that rely only on the four basic arithmetic operations and square root extraction. Newton-based or ninth-order methods are therefore not the preferred route for generating π at high precision, though they serve well as an introduction to iterative convergence and provide a useful baseline for comparison.

Chapter 2: Infinite Series for π

The second chapter covers the family of rapidly converging infinite series for π, including the formulas due to Ramanujan, the Chudnovsky brothers, and Borwein. These series share a common structure: each term in the summation contributes a fixed number of new correct decimal digits, so the number of terms required to reach p digits is simply p divided by the digits-per-term of the particular series. Ramanujan's 1914 formula adds approximately 8 decimal digits per term, while the Chudnovsky formula from 1987 adds approximately 14.18 digits per term, making it the most efficient of the classical hypergeometric series.

Taken alone, these series are still limited by the fact that evaluating many terms at full precision scales as $O(p^2)$. The chapter, therefore, places particular emphasis on the binary splitting technique, which restructures the series evaluation as a recursive divide-and-conquer computation. Rather than accumulating terms one at a time at full precision, binary splitting recursively splits the summation range in half, evaluating each half at reduced intermediate precision and combining the results. This brings the total cost down to $O(M(p)\cdot\log^2(p))$, where $M(p)$ is the cost of a p-digit multiplication, making the Chudnovsky series with binary splitting competitive with the AGM-based methods of chapter three. The chapter includes C++ implementations for all three series, both with and without binary splitting, along with a performance comparison that makes the benefit of binary splitting directly visible. The recommendation is to use the Chudnovsky series with binary splitting as the standard series-based method, and precisely this combination underlies the world record computation of 324 trillion digits of π in 2026.

Chapter 3: Higher Order Algorithms for π

The third chapter examines the family of AGM-based iterative algorithms developed from 1976 onwards. These methods differ fundamentally from the series-based approaches of Chapter 2. Rather than summing many terms, they iterate a small set of recurrences that converge to π at an exponential rate with the iteration count. The chapter opens with the Brent-Salamin algorithm, which achieves quadratic convergence using the

arithmetic-geometric mean, and its close relative, the Gauss-Legendre algorithm. Both require only one square root per iteration, along with a fixed number of additions and multiplications, and both double the number of correct digits with each step.

The chapter then surveys the higher-order methods discovered by the Borwein brothers during the 1980s and early 1990s: a cubic algorithm from 1991, a quartic algorithm from 1984, a quintic algorithm, and a nonic (ninth-order) algorithm. Each of these offers a higher convergence rate than Brent-Salamin, at the cost of additional square root and multiplication operations per step. The central finding of the chapter, supported by detailed benchmarks, is that the higher convergence rate of the Borwein methods is more than offset by the increased per-step complexity. Brent-Salamin and Gauss-Legendre are consistently faster than all Borwein variants by a factor of approximately two to two and a half across all tested digit counts. This leads to a clear practical recommendation: for high-order iterative computation of π, the Brent-Salamin algorithm is preferred. The Borwein variants are presented in full because they are mathematically important and historically significant, but they are not the right choice in a performance-critical implementation.

Chapter 4: The Spigot Algorithm

The fourth chapter introduces a qualitatively different approach to computing π: the spigot algorithm. Where all the methods in the preceding chapters compute π to p digits as a single block of output, requiring all intermediate values to be maintained at full precision throughout, a spigot algorithm produces the decimal digits of π one at a time in sequence. The name comes from the image of turning a tap and having digits flow out individually.

The chapter covers the Rabinowitz-Wagon spigot algorithm (1990) and the more refined bounded spigot due to Gibbons (2004), as well as the Gosper formula for π, based on a continued-fraction representation that produces base-10 digits directly. A central observation is that the most attractive property of the spigot approach, namely that it can, in principle, produce digits without requiring arbitrary precision arithmetic at all, holds only for a bounded number of digits: the integer coefficients involved grow without bound as more digits are generated, so truly unbounded computation still requires growing integer arithmetic. For an arbitrary precision library, the total cost of a spigot computation to p digits remains $O(p^2)$, with no path to the $O(M(p)\cdot\log^2(p))$ scaling achievable by binary splitting. The spigot algorithms are therefore presented as an interesting and self-contained family of methods with their own niche uses, rather than as a competitive alternative to series and iterative methods for large-scale arbitrary-precision computation.

Practical implementation of π Algorithms

1. Newton's & the 9th order fixed-point method for calculating π

Imagine that we do not have access to π at all, or at least not with the required precision; we would need to calculate it, typically through some iterative method, until the desired precision is achieved.

A simple, yet efficient method is finding π through the equation:

$$\sin(x)=0$$

Which has the general solution of $x=n\pi$; We are, of course, interested in finding the solution for n=1 and applying Newton's iteration formula of:

$$x_{n+1}=x_n-\frac{f(x_n)}{f'(x_n)}$$

We get:

$$x_{n+1}=x_n+\frac{\sin(x_n)}{\cos(x_n)} \quad \text{Or} \quad x_{n+1}=x_n+\tan(x_n)$$

To see how it works in practice, we can, for example, start with $x_0=2$ and obtain the following iteration.

Iteration	π	F(x)=sin(x)	F'(x)=cos(x)	Error
0	2.00	9.09E-01	-4.16E-01	1.14E+00
1	4.18503986326152	-8.64E-01	-5.03E-01	1.04E+00
2	2.46789367451467	6.24E-01	-7.82E-01	6.74E-01
3	3.26618627756911	-1.24E-01	-9.92E-01	1.25E-01
4	3.14094391231764	6.49E-04	-1.00E+00	6.49E-04
5	3.14159265368080	-9.10E-11	-1.00E+00	9.10E-11
6	3.14159265358979	1.23E-16	-1.00E+00	0.00E+00

We see quick convergence, and after six iterations, we have the solution to approx. this accuracy. 15 digits. The Newton method is said to have quadratic convergence, meaning that the number of correct digits doubles for each iteration. However, that is not what we observed in the above iterations. The convergence rate is cubic in the above iterations, meaning the number of correct digits triples with each iteration. The reason is that the error terms in a Newton iteration are $e_{n+1}=e_n^2\frac{f''(x)}{2f'(x)}$ and for f''(x)=-sin(x) => f''(π)=-sin(π)=0. The quadratic term vanished, and the leading error term is e_n^3, leading to a cubic convergence rate.

We also notice that

$$\lim_{x \to \pi} \sin(x) \sim 0$$

And

$$\lim_{x \to \pi} \cos(x) \sim 1$$

So we can simplify the iteration by simply ignoring cos(x) and iterate using:

$$x_{n+1} = x_n + \sin(x_n)$$

The first is that dropping $\cos(x_n) \approx -1$ is a valid approximation near π, saving the cost of a cosine evaluation without meaningfully changing the iteration and the convergence. This is a practical simplification, not a mathematical improvement.

Iteration	π	Sin(x)	Error
0	2.00	0.909297	1.14E+00
1	2.90929742682568	0.230212	2.32E-01
2	3.13950913306779	0.002084	2.08E-03
3	3.14159265208235	1.51E-09	1.51E-09
4	3.14159265358979	1.23E-16	0.00E+00

After four iterations, we get the same accuracy as in the normal case. The fact that it achieves the same accuracy after four rather than the original six iterations is not a sign that the new formula is faster. It still has cubic convergence like the original, tripling the number of accurate digits in each iteration. However, we avoided computing cos(x), making the algorithm twice as fast as the original. This is very important when using arbitrary precision arithmetic. The trigonometric function is very expensive time-wise to calculate.

Can we further improve on the Newton formula? The answer is yes. Instead of iterating where the new x_{n+1} is the tangent interception on the x-axis of the previous point x_n, we can use a polynomial curve of higher order. One use is the 9th-order fixed point iteration based on the truncated arcsin series applied to sin(x):

$$x_{n+1} = x_n + \sin(x_n) + \frac{1}{6}\sin^3(x_n) + \frac{3}{40}\sin^5(x_n) + \frac{5}{112}\sin^7(x_n)$$

It has much faster convergence, yielding 9 times as many correct digits per iteration. The deviation of the formula can be found in Appendix B.

Iteration	π	Sin(x) +6Sin(x)^3/40+3Sin(x)^5/40+5Sin(x)^7/112	Error
0	2	1.104169235	1.14E+00

1	3.10416923500490	0.037423419	3.74E-02
2	3.14159265358979	4.56341E-15	4.44E-15
3	3.14159265358979	1.22515E-16	0.00E+00

After only three iterations, we have found the solution with 15-digit accuracy.
Now, is it worth using these higher-order iterations? Assuming the most time-consuming calculation is sin(x), which is several magnitudes higher than regular arithmetic, the answer is yes, since we calculate $\sin(x_n)$ once in both places. Of course, we will also need to revise the nine-order method by rearranging it a little bit, applying a Horner-type recursion

$$x_{n+1}=x_n+\frac{1}{1680}\sin(x_n)\left(1680+\sin^2(x_n)\left(280+\sin^2(x_n)\left(126+75\sin^2(x_n)\right)\right)\right)$$

The constant $\frac{1}{1680}$ can be calculated before the iteration and converted into a multiplication, which is a much faster operation in arbitrary-precision arithmetic.

The Newton and ninth-order iteration code segments are listed below.

Newton (Cubic convergence)

Notice that we have "seeded" the iteration with the first approximation. 15-16 digits of π effectively avoiding 4 iterations. E.g., if we want π with 531,441 digits (3^{12}) but make our initial start guess with the first 16 digits (~32.5) of π, we only need to iterate through ⌈12-2.5⌉=10 iterations to get our result. For a large number of digits that will save us a tremendous amount of calculation, equivalent to approx. 20% time savings.

Now, what kind of guard digits (g) do we need to apply when we want a precision of p decimal digits? We have the sin(x) evaluation and the addition per iterations given a total of roughly $2\cdot10^{(-p+g)}$. Over N iterations ($N=\log_3(p)$) the accumulated rounding is $2N\cdot10^{(-p+g)}$. For this to stay below $0.5\cdot10^{-p}$, you need $g \geq \log_{10}(4N)$. For N=20 (more than trillions of digits), you get log10(4*20)~1.9 guard digits. With a choice of guard digit=2, we should be covered.

Algorithm 1.1 Standard Newton

```
// Standard newton iteration of PI given by x=x+sin(x)
static float_precision pi_newton(const uintmax_t digits)
   {
   const uintmax_t prec = digits + 2;
   // Set default 16 digits precision as a starting point.
   float_precision piprev(0,prec), pi("3.141592653589793", prec);

   if( digits <= 16)  // Do we already have the digits?
   {
      pi.precision(digits);
      return pi;
   }

   for (; pi!=piprev; )
      {
```

```
      piprev = pi;
      pi += sin(pi);
      }

   pi.precision(digits);
   return pi;
   }
```

Fixed-point 9th order convergence

The 9th order requires fewer iterations than the example above; it only requires 9^x=531,441 or approx. 6 iterations, while starting with the first 16 correct digits of π gives a net effect of approx. 3.5 iterations again, a saving of around 50-60%.

We can do the same analysis for guard digits as we did for the Newton method. Each step has more operations: sin(x) plus four multiplications and the final addition, giving roughly $6 \cdot 10^{-(p+g)}$ rounding per step. However, the 9th order method needs far fewer iterations, only $N = \lceil \log_9(p) \rceil$ steps, so the total accumulated rounding is $6N \cdot 10^{-(p+g)}$. You need $g \geq \log_{10}(12N)$. Since N never exceeds 8 for p up to 100 million digits, $\log_{10}(12 \cdot 8) \approx 1.98$, so 2 decimal guard digits are again always sufficient, but just barely: at p = 100,000, the requirement is $\log_{10}(72) \approx 1.86$, meaning 1 guard digit is not enough. The 9th order method is therefore slightly more demanding on guard digits than Newton, not less, because the per-step rounding from the extra polynomial operations slightly outweighs the benefit from fewer total iterations.

Algorithm 1.2 ninth-order fixed-point method

```
// 9th order newton iteration for PI using
// x=x+sin(x)+1/6Sin(x)^3+3/40sin(x)^5+5/112sin(x)^7
static float_precision pi_9order(const uintmax_t digits)
        {
        const uintmax_t prec = digits + 2;
        const float_precision c1680(1680), c280(280), c126(126), c75(75); // temp constants
        // Set default 16 digits precision a starting point.
        float_precision pi("3.141592653589793", prec);
        float_precision sinx(0,prec), sinxsq(0,prec), inv1680(1680,prec), r(0,prec);
//temp variables

        if (digits <= 16)  // Do we already have the digits?
        {
                pi.precision(digits);
                return pi;
        }

        inv1680 = inv1680.inverse();
        for (; ; )
                {
                sinx = sin(pi);
                sinxsq = sinx.square();
                r = inv1680 * (sinx*(c1680 + sinxsq*(c280 + sinxsq*(c126 + c75 * sinxsq))));
                if (pi + r == pi)
                        break;
                pi += r;
                }

        pi.precision(digits);
        return pi;
        }
```

Improvement of the Newton method?

Further improvements can be made by noting that a Newton iteration step is self-correcting. In the previous example of π with 531,441 digits, instead of starting with an arbitrary precision of 531,441 digits, we could start with a lower precision and then progressively increase it until we reach the desired precision in the final iteration. Although it can be found under different names, I usually prefer the term as dynamic precision or iterative deepening (increasing precision per iteration)

The two algorithms for progressively more digits to calculate π are listed below.

Algorithm 1.3 Newton Standard with Dynamic Precision

```
// Newton standard with iterative deepening
static float_precision pi_newton_deepening(const uintmax_t digits)
   {
   const uintmax_t prec = digits + 2;
   uintmax_t d;
   // Set default 16 digits precision as a starting point.
   float_precision piprev(0, prec), pi("3.141592653589793", prec);

   if (digits <= 16)  // Do we already have the digits?
   {
      pi.precision(digits);
      return pi;
   }

   for (d = std::min(digits, (uintmax_t)(3*16)); d!=prec || pi != piprev;)
      {
      piprev.precision(d);   // change the calculating precision
      pi.precision(d);

      piprev = pi;
      pi += sin(pi);
      // Increase the calculting precision by the convergence rate of 2
      d = std::min(prec, 3 * d );
      }

   pi.precision(digits);
   return pi;
   }
```

Algorithm 1.4 Newton ninth-order convergence using dynamic precision

```
// 9th order fixed-point with iterative deepening
static float_precision pi_9order_deepening(const uintmax_t digits)
   {
   const uintmax_t prec = digits + 2;
   uintmax_t d;
   // temp constants.
   const float_precision c1680(1680), c280(280), c126(126), c75(75);
   float_precision pi("3.141592653589793", prec);  // Set default 15 digits precision a 
starting point.
   //temp variables
   float_precision sinx(0,prec), sinxsq(0, prec), inv1680(1680, prec), r(0,prec), 
tmp_inv1680;

   if (digits <= 16)  // Do we already have the digits?
   {
      pi.precision(digits);
      return pi;
   }

   inv1680 = inv1680.inverse(); // 1/1680
```

```
for (d = std::min(digits, (uintmax_t)(16*9)); ;  )
    {
    // change the calculating precision
    pi.precision(d );
    sinx.precision(d);
    sinxsq.precision(d);
    r.precision(d);
    tmp_inv1680.precision(d);
    tmp_inv1680 = inv1680;  // Only extract what needed for a d precision digit

    sinx = sin(pi);
    sinxsq = sinx.square();
    r = tmp_inv1680 * (sinx*(c1680 + sinxsq*(c280 + sinxsq*(c126 + c75 * sinxsq))));
    if (d==prec && pi + r == pi )
       break;
    pi += r;
    // Increase the calculting precision by the convergence rate of 9
    d = std::min(prec, 9 * d);
    }

pi.precision(digits);
return pi;
}
```

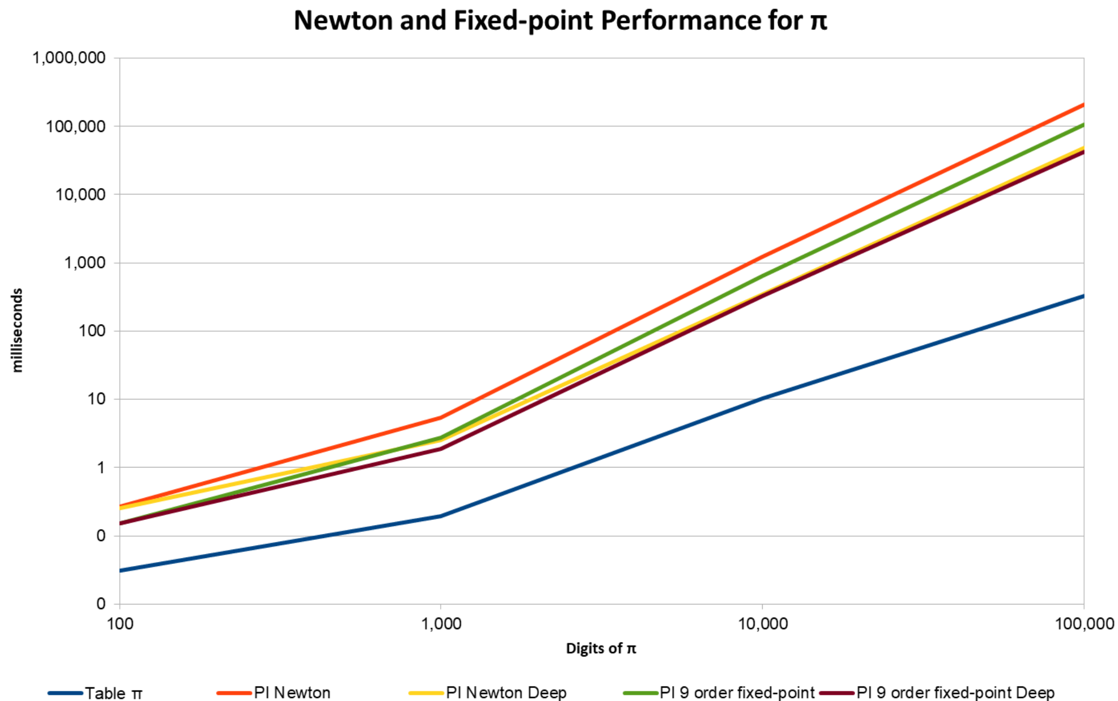
Performance

The performance improves significantly by using dynamic precision instead of regular iteration; however, it still doesn't match our standard algorithm in the arbitrary-precision package we are using, particularly as the number of π digits increases. The Newton algorithm for dynamic precision is, as expected, a huge improvement over the standard Newton methods; for example, it speeds up the Newton algorithm by a factor of approximately four over the standard Newton method and a factor of 5 in the 9th-order fixed-point method using dynamic precision. The ninth-order fixed-point method is approx. two times faster than the Newton cubic method.

However, in the end, the Newton method and the 9th-order fixed-point method won't match the performance of, e.g., the currently faster method based on the Chudnovsky series implemented via binary splitting (Table π), which shows much better scalability as the number of digits increases. The culprit is the slow calculation of sin(x) compared to the binary splitting method, which uses only basic arithmetic operations like +, -, *, /, and √.

See the chart below with a comparison to the Chudnovsky method described in section 2 of this paper (table π).

Practical implementation of π Algorithms



Recommendation for Newton's methods for generating π

The following recommendation arises for examining the Newton and 9th-order methods.

- The Newton method is not the fastest way of generating π and is therefore not recommended
- If you want to use a fixed-point iteration method, then use the ninth-order fixed-point method using dynamic precision.

2. Infinite series for π

In this section, we examine the famous Ramanujan, Chudnovsky, and Borwein infinite series for finding π. The series' performance in itself is not impressive compared to other methods. However, you can apply the binary splitting method, which significantly improves the performance of infinite series methods.

Ramanujan and π

Ramanujan was an Indian self-studying mathematician who, around 1910, quite astonished, invented the following infinite series formula for π.

Practical implementation of π Algorithms

$$\frac{1}{\pi}=\frac{2\sqrt{2}}{9801}\sum_{n=0}^{\infty}\frac{(4n)!(1103+26390n)}{(n!)^4 396^{4n}}$$

For each term in the series, it produces approx. eight more correct digits of π. In the quest to calculate π, this particular formula was used in 1985 to compute approx. 17 million digits of π. In the table below, we use column a_n to denote the nth term of the summations and the column ∑ to hold the accumulated sum of the first n terms.

$$a_n=\frac{(4n)!(1103+26390n)}{(n!)^4 396^{4n}}$$

terms	a_n	Σ	1/π	π	Error
0	1.10300E+03	1103.000000000	0.31831	3.14159273001331	7.64E-08
1	2.68320E-05	1103.000026832	0.31831	3.14159265358979	4.44E-16

From the table above, we get approx. eight correct digits after the first term and eight more digits that are correct after the second term. This means that, for example, if you want to calculate π to 1,000 digits, you need approximately. 1,000/8=125 terms of the Ramanujan series.

To make an effective recurrence of the above formula, we need to make some adjustments:

$$\frac{1}{\pi}=\frac{2\sqrt{2}}{9801}\sum_{n=0}^{\infty}\frac{(4n)!(1103+26390n)}{(n!)^4 396^{4n}} \Rightarrow$$

$$\frac{1}{\pi}=\frac{2\sqrt{2}}{9801}\left(1103\sum_{n=0}^{\infty}\frac{(4n)!}{(n!)^4 396^{4n}}+26390\sum_{n=0}^{\infty}\frac{(4n)!}{(n!)^4 396^{4n}}n\right)$$

Let $a_n=\frac{(4n)!}{(n!)^4 396^{4n}}$ and you get:

$$\frac{1}{\pi}=\frac{2\sqrt{2}}{9801}\left(1103\sum_{n=0}^{\infty}a_n+26390\sum_{n=0}^{\infty}a_n n\right)$$

Substitute $b_n=a_n n$ into the above and you get:

$$\pi=\frac{9801}{2\sqrt{2}\left(1103\sum_{n=0}^{\infty}a_n+26390\sum_{n=0}^{\infty}b_n\right)}$$

The only thing left is to calculate as a_n part of the recurrence. We can do that by evaluating:

$$\frac{a_n}{a_{n-1}} = \frac{\frac{(4n)!}{(n!)^4 396^{4n}}}{\frac{(4(n-1))!}{((n-1)!)^4 396^{4(n-1)}}} \quad =>$$

$$\frac{a_n}{a_{n-1}} = \frac{4n(4n-1)(4n-2)(4n-3)}{n^4 396^4} \quad =>$$

$$a_n = \frac{(4n-1)(4n-2)(4n-3)}{6147814464 n^3} a_{n-1}$$

We can now write up the complete recurrence:

Let the initial conditions be $a_0=1$, $a_s=0$, $b_s=0$ then for n=1,2,3… you get:

$$a_n = \frac{(4n-1)(4n-2)(4n-3)}{6147814464 n^3} a_{n-1}$$

$$a_s += a_n$$

$$b_n += a_n n$$

$$\pi = \frac{9801}{2\sqrt{2}(1103 a_s + 26390 b_s)}$$

To ensure sufficient guard digits when evaluating the series, we have derived the required guard digits from the error analysis in Appendix C.

Algorithm 2.1 Ramanujan Infinite series

```
// Ramanujan series that add 8 more correct digits per iteration
static float_precision pi_ramanujan(uintmax_t digits)
   {
   const size_t guard = (size_t)std::ceil(std::log10(3.12 * digits)) + 1;
   const size_t prec = digits + guard;
   uintmax_t n;
   const float_precision c1103(1103), c26390(26390);       // constants
   const float_precision c9801(9801),c6147814464(6147814464);      // constants
   float_precision pi(3, prec);
   float_precision an(1, prec), asum(1, prec), bsum(0, prec);
```

```
    float_precision c2sq2(0,prec);  float_precision tmp(0, prec); float_precision np3(0, 
prec);

    c2sq2 = _float_table(_SQRT2, prec );
    c2sq2.adjustExponent(1); // multiply by 2 to get 2*sqrt(2)
    for (n=1; ; ++n )
        {
        uintmax_t n4 = 4 * n;
        // Use 64bit integer arithmetic when possible. Assuming that n<~2G iterations is 
required
        if (n <= 2'479'700'524)
            np3 += float_precision(n*(3 * n - 3) + 1);
        if (n < 660'562)  // Up to 64bit arithmetic is safe
            tmp = float_precision((n4 - 1)*(n4 - 2)*(n4 - 3));
        else
            {// Worst case fallback
            tmp = float_precision(n4 - 1);
            tmp *= float_precision(n4 - 2);
            tmp *= float_precision(n4 - 3);
            }
        an *= tmp / (c6147814464 * np3 );
        if (asum + an == asum)
            break;
        asum += an;
        bsum += an * float_precision(n);
        }

    pi = c9801 / (c2sq2*(asum * c1103 + bsum* c26390));
    pi.precision(digits);
    return pi;
    }
```

Chudnovsky brothers

The Chudnovsky brothers found a variation of the Ramanujan infinite series for π in 1989 using the infinite series:

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

For each term in the series, it produces approx. 14 more correct digits of π, which is six digits more than the Ramanujan series per term. In 1994, the formula was used to calculate approx. 4 billion digits of π. Again in 2010 and 2011, it reached 10 billion digits of π, and finally in 2026, 314 trillion digits of π.

In the table below, we use column 'a_n' to denote the nth term of the summations and the column ∑ to hold the accumulated sum of the first n terms.

$$a_k=\frac{(-1)^n(6n)!(13591409+545140134n)}{(3n)!(n!)^3 640320^{4n+3/2}}$$

terms	a_n	Σ	1/π	π	Error

0	1.3591E+07	13591409	0.318309886183797	3.14159265358973	5.91E-14
1	-2.5538E-07	13591409	0.318309886183791	3.14159265358979	0

From the above table, we get approx. 14 correct digits after the first term and 14 more correct digits after the second term (which is outside the limit of the accuracy of the calculation since we only have approx. 15 digits of accuracy). This means that, for example, if you want to calculate π to 1000 digits, you need approximately. 1,000/14=72 terms of the Chudnovsky brother algorithm.

As we did for the Ramanujan series, we need to rewrite the above equations as follows:

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

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

Letting $a_n=\frac{(-1)^n(6n)!}{(3n)!(n!)^3 640320^{3n}}$ you get:

$$\frac{1}{\pi}=\frac{12}{640320^{\frac{3}{2}}}\left(13591409\sum_{n=0}^{\infty}a_n+545140134\sum_{n=0}^{\infty}a_n n\right) \Rightarrow$$

$$\frac{1}{\pi}=\frac{1359409\sum_{n=0}^{\infty}a_n+54514013\sum_{n=0}^{\infty}a_n n}{426880\sqrt{100005}} \Rightarrow$$

$$\pi=\frac{426880\sqrt{10005}}{13591409\sum_{n=0}^{\infty}a_n+545140134\sum_{n=0}^{\infty}a_n n}$$

We just need now to find an easier way to calculate a_n.

