

Exploring the Binary Splitting Method
By Henrik Vestermark (hve@hvks.com)

Abstract

This paper explores the binary splitting method, a divide-and-conquer strategy for summing rational expressions in series expansions of special constants. Binary splitting reduces arithmetic and memory overhead when computing large-precision constants by reorganizing series into partial numerators and denominators, often combined with additional product terms. We illustrate how this method applies to classic constants such as π , the Catalan constant G , the lemniscate constant ϖ , and Apéry's constant $\zeta(3)$. Each constant's series is transformed into the standard two- or three-variable binary splitting form, with sample JavaScript implementations to demonstrate the practicality of these techniques in a web-based environment. The findings show that binary splitting consistently shortens execution time and intermediate number sizes compared to naive term-by-term summation, which is critical in high-precision computations. Additionally, an online tool integrating these algorithms allows users to test various splitting approaches interactively, emphasizing how straightforward code can achieve reliable results even for millions of digits.

Introduction

High-precision computation of mathematical constants has long been an area of interest for theoretical and practical reasons. Constants such as e , π , Catalan's constant, the Lemniscate constant, and Apéry's constant appear in numerous mathematics and applied science areas. While series expansions may represent each constant, straightforward summation often leads to large intermediate numerators and denominators, increasing arithmetic cost and storage requirements. The binary splitting method addresses these issues by splitting the summation range into halves and merging partial results tree-likely, reducing the computational load and limiting numerical growth.

This paper's motivation is twofold. First, it aims to clearly describe how to systematically convert well-known series expansions into binary splitting form. Second, it offers practical examples in JavaScript and an online tool at www.hvks.com/Numerical/webbinarysplitting.html so readers can see the performance and accuracy benefits in action. While the method is broadly applicable, we focus on specific constants that showcase different patterns of factorials, binomial coefficients, and powers. By examining how these factors are decomposed, we can illustrate the flexibility of binary splitting under varied scenarios.

Four key advantages exist to using a binary splitting method for the constants above.

In the Divide-and-Conquer Structure, instead of summing terms sequentially, the method partitions the summation interval $[0,n]$ into smaller segments, computes partial sums (and products) independently, and then merges them.

Benefits of using Smaller Intermediate Values. Because numerators and denominators are merged and balanced, the peak size of intermediate big integers is reduced compared to naive summation.

Benefits of Exact Rational Arithmetic. Until the final conversion to a floating-point or decimal representation, partial results are stored as ratio forms, limiting the introduction of floating-point rounding.

Parallelization is easy to create. Each segment can be processed in isolation, making the method suitable for multi-core and distributed environments. The final merge steps require only a few large multiplications and additions.

After a brief overview of the binary splitting algorithm and its historical context, we step through derivations that transform standard series expansions into two-variable or three-variable binary splitting forms. We highlight examples of well-known constants like e , π , the Catalan constant, Apéry's constant, and the Lemniscate constant using various methods developed over the last 30 years but also show some new methods discovered in 2023 and 2024.

Finally, we compare the performance and convergence properties of each approach. A numerical test implemented in JavaScript demonstrates how quickly we can reach double-precision accuracy and beyond. The scripts rely on JavaScript BigInt to handle large integer multiplications, which is essential once partial numerators and denominators exceed the range of 64-bit floating-point types.

By combining direct derivations with real code examples, this paper aims to serve as both a tutorial and a reference for anyone seeking to apply binary splitting in their high-precision arithmetic work. The companion web tool, hosted at the author's site, provides an interactive setting to tweak parameters, visualize error curves, and observe how each method scales with larger terms. See <https://www.hvks.com/Numerical/webbinarysplitting.html>

A special thanks goes to Alexander Yee's brilliant website at www.numberworld.org, which contains all the algorithms, series, and binary splitting formulas I used in this paper to create my web-based tool. It is worth exploring his website to learn more about this numerical algorithm.