Letting:

Practical implementation of π Algorithms

$$\frac{a_n}{a_{n-1}}=\frac{\frac{(-1)^n(6n)!}{(3n)!(n!)^3 640320^{3n}}}{\frac{(-1)^{n-1}(6(n-1))!}{(3(n-1))!((n-1)!)^3 640320^{3(n-1)}}} \quad =>$$

$$\frac{a_n}{a_{n-1}}=-\frac{(6n-5)(6n-4)(6n-3)(6n-2)(6n-1)6n}{3n(3n-1)(3n-2)n^3 640320^3} =>$$

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

We can now write up the complete recurrence:

Let the initial conditions be $a_0=1,\ a_s=0,\ b_s=0$ then for n=1,2,3… you get:

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

$$a_s+=a_n$$

$$b_s+=a_n n$$

$$\pi=\frac{426880\sqrt{10005}}{13591409\,a_s+545140134\,b_s}$$

To ensure sufficient guard digits when evaluating the series, we have derived the required guard digits from the error analysis in Appendix D.

Algorithm 2.2 Chudnovsky Infinite series

```
// Chudnovsky series that add 14 more correct digits per iteration
static float_precision pi_chudnovsky(uintmax_t digits)
	{
	const size_t guard = (size_t)std::ceil(std::log10(2.9 * digits)) + 1;
	const size_t prec = digits + guard;
	uintmax_t n;
	const float_precision c426880(426'880), c10005(10'005, prec);   // constants
	const float_precision c13591409(13'591'409), c545140134(545'140'134);   // constants
	float_precision pi(3, prec);
	float_precision an(1, prec), asum(1, prec), bsum(0, prec);
	float_precision tmp(0, prec), np3(0, prec);
	const float_precision c3_over24(10'939'058'860'032'000ull, prec);// 640320^3/24

	for (n = 1; ; ++n)
		{// Use 64bit integer arithmetic when possible.
		//Assuming less than n<~2G iterations is required
		if (n <= 2'479'700'524)
			np3 += float_precision(n*(3 * n - 3) + 1);
```

```
            if (n < 635'130)
                    tmp = float_precision((6 * n - 5)*(2 * n - 1)*(6 * n - 1));
            else
                    {
                    tmp = float_precision(6 * n - 5);
                    tmp *= float_precision(2 * n - 1);
                    tmp *= float_precision(6 * n - 1);
                    }
            tmp /= (c3_over24* np3);
            an *= -tmp; // - is the toogle of the (-1)^n
            if (asum + an == asum)
                    break;
            asum += an;
            bsum += an * n;
            }

    pi = c426880 * sqrt(c10005)/(asum * c13591409 + bsum* c545140134);
    pi.precision(digits);
    return pi;
    }
```

Borwein Brothers

The Borwein brothers presented two infinite series in 1989 and 1993. The 1989 produced 25 correct digits per iteration, while the 1993 infinite series produced an impressive 50 correct digits per iteration. We will call it the Borwein 25 and Borwein 50 Infinite series.

Borwein 25

The Borwein brothers presented an infinite series for calculating π, which was published in 1989. This is similar to the Chudnovsky brothers but uses different constants and produces 25 correct digits for π per iteration.

$$\frac{1}{\pi}=12\sum_{n=0}^{\infty}\frac{(-1)^n(6n)!(A+nB)}{(3n)!(n!)^3C^{n+1/2}}$$

Where the constants A, B, and C are:

$$A=212175710912\sqrt{61}+1657145277365$$

$$B=13773980892672\sqrt{61}+107578229802750$$

$$C=(159999840\sqrt{61}+1249638720)^3$$

Due to the $\sqrt{61}$ The constant is now a floating-point constant and not an integer constant, as for the Ramanujan and Chudnovsky infinite series.

Each additional term will generate approximately 25 correct digits. As expected, we only need to calculate the first term, n=0, to be correct within the limitations of default IEEE-754 double-precision approximation of 15-16 digits. This means that, for example, if you want to calculate π to 1,000 digits, you need approximately. 1,000/25=40 terms of the Borwein brothers' algorithm.

In the table below, we use column 'a_n' to denote the nth term of the summations and the column ∑ to hold the accumulated sum of the first n terms

$$a_n = \frac{(-1)^n (6n)!(A+nB)}{(3n)!(n!)^3 C^{n+1/2}}$$

terms	aₙ	Σ	1/π	Π	Error
0	0.026526	0.026525824	0.318309886	3.14159265358979	0

As we did for the Ramanujan and Chudnovsky series, we need to rewrite the above equations to create an efficient algorithm as follows:

$$\frac{1}{\pi} = 12 \sum_{n=0}^{\infty} \frac{(-1)^n (6n)!(A+nB)}{(3n)!(n!)^3 C^{n+1/2}} =>$$

$$\frac{1}{\pi} = \frac{12}{\sqrt{C}} \left(\sum_{n=0}^{\infty} \frac{(-1)^n (6n)!}{(3n)!(n!)^3 C^n} A + \sum_{n=0}^{\infty} \frac{(-1)^n (6n)!}{(3n)!(n!)^3 C^n} Bn \right)$$

Letting $a_n = \frac{(-1)^n (6n)!}{(3n)!(n!)^3 C^n}$ You get:

$$\frac{1}{\pi} = \frac{12}{\sqrt{C}} \left(A \sum_{n=0}^{\infty} a_n + B \sum_{n=0}^{\infty} a_n n \right) =>$$

$$\pi = \frac{\sqrt{C}}{12 \left(A \sum_{n=0}^{\infty} a_n + B \sum_{n=0}^{\infty} a_n n \right)}$$

We just need to find an easier way to calculate a_n.

Letting:

$$\frac{a_n}{a_{n-1}}=\frac{\frac{(-1)^n(6n)!}{(3n)!(n!)^3C^n}}{\frac{(-1)^{n-1}(6(n-1))!}{(3(n-1))!((n-1)!)^3C^{n-1}}} \quad =>$$

$$\frac{a_n}{a_{n-1}}=-\frac{6n(6n-1)(6n-2)(6n-3)(6n-4)(6n-5)}{3n(3n-1)(3n-2)n^3C} \quad =>$$

$$a_n=-\frac{24(6n-5)(2n-1)(6n-1)}{n^3C}a_{n-1}$$

This is very similar to the result we got with Chudnovsky. Only the constant C differs.

We can now write up the complete Borwein 25 recurrence:

Let the initial conditions be $a_0=1,\ a_s=0,\ b_s=0$ then for n=1,2,3… you get:

$$a_n=-\frac{24(6n-5)(2n-1)(6n-1)}{Cn^3}a_{n-1}$$

$$a_s+=a_n$$

$$b_s+=a_n n$$

$$\pi=\frac{\sqrt{C}}{12(Aa_s+Bb_s)}$$

Where:

$$A=212175710912\sqrt{61}+1657145277365$$
$$B=13773980892672\sqrt{61}+107578229802750$$
$$C=(159999840\sqrt{61}+1249638720)^3$$

A detailed error bound analysis is carried out in Appendix E.

Algorithm 2.3 Borwein 25

```
// Borwein series that add 25 more correct digits per iteration
static float_precision pi_borwein25(unsigned int digits)
	{
	const size_t guard = (size_t)std::ceil(std::log10(2.64 * digits)) + 1;
	const size_t prec = digits + guard;
	uintmax_t n;
	const float_precision c212175710912(212'175'710'912),
c13773980892672(13'773'980'892'672);	// constants
	const float_precision c12(12), c159999840(159'999'840);
// constants
	float_precision A(1'657'145'277'365, prec), B(107'578'229'802'750, prec),
C(1'249'638'720, prec);
	float_precision c61sq(61, prec);
```

```
	float_precision pi(3, prec);
	float_precision an(1, prec), asum(1, prec), bsum(0, prec);
	float_precision tmp(0, prec), np3(0, prec);

	c61sq = sqrt(c61sq);  // Calculate sqrt(61) at the requested precision
	A += c212175710912 * c61sq;            // Finish up creation of constant A
	B += c13773980892672 * c61sq;   // Finish up creation of constant B
	C += c159999840 * c61sq;               // Finish up creation of constant C
	C *= C.square();                               // C = C^3; C = C.square() * C
	// Now iterate until no change in the asum
	for (n = 1; ; ++n)
		{// Use 64bit integer arithmetic when possible.
		// Assuming less than n<~2G iterations
		if (n <= 2'479'700'524)
			np3 += float_precision(n*(3 * n - 3) + 1);
		if (n < 220'188)
			tmp = float_precision((6 * n - 5)*(48 * n - 24)*(6 * n - 1));
		else
			{// more than wat 64bit can handle
			tmp = float_precision(6 * n - 5);
			tmp *= float_precision(48 * n - 24);
			tmp *= float_precision(6 * n - 1);
			}
		tmp /= C * np3;
		an *= -tmp;
		if (asum + an == asum)
			break;
		asum += an;
		bsum += an * n;
		}

	pi = sqrt(C) / ( (asum * A + bsum* B) * c12 );
	pi.precision(digits);
	return pi;
	}
```

Borwein 50

In 1993, the Borwein brothers published another series that finds approx. 50 correct digits per iteration.

$$\frac{1}{\pi}=\frac{1}{\sqrt{-C^3}}\sum_{n=0}^{\infty}\frac{(6n)!(A+nB)}{(3n)!(n!)^3C^{3n}}$$

Where the constants A, B, and C are:

$$A=A_1+A_2\sqrt{5}+384\sqrt{5}(A_3+A_4\sqrt{5})^{\frac{1}{2}}$$

Where:

$A_1=63365028312971999585426220$

$A_2=28337702140800842046825600$

$A_3=10891728551171178200467436212395209160385656017$

$A_4=4870929086578810225077338534541688721351255040$

$$B=B_1+B_2\sqrt{5}+2515968\sqrt{31101}(B_3\ +\ B_4\sqrt{5})^{\frac{1}{2}}$$

Where :

$B_1 = 7849910453496627210289749000$
$B_2 = 3510586678260932028965606400$
$B_3 = 26260208323789001636993322654444020882161$
$B_4 = 2799650273060444296577206890718825190235$

$$C = -C_1 - C_2\sqrt{5} - 1296\sqrt{5}\left(C_3 + C_4\sqrt{5}\right)^{\frac{1}{2}}$$

Where:

$C_1 = 214772995063512240$
$C_2 = 96049403338648032$
$C_3 = 10985234579463550323713318473$
$C_4 = 4912746253692362754607395912$

We notice that since C is negative, C^3 is also negative, and $-C^3$ is positive, so we can do a calculation of $\sqrt{-C^3}$.

Each additional term will generate approximately 50 correct digits. If you want, for example, to calculate π to 1000 digits, you need approximately. 1,000/50=20 terms of the Borwein brothers' algorithm.
As we did for the Ramanujan and Chudnovsky series, we need to rewrite the above equations to create an efficient algorithm as follows:

$$\frac{1}{\pi} = \frac{1}{\sqrt{-C^3}} \sum_{n=0}^{\infty} \frac{(6n)!(A+nB)}{(3n)!(n!)^3 C^{3n}} \Rightarrow$$

$$\frac{1}{\pi} = \frac{1}{\sqrt{-C^3}} \left(\sum_{n=0}^{\infty} \frac{(6n)!}{(3n)!(n!)^3 C^{3n}} A + \sum_{n=0}^{\infty} \frac{(6n)!}{(3n)!(n!)^3 C^{3n}} Bn \right)$$

Letting $a_n = \frac{(6n)!}{(3n)!(n!)^3 C^{3n}}$ You get:

$$\frac{1}{\pi} = \frac{1}{\sqrt{-C^3}} \left(A \sum_{n=0}^{\infty} a_n + B \sum_{n=0}^{\infty} a_n n \right) \Rightarrow$$

Practical implementation of π Algorithms

$$\pi = \frac{\sqrt{-C^3}}{\left(A\sum_{n=0}^{\infty} a_n + B\sum_{n=0}^{\infty} a_n n\right)}$$

We just need to find an easier way to calculate a_n.

Letting:

$$\frac{a_n}{a_{n-1}} = \frac{\dfrac{(6n)!}{(3n)!(n!)^3 C^{3n}}}{\dfrac{(6(n-1))!}{(3(n-1))!((n-1)!)^3 C^{3(n-1)}}} \quad =>$$

$$\frac{a_n}{a_{n-1}} = \frac{6n(6n-1)(6n-2)(6n-3)(6n-4)(6n-5)}{3n(3n-1)(3n-2)n^3 C^3} \quad =>$$

$$a_n = \frac{24(6n-5)(2n-1)(6n-1)}{n^3 C^3} a_{n-1}$$

Which is very similar to the result we got with Borwein 1989. Only the constant C differs. (C^3 versus C).

We can now write up the complete Borwein 1993 recurrence:

Let the initial conditions be $a_0 = 1$, $a_s = 0$, $b_s = 0$ then for n=1,2,3… you get:

$$a_n = \frac{24(6n-5)(2n-1)(6n-1)}{C^3 n^3} a_{n-1}$$

$$a_s += a_n$$

$$b_s += a_n n$$

$$\pi = \frac{\sqrt{-C^3}}{A a_s + B b_s}$$

Where the constants A, B, and C are:

$$A = A_1 + A_2\sqrt{5} + 384\sqrt{5}(A_3 + A_4\sqrt{5})^{\frac{1}{2}}$$

Where:

$A_1 = 63365028312971999585426220$
$A_2 = 28337702140800842046825600$
$A_3 = 10891728551171178200467436212395209160385656017$
$A_4 = 48709290865788102250773385345416887213512550 40$

$$B = B_1 + B_2\sqrt{5} + 2515968\sqrt{31101}\left(B_3 + B_4\sqrt{5}\right)^{\frac{1}{2}}$$

Where :

$B_1 = 7849910453496627210289749000$
$B_2 = 3510586678260932028965606400$
$B_3 = 2626020832378900163699332265444402088216 1$
$B_4 = 2799650273060444296577206890718825190235$

$$C = -C_1 - C_2\sqrt{5} - 1296\sqrt{5}\left(C_3 + C_4\sqrt{5}\right)^{\frac{1}{2}}$$

Where:

$C_1 = 214772995063512240$
$C_2 = 96049403338648032$
$C_3 = 1098523457946355032371331 8473$
$C_4 = 4912746253692362754607395912$

A detailed error bound analysis can be found in Appendix F.

Algorithm 2.4 Borwein 50

```
// Borwein series that add 50 more correct digits per iteration
static float_precision pi_borwein50(unsigned int digits)
	{
	const size_t guard = (size_t)std::ceil(std::log10(2.64 * digits)) + 1;
	const size_t prec = digits + guard;
	uintmax_t n;
	float_precision A(0, prec), B(0, prec), C(0, prec), C3(0,prec);
	float_precision csq5(5, prec);
	float_precision pi(3, prec);
	float_precision an(1, prec), asum(1, prec), bsum(0, prec);
	float_precision tmp(0, prec), np3(0, prec);

	csq5 = sqrt(csq5);		// Calculate sqrt(5) at the requested precision
	// Build Constant A
	A =  float_precision("63365028312971999585426220", prec);
	A += float_precision("28337702140800842046825600", prec)*csq5;
	tmp = float_precision("10891728551171178200467436212395209160385656017", prec);
	tmp += float_precision("48709290865788102250773385345416887213512550 40", prec)*csq5;
	tmp = sqrt(tmp);
	A += float_precision(384, prec)*csq5*tmp;
	// Build Constant B
	B = float_precision("7849910453496627210289749000", prec);
	B +=float_precision("3510586678260932028965606400", prec)*csq5;
	tmp = float_precision("6260208323789001636993322654444020882161", prec);
```

```
	tmp += float_precision("2799650273060444296577206890718825190235", prec)*csq5;
	tmp = sqrt(tmp);
	B += float_precision(2515968, prec)*sqrt(float_precision(3110, prec))*tmp;
	// Build Constant C
	C = float_precision(-214772995063512240, prec);
	C += float_precision(-96049403338648032, prec)*csq5;
	tmp = float_precision("10985234579463550323713318473", prec);
	tmp += float_precision("4912746253692362754607395912", prec)*csq5;
	tmp = sqrt(tmp);
	C += float_precision(-1296, prec)*csq5*tmp;
	// Build Constant C3=C^3
	C3 = C*C.square();		// C3 = C^3;

	// Now iterate until no change in the asum
	for (n = 1; ; ++n)
		{
		if (n <= 2'479'700'524) // Use 64bit integer arithmetic when possible. 
Assuming less than n<~2G iterations
			np3 += float_precision(n*(3 * n - 3) + 1);
		if (n < 220'188)
			tmp = float_precision((6 * n - 5)*(48 * n - 24)*(6 * n - 1));
		else
			{// more than wat 64bit can handle
			tmp = float_precision(6 * n - 5);
			tmp *= float_precision(48 * n - 24);
			tmp *= float_precision(6 * n - 1);
			}
		tmp /= C3 * np3;
		an *= tmp;
		if (asum + an == asum)
			break;
		asum += an;
		bsum += an * n;
		}

	loopcnt_borwein50 = n;
	pi = sqrt(-C3) / (asum * A + bsum* B);
	pi.precision(digits);
	return pi;
	}
```

The Binary splitting method for Ramanujan and Chudnovsky series

The methods described in the preceding sections evaluate the Ramanujan and Chudnovsky series by accumulating terms one at a time, performing each addition and multiplication at full working precision p + g. Since approximately N = p/8 terms are needed for Ramanujan and N = p/14 for Chudnovsky, the total cost scales as O(N · M(p)) where M(p) is the cost of a p-digit multiplication. Because N grows linearly with p, this gives an overall cost of O(p · M(p)), which is roughly O(p²) for schoolbook multiplication or O(p · log p · log log p) for FFT-based multiplication.

Binary splitting reduces this cost to O(M(p) · log²(p)) by restructuring the summation as a divide-and-conquer computation over integers rather than floating-point numbers.

Binary Splitting of the Ramanujan Infinite Series

The Ramanujan formula for π involves two separate partial sums accumulated over the same terms:

$$A = \sum a_n, \quad B = \sum n \cdot a_n \quad (n = 0, 1, \ldots, k-1)$$

Rather than accumulating A and B one term at a time in floating-point arithmetic, we observe that both sums can be expressed exactly using three integers P(0, k), Q(0, k), and R(0, k) sharing a common denominator, such that:

$$A = P(0, k) / Q(0, k) \quad B = R(0, k) / Q(0, k)$$

If the range [0, k) is split at the midpoint m, and we already know the integer triples for the left half [0, m) and the right half [m, k) separately, the two halves can be combined using exact integer arithmetic:

$$P(0, k) = P(0, m) \cdot Q(m, k) + P(m, k) \cdot R(0, m)$$

$$Q(0, k) = Q(0, m) \cdot Q(m, k)$$

$$R(0, k) = R(0, m) \cdot R(m, k)$$

This gives a recursive divide-and-conquer structure. At the leaves of the recursion, each interval [n, n+1) covers a single term whose P, Q, and R values are small integers computable directly from the recurrence coefficients. The recursion then merges pairs of intervals upward, doubling the covered range at each level, until the full integer triple P(0, N), Q(0, N), R(0, N) is assembled at the root.

The benefit of this structure is that all arithmetic during the merging phase is performed on integers of increasing but bounded size, rather than on floating-point numbers at full precision p throughout. The integers grow as the recursion deepens, but the multiplications at the upper levels of the tree, which involve the largest integers, are performed only once each. The multiplications at the lower levels, which are cheap because the integers are still small, are performed many times. The total bit-complexity across all multiplications in the recursion tree works out to $O(M(p) \cdot \log^2(p))$, a substantial improvement over the sequential $O(p \cdot M(p))$ cost of the term-by-term approach.

Once the integer triple P(0, N), Q(0, N), R(0, N) has been assembled, a single floating-point evaluation of the final formula at full precision p converts the exact rational result to the required approximation of π. This evaluation is performed once and represents only a small fraction of the total computation time.

The Chudnovsky series is handled in exactly the same way, with different recurrence coefficients determining the values of P, Q, and R at the leaves. Both series produce the same binary splitting structure and the same $O(M(p) \cdot \log^2(p))$ cost, though in practice the Chudnovsky series reaches the required precision with fewer terms ($N \approx p/14$ versus $N \approx p/8$ for Ramanujan), giving it a constant-factor advantage in addition to the shared asymptotic improvement.

It is worth noting that binary splitting is not restricted to these two series. Any hypergeometric series whose term ratio a_n/a_{n-1} is a rational function of n can be evaluated by the same method. The Borwein 25 and Borwein 50 series discussed in the preceding section also have rational term ratios and can in principle be accelerated by binary splitting, though in practice their faster per-term convergence already reduces N to the point where the sequential approach remains competitive at moderate precision.

Binary splitting of the Ramanujan infinite series

As explained above, instead of adding each series term, we try to find two integers P & Q that equate to the first k terms of the series:

$$\frac{1}{\pi}=\frac{2\sqrt{2}}{9801}\sum_{n=0}^{\infty}\frac{(4n)!(1103+26390n)}{(n!)^4 396^{4n}} =>$$

$$\frac{1}{\pi}=\frac{2\sqrt{2}}{9801}\frac{P}{Q}$$

Given the first k terms of the series, you get (see [15]), see also Appendix A for how we derived the binary splitting method from the Ramanujan Infinite series:

$$\frac{1}{\pi}=\frac{2\sqrt{2}}{9801}\frac{P(0,k)+1103Q(0,k)}{Q(0,k)} =>$$

$$\pi=\frac{9801Q(0,k)}{P(0,k)+1103Q(0,k)}\frac{1}{2\sqrt{2}}+O(96059301^{-k})$$

Where k, is found to satisfy the precision of the number. E.g., for precision P, we have equality:

$$10^{-p}< 96059301^{-k} => k>\frac{P\cdot\log(10)}{\log(96059301)}$$

We take k as the ceiling of $\left\lceil\frac{P\cdot\log(10)}{\log(96059301}\right\rceil$

To find the P(0,k) and Q(0,k) you can use a recursive formula for P(a,b) & Q(a,b) and a<b:

$$m=\frac{a+b}{2}\quad integer\quad division$$

$P(a,b)=P(a,m)Q(m,b)+P(m,b)R(a,m)$
$Q(a,b)=Q(a,m)Q(m,b)$
$R(a,b)=R(a,m)R(m,b)$

Where:
$P(b-1,b)=(1103+26390b)(2b-1)(4b-3)(4b-1)$
$Q(b-1,b)=3073907232b^3$
$R(b-1,b)=(2b-1)(4b-3)(4b-1)$

Since most computation uses arbitrary integer arithmetic, there is no loss of accuracy except for the final computation using arbitrary precision floating point. A conservative guard digit of 2 is used.

Algorithm 2.5 Binary splitting for Ramanujan π

```
// Ramanujan PI using the binary Splitting Method
static float_precision pi_ramanujan_binary(uintmax_t digits)
{
	const size_t prec = std::max((size_t)20U, digits + 2);
	const uintmax_t k = (uintmax_t)ceil(prec * log(10) / log(96059301ull));
	int_precision p, q, r;
	float_precision pi(0, prec);

	// Define the recursive binary splitting as a lambda.
	// std::function is required to give the lambda a named type
	// so it can capture itself by reference and call itself recursively.
	std::function<void(uintmax_t, uintmax_t, int_precision&, int_precision&,
int_precision&)> binarySplit;

	binarySplit = [&](uintmax_t a, uintmax_t b,
		int_precision& p, int_precision& q, int_precision& r)
		{
			if (b - a == 1)
			{
				if (b <= 832'256)
					r = int_precision((2 * b - 1) * (4 * b - 3) * (4 * b -
1));
				else
				{
					r = int_precision((4 * b - 3) * (4 * b - 1));
					r *= int_precision(2 * b - 1);
				}
				p = int_precision(1'103ull + 26'390ull * b);
				p *= r;
				if (b <= 2'642'245)
					q = int_precision(b * b * b);
				else
				{
					q = int_precision(b * b);
					q *= int_precision(b);
				}
				q *= int_precision(3'073'907'232ull);
				return;
			}

			const uintmax_t mid = (a + b) / 2;
			int_precision pp, qq, rr;

			binarySplit(a, mid, p, q, r);      // interval [a..mid]
			binarySplit(mid, b, pp, qq, rr);   // interval [mid..b]
```

```
                        // Reconstruct interval [a..b]
                        p = p * qq + pp * r;
                        q *= qq;
                        r *= rr;
                };

        binarySplit(0, k, p, q, r);

        pi = (float_precision(9801ull) * float_precision(q, prec))
                / (float_precision(p, prec) + float_precision(1103ull) * float_precision(q,
prec));
        pi *= float_precision(1) / sqrt(float_precision(8, prec));
        pi.precision(digits);
        return pi;
}
```

Binary splitting of the Chudnovsky infinite series

Instead of adding each series of terms, we try to find two integers, P & Q, that equate to the first k terms of the series.

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

$$\frac{1}{\pi}=\frac{\sqrt{10005}}{4270934400}\sum_{n=0}^{\infty}\frac{(-1)^n(6n)!(13591409+545140134n)}{(3n)!(n!)^3 640320^{3n}} =>$$

$$\frac{1}{\pi}=\frac{\sqrt{10005}}{4270934400}\frac{P}{Q}$$

Given the first k terms of the series, you get (see [15]). See also Appendix A for how the binary splitting method is derived from the Chudnovsky infinite series:

$$\frac{1}{\pi}=\frac{\sqrt{10005}}{4270934400}\frac{P(0,k)+13591409\,Q(0,k)}{Q(0,k)} =>$$

$$\pi=\frac{4270934400\cdot Q(0,k)}{P(0,k)+13591409\,Q(0,k)}\frac{1}{\sqrt{10005}}+O(151931373056000^{-k})$$

Where k, is found to satisfy the precision of the number. E.g. for precision P we have equality:

$$10^{-P}<151931373056000^{-k} => k>\frac{P\cdot\log(10)}{\log(151931373056000)}$$

We take k as the ceiling of $\left\lceil \frac{P\cdot\log(10)}{\log(151931373056000)} \right\rceil$

To find the P(0,k) and Q(0,k) you can use a recursive formula for P(a,b) & Q(a,b) and a<b:

$$m=\frac{a+b}{2}\quad integer\ \ division$$

P(a,b)=P(a,m)Q(m,b)+P(m,b)R(a,m)
Q(a,b)=Q(a,m)Q(m,b)
R(a,b)=R(a,m)R(m,b)

Where:
$P(b-1,b)=(13591409+545140134b)(2b-1)(6b-5)(6b-1)(-1)^b$
$Q(b-1,b)=10939058860032000b^3$
$R(b-1,b)= (2b-1)(6b-5)(6b-1)$

Since most computation uses arbitrary integer arithmetic, there is no loss of accuracy except for the final computation using arbitrary precision floating point. A conservative guard digit of 2 is used.

Algorithm 2.6 Binary splitting for Chudnovsky π

```
// The Chudnovsky series using the Binary splitting method
static float_precision pi_chudnovsky_binary(uintmax_t digits)
{
	const size_t prec = std::max((size_t)20U, digits + 2);
	const uintmax_t k = (uintmax_t)ceil(prec * log(10) / log(151'931'373'056'000ull));
	int_precision p, q, r;
	float_precision pi(0, prec);

	// Recursive binary splitting embedded as a lambda.
	// std::function is required to allow the lambda to call itself by name.
	std::function<void(uintmax_t, uintmax_t, int_precision&, int_precision&, 
int_precision&)>
		binarySplit;

	binarySplit = [&](uintmax_t a, uintmax_t b,
		int_precision& p, int_precision& q, int_precision& r)
		{
			if (b - a == 1)
			{
				if (b < 635'130)    // No overflow using 64-bit arithmetic
					r = int_precision((2 * b - 1) * (6 * b - 5) * (6 * b - 
1));
				else
				{		// No overflow if b <= 715'827'883 (~10B 
digits)
					r = int_precision((6 * b - 5) * (6 * b - 1));
					r *= int_precision(2 * b - 1);
				}
				p = int_precision(13'591'409ull + 545'140'134ull * b);
				p *= r;
				if (b & 0x1)
					p.change_sign();    // encodes the (-1)^n alternating 
sign
				if (b <= 2'642'245)
					q = int_precision(b * b * b);
				else
				{
```

```
                                q = int_precision(b * b);
                                q *= int_precision(b);
                        }
                        q *= int_precision(10'939'058'860'032'000ull);
                        return;
                }

                const uintmax_t mid = (a + b) / 2;
                int_precision pp, qq, rr;

                binarySplit(a, mid, p, q, r);       // interval [a..mid]
                binarySplit(mid, b, pp, qq, rr);    // interval [mid..b]

                // Reconstruct interval [a..b]
                p = p * qq + pp * r;
                q *= qq;
                r *= rr;
        };

        if (digits >= 1000)
        {   // Split at 3k/4 for the asynchronous two-segment case
                int_precision pp, qq, rr;
                binarySplit(0, k / 2, p, q, r);
                binarySplit(k / 2, k, pp, qq, rr);
                p = p * qq + pp * r;
                q *= qq;
                r *= rr;
        }
        else
                binarySplit(0, k, p, q, r);

        pi = (float_precision(4'270'934'400ull) * float_precision(q, prec))
                / (float_precision(p, prec) + float_precision(13'591'409ull) *
float_precision(q, prec));
        pi *= 1 / sqrt(float_precision(10'005, prec));
        pi.precision(digits);
        return pi;
}
```

Binary splitting of the Borwein25 infinite series

Borwein25 infinite series is given by:

$$\frac{1}{\pi}=12\sum_{n=0}^{\infty}\frac{(-1)^n(6n)!(A+nB)}{(3n)!(n!)^3C^{n+1/2}} =>$$

$$\frac{1}{\pi}=\frac{12}{\sqrt{C}}\frac{P}{Q}$$

Given the first k terms, you get (see Appendix A)

$$\pi=\frac{\sqrt{C}\cdot Q(0,k)}{12\cdot(P(0,k)+A\cdot Q(0,k))}$$

To find the P(0,k) and Q(0,k) you can use a recursive formula for P(a,b) & Q(a,b) and a<b: Please note that since the constant A, B, C & D is no longer integers (as it was for

the Ramanujan and Chudnovsky series) P(a,b), Q(a,b) is arbitrary precision floating point numbers, however, R(a,b) can still be kept as arbitrary precision integers)

$$m = \frac{a+b}{2} \quad integer \quad division$$

P(a,b)=P(a,m)Q(m,b)+P(m,b)R(a,m)
Q(a,b)=Q(a,m)Q(m,b)
R(a,b)=R(a,m)R(m,b)

Where:
$P(b-1,b)= (A+Bb)(2b-1)(6b-5)(6b-1)(-1)^b$
$Q(b-1,b)=Db^3$
R(b-1,b)= (2b-1)(6b-5)(6b-1)

Where A, B, and C are the constants from the description of Borwein25 infinite series and $D = \frac{C}{24}$.

An error analysis in Appendix G recommends a guard digit count of $\log_{10}$(digits) + 2.

Algorithm 2.7 Binary splitting for Borwein25 π

```
// Borwein series implemented using the binary splitting method
static float_precision pi_borwein25_binary(uintmax_t digits)
{
	const size_t guard = (size_t)std::ceil(std::log10((double)digits)) + 2;
	 const size_t prec = std::max((size_t)20U, digits + guard);
	const size_t prec = std::max((size_t)20U, digits + 2);

	const uintmax_t k = (uintmax_t)ceil(prec / 24) + 1;
	int_precision r;
	const float_precision c212175710912(212175710912), c13773980892672(13773980892672);
	const float_precision c12(12), c159999840(159999840);
	float_precision p(0, prec), q(0, prec), pi(0, prec);
	float_precision A(1657145277365, prec), B(107578229802750, prec), C(1249638720, 
prec);
	float_precision Cover24(24, prec), c61sq(61, prec);

	piSplitting = 0;
	c61sq = sqrt(c61sq);
	A += c212175710912 * c61sq;
	B += c13773980892672 * c61sq;
	C += c159999840 * c61sq;
	C *= C.square();		// C = C^3
	Cover24 = C / Cover24;		// Cover24 = C^3 / 24

	// Recursive binary splitting embedded as a lambda.
	// p and q remain float_precision throughout to match the original behaviour.
	std::function<void(uintmax_t, uintmax_t,
		float_precision&, float_precision&, int_precision&)>
		binarySplit;

	binarySplit = [&](uintmax_t a, uintmax_t b,
		float_precision& p, float_precision& q, int_precision& r)
```

```
		{
			++piSplitting;
			if (b - a == 1)
			{
				if (b < 635'130)
					r = int_precision((2 * b - 1) * (6 * b - 5) * (6 * b - 
1));
				else
				{
					r = int_precision((6 * b - 5) * (6 * b - 1));
					r *= int_precision(2 * b - 1);
				}
				p = A + B * float_precision(b);
				p *= r;
				if (b & 0x1)
					p.change_sign();
				if (b <= 2'642'245)
					q = float_precision(int_precision(b * b * b), prec);
				else
				{
					int_precision iq(b * b);
					iq *= int_precision(b);
					q = float_precision(iq, prec);
				}
				q *= Cover24;
				return;
			}

			const uintmax_t mid = (a + b) / 2;
			float_precision pp(0, prec), qq(0, prec);
			int_precision rr;

			binarySplit(a, mid, p, q, r);
			binarySplit(mid, b, pp, qq, rr);

			p = p * qq + pp * float_precision(r, prec);
			q *= qq;
			r *= rr;
		};

	binarySplit(0, k, p, q, r);
	pi = sqrt(C) * q / (c12 * (p + A * q));
	pi.precision(digits);
	return pi;
}
```

Binary splitting of the Borwein50 infinite series

Borwein50 Infinite series is given by:

$$\frac{1}{\pi}=\frac{1}{\sqrt{-C^3}}\sum_{n=0}^{\infty}\frac{(6n)!(A+nB)}{(3n)!(n!)^3C^{3n}}$$

Given the first k terms, you get (see Appendix A):

$$\pi=\frac{\sqrt{-C^3}\cdot Q(0,k)}{(P(0,k)+A\cdot Q(0,k))}$$

To find the P(0,k) and Q(0,k) you can use a recursive formula for P(a,b) & Q(a,b) and a<b: Please note that since the constant A, B, C & D is no longer integers (as it was for the Ramanujan and Chudnovsky series) P(a,b), Q(a,b) is arbitrary precision floating point numbers, however, R(a,b) can still be kept as arbitrary precision integers).

$$m = \frac{a+b}{2} \quad integer \quad division$$

P(a,b)=P(a,m)Q(m,b)+P(m,b)R(a,m)
Q(a,b)=Q(a,m)Q(m,b)
R(a,b)=R(a,m)R(m,b)

Where:
$P(b-1,b)= (A+Bb)(2b-1)(6b-5)(6b-1)(-1)^b$
$Q(b-1,b)=Db^3$
R(b-1,b)= (2b-1)(6b-5)(6b-1)

Where A, B, and C are the constants from the description of Borwein50 infinite series and $D = \frac{C^3}{24}$.

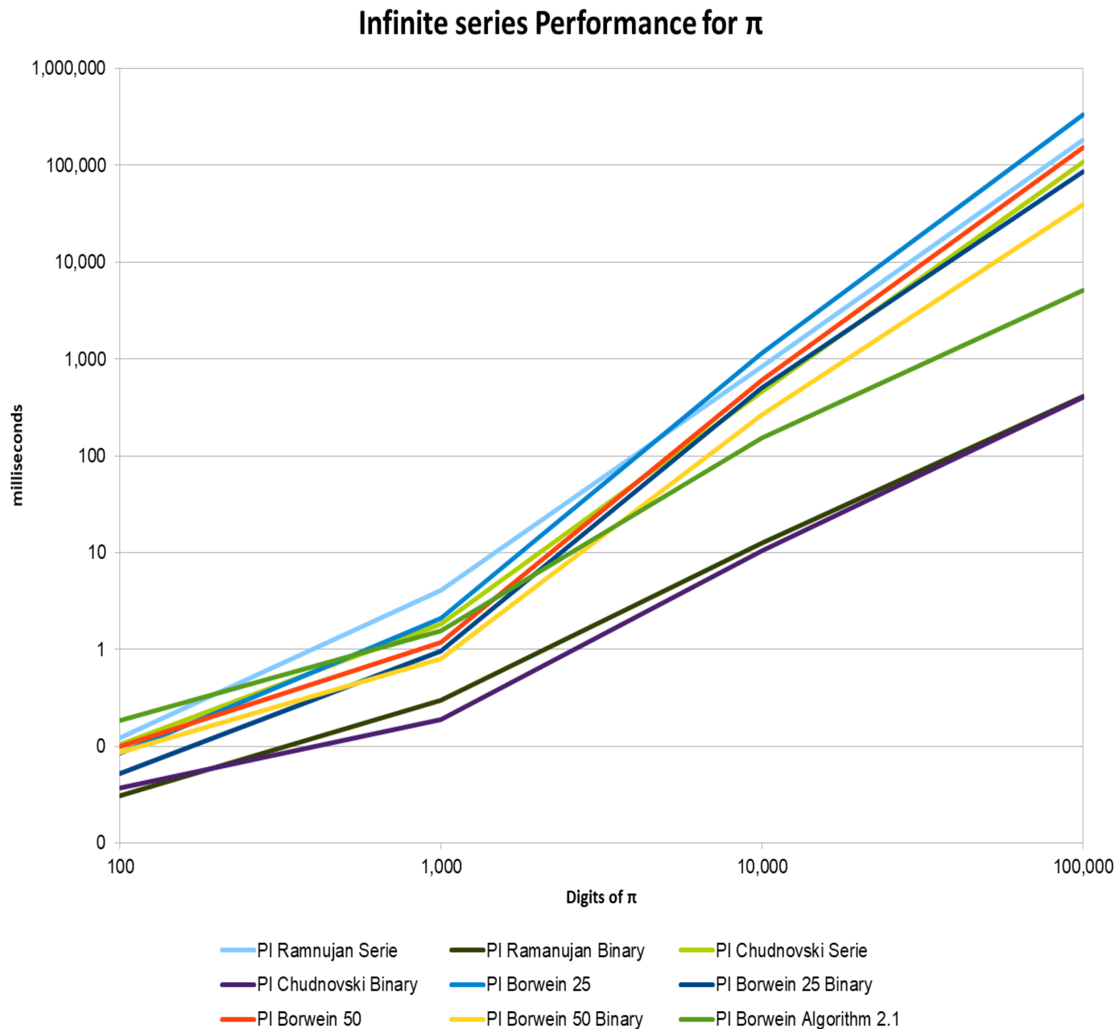
In Appendix H, an error analysis has revealed the same guard-digit computation as in the Borwein 25 binary-splitting method.

Algorithm 2.8 Binary splitting for Borwein50 π

```
// Binary splitting method using Borwein 50 Infinite series formula
static float_precision pi_borwein50_binary(uintmax_t digits)
{
	const size_t guard = (size_t)std::ceil(std::log10((double)digits)) + 2;
	const size_t prec = std::max((size_t)20U, digits + guard);
	const uintmax_t k = (uintmax_t)ceil(prec / 49.0) + 1;
	int_precision r;
	float_precision p(0, prec), q(0, prec), pi(0, prec);
	float_precision A(0, prec), B(0, prec), C(0, prec);
	float_precision Cover24(24, prec), csq5(5, prec), tmp(0, prec);

	piSplitting = 0;

	csq5 = sqrt(csq5);
	// Build Constant A
	A = float_precision("63365028312971999585426220", prec);
	A += float_precision("28337702140800842046825600", prec) * csq5;
	tmp = float_precision("10891728551171178200467436212395209160385656017", prec);
	tmp += float_precision("4870929086578810225077338534541688721351255040", prec) *
csq5;
	tmp = sqrt(tmp);
	A += float_precision(384, prec) * csq5 * tmp;
	// Build Constant B
	B = float_precision("7849910453496627210289749000", prec);
	B += float_precision("3510586678260932028965606400", prec) * csq5;
	tmp = float_precision("6260208323789001636993322654444020882161", prec);
	tmp += float_precision("2799650273060444296577206890718825190235", prec) * csq5;
	tmp = sqrt(tmp);
	B += float_precision(2515968, prec) * sqrt(float_precision(3110, prec)) * tmp;
```

```
        // Build Constant C
        C = float_precision(-214772995063512240, prec);
        C += float_precision(-96049403338648032, prec) * csq5;
        tmp = float_precision("10985234579463550323713318473", prec);
        tmp += float_precision("4912746253692362754607395912", prec) * csq5;
        tmp = sqrt(tmp);
        C += float_precision(-1296, prec) * csq5 * tmp;
        // Build C3 = C^3, then Cover24 = C^3 / 24
        C = C * C.square();   // C = C^3  (negative, since C < 0)
        Cover24 = C / Cover24;      // Cover24 = C^3 / 24

        // Recursive binary splitting embedded as a lambda.
        // A, B, Cover24 captured by reference from enclosing scope.
        // p and q remain float_precision to accommodate the irrational constants.
        std::function<void(uintmax_t, uintmax_t,
                float_precision&, float_precision&, int_precision&)>
                binarySplit;

        binarySplit = [&](uintmax_t a, uintmax_t b,
                float_precision& p, float_precision& q, int_precision& r)
                {
                        ++piSplitting;
                        if (b - a == 1)
                        {
                                if (b < 635'130)        // No overflow using 64-bit arithmetic
                                        r = int_precision((2 * b - 1) * (6 * b - 5) * (6 * b - 
1));
                                else
                                {                    // No overflow if b <= 715'827'883 (~34B 
digits)
                                        r = int_precision((6 * b - 5) * (6 * b - 1));
                                        r *= int_precision(2 * b - 1);
                                }
                                p = A + B * float_precision(b);
                                p *= r;
                                // No explicit sign change: alternating sign is encoded
                                // in the negativity of C^3, which flips sign via Cover24 each 
step
                                if (b <= 2'642'245)
                                        q = float_precision(int_precision(b * b * b), prec);
                                else
                                {
                                        int_precision iq(b * b);
                                        iq *= int_precision(b);
                                        q = float_precision(iq, prec);
                                }
                                q *= Cover24;
                                return;
                        }

                        const uintmax_t mid = (a + b) / 2;
                        float_precision pp(0, prec), qq(0, prec);
                        int_precision rr;

                        binarySplit(a, mid, p, q, r);
                        binarySplit(mid, b, pp, qq, rr);

                        // r is the LEFT half's r — used in the recurrence p(a,b) = 
p(a,m)*q(m,b) + p(m,b)*r(a,m)
                        p = p * qq + pp * float_precision(r, prec);
                        q *= qq;
                        r *= rr;
                };

        binarySplit(0, k, p, q, r);
        pi = sqrt(-C) * q / (p + A * q);  // -C > 0 since C = C^3 < 0
        pi.precision(digits);
        return pi;
}
```

Speed Comparison of the Infinite Series



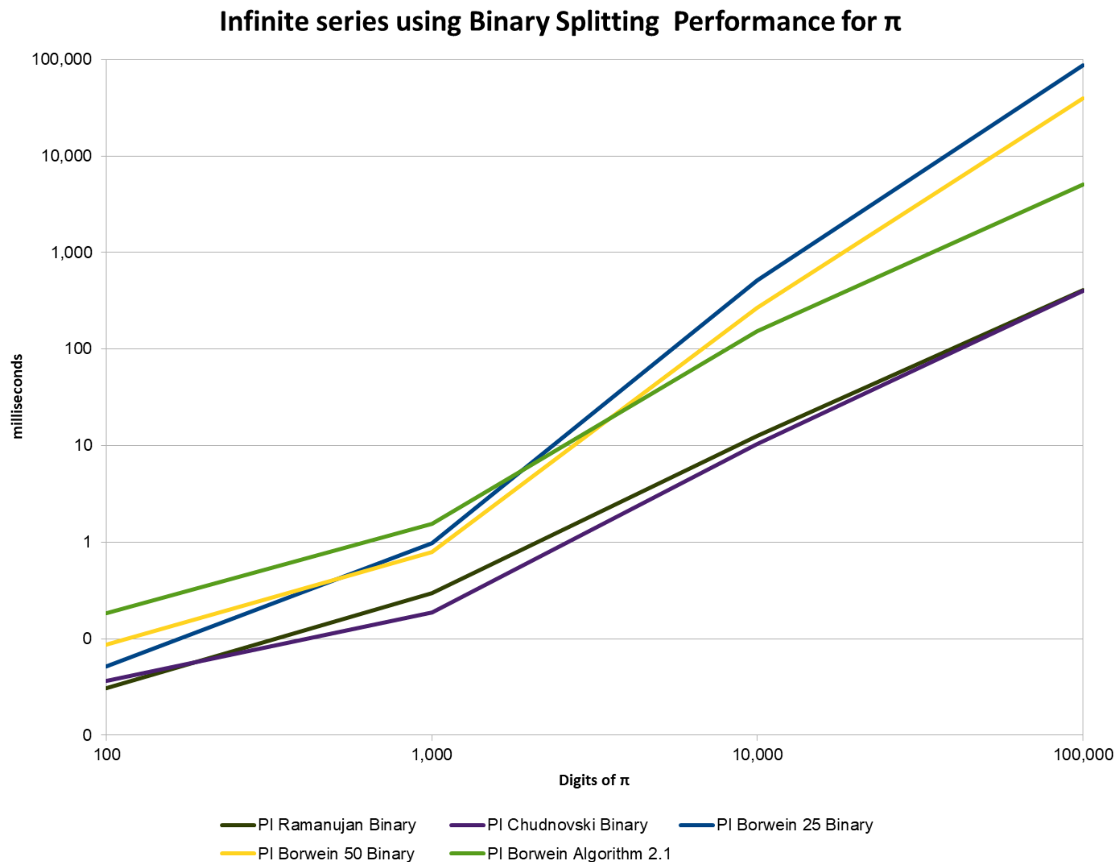
The chart above compares the Ramanujan, Chudnovsky, and Borwein series with the standard method and their binary-splitting counterparts.

The first observation is that all series, evaluated term by term, are substantially slower than their binary-splitting equivalents. This is expected given the fundamental difference in how the two approaches operate: the sequential series performs each addition at full working precision, while binary splitting restructures the computation as integer arithmetic over a recursion tree, deferring the single floating-point conversion to the very end.

Among the series, the Borwein 25 and Borwein 50 evaluations are slower than both the Ramanujan and Chudnovsky series, despite their higher per-term convergence rate. The

reason is that each Borwein term involves more arithmetic to construct, and the constant in the per-term cost outweighs the benefit of needing fewer terms. Within the Ramanujan-Chudnovsky pair, Chudnovsky has a consistent edge because it generates approximately 14.18 correct digits per term, compared to Ramanujan's 7.98, requiring roughly half as many iterations to achieve the same precision. Compared to Borwein algorithm 2.1 described in the next chapter, all sequential series evaluations are slower, which confirms that traditional term-by-term series evaluation is not the recommended approach for computing π at arbitrary precision.

The binary splitting results show a different ordering.



Ramanujan and Chudnovsky binary splitting achieve nearly identical performance, with Chudnovsky maintaining a modest edge from its higher per-term digit yield. At 100,000 digits both are approximately 200 to 450 times faster than their corresponding sequential series implementations, which is a substantial gain. The Borwein 25 and Borwein 50 binary splitting implementations are considerably slower than Ramanujan and Chudnovsky binary splitting, despite the Borwein series producing two to four times as many correct digits per split. The cause is the one identified in the guard-digit analysis: Borwein 25 and Borwein 50 require float-precision arithmetic throughout the recursion

because the constants A and B involve irrational numbers that cannot be represented as integers. Ramanujan and Chudnovsky perform all recursive merging in exact integer arithmetic, which is significantly faster at high precision. The Borwein binary-splitting implementations nonetheless show a large performance improvement over their sequential counterparts, confirming that the asymptotic advantage of binary splitting applies regardless of the arithmetic type used.

Recommendation for the Infinite Series

Although many other infinite series for π exist (Pi Formulas -- from Wolfram MathWorld), the Chudnovsky series, combined with binary splitting, appears to occupy a sweet spot where convergence rate, simplicity of the algebraic constants, and implementation efficiency are well-balanced. Over the past two decades, it has become the algorithm of choice for record-breaking computations of π. The recommendation here is therefore to use the Chudnovsky binary splitting method, which offers the best performance among the implementations examined in this paper. It is this method that underlies the record-breaking computation of π to 314 trillion digits in 2026.

3. Higher-order algorithm for π

In this section, we describe the family of higher-order iterative methods for computing π that were developed from the mid-1970s through the 1990s. These methods differ fundamentally from the infinite series in the previous chapter: rather than summing many terms, they iterate a small set of recurrence relations that converge to π at an exponential rate with the iteration count. Each iteration produces a fixed multiple of the previous number of correct digits, so only a handful of iterations is needed to reach any practical precision.

The chapter begins with the Brent-Salamin and Gauss-Legendre algorithms, both from 1976, which achieve quadratic convergence using the arithmetic-geometric mean. These are followed by several algorithms due to Jonathan and Peter Borwein, who extended the AGM framework to produce methods of cubic, quartic, quintic, and nonic order. A selection of their quadratic algorithms from 1984 and 1985 is also included for completeness and historical context.

As the benchmarks at the end of the chapter show, higher convergence order does not automatically translate into better wall-clock performance. The Brent-Salamin and Gauss-Legendre algorithms consistently outperform the higher-order Borwein variants because each additional order of convergence comes with a proportionally larger number of arithmetic operations per step. Brent-Salamin remains the recommended algorithm within this family.

Brent-Salamin method for π

This is another famous algorithm for quickly calculating π. First published in 1976 by Brent. It uses the arithmetic-geometric mean to calculate π. The convergence rate is quadratic in the mean for each iteration; it has twice as many correct digits as the previous iteration.

It uses the following recursion, starting with the initial conditions:

$$a_0=1,\ b_0=\frac{1}{\sqrt{2}},\ c_0=0.5$$

Then repeat for n=0,1,2… until sufficient accuracy has been obtained.

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

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

$$c_{n+1}=c_n-2^{n+1}(a_{n+1}-b_n)^2$$

$$\pi_{n+1}=2\frac{a_{n+1}^2}{c_{n+1}}$$

In [10] Borwein has $c_{n+1}=c_n-2^{n+1}(a_{n+1}^2-b_{n+1}^2)$ which is the same as the c_{n+1} above as shown here:

$$c_n-2^{n+1}(a_{n+1}-b_n)^2=$$

$$c_n-2^{n+1}(a_{n+1}^2+b_n^2-2a_{n+1}b_n)=$$

$$c_n-2^{n+1}(a_{n+1}^2+b_n^2-2(\frac{a_n+b_n}{2})b_n)=$$

$$c_n-2^{n+1}(a_{n+1}^2+b_n^2-b_n^2-a_n b_n)=$$

$$c_n-2^{n+1}(a_{n+1}^2-a_n b_n)=$$

$$c_n-2^{n+1}(a_{n+1}^2-b_{n+1}^2)$$

An iteration looks like this and after four iterations; we have reached π to 15 digits of accuracy.

Brent-Salami

Iteration	a	b	c	π	Error
0	1	0.707107	0.5		8.58E-01
1	0.853553	0.840896	0.457107	3.18767264271211	4.61E-02
2	0.847225	0.847201	0.456947	3.14168029329765	8.76E-05
3	0.847213	0.847213	0.456947	3.14159265389545	3.06E-10
4	0.847213	0.847213	0.456947	3.14159265358979	8.88E-16

Algorithm 3.1 Brent-Salamin

```
// Brent-Salamin Algorithm for pi
static float_precision pi_brent_salamin(uintmax_t digits)
	{
	const size_t prec = digits + 2 + (size_t)(log10(digits) + 0.5);
	const eptype limit = -(eptype)ceil((digits+1)*log2(10));
	float_precision a(1, prec), b(2, prec), sum(0.5, prec);
	float_precision ak(0, prec), bk(0, prec), ck(1, prec);
	float_precision ab(0, prec), asq(0, prec);
	float_precision pow2(1,digits), pi(3,digits);
	const float_precision c1(1), c2(2);

	b = c1 / sqrt(b);
	for ( ; !ck.iszero() && ck.exponent()>limit; )
		{
		ak = a + b;
		ak.adjustExponent(-1); // ak*=0.5
		ab = a * b;
		bk = sqrt(ab);
		asq = ak.square();
		ck = asq - ab;
		pow2.adjustExponent(+1);  // pow2*=2
		sum -= pow2*ck;
		a = ak; b = bk;
		}

	pi = c2 * asq / sum;
	return pi;
	}
// End Brent-Salamin
```

Gauss-Legendre method for π

Another algorithm, as good as the Brent-Salamin algorithm, is the Gauss-Legendre algorithm, a variation of the Brent-Salamin algorithm.