Contents

Abstract.....	1
Introduction.....	1
What is Binary Splitting?.....	5
Basic Series Form.....	5
Two-Variable Recursion.....	5
Benefits of the Method.....	6
Typical Workflow for converting a series representation into a binary splitting method.....	6
Historical Background.....	7
Advantages and Disadvantages of the Binary Splitting Method.....	7
Different Forms of Binary Splitting.....	7
Two-Variable Approach.....	8
Deriving the Binary splitting method for e	8
Three-Variable Approach.....	9
Deriving the Binary splitting method for Ramanujan π	9
Hypergeometric Generalizations.....	10
Practical Applications and Examples.....	11
Error Analysis.....	11
Eulers constant e	12
The Constant π	14
Binary splitting of the Ramanujan infinite series.....	14
Source Binary splitting for Ramanujan π	15
Binary splitting of the Chudnovsky infinite series.....	16
Source Binary splitting for Chudnovsky π	16
Catalan's constant G	18
Lupas Binary Splitting method.....	18
Source Binary splitting for Lupas Catalan constant.....	19
Guillera Binary Splitting method.....	21
Source Binary splitting for Guillera 2008 Catalan constant.....	21
Source Binary splitting for Guillera 2019 Catalan constant.....	23
Pilehrood binary splitting method.....	24
Source Binary splitting for Pilehrood-short Catalan constant.....	25
Source Binary splitting for Pilehrood-long Catalan constant.....	27
Zuniga binary splitting method.....	28
Source Binary splitting for Zuniga Catalan constant.....	29
Comparison of the Catalan Methods.....	30
Apéry's constant $\zeta(3)$ (Zeta(3)).....	32

Amdeberhan-Zeilberger series	32
Source Binary splitting for Amdeberhan-Zeilberger Zeta(3)	33
Wedeniwski series	34
Source Binary splitting for Wedeniwski Zeta(3)	35
Zuniga series (v)	36
Source Binary splitting for Zuniga Zeta(3)	36
Zuniga series (vi)	38
Comparison of the Apéry's Methods	38
Lemniscate constant ϖ	39
Zuniga many series	39
Guillera series	42
Comparison of the Lemniscate methods	43
Conclusion	43
References	44

What is Binary Splitting?

The binary splitting method is a computational approach that splits a summation or product into parts and then recombines these parts to produce numerically stable results. It has been employed in various series expansions for constants such as π , e , the Catalan constant, and others, and it often allows faster convergence and higher precision than more straightforward summation methods.

It is especially useful in high-precision arithmetic where straightforward summation might cause large intermediate values and heavy computational overhead.

The binary splitting method is a divide-and-conquer technique for evaluating sums of rational functions at high precision. Instead of adding terms one at a time, each of which may involve large numerators and denominators, we split the summation range into two parts, compute each part independently (often in separate recursive calls), and then merge those parts in a way that controls the size of intermediate results.

Basic Series Form

Many series for mathematical constants, like e , can be expressed in the form:

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

Where $P(k)$ and $Q(k)$ are integer (or polynomial) expressions in k , a key observation is that if you gather the entire sum into a single fraction, you get:

$$x = \frac{P(0, n)}{Q(0, n)}$$

Where:

- $P(0, n)$ represents an accumulated numerator (or partial sum),
- $Q(0, n)$ represents an accumulated denominator (or partial product).

Those two large integers (or polynomials) can then be combined just once at the end to produce a precise rational approximation of x .

Two-Variable Recursion

To build up $P(0, n)$ and $Q(0, n)$ without simply multiplying and adding all terms in a single pass, the binary splitting method breaks down the index interval $[0, n]$ recursively:

1. Split $[0, n]$ into $[0, m]$ and $[m+1, n]$ around a midpoint m .
2. Recursively compute: $P(0, m)$, $Q(0, m)$ and $P(m+1, n)$, $Q(m+1, n)$.
3. Merge those partial results with a small number of big-integer (or big-polynomial) multiplications and additions.

A common two-variable merge step for subranges $[a,m]$ and $[m+1,b]$ can look like:

$$\begin{aligned} P(a,b) &= P(a,m) Q(m+1,b) + P(m+1,b), \\ Q(a,b) &= Q(a,m) Q(m+1,b) \end{aligned}$$

(The exact formulas can vary slightly depending on how you define $P(k)$ and $Q(k)$. By doing this recursively, you avoid forming the product of all $Q(k)$ values in one linear chain, which can be unnecessarily large. Instead, each partial product is merged only once per recursive step.

Benefits of the Method

This approach has several benefits. First, it reduces intermediate growth by merging numerators and denominators in a balanced tree-like fashion; intermediate products and sums tend to be smaller than if you multiplied everything in a naive, single pass.