The Gauss-Legendre starts with the initial settings: $a_0=1,\ b_0=\frac{1}{\sqrt{2}},\ t_0=0.25$

Then repeat for n=0,1,2,3,…

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

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

$$t_{n+1}=t_n-2^n(a_n-a_{n+1})^2$$

$$\pi_{n+1}=\frac{(a_{n+1}+b_{n+1})^2}{4t_{n+1}}$$

Gauss-Legendre

Iteration	a	b	t	π	Error
0	1	0.707107	0.25	2.91421356237309	2.27E-01
1	0.853553	0.840896	0.228553	3.14057925052217	1.01E-03
2	0.847225	0.847201	0.228473	3.14159264621354	7.38E-09
3	0.847213	0.847213	0.228473	3.14159265358979	8.88E-16

The result is nearly identical to the Brent-Salamin algorithm.

Algorithm 3.2 Gauss-Legendre

```
// Begin Gauss-Legendre
static float_precision pi_gauss_legendre(uintmax_t digits)
	{
	const size_t prec = digits+5+(size_t)(log10(digits)+0.5);
	const eptype limit = -(eptype)ceil((digits + 2) * log2(10));
	const float_precision c0(0), c1(1), c4(4);
	float_precision a(1,prec), b(2,prec), pow2(0.5,digits);
	float_precision ak(0,prec), bk(0,prec), tk(0.25,prec);
	float_precision pi(3,digits), tmp(0,prec), sq(1,prec);

	b = c1 / sqrt(b);
	for ( ; sq != c0 && sq.exponent()>limit; )
		{
		ak = a + b;
		ak.adjustExponent(-1);  // ak*=0.5
		bk = sqrt(a * b);
		pow2.adjustExponent(+1);// pow2*=2
		sq = (a - ak);
		tk -= pow2*sq.square();
		a = ak;
		b = bk;
		}

	tmp = ak + bk;
	pi = (tmp * tmp) / (c4 * tk);
	return pi;
	}
// end Gauss-Legendre
```

Borwein Brothers Algorithms for PI

The Borwein brothers conducted extensive research on the subject and wrote an excellent book [4], “PI and the AGM,” in which they presented several interesting algorithms for

calculating π. Furthermore, they have published several articles over the years. References [8], [9], [10], [11] & [12] are great papers on the subject that also include a historical walkthrough of the progress in calculating π from ancient times to the present.

Borwein Quadratic Algorithm 2.1 from 1987

One of the early ones is their 1987 quadratic algorithm (Algorithm 2.1 in [4]), derived from the Gauss Arithmetic-Geometric Mean (AGM) iteration.

Let:

$$x_0=\sqrt{2},\ \ y_0=\sqrt[4]{2},\ \ \pi_0=2+\sqrt{2}$$

Now iterate through n=1,2,3…

$$x_n=\frac{1}{2}\left(\sqrt{x_{n-1}}+\frac{1}{\sqrt{x_{n-1}}}\right)$$

$$y_n=\frac{y_{n-1}\sqrt{x_n}+\frac{1}{\sqrt{x_n}}}{y_{n-1}+1}$$

$$\pi_n=\pi_{n-1}\frac{x_n}{y_{n-1}}$$

The algorithm is efficient, with quadratic convergence. Borwein states that the first nine iterations yield the first 1, 3, 8, 19, 41, 83, 170, 345, and 694 digits of π. This corresponds exactly with the result below of the first four iterations.

Borwein Algorithm 2.1

Iteration	x	π	y	Error
0	1.414214	3.41421356237309	1.189207	0.272621
1	1.015052	3.14260675394162	1.000673	0.001014
2	1.000028	3.14159266096604	1	7.38E-09
3	1	3.14159265358979	1	0

Algorithm 3.3 Borwein Quadratic 2.1

```
// Begin Borwein algorithm 2.1
static float_precision pi_borwein_algorithm2_1(uintmax_t digits)
	{
	const size_t prec = digits + 2 + (size_t)(log10(digits) + 0.5);
	const float_precision c1(1), c2(2);
	float_precision x(2,prec), y(0,prec), z(0,prec);
	float_precision xsq(0,prec), xsq_inv(0,prec);
```

```
	float_precision r(0,prec), rold(1,prec);
	float_precision pi(0,prec);

	// Initial setup.
	// x0 = sqrt(2), pi = 2 + sqrt(2), y = 2^(1/4)
	x = sqrt(x);
	xsq = sqrt(x);
	pi = x + c2;
	xsq_inv = xsq.inverse();
	y = xsq;

	// Iterate.
	// x = 0.5(sqrt(x)+1/sqrt(x)),
	// pi=pi((x+1)/(y+1)),
	// y=(y*sqrt(x)+1/sqrt(x))/(y+1)
	for ( ;; )
		{
		x = (xsq + xsq_inv);
		x.adjustExponent(-1);   // x*= 0.5
		r = (x + c1) / (y + c1);
		pi *= r;
		xsq = sqrt(x);
		xsq_inv = xsq.inverse();
		y = (y * xsq+xsq_inv) / (y + c1);
		// Stop when r==1 or when r doesn't change within an iteration
		if (r == c1 || rold == r)
			break;
		rold = r;
		}

	pi.precision(digits);   // Round to desired precision
	return pi;
	}
```

Borwein Quadratic 1984

Another Quadratic algorithm from 1984. The iteration goes as follows:

Let the initial conditions be: $x_0=\sqrt{2}, \quad y_0=0, \quad \pi_0=2+\sqrt{2}$

Then iterate through n=1,2,3…

$$x_n=\frac{1}{2}\left(\sqrt{x_{n-1}}+\frac{1}{\sqrt{x_{n-1}}}\right)$$

$$y_n=\frac{(1+y_{n-1})\sqrt{x_{n-1}}}{x_{n-1}+y_{n-1}}$$

$$\pi_n=\pi_{n-1}y_n\frac{x_n+1}{y_n+1}$$

π Borwein 1984

Iteration	x	y	π	Error
0	1.414214	0	3.41421356237309	0.272621
1	1.015052	0.840896	3.14260675394162	0.001014

2	1.000028	0.999327	3.14159266096604	7.38E-09
3	1	1	3.14159265358979	0

Compared to Borwein's 1987 algorithm, it is identical in the iteration steps.

Algorithm 3.4 Borwein Quadratic 1984

```
// Borwein Quadratic 1984
static float_precision pi_borwein_1984(unsigned int digits)
	{
	const size_t prec = digits + 2 + (size_t)(log10(digits) + 0.5);
	const float_precision c1(1), c2(2);
	float_precision x(2, prec), y(0, prec);
	float_precision xsq(0, prec), xsq_inv(0, prec);
	float_precision r(0, prec), rold(1, prec);
	float_precision pi(0, prec);

	// Initial Setup
	// x0 = sqrt(2), pi = 2 + sqrt(2), y = 0
	x = sqrt(x);
	xsq = sqrt(x);
	pi = x + c2;
	xsq_inv = xsq.inverse();
	// Iterate.
	// y=((1+y)*sqrt(x))/(x+y+1),
	// x = 0.5(sqrt(x)+1/sqrt(x)),
	// pi=pi(y(x+1)/(y+1))
	for (;;)
		{
		y = (c1 + y)*xsq/(x+y);
		x = (xsq + xsq_inv);
		x.adjustExponent(-1);  // x*=c05;
		r = (x + c1) / (y + c1);
		r *= y;
		pi *= r;
		xsq = sqrt(x);
		xsq_inv = xsq.inverse();
		// Stop when r gets 1 or r doesn't change within an iteration
		if (r == c1 || rold == r)
			break;
		rold = r;
		}

	pi.precision(digits);   // Round to desired precision
	return pi;
	}
```

Borwein Quadratic algorithm 1985

Another algorithm, published shortly after the 1984 algorithm, is the one below; it still has quadratic convergence and is similar in performance to the 1984 algorithm.

Let the initial conditions be: $a_0=6-4\sqrt{2},\ y_0=\sqrt{2}-1$

And then for n=1,2,3…

$$f\left(y_{n-1}\right)=\sqrt[4]{1-y_{n-1}^4}$$

$$y_n = \frac{1 - f(y_{n-1})}{1 + f(y_{n-1})}$$

$$a_n = a_{n-1}(1 + y_n)^4 - 2^{2n+1} y_n (1 + y_n + y_n^2)$$

$$\pi_n = \frac{1}{a_n}$$

Iteration	Y_n	$f(y_n)$	a_n	π_n	Error
0	4.14E-01	0.992558024001326	0.343145750507619	2.91421356237310	0.22737
1	3.73E-03	0.999999999951354	0.318309886931161	3.14159264621355	7.38E-09
2	2.43E-11	1.000000000000000	0.318309886183791	3.14159265358979	1.33E-15

This particular algorithm has been used as one of the methods in the quest to calculate π to the highest number of digits. In 1986, it was used to compute 29.4 million digits of π, and reach over 6 billion digits in 1995 [8], and in 2009, Takahashi calculated more than 2.5 trillion digits using this algorithm [9]

Algorithm 3.5 Borwein Quadratic 1985

```
// Borwein quadratic algorithm from 1985
static float_precision pi_borwein_1985(unsigned int digits)
	{
	const size_t prec= digits + 2 + (size_t)(log10(digits) + 0.5);
	const float_precision c1(1), c6(6);
	float_precision a(2, prec), y(2, prec);
	float_precision pow2(2, prec), fy(0, prec);
	float_precision an(0, prec), yn(1, prec),  yn1(0,prec);
	float_precision pi(0, prec), tmp(0,prec);

	// Initial Setup
	a = sqrt(a);                    // sqrt(2)
	y = a - c1;                     // sqrt(2)-1
	a.adjustExponent(+2);           // 4*sqrt(2)
	a = c6 - a;                     // 6-4*sqrt(2)
	for ( ; ; )
		{
		fy = y.square();         // y^2
		fy = fy.square();        // y^4
		fy = c1 - fy;            // 1 - y^4
		fy = nroot(fy, 4);       //(1-y^4)^(1/4)
		yn = (c1-fy)/(c1+fy);    // (1-f(y))/(1+f(y))
		// stop if yn ==0 => an==a and no further improvement
		if (yn.iszero())
			break;
		pow2.adjustExponent(+2); // pow2*=4;  2^(2n+1)
		yn1 = c1 + yn;           // 1+y
		tmp = yn1.square();      // (1+y)^2
		tmp = tmp.square();      // (1+y)^4
		an = a*tmp - pow2*yn*(yn1 + yn*yn);  // a=a*(1+y)^4+2^(2n+1)(1+y+y^2)
		a = an;
		y = yn;
		}
```

```
        pi = an.inverse();                  // pi = 1/a
        pi.precision(digits);               // Round to precision
        return pi;
        }
```

Borwein Cubic 1991

Through continued research on convergent series, the Borwein brothers discovered additional series with higher-order convergence rates than the four mentioned above. The Cubic version was from 1991 and goes as follows:

Let the initial conditions be: $a_0 = \frac{1}{3}, \quad s_0 = \frac{\sqrt{3}-1}{2}$

And then for n=1,2,3…

$$r_n = \frac{3}{1 + 2\sqrt[3]{1 - s_{n-1}^3}}$$

$$s_n = \frac{r_n - 1}{2}$$

$$a_n = r_n^2 a_{n-1} - 3^{n-1}\left(r_n^2 - 1\right)$$

$$\pi_n = \frac{1}{a_n}$$

Then π_n converges cubically, meaning that with each iteration the number of correct digits triples.

π Borwein 1991

Iteration	r	s	a	π	Error
0		0.366025	0.333333	3.0000000000000	0.141593
1	1.011205	0.005602	0.31831	3.1415905852059	2.07E-06
2	1	1.95E-08	0.31831	3.1415926535898	1.11E-14

Algorithm 3.6 Borwein Cubic 1991

```
// Borwein cubic algorithm from 1991
static float_precision pi_borwein_cubic_1991(unsigned int digits)
        {
        const size_t prec = digits + 2 + (size_t)(log10(digits) + 0.5);
        bool first_time = true;
        const float_precision c1(1), c3(3);
        float_precision a(3, prec), r(0,prec), s(3, prec);
        float_precision pow3(1, prec), rsq(0, prec);
        float_precision pi(0, prec);
```

```
        a = a.inverse();                    // 1/3
        s =(sqrt(s) - c1);                  // (sqrt(3)-1)
        s.adjustExponent(-1);               // s*= 0.5
        for ( ; ;)
                {
                r = nroot(c1-s.square()*s, 3);  // (1-s^3)^(1/3)
                // Break if r==1 => no further improvement
                if (r == c1)
                        break;                          //Most common break point
                r.adjustExponent(+1);   // 2(1-s^3)^(1/3)
                r += c1;                // 1+2(1-s^3)^(1/3)
                r = c3 / r;             // 3/(1+2(1-s^3)^(1/3))
                // Break if r==1 => no further improvement
                if (r == c1)
                        break;          // Break point rarely
                s = ( r - c1 );         // (r-1)
                s.adjustExponent(-1);   // 0.5*(r-1)
                if (first_time == true)
                        {
                        // pow3=3^0=1 already initialized
                        first_time = false;
                        }
                else
                        pow3 *= c3;                     // 3^(n-1)
                rsq = r.square();
                a = a * rsq - pow3 *(rsq - c1);
                }

        pi = a.inverse();                   // pi = 1/a
        pi.precision(digits);               // Round to precision
        return pi;
        }
```

Borwein Quintic (fifth order) Algorithm

As Borwein conducted further research, they discovered a higher-order convergence algorithm for π.

This is the quintic algorithm, where the number of correct digits quintuples after each iteration.

Let the initial conditions be: $a_0=\frac{1}{2}, \quad s_0=5(\sqrt{5}-2)$

Then for n=1,2,3…

$$x_n=\frac{5}{s_{n-1}}-1$$

$$y_n=(x_n-1)^2+7$$

$$z_n=\sqrt[5]{\frac{1}{2}x_n\left(y_n+\sqrt{y_n^2-4x_n^3}\right)}$$

$$a_n=s_{n-1}^2a_{n-1}-5^{n-1}\left(\frac{s_{n-1}^2-5}{2}+\sqrt{s_{n-1}\left(s_{n-1}^2-2s_{n-1}+5\right)}\right)$$

$$s_n=\frac{25}{\left(z_n+\frac{x_n}{z_n}+1\right)^2s_{n-1}}$$

$$\pi_n=\frac{1}{a_n}$$

As you can see, it requires only two iterations to get the result.

π Borwein Quintic

Iteration	x	y	z	a	s	π	error
0				0.5	1.18034	n	
1	3.236068	12	1.89003 9	0.318316	1.000001	3.14153694751872	5.57E-05
2	3.999997	15.99998	2	0.31831	1	3.14159265358980	4.44E-15

Algorithm 3.7 Borwein quintic (fifth order convergence)

```
// Borwein quintic algorithm
static float_precision pi_borwein_quintic(unsigned int digits)
	{
	const size_t prec = digits + 2 + (size_t)(log10(digits) + 0.5);
	bool first_time = true;
	const float_precision c0(0), c1(1), c2(2), c05(0.5), c4(4), c5(5), c7(7);
	const float_precision c16(16), c25(25);
	float_precision a(0.5, prec), an(0, prec), s(5, prec);
	float_precision x(0, prec), y(0, prec), z(0, prec);
	float_precision pow5(1, prec), ssq(0, prec), tmp(0,prec);
	float_precision pi(0, prec);
	const eptype limit = -(eptype)ceil((digits + 2)*log2(10));

	s = c5*(sqrt(s) - c2);			// 0.5*(sqrt(5)-1)
	for(; ;)
		{
		// Build x
		x = c5 / s - c1;		// 5/s-1
		// Build y
		y = x - c1; y *= y; y += c7;// (x-1)^2+7
		// Build z
		z = y.square() - c4*x*x.square();// y^2-4x^3
		// due to potential small rounding errors you could get a very small negative
number
		// if z<0 then set z=0 and continue
		if (z < c0)
			z = c0;
		else
			z = sqrt(z);
		z = x*(y + z);			// x*(y+sqrt(y^2-4x^3))
		z.adjustExponent(-1);		// 0.5*x*(y+sqrt(y^2-4x^3))
```

```
            z = nroot(z, 5);                // ( 0.5*x*(y+sqrt(y^2-4x^3)))^(1/5)
            // Build an
            ssq = s.square();               // s*s
            tmp = s*(ssq - c2 * s + c5);    // s^*(s^2-2*s+5)
            tmp = sqrt(tmp);                // sqrt(s^*(s^2-2*s+5))
            tmp += c05*(ssq - c5);          // sqrt(s^*(s^2-2*s+5))+0.5(s^2-5)
            if (first_time == true)
                    {
                    //pow5 = c1;            // 5^0=1
                    first_time = false;
                    }
            else
                    pow5 *= c5;             // 5^n
            tmp *= pow5;
            an = ssq*a - tmp;
            // Build s
            tmp = z + x / z + c1;           // z+x/z+1
            tmp = tmp.square() * s;         // s(z+x/z+1)^2
            tmp = c25 / tmp;                // 25/(s(z+x/z+1)^2)
            if ((tmp - s).exponent() < limit)
                    break;
            s = tmp;                        // 25/(s(z+x/z+1)^2)
            if (s == c1)
                    break;
            a = an;
            }

    pi = an.inverse();              // pi = 1/a
    pi.precision(digits);           // Round to precision
    return pi;
    }
```

Borwein Nonic (ninth order) algorithm

This is the Nonic convergence algorithm, where the number of correct digits is 9 times more after each iteration.

Let the initial conditions be: $a_0=\frac{1}{3}$, $r_0=\frac{\sqrt{3}-1}{2}$, $s_0=\sqrt[3]{1-r_0^3}$

Then for n=1,2,3…

$$t_n=1+2r_{n-1}$$

$$u_n=\sqrt[3]{9r_{n-1}\left(1+r_{n-1}+r_{n-1}^2\right)}$$

$$v_n=t_n^2+t_nu_n+u_n^2$$

$$w_n=\frac{27\left(1+s_{n-1}+s_{n-1}^2\right)}{v_n}$$

$$a_n=w_na_{n-1}+3^{2n-3}\left(1-w_n\right)$$

$$s_n = \frac{(1-r_{n-1})^3}{(t_n+2u_n)v_n}$$

$$r_n = \sqrt[3]{1-s_n^3}$$

$$\pi_n = \frac{1}{a_n}$$

As you can see, we use only two iterations to get the best result possible using approximately 15 digits of precision in our calculation.

π Borwein Nonic

Iteration	T	u	v	w	a	s	r	π	error
0					0.33333	0.9834	0.366		
1	1.732	1.703	8.851	9	0.33333	0.0056	1	3.0000000000000	0.14159
2	3	3	27	1.005	0.31831	8.3E-25	1	3.1415926535898	5.77E-15

It should be noted that although the Quintic and the Nonic algorithms have considerably higher convergence rates than the Quadratic and Cubic algorithms, the extra computational work is not justified by the fewer iterations.

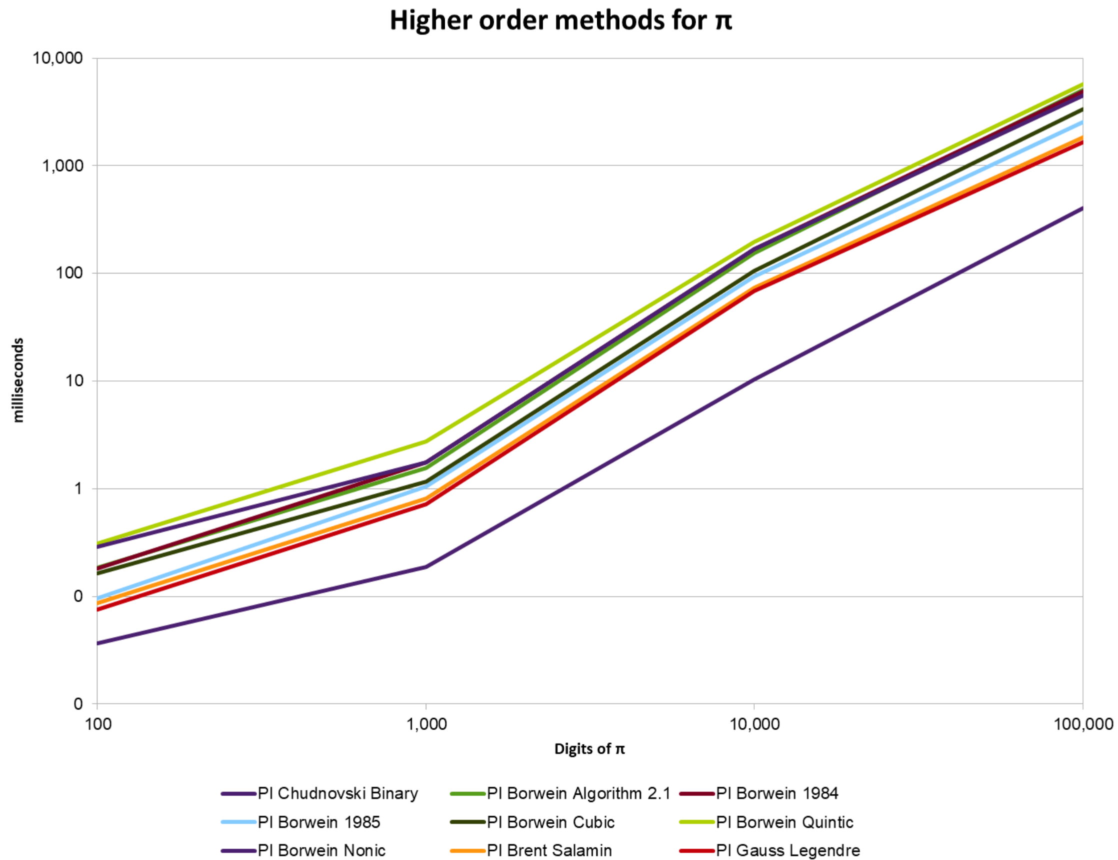
Algorithm 3.8 Borwein Nonic (9th order convergence)

```
// Borwein nonic order algorithm
static float_precision pi_borwein_nonic(unsigned int digits)
	{
	const size_t prec = digits + 2 + (size_t)(log10(digits) + 0.5);
	bool first_time = true;
	const float_precision c1(1), c2(2), c3(3), c05(0.5), c9(9), c27(27);
	float_precision a(3, prec), an(0, prec), s(5, prec);
	float_precision u(0, prec), t(0, prec), r(3, prec);
	float_precision pow3(0, prec), v(0, prec), w(0, prec);
	float_precision pi(0,prec);
	const eptype limit = -(eptype)ceil((digits + 2)*log2(10));
	size_t loopcnt = 1;

	a = a.inverse();                                        // 1/3
	r = c05 * ( sqrt(r) - c1);              // (sqrt(3)-1)/2
	s = nroot(c1-r*r*r,3);                  // (1-r)^(1/3)
	for (; ;++loopcnt)
		{
		// Build t
		t = c1 + c2*r;                  // 1+2r
		// Build u
		u = c1 + r + r * r;             // 1+r+r^2
		u *= c9 * r;                    // 9r(1+r+r^2)
		u = nroot(u, 3);                // (9r(1+r+r^2))^(1/3)
		// Build v
		v = t*t + t*u + u*u;            // t^2+tu+u^2
		// Build w
		w = c27*(c1 + s + s*s);         // 27(1+s+s^2)
		w /= v;                         // 27(1+s+s^2)/v
		// Build an
		if (first_time == true)
			{
```

```
                pow3 = a;                // 3^(-1)=1/3 same as the initial a
                first_time = false;
                }
        else
                pow3 *= c9;              // 3^(2n-1)
        an = w*a+pow3*(c1 - w);         // w*a+3^(2n-1)(1-w)
        // Build s
        s = (c1 - r);                   // (1-r)
        s *= s.square();                // (1-r)^3
        s /= (t + 2 * u)*v;             // (1-r)^3/((t+2u)*v)
        // Build  r
        r = c1 - s.square() * s;        // 1-s^3
        r = nroot(r, 3);                // (1-s^3)^(1/3)
        pi = an - a;
        if (loopcnt>1 && (r==c1||an == a||(pi).exponent()<limit))
                break;
        a = an;
        }

    pi = an.inverse();              // pi = 1/a
    pi.precision(digits);           // Round to precision
    return pi;
    }
```

Performance of higher-order methods for π

The performance of the higher-order methods is listed below. It is not surprising that higher-order convergence methods like the Borwein Quintic and Nonic versions do not perform better than quadratic convergence methods. Usually, the promise of these higher-order methods is eaten up by the much higher complexity of each iteration step. A clear winner is the Brent-Salamin & Gauss-Legendre methods (very similar), which are approx. 2.5 times faster than Borwein's 1984 methods and approx. 2-2.5 times faster than Borwein Quintic and Nonic methods. As a reference the chart also include the Chudnovsky binary splitting method for π.



Recommendation for higher-order π

All the higher-order methods are within a factor of three of each other from top to bottom. My recommendation for higher-order methods is to use the Brent-Salamin method. Higher-order methods like Borwein's 9th- or 5th-order methods, although they have a faster convergence rate, have higher complexity than the performance gain, and are therefore not recommended. Also note that the Chudnovsky Binary splitting method is far superior to any higher-order method, making it the overall choice for fast computation of π to any number of digits.

4. Spigot Algorithm

The spigot algorithm represents a fundamentally different approach to computing π from all the methods described in the preceding chapters. Where the Newton, AGM, and series-based methods compute π to p digits as a single block of output using arbitrary precision arithmetic, a spigot algorithm produces the decimal digits of π one at a time in sequence, extracting each new digit before moving on to the next. The name comes from the image of turning a tap and having digits flow out individually.

The practical appeal of spigot algorithms is that they require only basic integer arithmetic: addition, subtraction, multiplication, and division, with no floating-point operations and no arbitrary precision library. The implementations in this chapter operate entirely within 64-bit integer arithmetic, making them self-contained and straightforward to implement on any platform.

However, this simplicity comes with a significant limitation. The total cost of computing p digits of π using a spigot algorithm is $O(p^2)$. To produce each successive digit, the algorithm must work through all preceding states, and this work grows linearly with the number of digits already produced. There is no path to the $O(M(p) \log^2(p))$ scaling of binary splitting methods. As the performance comparisons at the end of the chapter show, the spigot algorithms are overtaken by the Brent-Salamin method beyond approximately 1,000 digits and fall further behind as precision increases.

This chapter should therefore be read as an exploration of an elegant and historically interesting family of algorithms rather than as a practical recommendation for arbitrary-precision computation. For serious computation of π at high precision, the binary splitting methods of the preceding chapters are always preferable. The spigot algorithm has its niche in constrained environments where no arbitrary-precision library is available, and a modest number of digits is required, but beyond that niche, it is not competitive.

The spigot algorithm for calculating π was discovered by Rabinowitz and Wagon in 1990. See [12]. The formula is remarkably simple and does not require any fancy computing, just basic operations like add, subtract, multiply, and divide.

However, to compute the n-th digit of π, you still need to calculate all the preceding n-1 digits. Although the formula was invented solely by Plouffe in 1995, the paper by Bailey, Borwein, and Plouffe describes the methods in detail. [11].

The algorithm goes as follows:

$$\pi = \sum_{n=0}^{\infty} \frac{1}{16^n}\left(\frac{4}{8n+1} - \frac{2}{8n+4} - \frac{1}{8n+5} - \frac{1}{8n+6}\right)$$

This algorithm is not faster than the higher-order algorithm just described, but what is interesting is that it requires only basic operators. It takes nine iterations to get to approx. 15-digit accuracy.

Iteration	Σ	π	Error
0	3.133333	3.13333333333333	8.26E-03
1	8.09E-03	3.14142246642247	1.70E-04
2	1.65E-04	3.14158739034658	5.26E-06
3	5.07E-06	3.14159245756744	1.96E-07
4	1.88E-07	3.14159264546034	8.13E-09
5	7.77E-09	3.14159265322809	3.62E-10
6	3.45E-10	3.14159265357288	1.69E-11
7	1.61E-11	3.14159265358897	8.20E-13
8	7.80E-13	3.14159265358975	4.09E-14
9	3.89E-14	3.14159265358979	1.78E-15

Not very impressive, but it was also discovered that this algorithm could be used to calculate the next digit of π without knowledge of all the preceding digits. This is in sharp contrast to the higher-order methods, where you are bound to perform the iteration with the number of digits you need to calculate π. Even if you need considerably fewer iterations to calculate π for the higher-order methods, e.g., 1 million digits of π can be done in approx. 13 iterations for cubic convergence methods, then you still have to perform all the calculations using 1 million digits precision, which, even with speedup methods, can be slow.

The spigot algorithm is based on the expansion for π:

$$\pi=\sum_{i=0}^{\infty}\frac{(i!)^2 2^{i+1}}{(2i+1)!}$$

THis series expands into a Horner-type schema.

To see that we can just run the first couple of expansions e.g. n=0,1,2:

$$\pi=\sum_{i=0}^{\infty}\frac{(i!)^2 2^{i+1}}{(2i+1)!}=\frac{1*2}{1}+\frac{1*2^2}{3!}+\frac{(2!)^2*2^3}{5!}+\ldots=$$

$$2+2\frac{1*2}{2*3}+2\frac{2*2*2*2}{2*3*4*5}+\ldots=$$

$$2+2\frac{1}{3}+2\frac{1}{3}\frac{2}{5}+\ldots=2+\frac{1}{3}\left(2+\frac{2}{5}(2,\ldots)\right)$$

Practical implementation of π Algorithms

This series expands out using the Horner schema:

$$\pi = 2 + \frac{1}{3}\left(2 + \frac{2}{5}\left(2 + \frac{3}{7}\left(\ldots\left(2 + \frac{i}{2i+1}(\ldots)\right)\right)\right)\right)$$

This is known to be a mixed-radix base $c = \left(\frac{1}{3}, \frac{2}{5}, \frac{3}{7}, \frac{4}{9}, \ldots\right)$ with respect to π=(2;2,2,2,..).

You can set up a simple Excel sheet to calculate the digits of π as shown below; see [12] for a detailed explanation of the formula in each cell. The π digit appears in the gray column below as 3.1415. Now the number of terms you would need to calculate n digits of π is bound by $(\frac{10n}{3} + 1)$ see [13].

Spigot Algorithm

	π		**1/3**	**2/5**	**3/7**	**4/9**	**5/11**	**6/13**	**7/15**	**8/17**	**9/19**	**10/21**	**11/23**	**12/25**	**13/27**	**14/29**
Initialize		2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
		2	2	2	2	2										
Scale		0	0	0	0	0	20	20	20	20	20	20	20	20	20	20
		1	1	1	1	1										
Carry	**3**	0	2	2	2	0	12	7	8	9	0	0	0	0	0	
		3	3	3	3	3										
Sum		0	2	2	2	0	32	27	28	29	20	20	20	20	20	20
remainders		0	2	2	4	3	10	1	13	12	1	20	20	20	20	20
			2	2	4	3	10		13	12		20	20	20	20	20
Scale		0	0	0	0	0	0	10	0	0	10	0	0	0	0	0
		1	2	3	4	6					17	16	15	13		
Carry	**1**	3	0	3	0	5	48	98	88	81	0	5	6	0	90	
		1	4	5	8	9	14	10	21	20	18	36	35	33	29	20
Sum		3	0	3	0	5	8	8	8	1	0	5	6	0	0	0
remainders		3	1	3	3	5	5	4	8	14	9	8	11	5	20	26
		3	1	3	3	5				14			11		20	26
Scale		0	0	0	0	0	50	40	80	0	90	80	0	50	0	0
		1	2	3	4	4								14	12	
Carry	**4**	1	4	0	0	5	54	77	96	72	80	88	84	3	0	
		4	3	6	7	9	10	11	17	21	17	16	19	19	32	26
Sum		1	4	0	0	5	4	7	6	2	0	8	4	3	0	0

remainders		1	1	0	0	5	5	0	11	8	18	0	10	18	23	28
		1	1			5			11		18		10	18	23	28
Scale		0	0	0	0	0	50	0	0	80	0	0	0	0	0	0
				1	4	4				10		12	15	16	13	
Carry	**1**	5	6	5	0	0	42	91	88	8	50	1	6	9	5	
		1	1	1	4	9			19	18	23	12	25	34	36	28
Sum		5	6	5	0	0	92	91	8	8	0	1	6	9	5	0

remainders		5	1	0	5	0	4	0	3	1	2	16	3	24	14	19
		5	1		5							16		24	14	19
Scaler		0	0	0	0	0	40	0	30	10	20	0	30	0	0	0
				2		2					11		15	10		
Carry	**5**	6	8	4	8	0	6	21	24	54	0	88	6	4	90	
		5	1	2	5	2					13	24	18	34	23	19
Sum		6	8	4	8	0	46	21	54	64	0	8	6	4	0	0

Thanks to Dik Winter and Achim Flammenkamp, a condensed C-language version of the algorithm that produces 4 digits at a time using only integer arithmetic was published. That version was later beautified by Gibbons and bought below. The algorithm is bounded, meaning that it requires the desired number of digits you want to calculate π prior. Gibbons [13] presents another unbounded algorithm that processes a steady stream of π digits. The algorithm below procedures 4 digits of π per iteration. The number 14 below is coming from the number of terms formula above: $(\frac{10n}{3}+1)=(\frac{10*4}{3}+1)=14$

Algorithm 4.1 Gibbons spigot

```
#define NDIGITS 15000               /*max.digits to compute*/
#define LEN      (NDIGITS/4+1)*14 /*nec.arraylength*/
Int a[LEN];                         /*arrayof4digit-decimals*/
Int b;                              /*nominatorprev.base*/
Int c=LEN;                          /*index*/
Int d=0;                            /*accumulatorandcarry*/
Int e=0;                            /*saveprev.4digits*/
Int f=10000;                        /*newbase,4dec.digits*/
Int g;                              /*denomprev.base*/
Int h=0;                            /*initswitch*/
//Spigotalgorithms 4
Int main(){
      for(;(b=c-=14)>0;)            /*outerloop:4digits/loop*/
      {
      for(;--b>0;)                  /*innerloop:radixconv*/
      {
      d*=b;                         /*acc*=nom.prevbase*/
      if(h==0)
```

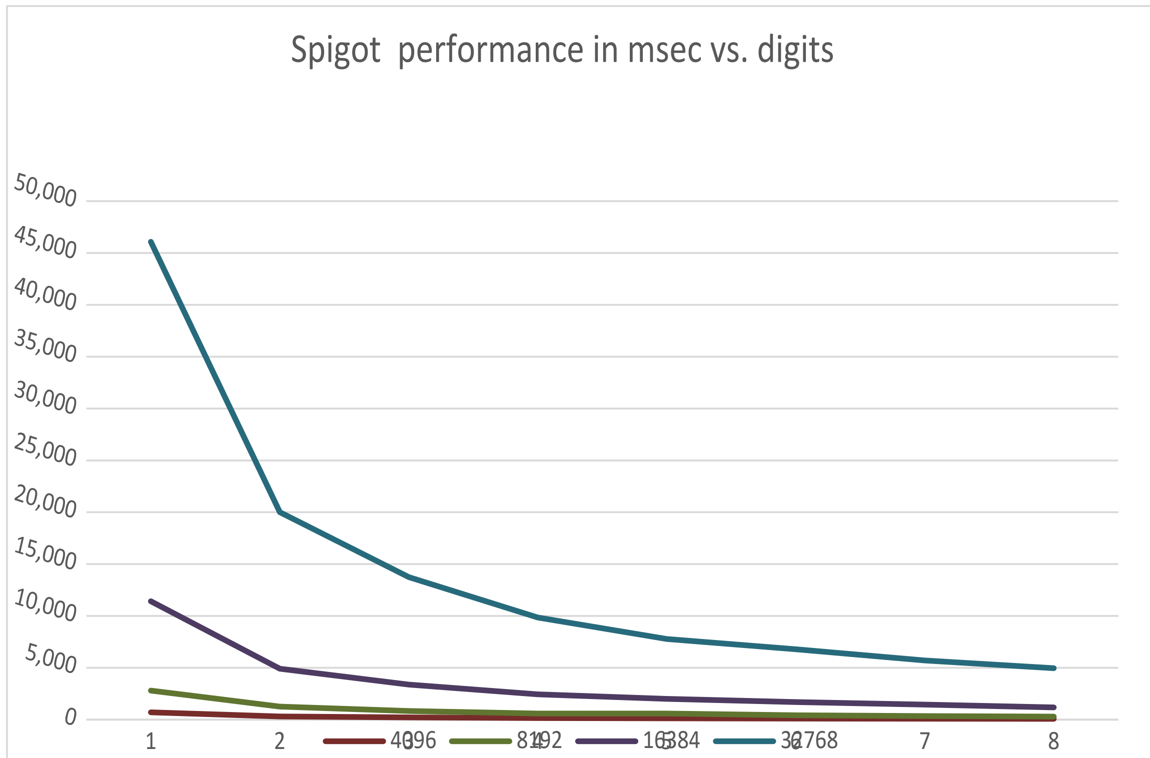
```
    d+=2000*f;           /*firstouterloop*/
    else
         d+=a[b]*f;           /*non-firstouterloop*/
    g=b+b-1;                  /*denomprev.base*/
    a[b]=d%g;
    d/=g;                     /*savecarry*/
    }
    h=printf("%04d",e+d/f);   /*printprev 4 digits*/
    d=e=d%f;                  /*savecurrent 4 digits*/
    }
    return0;
    }
```

The Algorithm above can deliver approx. 15,000 digits of π without going into overflow. Let us try to improve that and make it more useful to generate more π digits.

The first improvement we can make is to use 32-bit unsigned integer arithmetic instead of signed integer arithmetic, which will allow the above algorithm to handle 30,723 digits of π before overflow occurs. See below.

The biggest issue is overflow in the accumulator variable d. Since we are initially multiplying 2,000 by the constant F, which is 10,000, and adding it to the accumulator d, we add a digit of magnitude of $2*10^7$. The maximum digit that can be held using 32-bit unsigned integer arithmetic is $\sim 4*10^{9,}$ and that is why the algorithm goes into overflow shortly after 30,000 digits of π have been calculated.

The next thing we can do is lower the number of digits to add per iteration; instead of 4 digits at a time, we can use, e.g., 3, which increases the number of digits for π to a maximum of 293,261 before overflow occurs. Continuing down that road, we can increase the number of digits for PI to ~2.8 million, and to more than 26 million if we find only one digit at a time. However, this is not without a time penalty. See the picture below, which shows the time in milliseconds as a function of the number of digits of π you need to calculate. The test was performed on an Intel i7 quad-core CPU with a 2.6 GHz clock frequency. If you follow the blue line (π with 32,767 digits), you can see that below four digits at a time, you see a dramatic increase in time. On the other hand, if you increase the number of digits per iteration to eight, you make the algorithm twice as fast. However, that is not possible with the below algorithm, which only uses 32-bit integer arithmetic. The maximum number of digits it can handle is 5 at a time, but that limits the digits of π to 3,474 before it overflows.



Gibbon's version of π has a flaw that is not exposed with four digits at a time, with the limited number of digits it can generate, but is visible when lowering the number of digits. And that is an overflow in the printout of e+d/f in the statement :

h=printf("%04d",e+d/f);

The issue is that it sometimes generates a carry that is not added to the π digit of the previous digit, and therefore fails to produce the correct result for π. Instead, we change it to accumulate the π digits into a std::string from the C++ Standard Library. That way, when a carry is detected, we can correctly propagate it back into the already-calculated digit.

The 64-bit version of the final algorithm is listed below.

Algorithm 4.2 64-bit Spigot

```
// 64bit version of the spigot algorithm.
// Notice acc, a, g needs to be unsigned 64bit.
// Emperisk for pi to 2^n digits, acc need to hold approx 2^(n+17) numbers. while a[] and g needs
approx 2^(n+3) numbers
// a[] & g could potential be unsigned long (32bit) going to a max of 2^29 digit or 536millions
digit of PI. but with
// unsigned 64bit you can do nearly "unlimited"
// By default it collects 4 digits at a time, parameter no_dig can adjust that from 1 to 8
std::string pi_spigot_64( const int digits, int no_dig = 4 )
        {
        static const unsigned long f_table[]=
{0,10,100,1000,10000,100000,1000000,10000000,100000000};
        static const unsigned long f2_table[] = {0,2,20,200,2000,20000,200000,2000000,20000000};
        const int TERMS = (10 * no_dig / 3 + 1);          // Number of Terms needed
```

```
        bool first_time = true;                             // First time in loop flag
        bool overflow_flag = false;                         // Overflow flag
        char buffer[9];
        std::string ss;                     // The String that holds the calculated PI
        long b, c;                          // Loop counters
        int carry, no_carry = 0; // Outer loop carrier, plus no of carrier adjustment counts
        unsigned long f, f2;                // New base 1 decimal digits at a time
        unsigned long dig_n = 0;            // dig_n holds the next no_dig digit to add
        unsigned long e = 0;                // Save previous 4 digits
        unsigned long long acc = 0, g = 0, tmp64;
        ss.reserve(digits + 16);            // Reserve the string size to be able to accumulate all
digits plus 8
        if (no_dig > 8) no_dig = 8;         // ensure no_dig<=8
        if (no_dig < 1) no_dig = 1;         // Ensure no_dig>0
        c = (digits / no_dig + 1) * no_dig;         // Since we do collect PI in trunks of no_dig
digit at a time we need to ensure digits are divisible by no_dig.
        if (no_dig == 1) c++;               // Extra guard digit for 1 digit at a time.
        c = (c / no_dig + 1) * TERMS;       // c ensure that the digits we seek is divisible by
no_dig
        f = f_table[no_dig];                // Load the initial f
        f2 = f2_table[no_dig];              // Load the initial f2
        std::vector<uintmax_t> a(c);        // Vector of 8 digits decimals

        // b is the nominator previous base; c is the index
        for (; (b = c -= TERMS) > 0 && overflow_flag==false; first_time=false)
                {
                for (; --b > 0 && overflow_flag==false;)
                        {
                        if (acc > ULLONG_MAX / b) overflow_flag = true;   // Check for overflow
                        acc *= b;                           // Accumulator *= nom previous base
                        tmp64 = f;
                        if (first_time==true)               // Test for the first run in the main
loop
                                tmp64 *= f2;                // First outer loop. a[b] is not yet
initialized
                        else
                                tmp64 *= a[b];              // Non first outer loop. a[b] is
initialized in the first loop
                        if (acc > ULLONG_MAX - tmp64) overflow_flag = true;// Check for overflow
                        acc += tmp64;                       // add it to accumulator
                        g = b + b - 1;                      // denominated previous base
                        a[b] = acc % g;                     // Update the accumulator
                        acc /= g;                           // save carry
                        }
                dig_n =(unsigned long)( e + acc / f );     // Get previous no_dig digits. Could
occasionally be no_dig+1 digits in which case we have to propagate back the extra digit.
                // Check for extra carry to propagate back into the current sum of PI digits
                carry = (unsigned)( dig_n / f);
                // Eliminate the extra carrier. Now l contains no_dig digits to add to the string
                dig_n %= f;
                // Add the carrier to the existing number for PI calculate so far.
                if (carry > 0)
                        {
                        ++no_carry;
                                // Keep count of how many carriers detect
                        for (size_t i = ss.length(); carry > 0 && i > 0; --i)
                        // Loop and propagate back the extra carrier to the existing
                        // PI digits found so far
                                {
                                        // It can handle multiple carry-back propagations
                                int new_digit;
                                new_digit = (ss[i - 1] - '0') + carry;   // Calculate new digit
                                carry = new_digit / 10;  // Calculate new carry if any
                                // Put the adjusted digit back in our PI digit list
                                ss[i - 1] = new_digit % 10 + '0';
                                }
                        }

                for(int i = no_dig-1; i >= 0; --i)
```

```
                    {
                    if (dig_n > 0)
                            {
                            buffer[i] = (dig_n % 10)+'0'; dig_n /= 10;
                            }
                    else
                            buffer[i] = '0';
                    }
            // Print previous no_dig digits to buffer
            ss += std::string(buffer,no_dig);
                    // Add it to PI string
            if(first_time==true)
                    ss.insert(1, ".");
                    // add the decimal point after the first digit to create 3.14...
            acc = acc % f;
                    // save current no_dig digits and repeat loop
            e = (unsigned long)acc;
            }
    // Remove the extra digits that we didn't request but used as guard digits
    ss.erase(digits+1);
    if (overflow_flag == true)
            ss = std::string("Overflow:") + ss;
    // Set overflow in the return string
    return ss;      // Return Pi with the number of digits
    }
```

Gosper formula for π

As it has been mentioned in the previous section, Rabinowitz-Wagon Spigot Algorithms for π require approximately 3.3 terms per digit of π. However, Gosper's page formula for π can also be used, and it is more efficient because it requires fewer terms to evaluate per digit. The number of digits you get is approx. 1.1 digit, or if you evaluate 10 terms, you get 11 valid digits of π. This is approx. 3 times less work to perform per digit; however, as always, you do not get things for free. Each term is a little bit more complicated to handle, and you quickly reach the limit of the integer representation, so for all practical purposes, you need to implement this algorithm using 64-bit integer arithmetic only.

The Gosper formula is:

$$\pi = 3 + 2\sum_{n=1}^{\infty} \frac{n(5n+3)(2n-1)!(n!)}{2^{n-1}(3n+2)!}$$

Which expands into this series:

$$\pi = 3 + \frac{1}{60}\left(8 + \frac{2\times 3}{7\times 8\times 3}\left(13 + \frac{3\times 5}{10\times 11\times 3}\left(18 + \frac{4*7}{13*14*3}(\ldots)\right)\right)\right)$$

And:

Practical implementation of π Algorithms

$$\pi=3+\frac{1}{60}\left(8+\frac{6}{168}\left(13+\frac{15}{330}\left(18+\frac{28}{816}\left(5n-2+\frac{n(2n-1)}{3(9(n^2+n)+2)}(\ldots)\right)\right)\right)\right)$$

This is the way we want to have the series expanded so we can quickly identify the different Spigot elements. This is a well-known mixed-radix base $c=\left(\frac{1}{60},\frac{6}{168},\frac{15}{330},\frac{28}{816},\frac{n(2n-1)}{3(9(n^2+n)+2)},\ldots\right)$ with respect to π=(3;8,13,18,5n-2,..).

As the fraction of two terms is always smaller than $\frac{1}{13}$ You would get d precision with d $=\frac{\log(10^n)}{\log(13)}=>d\approx\frac{n}{0.9}$ terms.

The new, simple Excel sheet that calculates the digits of π is shown below; see [17] for a detailed explanation of the formula in each cell. The π digit appears in the gray column below as 3.1415. Now, the number of terms you would need to calculate n digits of π is bound by digits/0.9. In the table below, we see that we only need six terms to obtain approximately seven correct digits, which is far fewer than the Rabinowitz-Wagon algorithm requires. However, you also notice that the mixed radix-based $\frac{A}{B}$ quickly reaches high values that can cause overflow if not carefully managed.

Spigot π - Gosper							
	Terms	0	1	2	3	4	5
	A		1	6	15	28	45
	B		60	168	330	546	816
Initialize		3	8	13	18	23	28
Scale		30	80	130	180	230	280
Carry	**3**	1	0	0	0	0	
Sum		31	80	130	180	230	280
remainder s		1	20	130	180	230	280
Scale		10	200	1300	180 0	2300	2800
Carry	**1**	4	48	75	112	135	
Sum		14	248	1375	191	2435	2800

					2		
remainders		4	8	31	262	251	352
Scale		40	80	310	2620	2510	3520
Carry	**4**	1	12	120	112	180	
Sum		41	92	430	2732	2690	3520
remainders		1	32	94	92	506	256
Scale		10	320	940	920	5060	2560
Carry	**1**	5	30	45	252	135	
Sum		15	350	985	1172	5195	2560
remainders		5	50	145	182	281	112
Scaler		50	500	1450	1820	2810	1120
Carry	**5**	9	54	75	140	45	
Sum		59	554	1525	1960	2855	1120
remainders		9	14	13	310	125	304
Scaler		90	140	130	3100	1250	3040
Carry	**9**	2	6	135	56	135	
Sum		92	146	265	3156	1385	3040
remainders		2	26	97	186	293	592
Scaler		20	260	970	1860	2930	5920
Carry	**2**	4	36	90	140	315	
Sum		24	296	1060	2000	3245	5920

With only six terms, we get seven correct digits of π (3.141592). The only drawback with the Gosper algorithm over the Rabinowitz-Wagon Spigot Algorithms for π is that the mixed radix-based $\left(\frac{1}{60},\frac{6}{168},\frac{15}{330},\frac{28}{816},\ldots,\frac{n(2n-1)}{3(9(n^2+n)+2)},\ldots\right)$ yield higher than the

Rabinowitz-Wagon algorithm that used $\left(\frac{1}{3},\frac{2}{5},\frac{3}{7},\frac{4}{9},\ldots,\frac{n}{2n+1}\right)$. This leads to faster overflow even when using 64-bit integer arithmetic. On the other hand, we do not need as many terms as the Rabinowitz-Wagon Algorithm. For a wanted precision of *d* digits, we need the Rabinowitz-Wagon algorithm n=($\frac{10d}{3}+1$). For the Gosper algorithm, we need $n\sim\frac{d}{0.9}$.

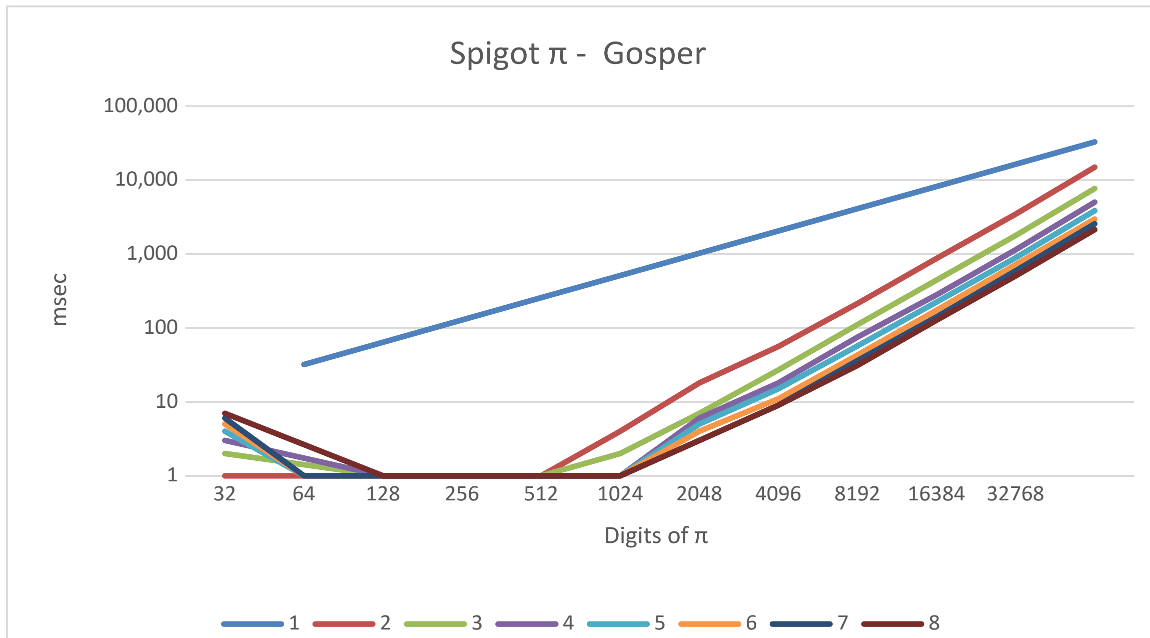
Dividing the two formulas, you get a ratio of $\sim 3+\frac{0.9}{d}$=> or 3 for a larger number of d. e.g let's assume you need to find 1,000,000 digits precision of π. The largest term needs for Gosper even with the reduced number of terms is $\frac{\sim 6.67^{11}}{\sim 9^{12}}$ while Rabinowitz-Wagon is $\frac{10^6}{\sim 2x10^6}$. We should expect the Gosper algorithm to overflow faster for large d than the Rabinowitz-Wagon algorithm does.