You always have an Exact Rational Representation. Until the final division (if you convert to a floating point), the partial sums are stored as exact ratios, and no floating-point rounding error is introduced at intermediate stages.

It is very suitable for large-precision constants. Controlling arithmetic cost is critical if you aim for hundreds, thousands, or even billions of digits. Binary splitting keeps the number of large multiplications logarithmic in the number of terms.

It is very easy to parallelize because each half of the summation $[a,m]$ and $[m+1,b]$ is independent; you can compute them on different threads or machines and then merge them at the end.

Typical Workflow for converting a series representation into a binary splitting method

Rewrite the Series by starting with a known formula for your constant x (for instance, $\sum 1/k!$ for e , or a factorial-based series for π). Express each term in the form. $\sum_{k=1}^n P(k) \prod_{i=1}^k \frac{1}{Q(i)}$

Identify $P(k)$ and $Q(k)$ and make sure $P(k)$ and $Q(k)$ are integer (or polynomial) expressions that will let you construct partial products and sums efficiently.

Define the Recursion and choose how to split the interval and how to merge partial results. Write out the exact form for $P(a,b)$ and $Q(a,b)$.

Implement a Big Integer (or Big Polynomial) Library since standard floating-point types of 64 bits are insufficient. Use or implement data structures that can hold very large numbers or use a language that supports arbitrary precision integers directly.

Recursive Computation of $P(0,n)$ and $Q(0,n)$ via divide-and-conquer.

Do a final conversion at the end, converting $\frac{P(0,n)}{Q(0,n)}$ To the precision you desire (decimal, binary, etc.).

Historical Background

Research by Richard Brent in the 1970s and later contributions by the Chudnovsky brothers brought attention to binary splitting. Brent and others investigated algorithms for faster computation of constants, which led to insights about handling large integer arithmetic more effectively. The Chudnovsky brothers further developed methods for π that used the binary splitting idea, aiding in large-scale calculations that set records for decimal expansions of π .

During that time, researchers explored ways to minimize multiplication costs. Algorithms like Karatsuba, Toom-Cook, and later the Schönhage-Strassen and the Fürer or Harvey-van der Hoeven multiplication methods also became part of an ecosystem of techniques. Binary splitting fits nicely into this, ensuring the number of large multiplications is kept in check while preserving exact numerators and denominators up to very high precision.

Advantages and Disadvantages of the Binary Splitting Method

There are several advantages to using the binary splitting method.

Splitting ranges in two is very efficient with large integers. Intermediate products stay smaller than if one multiplies everything in one long chain. This can be a tremendous saving, especially for computations requiring millions or billions of digits of accuracy.

It always has an exact rational representation until the final step. Rounding error is not introduced at each partial sum; it is only introduced once, if at all, when the fraction is converted to a floating-point or decimal representation.

The Binary splitting method has a high parallel potential and naturally lends itself to divide-and-conquer strategies. Large computations can be split among multiple processors or threads, each working on a subrange. Final merging is straightforward, requiring only a few large multiplications and additions.

There are, of course, also some disadvantages.

Direct summation involves a simple loop, while binary splitting requires recursive structures and careful tracking of numerator and denominator functions. This leads to higher implementation complexity, which can be mitigated using recursive procedures.

You often need to rewrite a series of presentations into a form suitable for a binary splitting process. One must find a representation of the series where each term is expressed as a rational function $N(k)/D(k)$. In some cases, additional transformations or manipulations may be necessary to get it into a convenient form. However, sometimes, you can achieve optimization potential by rewriting it into a binary splitting form.

Lastly, there could be some disadvantages regarding memory use. While intermediate product sizes are smaller than naive methods, large computations still require substantial memory for storing partial results and big integers.

Different Forms of Binary Splitting

The most common one is the two and three-variable form of binary splitting.

Two-Variable Approach

The Binary Recursion algorithm for the two-variable splitting

The two-variable binary splitting algorithm is on the form:

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

Algorithm two variable form. Given $a < m < b$.

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

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

$$Q(a, b) = Q(a, m)Q(m, b)$$

Where:

$$P(b-1, b) = P(b)$$

$$Q(b-1, b) = Q(b)$$

Deriving the Binary splitting method for e

For example, we can derive the binary splitting method for Euler's e constant.