Algorithm 4.3 Gosper 64-bit

```
// Gosper algorithm
// A Column: 1,6,15,28,45,...  2n(n-1)-n
// B Column: 60, 168, 330,  546,  816,... 3(9(n+1)n+2)
// Initialization values:3, 8, 13, 18, 23 28,... 5n-2
std::string pi_spigot_gosper_64(int digits, int no_dig = 1)
        {
        static unsigned long f_table[] = { 1, 10, 100, 1000, 10000, 100000, 1000000, 10000000, 
100000000 };
        bool first_time = true;
        bool overflow_flag = false;
        char buffer[9];
        std::string ss;
        int dig;
        unsigned int car, no_carry = 0;
        unsigned int no_terms;           // No of terms to complete as a function of digits
        unsigned long f, f2;             // New base 1 decimal digits at a time
        unsigned long dig_n;             // dig_n holds the next no_dig digit to add
        unsigned _int64 carry, a, b, tmp64;
        ss.reserve(digits + 16);

        if (no_dig > 8) no_dig = 8;      // ensure no_dig<=5
        if (no_dig < 1) no_dig = 1;      // Ensure no_dig>0
        // Since we do it in trunks of no_dig digits at a time we need to ensure digits are 
divisible with no_dig.
        dig = (digits / no_dig + (digits%no_dig>0 ? 1 : 0)) * no_dig;
        dig += no_dig;                   // Extra guard digits
        no_terms = (unsigned int)(dig * 0.9) + 1; // Calculate the number of terms needed
        std::vector<uintmax_t> acc(no_terms+1);// Vector of the accumulator
        f = f_table[no_dig];             // Load the initial f
        f2 = f_table[no_dig - 1];        // Load the initial f2

        for (int i = dig; i >= 0 && overflow_flag == false; i -= no_dig, first_time = false)
                {
                carry = 0;
                no_terms = (unsigned int)(i * 0.9) + 1; // Calculate the number of terms needed
```

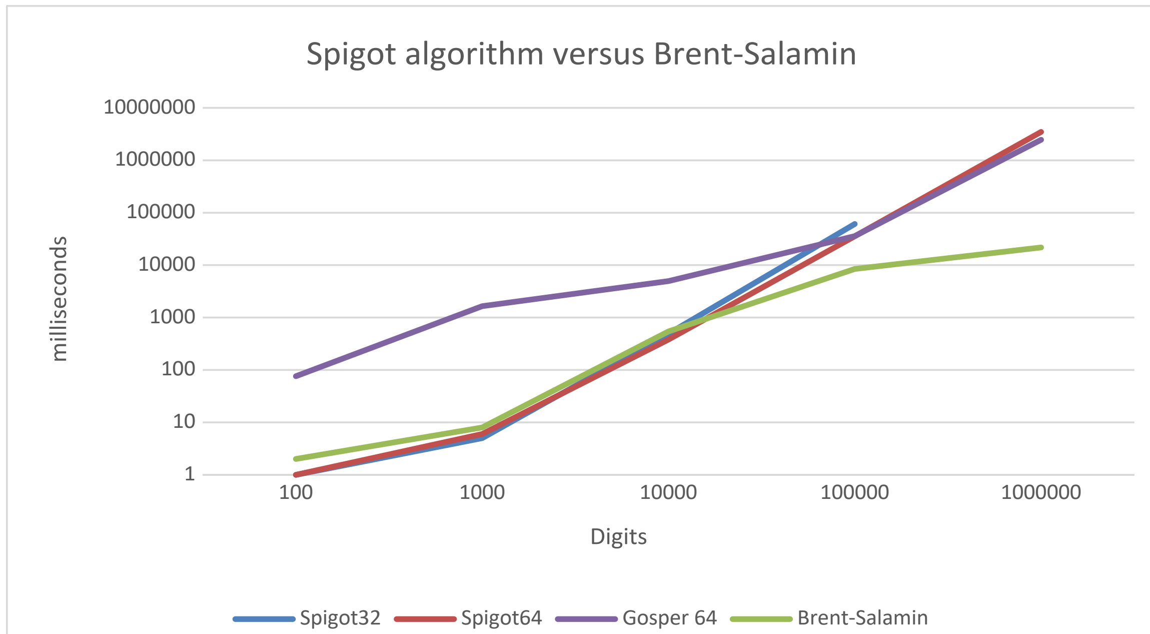
```
            for (int j = no_terms; j>0 && overflow_flag == false; --j)
                    {
                    a = 2 * (j + 1) - 1;     // Create Column A terms
                    a *= (j + 1);    // Take the previous column A and multiply it with carry
                    if (carry > ULLONG_MAX / a)
                            overflow_flag = true;    // Check for overflow
                    carry *= a;
                    tmp64 = f;
                    if (first_time == true)
                            {
                            tmp64 *= f2;
                            tmp64 *= (5 * (j + 1) - 2);      // Create the initialized value
                            }
                    else
                            tmp64 *= acc[j];
                    if (carry > ULLONG_MAX - tmp64)
                            overflow_flag = true;
                    carry += tmp64;
                    b = j;          //Assign it to 64bit variable b to avoid 32bit overflow.
                    b = 3 * (9 * (b + 1)*b + 2);             // Create Column B terms
                    acc[j] = carry % b;
                    carry /= b;
                    }

            if (first_time == true)
                    {
                    tmp64 = f; tmp64 *= 3 * f2;
                    acc[0] = (tmp64 + carry);
                    }
            else
                    acc[0] = acc[0] * f + carry;
            dig_n = (unsigned)(acc[0] / f);
            car = (unsigned)(dig_n / f);
            dig_n %= f;
            // Add the carry to the existing number for PI calculated so far.
            if (car > 0)
                    {
                    ++no_carry;
                    for (int j = ss.length(); car > 0 && j > 0; --j)
                            {
                            int dd;
                            dd = (ss[j - 1] - '0') + car;
                            car = dd / 10;
                            ss[j - 1] = dd % 10 + '0';
                            }
                    }
            for (int i = no_dig - 1; i >= 0; --i)
                    {
                    if (dig_n > 0)
                            {
                            buffer[i] = (dig_n % 10) + '0'; dig_n /= 10;
                            }
                    else
                            buffer[i] = '0';
                    }
            ss += std::string(buffer);
            acc[0] %= f;
            }

    ss.insert(1, ".");// add a come after the first digit to create 3.14...
    if (overflow_flag == false)
            ss.erase(digits + 1); // Remove the extra digits that we didnt requested.
    else
            ss = std::string("Overflow:") + ss;
    return ss;
    }
```

The above-mentioned algorithm can compute π, and in each loop, we can find between 1 and 8 digits. Not surprisingly, the more digits we find per loop, the faster the overall algorithm is, as shown in the diagram below. The numbers 1 to 8 refer to how many digits we find per loop, and in the calculation, π with digits from 32 to 32,768 digits, and the timing on the Y-axis is in milliseconds.



Notice the scale is logarithmic; to find π, eight digits at a time is approximately 10 times faster than applying the algorithm one digit at a time

The time comparison shows that the 32-bit algorithm is faster within the range of π digits that both algorithms can handle. This is not a surprise, since 64-bit integer arithmetic is more time-consuming than its 32-bit counterpart. However, as the number of digits increases above 1,000 digits, the Spigot algorithm is less efficient than the Brent-Salamin algorithm. It is interesting to note that the 64-bit spigot is faster than the Gosper version for up to 100,000 digits, whereas the Gosper algorithm performs faster.



Recommendation for Spigot-π algorithms.

- Always use the 64-bit version since it allows you to do 8 digits at a time, which is faster than the 32-bit version.
- Consider the Gosper version when the number of digits exceeds 100,000.

Overall Recommendation for π algorithms.

- The Chudnovsky binary splitting method is the fastest of them all. It is no surprise that this algorithm is also being used to set records in calculating many digits of π. The Brent-Salamin method is fast, but the binary splitting method is in a league of its own.
- If the Binary splitting method is not an option, then I recommend the Brent-Salamin versions.
- If there is no arbitrary precision library available, then the Spigot 64-bit or Gosper 64-bit can be useful.

Reference

1. Arbitrary precision library package. Arbitrary Precision C++ Packages (hvks.com)
2. The World of π. www.pi314.net/eng/salamin.php - Oct 5, 2016
3. The Math Forum: http://mathforum.org/library/drmath/view/58283.html - Oct 5, 2016
4. Borwein, Pi and the AGM, Volume 4, John Willey & Sons Inc, New York, NY 1998
5. https://en.wikipedia.org/wiki/Borwein%27s_algorithm – Oct 6, 2016
6. https://en.wikipedia.org/wiki/Bailey%E2%80%93Borwein%E2%80%93Plouffe_formula – Oct 6, 2016
7. https://en.wikipedia.org/wiki/Approximations_of_%CF%80 – Oct 6, 2016
8. P. Borwein – The Amazing number π
9. Bailey, Borwein, Plouffe, The Quest for Pi, June 25, 1996
10. J. Borwein, Ramanujan and PI, May 3 2012
11. J Borwein, The life of Pi: From Archimedes to Eniac and Beyond, June 19, 2012
12. Bailey, Borwein, Plouffe, "on the rapid Computation of Various Polylogarithmic Constants 1996. http://www.cecm.sfu/personal/pborwein
13. Rabinowitz & Wagon, A Spigot Algorithm for the Digits of Pi, The American Mathematical Monthly, 102 (1995) page 195-203.
14. Unbounded Spigot Algorithms for the Digits of PI
15. Binary Splitting Recursion Library (numberworld.org)
16. The world of π. http://www.pi314.net/eng/goutte.php - Dec 28, 2016

Appendix A

The binary splitting method for Ramanujan, Chudnovsky, and the Borwein π is outlined below.

The Binary Recursion algorithm for the three-variable splitting

The three-variable binary splitting recursion goes as follows [15]:

$$x=\sum_{k=1}^{n}\frac{P(k)}{R(k)}\prod_{i=1}^{k}\frac{R(i)}{Q(i)}=\frac{P(0,n)}{Q(0,n)}$$

Given a<m<b.

$m=\frac{a+b}{2}$ *integer division*
P(a,b)=P(a,m)Q(m,b)+P(m,b)R(a,m)
Q(a,b)=Q(a,m)Q(m,b)
R(a,b)=R(a,m)R(m,b)

Where:
P(b+1,b)=P(b)
Q(b-1,b)=Q(b)
R(b-1,b)=R(b)

Deriving the Binary splitting method for Ramanujan π

The binary splitting algorithm is on the form:

$$x=\sum_{k=1}^{n}\frac{P(k)}{R(k)}\prod_{i=1}^{k}\frac{R(i)}{Q(i)}$$

In addition, we need to get the Ramanujan π series into that form.

Starting with the Ramanujan series for π:

$$\frac{1}{\pi}=\frac{2\sqrt{2}}{9801}\sum_{k=0}^{\infty}\frac{(4k)!(1103+26390k)}{(k!)^4 396^{4k}}$$

The first step is to split up the factorial into product notation from the rest of the series.

Noting (4k)! in product notation is: $\prod_{i=1}^{k}(4i)(4i-1)(4i-2)(4i-3)$ and

$(k!)^4$ is: $\prod_{i=1}^{k} i^4$

$$\frac{1}{\pi}=\frac{2\sqrt{2}}{9801}\sum_{k=0}^{\infty}\frac{1103+26390k}{396^{4k}}\prod_{i=1}^{k}\frac{(4i)(4i-1)(4i-2)(4i-3)}{i^4}$$

Aligning the start index to one and moving out the constant at k=0 yields:

$$\frac{1}{\pi}=\frac{1103\sqrt{2}}{9801}+\frac{2\sqrt{2}}{9801}\sum_{k=1}^{\infty}\frac{1103+26390k}{396^{4k}}\prod_{i=1}^{k}\frac{(4i)(4i-1)(4i-2)(4i-3)}{i^4}$$

Simplifying the product notation part:

$$\frac{1}{\pi}=\frac{1103\sqrt{2}}{9801}+\frac{2\sqrt{2}}{9801}\sum_{k=1}^{\infty}\frac{1103+26390k}{396^{4k}}\prod_{i=1}^{k}\frac{(4i-1)(2i-1)(4i-3)}{\frac{1}{8}i^3}$$

Rearranging the formula to fit into the binary splitting form:

$$\frac{1}{\pi}=\frac{1103\sqrt{2}}{9801}+\frac{2\sqrt{2}}{9801}\sum_{k=1}^{\infty}\frac{(4k-1)(2k-1)(4k-3)(1103+26390k)}{(4k-1)(2k-1)(4k-3)396^{4k}}\prod_{i=1}^{k}\frac{(396^4)(4i-1)(2i-1)(4i-3)}{\frac{1}{8}(396^4)i^3}$$

Simplify:

$$\frac{1}{\pi}=\frac{1103\sqrt{2}}{9801}+\frac{2\sqrt{2}}{9801}\sum_{k=1}^{\infty}\frac{(4k-1)(2k-1)(4k-3)(1103+26390k)}{(4k-1)(2k-1)(4k-3)}\prod_{i=1}^{k}\frac{(4i-1)(2i-1)(4i-3)}{3073907232i^3}$$

Now identify the P(k), Q(k) and R(K) you get:

$$P(k)=(1103+26390k)(4k-1)(2k-1)(4k-3)$$
$$Q(k)=3073907232k^3$$
$$R(k)=(4k-1)(2k-1)(4k-3)=32k^3-48k^2+22k-3$$

Replacing the series with P(0,k) and Q(0,k) yields:

$$\frac{1}{\pi}=\frac{1103\sqrt{2}}{9801}+\frac{2\sqrt{2}}{9801}\frac{P(0,k)}{Q(0,k)}=>$$

$$\pi=\frac{9801\,Q(0,k)}{P(0,k)+11033\,Q(0,k)}\frac{1}{2\sqrt{2}}$$

Which is the Binary Splitting algorithm for Ramanujan π.

The linear convergent cost, which is an expression of the computational speed can be found using the formula in [15]:

$$Relative\ \ cost=\frac{4D}{\log\left(\frac{C_q}{C_r}\right)}$$

And you get: $\frac{4\cdot 3}{\log\left(\frac{3073907232}{32}\right)}=0.653$

Deriving the Binary splitting method for Chudnovsky π

Chudnovsky infinite series for π:

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

$$\frac{1}{\pi}=\frac{\sqrt{10005}}{4270934400}\sum_{k=0}^{\infty}\frac{(-1)^k(6k)!(13591409+545140134k)}{(3k)!(k!)^3 640320^{3k}}=>$$

Separate the factorial into product notation:

Where (6k)! in product notation is: $\prod_{i=1}^{k}6i(6i-1)(6i-2)(6i-3)(6i-4)(6i-5)$

(3k)! is: $\prod_{i=1}^{k}(3i)(3i-1)(3i-2)$

And $(k!)^3$ is: $\prod_{i=1}^{k}i^3$

Practical implementation of π Algorithms

$$\frac{1}{\pi}=\frac{\sqrt{10005}}{4270934400}\sum_{k=0}^{\infty}\frac{(-1)^k(13591409+545140134k)}{640320^{3k}}\prod_{i=1}^{k}\frac{6i(6i-1)(6i-2)(6i-3)(6i-4)(6i-5)}{3i(3i-1)(3i-2)i^3}$$

Aligning index start and simplify you get:

$$\frac{1}{\pi}=\frac{13591409\sqrt{10005}}{4270934400}+\frac{\sqrt{10005}}{4270934400}\sum_{k=1}^{\infty}\frac{(-1)^k(13591409+545140134k)}{640320^{3k}}\prod_{i=1}^{k}\frac{24(6i-1)(2i-1)(6i-5)}{i^3}$$

Rearranging the formula to fit into the binary splitting form:

$$\frac{1}{\pi}=\frac{13591409\sqrt{10005}}{4270934400}+\frac{\sqrt{10005}}{4270934400}\sum_{k=1}^{\infty}\frac{(-1)^k(6k-1)(2k-1))(6k-5)(13591409+545140134k)}{(6k-1)(2k-1))(6k-5)640320^{3k}}\prod_{i=1}^{k}\frac{640320^3\cdot 24(6i-1)(2i-1)(6i-5)}{640320^3\cdot i^3}$$

Simplify:

$$\frac{1}{\pi}=\frac{13591409\sqrt{10005}}{4270934400}+\frac{\sqrt{10005}}{4270934400}\sum_{k=1}^{\infty}\frac{(-1)^k(6k-1)(2k-1))(6k-5)(13591409+545140134k)}{(6k-1)(2k-1)(6k-5)}\prod_{i=1}^{k}\frac{(6i-1)(2i-1)(6i-5)}{10939058860032000\cdot i^3}$$

Now identify the P(k), Q(k) and R(K) you get:

$$P(k)=(-1)^k(13591409+545140134k)(6k-1)(2k-1)(6k-5)$$
$$Q(k)=10939058860032000k^3$$
$$R(k)=(6k-1)(2k-1)(6k-5)=72k^3-108k^2+46k-5$$

Replacing the series with P(0,k) and Q(0,k) yields:

$$\frac{1}{\pi}=\frac{13591409\sqrt{10005}}{4270934400}+\frac{\sqrt{10005}}{4270934400}\frac{P(0,k)}{Q(0,k)}=>$$

$$\pi=\frac{4270934400\,Q(0,k)}{P(0,k)+13591409\,Q(0,k)}\frac{1}{\sqrt{10005}}$$

Which is the Binary Splitting algorithm for Chudnovsky π.

The relative cost is $Relative\ \ cost=\frac{4D}{\log(\frac{C_q}{C_r})}=\frac{4\cdot 3}{\log(\frac{10939058860032000}{72})}=0.367$

Deriving the Binary splitting method for Borwein25 π

Practical implementation of π Algorithms

The Binary Splitting algorithm for Borwein25 for π.

Borwein 25 series for π:

$$\frac{1}{\pi}=12\sum_{k=0}^{\infty}\frac{(-1)^k(6k)!(A+kB)}{(3k)!(k!)^3C^{k+1/2}}=>$$

$$\frac{1}{\pi}=\frac{12}{\sqrt{C}}\sum_{k=0}^{\infty}\frac{(-1)^k(6k)!(A+kB)}{(3k)!(k!)^3C^{k}}=>$$

Separate the factorial into product notation and move the constant a k=0 out of the summation:

Where (6k)! in product notation is: $\prod_{i=1}^{k}6i(6i-1)(6i-2)(6i-3)(6i-4)(6i-5)$

(3k)! is: $\prod_{i=1}^{k}(3i)(3i-1)(3i-2)$

And $(k!)^3$ is: $\prod_{i=1}^{k}i^3$ yields:

$$\frac{1}{\pi}=\frac{12A}{\sqrt{C}}+\frac{12}{\sqrt{C}}\sum_{k=1}^{\infty}\frac{(-1)^k(A+kB)}{C^k}\prod_{i=k}^{k}\frac{6i(6i-1)(6i-2)(6i-3)(6i-4)(6i-5)}{3i(3i-1)(3i-2)i^3}$$

Simplify:

$$\frac{1}{\pi}=\frac{12A}{\sqrt{C}}+\frac{12}{\sqrt{C}}\sum_{k=1}^{\infty}\frac{(-1)^k(A+kB)}{C^k}\prod_{i=k}^{k}\frac{24(6i-1)(2i-1)(6i-5)}{i^3}$$

Rearranging the formula to fit into the binary splitting form:

$$\frac{1}{\pi}=\frac{12A}{\sqrt{C}}+\frac{12}{\sqrt{C}}\sum_{k=1}^{\infty}\frac{(-1)^k(6i-1)(2i-1)(6i-5)(A+kB)}{(6i-1)(2i-1)(6i-5)C^k}\prod_{i=k}^{k}\frac{C\cdot(6i-1)(2i-1)(6i-5)}{\frac{C}{24}\cdot i^3}$$

Simplify:

$$\frac{1}{\pi}=\frac{12A}{\sqrt{C}}+\frac{12}{\sqrt{C}}\sum_{k=1}^{\infty}\frac{(-1)^k(6i-1)(2i-1)(6i-5)(A+kB)}{(6i-1)(2i-1)(6i-5)}\prod_{i=k}^{k}\frac{(6i-1)(2i-1)(6i-5)}{\frac{C}{24}\cdot i^3}$$

Now identify the P(k), Q(k) and R(K) you get:

$$P(k)=(-1)^k(A+Bk)(6k-1)(2k-1)(6k-5)$$

$$Q(k)=\frac{C}{24}k^3$$

$$R(k)=(6k-1)(2k-1)(6k-5)=72k^3-108k^2+46k-5$$

Replacing the series with P(0,k) and Q(0,k) yields:

$$\frac{1}{\pi}=\frac{12A}{\sqrt{C}}+\frac{12}{\sqrt{C}}\frac{P(0,k)}{Q(0,k)}=>$$

$$\pi=\frac{\sqrt{C}\cdot Q(0,k)}{12\cdot(P(0,k)+A\cdot Q(0,k))}$$

Which is the Binary Splitting algorithm for Borwein 25 π.

The relative cost is:

$$Relative\ \ cost=\frac{4D}{\log\left(\frac{C_q}{C_r}\right)}=\frac{4\cdot 3}{\log\left(\frac{(159999840\sqrt{61}+1249638720)^3}{24\cdot 72}\right)}=0.208$$

Deriving the Binary splitting method for Borwein50 π

Borwein series that find approx. 50 correct digits per iteration:

$$\frac{1}{\pi}=\frac{1}{\sqrt{-C^3}}\sum_{k=0}^{\infty}\frac{(6k)!(A+nB)}{(3k)!(k!)^3C^{3k}}$$

Where the constants A, B, and C are:

$$A=A_1+A_2\sqrt{5}+384\sqrt{5}(A_3+A_4\sqrt{5})^{\frac{1}{2}}$$

Where:

$$A_1=63365028312971999585426220$$
$$A_2=28337702140800842046825600$$
$$A_3=10891728551171178200467436212395209160385656017$$
$$A_4=4870929086578810225077338534541688721351255040$$

$$B=B_1+B_2\sqrt{5}+2515968\sqrt{31101}\left(B_3\ +\ B_4\sqrt{5}\right)^{\frac{1}{2}}$$

Where :

$B_1=784991045349662721028974900$
$B_2=351058667826093202896560640$
$B_3=626020832378900163699332265444402088216 1$
$B_4=2799650273060444296577206890718825190235$

$$C=-C_1-C_2\sqrt{5}-1296\sqrt{5}\left(C_3\ +\ C_4\sqrt{5}\right)^{\frac{1}{2}}$$

Where:
$C_1=214772995063512240$
$C_2=96049403338648032$
$C_3=1098523457946355032371331847 3$
$C_4=4912746253692362754607395912$ Where the constants A, B, and C are:

Separate the factorial into product notation and move the constant a k=0 out of the summation:

Where (6k)! in product notation is: $\prod_{i=1}^{k} 6i(6i-1)(6i-2)(6i-3)(6i-4)(6i-5)$

(3k)! is: $\prod_{i=1}^{k} (3i)(3i-1)(3i-2)$

And $(k!)^3$ is: $\prod_{i=1}^{k} i^3$ yields:

$$\frac{1}{\pi}=\frac{A}{\sqrt{-C^3}}+\frac{1}{\sqrt{-C^3}}\sum_{k=1}^{\infty}\frac{(A+kB)}{C^{3k}}\prod_{i=k}^{k}\frac{6i(6i-1)(6i-2)(6i-3)(6i-4)(6i-5)}{3i(3i-1)(3i-2)i^3}$$

Simplify:

$$\frac{1}{\pi}=\frac{A}{\sqrt{-C^3}}+\frac{1}{\sqrt{-C^3}}\sum_{k=1}^{\infty}\frac{(A+kB)}{C^{3k}}\prod_{i=k}^{k}\frac{24(6i-1)(2i-1)(6i-5)}{i^3}$$

Rearranging the formula to fit into the binary splitting form:

Practical implementation of π Algorithms

$$\frac{1}{\pi} = \frac{A}{\sqrt{-C^3}} + \frac{1}{\sqrt{-C^3}} \sum_{k=1}^{\infty} \frac{(6i-1)(2i-1)(6i-5)(A+kB)}{(6i-1)(2i-1)(6i-5)C^{3k}} \prod_{i=k}^{k} \frac{C^3 \cdot 24(6i-1)(2i-1)(6i-5)}{C^3 \cdot i^3}$$

Simplify:

$$\frac{1}{\pi} = \frac{A}{\sqrt{-C^3}} + \frac{1}{\sqrt{-C^3}} \sum_{k=1}^{\infty} \frac{(6i-1)(2i-1)(6i-5)(A+kB)}{(6i-1)(2i-1)(6i-5)} \prod_{i=k}^{k} \frac{(6i-1)(2i-1)(6i-5)}{\frac{C^3}{24} \cdot i^3}$$

Now identify the P(k), Q(k) and R(K) you get:

$$P(k) = (A+Bk)(6k-1)(2k-1)(6k-5)$$

$$Q(k) = \frac{C^3}{24} k^3$$

$$R(k) = (6k-1)(2k-1)(6k-5) = 72k^3 - 108k^2 + 46k - 5$$

Replacing the series with P(0,k) and Q(0,k) yields:

$$\frac{1}{\pi} = \frac{A}{\sqrt{-C^3}} + \frac{1}{\sqrt{-C^3}} \frac{P(0,k)}{Q(0,k)} \Rightarrow$$

$$\pi = \frac{\sqrt{-C^3} \cdot Q(0,k)}{(P(0,k) + A \cdot Q(0,k))}$$

Which is the Binary Splitting algorithm for Borwein 50 π.

The relative cost is $Relative\ \ cost = \frac{4D}{\log\left(\frac{C_q}{C_r}\right)} = 0.103$

Appendix B

Derivation of the Higher-Order Sine Iterations for π

The paper presents in section 1 a ninth-order iteration for π based on sin(x) without deriving where the formula comes from or how its coefficients are obtained. This section provides that derivation in full, shows that the ninth-order case is one member of an infinite family of methods of every odd order, and derives the complete family up to the thirteenth order. The derivation reveals that the coefficients are not arbitrary: they are precisely the coefficients of the Taylor series of arcsin(x), which gives the family a clean, closed-form expression and makes it straightforward to extend to any desired order.