The two-variable binary splitting algorithm is on the form:

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

In addition, we have the Taylor series for e:

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

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

Noting $k!$ in product notation is: $\prod_{i=1}^k i$

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

We first align the start index to one and moving out the constant at $k=0$ yields:

$$e = 1 + \sum_{k=1}^{\infty} 1 \prod_{i=1}^k \frac{1}{i}$$

And simplify.

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

Now identify the $P(k)$ and $Q(k)$ you get:

$$\begin{aligned} P(k) &= 1 \\ Q(k) &= k \end{aligned}$$

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

$$e = 1 + \frac{P(0,n)}{Q(0,n)} = \frac{P(0,n) + Q(0,n)}{Q(0,n)}$$

Which is the Binary Splitting algorithm for e .

Three-Variable Approach

Sometimes, a single fraction is not enough. For instance, if each term in the series involves factorials or powers in a more elaborate combination, you might break the computation into three sequences instead of just the numerator and denominator. You might define:

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

Each captures part of the partial product or sum. The recursive step then merges all three, similar to the two-variable approach but with more terms in each merge. We can use the Ramanujan π series as an example of how to derive it from the series form.

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 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:

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

Simplify:

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

$$\begin{aligned} \frac{1}{\pi} &= \frac{1103\sqrt{2}}{9801} + \frac{2\sqrt{2}}{9801} \sum_{k=1}^n \frac{(4k-1)(2k-1)(4k-3)(1103 + 26390k)}{(4k-1)(2k-1)(4k-3)} \prod_{i=1}^k \frac{(4i-1)(2i-1)(4i-3)}{3073907232i^3} \\ &= \frac{1103\sqrt{2}}{9801} + \frac{2\sqrt{2}}{9801} \frac{P(0,n)}{Q(0,n)} \end{aligned}$$

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,n) and Q(0,n) yields:

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

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

Which is the Binary Splitting algorithm for Ramanujan π .

Hypergeometric Generalizations

In more advanced cases, especially when dealing with hypergeometric series, you can have multiple Pochhammer symbols in the numerator and denominator. You may end up tracking several sequences or using hypergeometric-based recursion relations. Once the hypergeometric representation is known, a general solution to many series expansions can be found.

Practical Applications and Examples

In practice, it has been shown that calculating constants like e , π , the Catalan constant, the Lemniscate constant, $\zeta(3)$, $\zeta(5)$, $\log(2)$, $\log(3)$, $\log(5)$, and others is faster than using the series summation for these constants.

Error Analysis

When summing a series with binary splitting, the remainder after n terms is usually about the size of the next term (or smaller). In many formulas for constants, the terms tend to shrink rapidly, often geometrically. If the ratio between consecutive terms is $q \ll 1$, the tail from $n+1$ to infinity is of the order of $T_{n+1}/(1-q)$. With clever series selections (like the Chudnovsky approach for π), the convergence can be extraordinarily fast, making high-precision calculations viable.

Summing up

The binary splitting method is a valuable tool in high-precision arithmetic. While a simple loop might be enough for modest precision, once you aim for millions or billions of digits, binary splitting offers performance and memory usage advantages. Whether computing classical constants like e and π or exploring less common values such as Catalan's and Lemniscate constants, this approach provides a systematic pathway to rational partial sums and precise final results.

Understanding how to tailor the numerator and denominator functions (or multiple sequences in more advanced settings) gives one a significant edge in large-scale computations. Researchers and engineers working on expanded decimal expansions have relied on binary splitting for decades, and it remains a fundamental strategy for any serious high-precision work today.

This paper will use JavaScript to show the practical code for the binary splitting method for many of these constants, as presented on the author's web page at www.hvks.com/Numerical/binarysplitting.html. This tool allows you to experiment with different splitting and constants to analyze the convergence rate and error in the computation.