Setting Up the Error Equation

x_n be the current approximation and write the error as $e_n = x_n - \pi$, so that $x_n = \pi + e_n$. The key identity is: $\sin(x_n) = \sin(\pi + e_n) = -\sin(e_n)$

Expanding the right-hand side as a Taylor series in the error:

$$\sin(x_n) = -e_n + e_n^3/6 - e_n^5/120 + e_n^7/5040 - e_n^9/362880 + \cdots$$

$x_{n+1} = x_n + \sin(x_n)$ (noting that $f'(x) = \cos(x) \to -1$ near π, so the Newton update adds rather than subtracts) gives:

$$e_{n+1} = e_n + \sin(x_n) = e_n - e_n + e_n^3/6 - \cdots = e_n^3/6 - \cdots$$

$e_n^3/6$, so the error cubes at each step. The number of correct digits, therefore, triples with each iteration, which is what is meant by cubic convergence in this context.

Constructing Higher-Order Iterations

To achieve a higher convergence order, we add correction terms in odd powers of sin(x). The iteration takes the general form:

$$x_{n+1} = x_n + a_1 \sin(x_n) + a_3 \sin^3(x_n) + a_5 \sin^5(x_n) + \cdots$$

$\sin^k(x) \approx (-e)^k$. Even powers would contribute positively and cannot cancel the e, e^3, e^5, $\cdots$ terms we need to eliminate. Each odd power of sin(x) gives access to the corresponding odd power of e plus higher-order corrections.

The error in the updated estimate becomes:

$$e_{n+1} = e_n + a_1 \sin(x_n) + a_3 \sin^3(x_n) + \cdots$$

$\sin(x_n)$ in terms of e_n and collect terms by power of e. Writing $s = \sin(x_n)$ for brevity:

$$s = -e + e^3/6 - e^5/120 + e^7/5040 - \cdots$$

$$s^3 = -e^3 + e^5/2 - 13e^7/120 + \cdots$$

$$s^5 = -e^5 + 5e^7/6 - \cdots$$

$s^7 = -e^7 + \cdots$

To cancel the e^1 term we require:

$$1 - a_1 = 0 \implies a_1 = 1$$

To cancel the e^3 term, once $a_1 = 1$ is fixed:

$$1/6 - a_3 = 0 \implies a_3 = 1/6$$

To cancel e^5, with a_1 and a_3 now fixed:

$$-1/120 + a_3/2 - a_5 = 0 \implies a_5 = -1/120 + 1/12 = 3/40$$

Each further coefficient is determined uniquely by the requirement that the coefficient of the next odd power of e is zero. This is a triangular system: each new coefficient a_{2k+1} is solved directly from the accumulated constraint at order e^{2k+1}, with no freedom remaining once $a_1 = 1$ is fixed.

The Connection to the arcsin Series

The pattern in the coefficients reveals something elegant. The Taylor series of arcsin(x) about x = 0 is:

$$\arcsin(x) = x + x^3/6 + 3x^5/40 + 5x^7/112 + 35x^9/1152 + 63x^{11}/2816 + \cdots$$

These are exactly the coefficients a_1, a_3, a_5, a_7, a_9, a_{11} derived above. This is not a coincidence. The iteration is constructing a polynomial approximation to $\arcsin(\sin(x_n))$ truncated at a chosen degree. Near π, $\sin(x) = -\sin(e)$ and $\arcsin(-\sin(e)) = -e$, so the exact iteration:

$$x_{n+1} = x_n + \arcsin(-\sin(x_n))$$

Would converge to π in a single step from any starting point within the correct range. Truncating the arcsin series at degree 2K−1 gives an iteration that cancels error terms through e^{2k-1}, leaving e^{2k+1} as the leading residual, and thereby achieving (2K+1)-th order convergence.

The general coefficient is given in closed form by the central binomial formula:

$$a_{2k+1} = C(2k, k) / (4^k (2k+1)) \quad \text{for } k = 0, 1, 2, \cdots$$

where C(2k, k)= is the central binomial coefficient: $\binom{2k}{k}=\frac{(2k)!}{k!k!}$. This formula generates the complete sequence: 1, 1/6, 3/40, 5/112, 35/1152, 63/2816, 231/13312, ⋯, and any term can be computed directly without solving the triangular system.

The Complete Family of Iterations

Each family member uses a different number of terms in the arcsin series, resulting in different convergence orders. The table below summarises the family from order 3 to

order 13, showing the convergence order, the number of $\sin^k$ evaluations required, and the digit-multiplication factor per iteration.

Order	Terms in arcsin	Digits factor	$\sin^k$ calls	Iteration formula
3	1	×2	1	$x + \sin(x)$
5	2	×4 (×5*)	1	$x + \sin(x)\cdot(6 + s^2) / 6$
7	3	×6 (×7*)	1	$x + \sin(x)\cdot(1 + s^2\cdot(1/6 + s^2\cdot 3/40))$
9	4	×8 (×9*)	1	$x + \sin(x)\cdot(1680 + s^2\cdot(280 + s^2\cdot(126 + s^2\cdot 75))) / 1680$
11	5	×10 (×11*)	1	$x + \sin(x)\cdot(40320 + s^2\cdot(6720 + s^2\cdot(3024 + s^2\cdot(1800 + s^2\cdot 1225)))) / 40320$
13	6	×12 (×13*)	1	$x + \sin(x)\cdot(887040 + s^2\cdot(147840 + s^2\cdot(66528 + s^2\cdot(39600 + s^2\cdot(26950 + s^2\cdot 19845))))) / 887040$

* The digit multiplication factor is strictly the order of convergence, but in practice, the effective gain per step is slightly less than the theoretical order because the leading error coefficient is not 1. The values shown without an asterisk are the theoretical digits-doubled-per-step lower bound; values with an asterisk are the nominal order.

Horner Form and Single sine Evaluation

A critical implementation observation is that each family member requires only a single evaluation of sin(x) per iteration, regardless of the convergence order. This is because all terms are powers of the same quantity $s = \sin(x_n)$. Once s is computed, all subsequent operations are multiplications and additions on the already-computed value and its successive squares.

The most efficient evaluation uses Horner's scheme, nesting the polynomial in s^2 from the inside outward. For the ninth-order case this gives:

$$x_{n+1} = x_n + s \cdot (1680 + s^2 \cdot (280 + s^2 \cdot (126 + s^2 \cdot 75))) / 1680$$

which is exactly the form appearing in the paper. The denominator 1680 = lcm(6, 40, 112) × ⋯ is the least common multiple of the denominators of the arcsin coefficients through degree 7, chosen to clear fractions while keeping integer arithmetic in the inner loop.

For the eleventh-order case, the denominator is 40320 = 8!, and for the thirteenth-order case, it is 887040. In general, the denominators grow rapidly, which is of no concern in arbitrary precision arithmetic but is worth noting for fixed-precision implementations.

Is There a Practical Benefit Beyond Ninth Order?

The family extends indefinitely: the eleventh-order method uses five arcsin terms, the thirteenth uses six, and so on. Each additional term costs one extra multiplication by s^2, which is cheap in arbitrary precision arithmetic relative to the cost of computing $\sin(x_n)$ itself.

However, the fundamental bottleneck of the entire family remains the single evaluation of $\sin(x_n)$ per iteration. At arbitrary precision, computing sine to p digits requires an internal series evaluation that itself costs $O(M(p)\cdot\sqrt{p})$ operations or similar. This is far more expensive than the handful of multiplications added by going from ninth to eleventh or thirteenth order. The gain from raising the convergence order from 9 to 11 is that the number of iterations decreases by a factor of $\log(9)/\log(11) \approx 0.92$, a saving of roughly 8%. This is outweighed in practice by the increased polynomial evaluation cost only marginally, so there is a mild theoretical argument for using eleventh or thirteenth order if the sine evaluation is the sole bottleneck.

In practice, however, all members of this family are dominated by the methods of section three, which achieve the same or higher effective convergence rates using only the four basic arithmetic operations and the square root, without any trigonometric evaluation. The AGM-based methods have no sin(x) inner loop to pay for, so the entire cost-per-digit comparison is decided before the convergence order is even relevant. The Newton-sine family is therefore best understood as a clean mathematical construction rather than a practical competitor for high-precision computation of π.

The table below shows the theoretical iteration count required to reach one million digits, starting from a precision of 16 digits, for each member of the family, to make the diminishing returns of higher order concrete.

Order	Iterations to 10^6 digits	Relative to order-3 baseline
3	19.9	1.00 (baseline)
5	12.7	0.64
7	10.1	0.51
9	8.6	0.43
11	7.7	0.39
13	7.1	0.36

The iteration counts are computed as $\log_2(10^6 / 16) / \log_2(\text{order})$. The improvement from order 9 to order 11 saves fewer than one iteration at this scale, confirming that the ninth-order method is a practical stopping point within the family. Going to order 13 or beyond yields diminishing returns that are unlikely to justify the additional code complexity.

Appendix C

Guard Digit Analysis for the Ramanujan Series

When implementing an infinite series for π in arbitrary precision arithmetic, the working precision must exceed the target precision by a sufficient number of guard digits to absorb the rounding errors accumulated over the course of the computation. A flat empirical constant is inadequate because the number of terms required, and therefore the accumulated rounding, grows with the target precision. This section derives a rigorous lower bound on the number of guard digits needed for the Ramanujan series implementation.

The Series and Its Convergence Rate

The Ramanujan formula for π is:

$$\pi = 9801 / (2\sqrt{2} \cdot (1103 \cdot A + 26390 \cdot B))$$

where the partial sums A and B are accumulated over N terms:

$$A = \sum a_n, \quad B = \sum n \cdot a_n \quad (n = 0, 1, 2, \ldots, N-1)$$

and the term recurrence is:

$$a_0 = 1, \quad a_n = a_{n-1} \cdot (4n-1)(4n-2)(4n-3) / (6147814464 \cdot n^3)$$

The asymptotic ratio of consecutive terms for large n is:

$$a_n / a_{n-1} \rightarrow (4n)^3 / (6147814464 \cdot n^3) = 64 / 6147814464 = 1 / 96059601 = 256 / 396^4$$

This gives a convergence rate of $\log_{10}(96059601) \approx 7.983$ decimal digits per term, which is the well-known figure of approximately 8 digits per term. The number of terms required to reach p decimal digits of precision is therefore:

$$N \geq \lceil p / 7.983 \rceil \approx \lceil p / 8 \rceil$$

Structure of the Denominator Sum

Before analyzing rounding errors, it is important to observe the relative magnitudes of the two partial sums. The initial term is $a_0 = 1$, and all subsequent terms are geometrically decreasing with ratio $r \approx 1/96059601$. Consequently, $A \approx 1$ throughout the summation, with corrections at the level of 10^{-9}. The partial sum B is even smaller: its first term is $1 \cdot a_1 \approx 9.76 \cdot 10^{-10}$, and B remains many orders of magnitude below 1 for all practical precisions.

Numerically, the ratio of the two contributions to the denominator sum $S = 1103 \cdot A + 26390 \cdot B$ satisfies:

$$26390 \cdot B / (1103 \cdot A) \approx 24 \cdot B \approx 24 \cdot (N/2) \cdot a_1 \approx 12\, N \cdot 10^{-9}$$

For $p = 10{,}000$ digits, this ratio is approximately $1.5 \cdot 10^{-5}$, confirming that A dominates S overwhelmingly. Despite its small magnitude, the B term cannot be ignored in the error budget because it is amplified by the factor $26390/1103 \approx 24$ relative to A when both contribute rounding errors to S.

Sources of Rounding Error

Throughout the computation, the working precision is $p + g$ decimal digits, where g is the number of guard digits. All arithmetic operations are assumed to produce results with absolute rounding error bounded by $0.5 \cdot 10^{-n}$ where $n = p + g$ is the working precision. The five sources of error are analyzed in turn.

Truncation error. The loop terminates when the next term satisfies $(A + a_n) = A$ in working precision arithmetic, meaning $a_n < 0.5 \cdot 10^{-n}$. The truncation error in the final result is therefore already controlled to within working precision and contributes nothing to the guard digit requirement.

Rounding in the a_n recurrence. Each term update involves two multiplications and one division, contributing at most $1.5 \cdot |a_n| \cdot 10^{-n}$ of absolute error per step. Since the resulting contaminated a_n is then added to A and B, this error is absorbed into the accumulation error analyzed below and need not be counted separately.

Accumulation error in A. At each of the N additions $A \leftarrow A + a_n$, the result is rounded to working precision with absolute error at most $0.5 \cdot 10^{-n}$. Since $A \approx 1$ throughout, the ulp of A is 10^{-n} regardless of how many terms have been added. Over N additions, the total accumulated absolute error in A is bounded by:

$$\varepsilon_A \le N \cdot 0.5 \cdot 10^{-(p+g)}$$

Accumulation error in B. Each addition $B \leftarrow B + n \cdot a_n$ introduces a rounding error of at most $0.5 \cdot 10^{-n}$ in absolute terms, since all operations are performed at working precision $p + g$ regardless of the magnitude of the operands. Over N additions:

$$\varepsilon_B \le N \cdot 0.5 \cdot 10^{-(p+g)}$$

Error from √2. The constant $2\sqrt{2}$ is precomputed at precision $p + g$, contributing an absolute error of at most $0.5 \cdot 10^{-(p+g)}$ to the final result. This is negligible compared to the accumulation terms.

Final formula rounding. The evaluation of $9801 / (2\sqrt{2} \cdot (1103 \cdot A + 26390 \cdot B))$ involves four arithmetic operations, contributing at most $2 \cdot 10^{-(p+g)}$ to the absolute error in π. This is also negligible.

Combined Error in the Denominator Sum

The two accumulation errors propagate into the denominator sum $S = 1103 \cdot A + 26390 \cdot B$. The absolute error in S is:

$$\varepsilon_S \leq 1103 \cdot \varepsilon_A + 26390 \cdot \varepsilon_B \leq (1103 + 26390) \cdot N \cdot 0.5 \cdot 10^{-(p+g)} \leq 27493 \cdot N \cdot 0.5 \cdot 10^{-(p+g)}$$

Since $\pi \approx 9801 / (2\sqrt{2} \cdot S)$ and $S \approx 1103$ (dominated by the $1103 \cdot A$ term with $A \approx 1$), the relative error in π due to rounding in S is approximately:

$$|\delta\pi / \pi| \approx \varepsilon_S / S \leq 27493 \cdot N \cdot 0.5 \cdot 10^{-(p+g)} / 1103 \approx 12.5 \cdot N \cdot 10^{-(p+g)}$$

Substituting $N \approx p/8$:

$$|\delta\pi / \pi| \leq (12.5 / 8) \cdot p \cdot 10^{-(p+g)} \approx 1.56 \cdot p \cdot 10^{-(p+g)}$$

Deriving the Guard Digit Requirement

For the computed value of π to be correct to p decimal digits the relative error must satisfy $|\delta\pi / \pi| < 0.5 \cdot 10^{-p}$. Substituting the bound above:

$$1.56 \cdot p \cdot 10^{-(p+g)} < 0.5 \cdot 10^{-p}$$

$$10^g > 1.56 \cdot p / 0.5 = 3.12 \cdot p$$

$$g > \log_{10}(3.12 \cdot p) = \log_{10}(p) + \log_{10}(3.12)$$

$$g > \log_{10}(p) + 0.494$$

Rounding up to the nearest integer, the minimum number of guard digits required is:

$$g_{min} = \lceil \log_{10}(p) + 0.494 \rceil = \lceil \log_{10}(3.12 \cdot p) \rceil$$

Adding one further decimal digit of safety margin to account for error accumulation in intermediate quantities that the bound treats as exact, the recommended working guard is:

$$g_{rec} = \lceil \log_{10}(p) \rceil + 2$$

This formula has a natural interpretation: the guard-digit requirement increases by one decimal place each time the target precision increases by a factor of 10. It replaces any flat constant with a value that is provably sufficient at all precisions.

Numerical Values

The table below lists the rigorously derived minimum guard digits and the recommended working guard for selected target precisions, along with the number of series terms required for each precision.

Target p (digits)	Terms N ≈ p/8	$g > \log_{10}(3.12p)$	$g_{min} = \lceil \dots \rceil$	$g^{Rec} = \lceil \log_{10}(p) \rceil + 2$
100	13	2.494	3	4
1,000	125	3.494	4	5
10,000	1,250	4.494	5	6

100,000	12,500	5.494	6	7
1,000,000	125,000	6.494	7	8
10,000,000	1,250,000	7.494	8	9

Summary of the Error Budget

The table below collects all five error contributions and their orders of magnitude relative to the target precision 10^{-p}, evaluated at p = 10,000 and g = 6 (the recommended working guard at that precision).

Error source	Bound	At p=10,000, g=6	Dominant?
Truncation	Controlled by loop exit	$< 10^{-n}$	No
a_n recurrence rounding	Subsumed in accumulation	$< 10^{-n}$	No
A accumulation	$N \cdot 0.5 \cdot 10^{-n}$	$6.25 \cdot 10^{-10006}$	Partial
B accumulation (amplified)	$24N \cdot 0.5 \cdot 10^{-n}$	$1.5 \cdot 10^{-10004}$	Yes
$2\sqrt{2}$ precomputation	$0.5 \cdot 10^{-n-1}$	$5 \cdot 10^{-10007}$	No
Final formula (4 ops)	$2 \cdot 10^{-n}$	$2 \cdot 10^{-10006}$	No
Total	$1.56p \cdot 10^{-n}$	$< 0.5 \cdot 10^{-10000}$	

The dominant contribution is the B accumulation amplified by the factor 26390/1103 ≈ 24. At p = 10,000 with g = 6, the total rounding error is comfortably below $0.5 \cdot 10^{-p}$, confirming that the recommended formula $g^{Rec} = \lceil \log_{10}(p) \rceil + 2$ provides a rigorous safety margin at all target precisions.

Appendix D

Guard Digit Analysis for the Chudnovsky Series

The Chudnovsky series is the fastest-converging hypergeometric series for π in practical use, adding approximately 14.18 decimal digits per term compared to approximately 7.98 for the Ramanujan series. Despite this faster convergence, the same category of accumulated rounding error arises during the summation, and the guard digit requirement must be determined rigorously rather than chosen empirically. This section derives a rigorous lower bound on the number of guard digits required, following the same structure as the Ramanujan analysis and allowing a direct comparison between the two series.

The Series and Its Convergence Rate

The Chudnovsky formula for π is:

$$\pi = 426880 \cdot \sqrt{10005} / (13591409 \cdot A + 545140134 \cdot B)$$

where the partial sums A and B are accumulated over N terms:

$$A = \sum a_n, \quad B = \sum n \cdot a_n \quad (n = 0, 1, 2, \ldots, N-1)$$

with the term recurrence:

$$a_0 = 1, \quad a_n = -a_{n-1} \cdot (6n-5)(2n-1)(6n-1) / (640320^3/24 \cdot n^3)$$

The alternating sign, encoded by the leading minus, reflects the $(-1)^n$ factor in the original Chudnovsky formula. The asymptotic ratio of consecutive term magnitudes for large n is:

$$|a_n| / |a_{n-1}| \rightarrow (6n)^2(2n)(6n) / (640320^3/24 \cdot n^3) = 1728 / 640320^3 = 1 / 151\,931\,373\,056\,000$$

Taking the base-10 logarithm gives the convergence rate:

$$\log_{10}(1728/640320^3) = \log_{10}(151\,931\,373\,056\,000) \approx 14.1816 \text{ decimal digits per term}$$

The number of terms required to reach p decimal digits of precision is therefore:

$$N \geq \lceil p / 14.1816 \rceil \approx \lceil p \cdot 0.07051 \rceil$$

Structure of the Denominator Sum

The initial term is $a_0 = 1$ and the first correction is $|a_1| = 5 \cdot 24 / 640320^3 \approx 4.57 \cdot 10^{-16}$, which is negligible. As a result, $A \approx 1$ throughout the summation, with alternating corrections at the level of 10^{-16} and smaller. The partial sum B is comparably tiny, with its first contributing term being $|1 \cdot a_1| \approx 4.57 \cdot 10^{-16}$.

The ratio of the two contributions to the denominator sum $S = 13591409 \cdot A + 545140134 \cdot B$ satisfies:

$$545140134 \cdot |B| / (13591409 \cdot A) \approx 40.1 \cdot |B|$$

For p = 10,000, this ratio is approximately $6.5 \cdot 10^{-12}$, confirming that A dominates the denominator overwhelmingly. Despite B's small magnitude relative to A, the coefficient 545140134 is approximately 40 times larger than 13591409, so B contributes a rounding error amplified by this same factor when both accumulations propagate into S.

Sources of Rounding Error

Throughout the computation, the working precision is p + g decimal digits. All arithmetic operations produce results with absolute rounding error bounded by $0.5 \cdot 10^{-n}$ where n = p + g. The five sources of error are:

Truncation error. The loop exits when $(A + a_n) = A$ in working precision arithmetic, meaning $|a_n| < 0.5 \cdot 10^{-n}$. The remaining tail is therefore already absorbed into the working precision and contributes nothing to the guard-digit requirement.

Rounding in the a_n recurrence. Each term update involves two multiplications and one division, contributing at most $1.5 \cdot |a_n| \cdot 10^{-n}$ of absolute error per step. Since the contaminated a_n is subsequently added to A and B, this error is absorbed into the accumulation error below and need not be counted separately.

Accumulation error in A. At each of the N additions $A \leftarrow A + a_n$, the result is rounded with absolute error at most $0.5 \cdot 10^{-n}$. Since $A \approx 1$ throughout, the unit in the last place of A is 10^{-n} regardless of the iteration count. Over N additions, the total accumulated absolute error in A is bounded by:

$$\varepsilon_A \leq N \cdot 0.5 \cdot 10^{-(p+g)}$$

Accumulation error in B. Each addition $B \leftarrow B + n \cdot a_n$ rounds with absolute error at most $0.5 \cdot 10^{-n}$, since the computation is performed at working precision p + g regardless of the magnitude of the operands. Over N additions:

$$\varepsilon_B \leq N \cdot 0.5 \cdot 10^{-(p+g)}$$

Error from √10005. The constant √10005 is computed at precision p + g, contributing an absolute error of at most $0.5 \cdot 10^{-n}$. Since it enters the numerator multiplicatively, its relative contribution to π is $0.5 \cdot 10^{-n}$, which is negligible compared to the accumulation terms.

Final formula rounding. The evaluation of $426880 \cdot \sqrt{10005} / (13591409 \cdot A + 545140134 \cdot B)$ involves four arithmetic operations contributing at most $2 \cdot 10^{-n}$ in absolute error. This is also negligible.

Combined Error in the Denominator Sum

The two accumulation errors propagate into the denominator sum $S = 13591409 \cdot A + 545140134 \cdot B$. The absolute error in S is:

$$\varepsilon_S \leq 13\,591\,409 \cdot \varepsilon_A + 545\,140\,134 \cdot \varepsilon_B \leq 558\,731\,543 \cdot N \cdot 0.5 \cdot 10^{-(p+g)}$$

Since π is inversely proportional to S, and S ≈ 13591409 (dominated by 13591409 · A with A ≈ 1), the relative error in π arising from rounding in S is:

$|\delta\pi / \pi| \approx \varepsilon_S / S \leq (558\,731\,543 / 13\,591\,409) \cdot N \cdot 0.5 \cdot 10^{-(p+g)} \approx 20.55 \cdot N \cdot 10^{-(p+g)}$

Substituting N = p / 14.1816:

$$|\delta\pi / \pi| \leq 1.449 \cdot p \cdot 10^{-(p+g)}$$

Deriving the Guard Digit Requirement

For the computed value of π to be correct to p decimal digits, the relative error must satisfy $|\delta\pi / \pi| < 0.5 \cdot 10^{-p}$. Substituting the bound above:

$$1.449 \cdot p \cdot 10^{-(p+g)} < 0.5 \cdot 10^{-p}$$

$$10^g > 2 \cdot 1.449 \cdot p = 2.899 \cdot p$$

$$g > \log_{10}(2.899 \cdot p) = \log_{10}(p) + \log_{10}(2.899) \approx \log_{10}(p) + 0.462$$

Rounding up to the nearest integer gives the minimum number of guard digits:

$$g_{min} = \lceil \log_{10}(p) + 0.462 \rceil = \lceil \log_{10}(2.899 \cdot p) \rceil$$

Adding one further decimal digit of safety margin to account for secondary error sources treated as exact in the bound, the recommended working guard is:

$$g_{rec} = \lceil \log_{10}(p) \rceil + 2$$

Numerical Values

The table below gives the rigorously derived minimum guard digits and the recommended working guard for selected target precisions, alongside the number of series terms required at each precision.

Target p (digits)	**Terms N ≈ p/14.18**	**$g > \log_{10}(2.899 \cdot p)$**	**$g_m{}^I{}_n = \lceil \dots \rceil$**	**$g^{Rec} = \lceil \log_{10}(p) \rceil + 2$**
100	8	2.462	3	4
1,000	71	3.462	4	5
10,000	706	4.462	5	6
100,000	7,052	5.462	6	7
1,000,000	70,514	6.462	7	8
10,000,000	705,137	7.462	8	9

Comparison with the Ramanujan Series

The guard digit formulae for both series have the same logarithmic structure but differ in their additive constants. The rigorous bounds are:

Ramanujan: $g > \log_{10}(p) + 0.494$

Chudnovsky: $g > \log_{10}(p) + 0.462$

The Chudnovsky constant 0.462 is slightly smaller than the Ramanujan constant 0.494. This reflects the faster convergence of the Chudnovsky series: since $N = p/14.18$ versus $N = p/7.98$ for Ramanujan, the number of rounding events is roughly 1.78 times smaller for the same target precision, which reduces the accumulated error proportionally. Despite this difference, both constants are less than 0.5, meaning both series round up to the same g_min at every practical precision level. The recommended formula $g_rec = \lceil \log_{10}(p) \rceil + 2$ is therefore equally valid and sufficient for both series.

Summary of the Error Budget

The table below collects all five error contributions at p = 10,000 and g = 6, confirming that the total rounding error is comfortably below $0.5 \cdot 10^{-p}$.

Error source	Bound	At p=10,000, g=6	Dominant?
Truncation	Controlled by loop exit	$< 10^{-n}$	No
a_n recurrence rounding	Subsumed in accumulation	$< 10^{-n}$	No
A accumulation	$N \cdot 0.5 \cdot 10^{-n}$	$3.53 \cdot 10^{-10006}$	Partial
B accumulation (amplified ·40)	$40N \cdot 0.5 \cdot 10^{-n}$	$1.41 \cdot 10^{-10004}$	Yes
√10005 precomputation	$0.5 \cdot 10^{-n-1}$	$5 \cdot 10^{-10007}$	No
Final formula (4 ops)	$2 \cdot 10^{-n}$	$2 \cdot 10^{-10006}$	No
Total	$1.449p \cdot 10^{-n}$	$< 0.5 \cdot 10^{-10000}$	

The dominant contribution is the B accumulation amplified by the ratio 545140134 / 13591409 ≈ 40. At p = 10,000 with g = 6, the total rounding error is well below $0.5 \cdot 10^{-p}$, confirming that $g^{Rec} = \lceil \log_{10}(p) \rceil + 2$ provides a rigorous and sufficient safety margin for the Chudnovsky series at all target precisions.

Appendix E

Guard Digit Analysis for the Borwein 25-Digit Series

The Borwein 25-digit series is a hypergeometric series for π derived from a modular equation of level 25, discovered by Jonathan and Peter Borwein. It adds approximately 24.96 decimal digits per term, making it significantly faster converging than either Ramanujan (7.98 digits per term) or Chudnovsky (14.18 digits per term). This section derives a rigorous lower bound on the required number of guard digits, following the same analytical framework applied to those two series.

The Series and Its Convergence Rate

The Borwein 25 formula for π is:

$$\pi = \sqrt{C} / (12 \cdot (A \cdot S_A + B \cdot S_B))$$

where the constants are:

$$A = 1\,657\,145\,277\,365 + 212\,175\,710\,912 \cdot \sqrt{61}$$

$$B = 107\,578\,229\,802\,750 + 13\,773\,980\,892\,672 \cdot \sqrt{61}$$

$$C = (1\,249\,638\,720 + 159\,999\,840 \cdot \sqrt{61})^3$$

and the partial sums are accumulated over N terms:

$$S_A = \sum a_n, \quad S_B = \sum n \cdot a_n \quad (n = 0, 1, 2, \ldots, N-1)$$

with the term recurrence:

$$a_0 = 1, \quad a_n = -a_{n-1} \cdot (6n-5)(48n-24)(6n-1) / (C \cdot n^3)$$

The factor $(48n - 24) = 24(2n - 1)$, so the numerator is $24(6n-5)(2n-1)(6n-1)$, which is structurally identical to the Chudnovsky recurrence, with the constant C scaled to reflect the modular equation of level 25. The asymptotic ratio of consecutive term magnitudes for large n is:

$$|a_n| / |a_{n-1}| \to 1728 \cdot 24 / C \text{ ... wait, more precisely:}$$

$$|a_n| / |a_{n-1}| \to (6n)^2(48n)(6n) / (C \cdot n^3) = 1728 / C \approx 1.107 \cdot 10^{-25}$$

Taking the base-10 logarithm:

$$\log_{10}(1728/C) \approx 24.956 \text{ decimal digits per term}$$

The number of terms required to reach p decimal digits of precision is therefore:

$$N \geq \lceil p / 24.956 \rceil \approx \lceil p \cdot 0.04007 \rceil$$

Numerical Values of the Constants

Evaluating the constants numerically gives:

$$A \approx 3.3143 \cdot 10^{12}, \quad B \approx 2.1516 \cdot 10^{14}, \quad C \approx 1.5611 \cdot 10^{28}$$

The ratio $B/A \approx 64.92$ is the key amplification factor for this series. It is substantially larger than the corresponding ratios for Ramanujan (≈24) and Chudnovsky (≈40), reflecting the larger coefficient B relative to A in the modular equation underlying this series. As will be seen below, the faster convergence rate (fewer terms, N) more than compensates for the larger amplification factor in the final guard-digit requirement.

Structure of the Denominator Sum

The initial term is $a_0 = 1$ and the first correction is $|a_1| = 120/C \approx 7.69 \cdot 10^{-27}$, which is many orders of magnitude smaller than 1. Consequently S

$A \approx 1$ throughout the summation. The partial sum S_B is comparably tiny. The ratio of the B contribution to the A contribution in the denominator $S = A \cdot S_A + B \cdot S_B$ satisfies:

$$B \cdot |S_B| / (A \cdot S_A) \approx 64.92 \cdot |S_B|$$

For p = 10,000, this ratio is approximately 10^{-22}, confirming that S^A dominates the denominator overwhelmingly at all practical precisions.

Sources of Rounding Error

Throughout the computation, the working precision is p + g decimal digits. All arithmetic operations produce results with absolute rounding error bounded by $0.5 \cdot 10^{-n}$ where n = p + g. The error sources are:

Truncation error. The loop exits when $(S_A + a_n) = S_A$ in working precision, so the remaining tail is absorbed into working precision. No contribution to the guard digit requirement.

Rounding in the a_n recurrence. Each term update involves two multiplications and one division. The resulting error in a_n is subsumed into the accumulation error below.

Accumulation error in S_A. Since $S_A \approx 1$ throughout, each of the N addition rounds has an absolute error at most $0.5 \cdot 10^{-n}$. Over N additions:

$$\varepsilon_A \le N \cdot 0.5 \cdot 10^{-(p+g)}$$

Accumulation error in S_B. Each addition $S_B \leftarrow S_B + n \cdot a_n$ rounds with absolute error at most $0.5 \cdot 10^{-n}$ at working precision. Over N additions:

$$\varepsilon_B \le N \cdot 0.5 \cdot 10^{-(p+g)}$$

Error from $\sqrt{C}$. The constant C is evaluated once at full working precision. Its square root enters the numerator and contributes a relative error of at most $0.5 \cdot 10^{-n}$, which is negligible compared to the accumulation terms.

Final formula rounding. The evaluation of $\sqrt{C} / (12 \cdot (A \cdot S_A + B \cdot S_B))$ involves four arithmetic operations contributing at most $2 \cdot 10^{-n}$. This is also negligible.

Combined Error in the Denominator Sum

The absolute error in $S = A \cdot S^A + B \cdot S^B$ is:

$$\varepsilon_S \le A \cdot \varepsilon_A + B \cdot \varepsilon_B \le (A + B) \cdot N \cdot 0.5 \cdot 10^{-(p+g)}$$

$$\le 2.1847 \cdot 10^{14} \cdot N \cdot 0.5 \cdot 10^{-(p+g)}$$

Since π is related inversely to S, and $S \approx A \approx 3.3143 \cdot 10^{12}$, the relative error in π is:

$$|\delta\pi/\pi| \approx \varepsilon_S/S \le (2.1847 \cdot 10^{14} / 3.3143 \cdot 10^{12}) \cdot N \cdot 0.5 \cdot 10^{-(p+g)} \approx 32.96 \cdot N \cdot 10^{-(p+g)}$$

Substituting N = p / 24.956:

$$|\delta\pi/\pi| \le 1.321 \cdot p \cdot 10^{-(p+g)}$$

Deriving the Guard Digit Requirement

For the result to be correct to p decimal digits, the relative error must satisfy $|\delta\pi/\pi| < 0.5 \cdot 10^{-p}$:

$$1.321 \cdot p \cdot 10^{-(p+g)} < 0.5 \cdot 10^{-p}$$

$$10^{g} > 2 \cdot 1.321 \cdot p = 2.641 \cdot p$$

$$g > \log_{10}(2.641 \cdot p) = \log_{10}(p) + \log_{10}(2.641) \approx \log_{10}(p) + 0.422$$

Rounding up to the nearest integer gives the minimum guard digits:

$$g_{min} = \lceil \log_{10}(2.641 \cdot p) \rceil$$

With one additional digit of safety margin the recommended working guard is:

$$g_{rec} = \lceil \log_{10}(p) \rceil + 2$$

Numerical Values

Target p	Terms N	$g > \log_{10}(2.641\cdot p)$	$g_{m}{}^{I}{}_{n}$	g^{Rec}	$g = \lceil\log_{10}(2p)\rceil+1$
100	5	2.422	3	4	4
1,000	41	3.422	4	5	5
10,000	401	4.422	5	6	6
100,000	4,008	5.422	6	7	7
1,000,000	40,071	6.422	7	8	8

The final column shows the guard produced by the formula $\lceil\log_{10}(2p)\rceil + 1$ used in the implementation. Comparing with $g_{m}{}^{I}{}_{n}$, the implementation formula is always sufficient, providing one extra digit of margin at every precision level. This is consistent with the analysis: $\lceil\log_{10}(2p)\rceil + 1 = \lceil\log_{10}(p) + 0.301\rceil + 1$ produces values that exceed $g_{m}{}^{I}{}_{n} = \lceil\log_{10}(p) + 0.422\rceil$ by exactly one at every power of ten boundaries.

Comparison Across All Three Series

The table below summarizes the error-bound analysis of all three hypergeometric series for π. The three series share the same analytical structure and differ only in their convergence rate and coefficient amplification factor.

Series	Digits/term	B/A ratio	Constant C	$g > \log_{10}(p)+$	g^{Rec}
Ramanujan	7.983	≈24	3.120	0.494	$\lceil\log_{10}(p)\rceil + 2$
Chudnovsky	14.182	≈40	2.899	0.462	$\lceil\log_{10}(p)\rceil + 2$
Borwein 25	24.956	≈65	2.641	0.422	$\lceil\log_{10}(p)\rceil + 2$

The pattern across the three series is consistent and instructive. As the convergence rate increases, the number of terms, N, decreases proportionally, thereby reducing the accumulated rounding error. The B/A amplification ratio increases across the three series (24, 40, 65), but the reduction in N more than compensates. The net effect is that the constant in the guard digit formula decreases monotonically from 3.120 through 2.899 to 2.641, and the additive term $\log_{10}(C)$ decreases from 0.494 through 0.462 to 0.422. All three constants are below 0.5, which means all three series round up to the same $g_m{}^I{}_n$ and require the same recommended formula $g^{Rec} = \lceil\log_{10}(p)\rceil + 2$. A single guard digit formula, therefore, covers all three series rigorously.

Appendix F

Guard Digit Analysis for the Borwein 50-Digit Series

The Borwein 50-digit series is a hypergeometric series for π derived from a modular equation of level 50. It adds approximately 50.56 decimal digits per term, making it the fastest converging of the four hypergeometric series examined in this paper. The series is structurally similar to the Borwein 25 series but uses an irrational constant C, constructed from √5 and √61, which is negative by construction. This section derives the rigorous guard-digit bound and verifies the guard-digit formula used in the implementation.

The Series and Its Convergence Rate

The Borwein 50 formula for π is:

$$\pi = \sqrt{(-C^3)} / (A \cdot S_A + B \cdot S_B)$$

where the three constants A, B, C involve √5 and nested square roots:

$A = 63\,365\,028\,312\,971\,999\,585\,426\,220 + 28\,337\,702\,140\,800\,842\,046\,825\,600 \cdot \sqrt{5} + 384 \cdot \sqrt{5} \cdot \sqrt{(\ldots)}$

$B = 7\,849\,910\,453\,496\,627\,210\,289\,749\,000 + 3\,510\,586\,678\,260\,932\,028\,965\,606\,400 \cdot \sqrt{5} + 2\,515\,968 \cdot \sqrt{3110} \cdot \sqrt{(\ldots)}$

$C = -214\,772\,995\,063\,512\,240 - 96\,049\,403\,338\,648\,032 \cdot \sqrt{5} - 1296 \cdot \sqrt{5} \cdot \sqrt{(\ldots)}$

The constant C is negative by construction, since all three of its constituent terms are negative. Consequently, C^3 is also negative and $\sqrt{(-C^3)} = \sqrt{|C^3|}$ is real and positive. Evaluating numerically:

$$A \approx 2.5346 \cdot 10^{26}, \quad B \approx 3.1400 \cdot 10^{28}, \quad |C| \approx 8.5909 \cdot 10^{17}$$

$$|C^3| \approx 6.3404 \cdot 10^{53}, \quad \sqrt{|C^3|} \approx 7.9627 \cdot 10^{26}$$

The partial sums are accumulated over N terms:

$S_A = \sum a_n, \quad S_B = \sum n \cdot a_n \quad (n = 0, 1, 2, \ldots, N-1)$

with the term recurrence:

$a_0 = 1, \quad a_n = a_{n-1} \cdot (6n-5)(48n-24)(6n-1) / (C^3 \cdot n^3)$

Note that, unlike the Borwein 25 recurrence, there is no explicit negation in the update. The alternating sign is carried implicitly through the negative denominator C^3: since $C^3 < 0$, each factor $(6n-5)(48n-24)(6n-1)/C^3$ is negative, so multiplication by a negative ratio each step produces the correct alternation of signs in the partial sums.

The asymptotic ratio of consecutive term magnitudes for large n is:

$|a_n| / |a_{n-1}| \to 1728 / |C^3| \approx 2.725 \cdot 10^{-51}$

Taking the base-10 logarithm:

$\log_{10}(1728/|C^3|) \approx 50.565$ decimal digits per term

The number of terms required to reach p decimal digits of precision is therefore:

$N \geq \lceil p / 50.565 \rceil \approx \lceil p \cdot 0.01978 \rceil$

Structure of the Denominator Sum

The initial term is $a_0 = 1$ and $|a_1| = 120/|C^3| \approx 1.89 \cdot 10^{-52}$, which is negligible. $S_A \approx 1$ throughout. The ratio B/A ≈ 123.88 is the amplification factor for this series, substantially larger than in the previous three series (24, 40, 65 for Ramanujan, Chudnovsky, and Borwein 25 respectively). However, the convergence rate is also the fastest, reducing N to approximately p/50.6 terms, which more than compensates.

Sources of Rounding Error

The error analysis follows the same structure as the three preceding series. Throughout the computation, the working precision is p + g decimal digits, and all operations produce results with absolute rounding error bounded by $0.5 \cdot 10^{-n}$ where n = p + g.

Truncation error. Controlled by the loop exit condition. No contribution to the guard digit requirement.

Rounding in the a_n recurrence. Subsumed into the accumulation error.

Accumulation error in S_A. Since $S_A \approx 1$ throughout:

$\varepsilon_A \leq N \cdot 0.5 \cdot 10^{-(p+g)}$

Accumulation error in S_B. Each addition rounds at working precision:

$\varepsilon_B \leq N \cdot 0.5 \cdot 10^{-(p+g)}$

Error from $\sqrt{|C^3|}$. Computed once at working precision, contributing a relative error of at most $0.5 \cdot 10^{-n}$. Negligible.

Final formula rounding. Four arithmetic operations contributing at most $2 \cdot 10^{-n}$. Negligible.

Combined Error in the Denominator Sum

The absolute error in $S = A \cdot S^A + B \cdot S^B$ is:

$$\varepsilon_S \le (A + B) \cdot N \cdot 0.5 \cdot 10^{-(p+g)} \approx 3.165 \cdot 10^{28} \cdot N \cdot 0.5 \cdot 10^{-(p+g)}$$

Since π is inversely proportional to S and $S \approx A \approx 2.535 \cdot 10^{26}$:

$$|\delta\pi/\pi| \le (A+B)/(2A) \cdot N \cdot 10^{-(p+g)} \approx 62.44 \cdot N \cdot 10^{-(p+g)}$$

Substituting $N = p / 50.565$:

$$|\delta\pi/\pi| \le 1.235 \cdot p \cdot 10^{-(p+g)}$$

Deriving the Guard Digit Requirement

For the result to be correct to p decimal digits:

$$1.235 \cdot p \cdot 10^{-(p+g)} < 0.5 \cdot 10^{-p}$$

$$10^g > 2 \cdot 1.235 \cdot p = 2.470 \cdot p$$

$$g > \log_{10}(2.470 \cdot p) = \log_{10}(p) + 0.393$$

Rounding up:

$$g_{min} = \lceil \log_{10}(2.470 \cdot p) \rceil$$

With one digit of safety margin:

$$g_{rec} = \lceil \log_{10}(p) \rceil + 2$$

Unified Comparison Across All Four Series

The table below assembles the complete results for all four hypergeometric series. The trend across all four is consistent: as the convergence rate increases, N decreases, and, despite the B/A amplification ratio increasing, the net effect is a slowly decreasing constant in the guard-digit formula.

Series	Digits/term	B/A	Constant C	$g > \log_{10}(p)+$	g^{Rec}
Ramanujan	7.983	~24	3.120	0.494	$\lceil \log_{10}(p) \rceil + 2$
Chudnovsky	14.182	~40	2.899	0.462	$\lceil \log_{10}(p) \rceil + 2$
Borwein 25	24.956	~65	2.641	0.422	$\lceil \log_{10}(p) \rceil + 2$
Borwein 50	50.565	~124	2.470	0.393	$\lceil \log_{10}(p) \rceil + 2$

All four constants (3.120, 2.899, 2.641, 2.470) lie below 4, and all four additive terms (0.494, 0.462, 0.422, 0.393) lie below 0.5. This means all four series round up to the

same $g_m{}^I{}_n$ at every practical precision, and a single formula $g^{Rec} = \lceil \log_{10}(p) \rceil + 2$ is rigorously sufficient for all four. The implementation can therefore use a single unified guard-digit formula regardless of the series selected.

The decreasing trend in the constant also confirms that faster convergence is genuinely beneficial not only in terms of iteration count but also in terms of precision management overhead: faster series require proportionally fewer guard digits, reinforcing their advantage at high precision.

Appendix G

Guard Digit Requirement for the Borwein 25 Binary Splitting Implementation

The Ramanujan and Chudnovsky binary splitting implementations perform all recursion in exact integer arithmetic, so rounding errors enter only in the final floating-point conversion step. As shown earlier, this reduces the guard-digit requirement to a constant $g = 2$, regardless of precision.

The Borwein 25 binary-splitting implementation differs fundamentally. The constants A and B both involve $\sqrt{61}$, which cannot be represented exactly as integers. As a result, p and q must remain float_precision throughout the recursion rather than int_precision, and rounding errors accumulate at every merge node in the recursion tree.

Error Accumulation in the Recursion Tree

The recursion tree for $k \approx p/24$ terms has k leaf nodes and $k - 1$ internal merge nodes. At each leaf approximately 5 float_precision operations construct p and q. At each internal merge node, 3 float_precision operations combine the left and right subtrees:

$$p = p \cdot q_{right} + p_{right} \cdot r_{left}$$

$$q = q \cdot q_{right}$$

Each operation introduces a relative rounding error bounded by $0.5 \cdot 10^{-(p+g)}$. Summing over all nodes in the tree gives approximately 8k total float_precision operations, so the total relative error in the final p is bounded by:

$$|\delta p / p| \le 8k \cdot 0.5 \cdot 10^{-(p+g)} = 4k \cdot 10^{-(p+g)}$$

Since π depends on p through the final formula, the relative error in π has the same bound. Substituting $k \approx p/24$:

$$|\delta\pi / \pi| \le (p/6) \cdot 10^{-(p+g)}$$

Deriving the Guard Digit Requirement

For the result to be correct to p decimal digits the relative error must satisfy $|\delta\pi/\pi| < 0.5 \cdot 10^{-p}$:

$(p/6) \cdot 10^{-(p+g)} < 0.5 \cdot 10^{-p}$

$10^{g} > p/3$

$g > \log_{10}(p/3) = \log_{10}(p) - 0.477$

Rounding up and adding one digit of safety margin gives the recommended guard:

$g_{rec} = \lceil \log_{10}(p) \rceil + 2$

This is the same formula derived for the sequential Borwein 25 series, which is appropriate since the hybrid binary splitting accumulates the same category of rounding errors, structured as a tree rather than a sequential loop.

Comparison with Pure Integer Binary Splitting

The table below summarizes the guard-digit requirement and the arithmetic type used in each of the three binary-splitting implementations covered in this paper.

Series	p and q type	Rounding enters	Guard formula	g at p=10,000
Ramanujan	int_precision	Final formula only	constant g = 2	2
Chudnovsky	int_precision	Final formula only	constant g = 2	2
Borwein 25	float_precision	Every merge node	$\lceil\log_{10}(p)\rceil + 2$	6

The table makes it clear that the Borwein 25 binary splitting method achieves its asymptotic speedup of $O(M(p) \cdot \log^2(p))$ over the sequential $O(p \cdot M(p))$ cost, but it does not reduce the precision overhead. If a pure integer formulation were available, the guard could be reduced to the constant g = 2 of the Ramanujan and Chudnovsky cases. Since A and B involve √61, no such formulation exists, and the logarithmic guard is unavoidable for this series.

Appendix H

Guard Digit Requirement for the Borwein 50 Binary Splitting Implementation

The Borwein 50 binary splitting implementation has the same structural property as Borwein 25: the constants A and B involve √5 and nested square roots that cannot be represented as integers, so p and q must remain float_precision throughout the recursion. Rounding errors accumulate at every merge node in the recursion tree, and the error analysis follows the same path as for Borwein 25.

Error Accumulation in the Recursion Tree

The recursion tree for $k \approx p/49$ terms has k leaf nodes and $k - 1$ internal merge nodes. At each leaf, approximately 5 float_precision operations construct p and q, and at each internal merge node, 3 float_precision operations combine the subtrees:

$$p = p \cdot q_{right} + p_{right} \cdot r_{left}$$

$$q = q \cdot q_{right}$$

Each operation introduces a relative rounding error bounded by $0.5 \cdot 10^{-(p+g)}$. Summing over all nodes gives approximately 8k total float_precision operations, so the total relative error in π is bounded by:

$$|\delta\pi / \pi| \leq 4k \cdot 10^{-(p+g)}$$

Substituting $k \approx p/49$:

$$|\delta\pi / \pi| \leq (4p/49) \cdot 10^{-(p+g)} \approx (p/12) \cdot 10^{-(p+g)}$$

Deriving the Guard Digit Requirement

For the result to be correct to p decimal digits:

$$(p/12) \cdot 10^{-(p+g)} < 0.5 \cdot 10^{-p}$$

$$10^g > p/6$$

$$g > \log_{10}(p/6) = \log_{10}(p) - 0.778$$

Rounding up and adding one digit of safety margin gives:

$$g_{rec} = \lceil \log_{10}(p) \rceil + 2$$

This is the same formula as Borwein 25. The faster convergence of Borwein 50 ($k \approx p/49$ versus $k \approx p/24$) means the constant in the bound is slightly smaller, but both series round up to the same g_rec at every practical precision.

Comparison of Borwein 25 and Borwein 50 Binary Splitting

The table below confirms that both Borwein binary-splitting implementations require the same guard-digit formula at every precision, despite Borwein 50 using roughly half as many terms.

Target p	B25 k	B25 g_min	B50 k	B50 g_min	g_rec
100	6	2	4	2	4
1,000	43	3	22	3	5
10,000	418	4	206	4	6
100,000	4,168	5	2,042	5	7
1,000,000	41,668	6	20,410	6	8

As with Borwein 25, the binary splitting method delivers its asymptotic speedup over the sequential series but does not reduce the precision overhead. The guard digit cost is logarithmic in both cases because the irrational nature of the constants A and B forces floating-point arithmetic throughout the recursion.

Appendix I

Guard Digit Analysis for the Brent-Salamin Algorithm

Iteration Count

The Brent-Salamin algorithm has quadratic convergence, meaning each iteration doubles the number of correct digits. Starting from the initial conditions with approximately one correct digit, the number of iterations needed to reach p decimal digits is at most:

$$N \le \lceil \log_2(p) \rceil + 2$$

This grows very slowly: fewer than 10 iterations suffice for p = 100, and fewer than 23 iterations for p = 1,000,000.

Operations Per Iteration

Each iteration of the loop performs the following float_precision operations, excluding the two adjustExponent calls, which are exact bit shifts introducing no rounding:

$ak = a + b$ — one addition

$ab = a \cdot b$ — one multiplication

$bk = \sqrt{(ab)}$ — one square root

$asq = ak^2$ — one squaring

$ck = asq - ab$ — one subtraction

$sum \mathrel{-}= pow2 \cdot ck$ — one multiplication and one subtraction

This gives 7 float_precision operations per iteration, each contributing an absolute rounding error bounded by 0.5×10^{-n} where $n = p + g$ is the working precision.

Error Accumulation in the Partial Sum

The variable sum accumulates the weighted corrections over N iterations. Its value at convergence is approximately 0.457, bounded away from zero throughout. Each iteration contributes at most two rounding events to sum (one multiply and one subtract), giving a total absolute error:

$$\varepsilon_{sum} \le N \times 10^{-(p+g)}$$

Error Accumulation in asq

The variable asq = a^{k2} is computed in the final iteration and holds the converged value $a_\infty^2 \approx 0.718$. Errors in ak propagate from the addition and the AGM iteration. Over N iterations, the accumulated absolute error in asq is bounded conservatively by:

$$\varepsilon_{asq} \leq N \times 1.5 \times 10^{-(p+g)}$$

Total Error in π

The final formula is $\pi = 2 \cdot asq / sum$. The relative error in π receives contributions from both asq and sum:

$$|\delta\pi / \pi| \leq |\delta(asq) / asq| + |\delta(sum) / sum| + \varepsilon_{formula}$$

Substituting the bounds, with asq ≈ 0.718 and sum ≈ 0.457:

$$|\delta(asq) / asq| \leq N \times 2.1 \times 10^{-(p+g)}$$

$$|\delta(sum) / sum| \leq N \times 2.2 \times 10^{-(p+g)}$$

$$\varepsilon_{formula} \leq 1.0 \times 10^{-(p+g)} \quad \text{(2 ops in final formula)}$$

The combined bound is:

$$|\delta\pi / \pi| \leq 5 \times N \times 10^{-(p+g)}$$

Deriving the Guard Digit Requirement

For the result to be correct to p decimal digits, the relative error must satisfy $|\delta\pi/\pi| < 0.5 \times 10^{-p}$:

$$5 \times N \times 10^{-(p+g)} < 0.5 \times 10^{-p}$$

$$10^g > 10 \times N$$

$$g > \log_{10}(10N) = 1 + \log_{10}(N)$$

Substituting $N \leq \log_2(p) + 2$:

$$g > 1 + \log_{10}(\log_2(p) + 2)$$

This grows extremely slowly. Even at p = 1,000,000, the requirement is only g > 2.34, so g_min = 3 at all practical precisions.

Verification of the Implementation Formula

The implementation uses guard = 5 + round($\log_{10}$(digits)), which gives:

Target p	N (iterations)	g > 1+$\log_{10}$(N)	g_min	g_code = 5+round($\log_{10}$(p))
100	9	1.954	2	7
1,000	12	2.079	3	8
10,000	16	2.204	3	9

100,000	19	2.279	3	10
1,000,000	22	2.342	3	11

The implementation formula provides a comfortable margin of 4 to 8 guard digits beyond the rigorous minimum. This generous margin reflects the conservative nature of the bound: in practice, errors from different iterations tend to partially cancel rather than accumulate maximally, so the actual error is likely several times smaller than the bound suggests. The formula is correct and well-chosen, with the $\log_{10}$(digits) term tracking the slow growth of N with precision and the flat +5 providing the safety margin.