Eulers constant e

Euler constant e is defined as:

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

and is approximately $e \sim 2.71828182845904523536$

We can transform this into a binary splitting method by noticing that it is a hyperdescent type that evaluates the following series:

$$e = 1 + \sum_{k=1}^n P(k) \prod_{i=1}^k \frac{1}{Q(i)} = 1 + \frac{P(0,n)}{Q(0,n)} + O\left(\frac{1}{n!}\right)$$

Where $P(0,n)$, $Q(0,n)$ is a function of n that returns an integer. The algorithm for the binary splitting method for e can be expressed as a recursion over the splitting interval (a,b):

Algorithm: Binary splitting method for e(a,b)

```

set m =  $\frac{a+b}{2}$  integer division
P(a,b)=P(a,m)Q(m,b)+P(m,b)
Q(a,b)=Q(a,m)Q(m,b)
And:
P(b-1,b)=1
Q(b-1,b)=b

```

You continue this recursive breakdown until $a+1=b$ and let the formula reverse bottom up. In the end, you find e by (see appendix):

$$e = \frac{P(0,n) + Q(0,n)}{Q(0,n)} + O\left(\frac{1}{n!}\right)$$

The following JavaScript source code implements this:

Source Binary splitting for e

```

function binarySplitE(a, b) {
  // Base case: interval [a..b] of length 1
  if (b - a === 1)
    return { p: 1n, q: BigInt(b) };
  const mid = Math.floor((a + b) / 2);
  // Recurse on [a..mid) and [mid..b)
  const left = binarySplitE(a, mid);
  const right = binarySplitE(mid, b);
  // Merge results
  const p = left.p * right.q + right.p;
  const q = left.q * right.q;
  return { p, q };
}

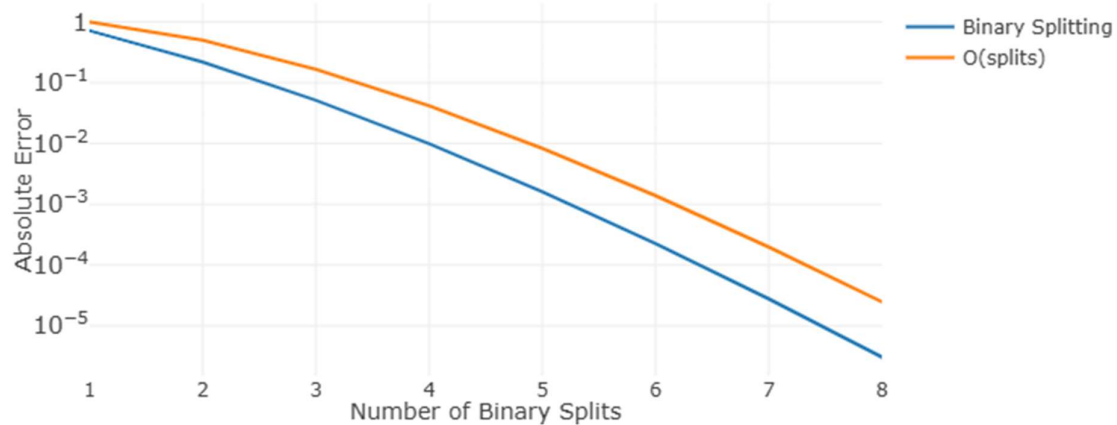
```

Example:

```

const result=binarySplitE(0,5); // Split it 5 times
const e=Number(result.p+result.q)/Number(result.q);

```

Binary Splitting Error of e Binary Splitting of e for:

Value: 2.716666666666667

Error: -1.62e-3

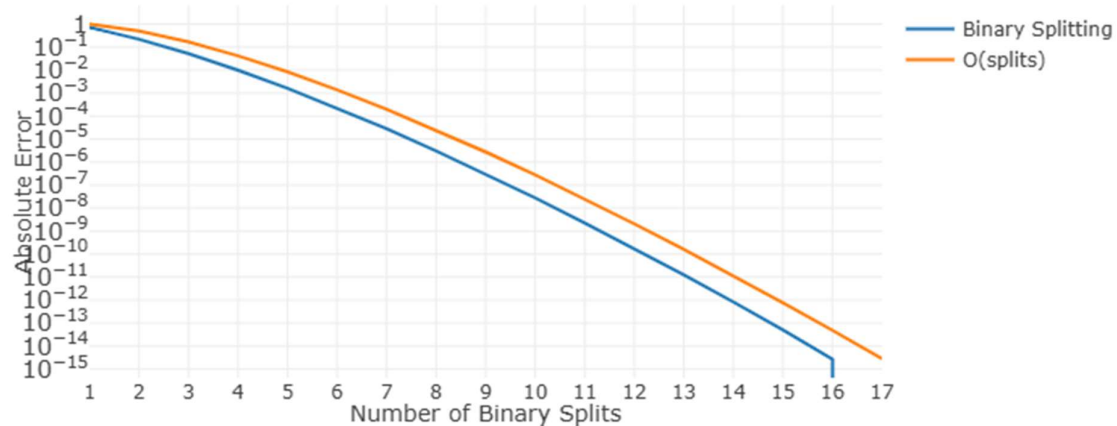
O(5) : 8.33e-3

Requested Splitting Terms: 5

Actual Splitting Terms: 5

The blue curve is the absolute error as a function of the number of splitting terms. The orange curve is the theoretical $O(n)$ as a function of the splits.

Notice that we take advantage of the BigInt type (arbitrary precision) in JavaScript since $P(0,n)$ and $Q(0,n)$ can be large. After five splits, the result was 2.71666, with an absolute error of 1.62E-3. We can increase the number of splits to, e.g., 17 and get a result with an error of $\sim 2E-15$. See below.

Binary Splitting Error of e 

With 17 splits, the final $P(0,17)$ and $Q(0,17)$ are 611171244308690 and 355687428096000, respectively. This underscores the need to implement the binary splitting algorithm using JavaScript BigInt types.

The Constant π

The binary splitting method has proven very efficient for evaluating π , particularly when paired with series expansions discovered by Ramanujan and the Chudnovsky brothers. Ramanujan introduced multiple rapidly converging formulas for π that rely on factorial-like terms and algebraic manipulations, which can be reorganized into a structure suitable for binary splitting. By breaking the infinite series into smaller intervals and merging partial sums and products, these formulas achieve remarkable precision with fewer iterations than more traditional approaches.

The Chudnovsky formula has been especially influential, providing one of the fastest-known series for computing π . When paired with binary splitting, it has powered groundbreaking calculations, culminating in the current record of 100 trillion digits. This technique works by minimizing the accumulation of round-off errors, allowing each partial sum to be computed and combined with high accuracy. The ability to handle large-scale parallel computations has further advanced its appeal, making it the preferred method for those aiming to push the limits of π 's computed digits.

For computing π , we can transform the infinite series of Ramanujan and Chudnovsky into a binary splitting method. The Chudnovsky binary splitting method currently holds the record for computing π with 100 trillion digits.

Binary splitting of the Ramanujan infinite series

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

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

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

Given the first k terms of the series, you get (see [3]):

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

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

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:
Algorithm for Ramanujan π .

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

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

$$Q(a,b) = Q(a,m)Q(m,b)$$

$$R(a,b) = R(a,m)R(m,b)$$

Where:

$$P(b-1,b)=(1103+26390b)(2b-1)(4b-3)(4b-1)$$

$$Q(b-1,b)=3073907232b^3$$

$$R(b-1,b)=(2b-1)(4b-3)(4b-1)$$

Source Binary splitting for Ramanujan π

```
function binarySplitRamanujan(a, b) {
  // Base case: interval [a..b) of length 1
  if (b - a === 1) {
    const r = BigInt((2 * b - 1) * (4 * b - 3) * (4 * b - 1));
    const p = BigInt(1103 + 26390 * b) * r;
    const q = BigInt(b ** 3) * 3073907232n;
    return { p, q, r };
  }

  const mid = Math.floor((a + b) / 2);
  // Recurse on [a..mid) and [mid..b)
  const left = binarySplitRamanujan(a, mid);
  const right = binarySplitRamanujan(mid, b);

  // Reconstruct interval [a..b] and return p, q & r
  const p = left.p * right.q + right.p * left.r;
  const q = left.q * right.q;
  const r = left.r * right.r;
  return { p, q, r };
}
```

Example:

```
const result=binarySplitRamanujan(0,2); // Split it 2 times
const pi = Number(9801n * result.q) / Number(result.p + 1103n * result.q) /
Math.sqrt(8);
```

Binary Splitting Error of pi-Ramanujan



Note the orange curve overlaps the blue curve.

Binary Splitting of pi-Ramanujan for:

Value: 3.1415926535897936

Error: 4.44e-16

O(1) : 1.04e-8

Requested Splitting Terms: 1

Actual Splitting Terms: 1

Notice that we already have an error of 1e-16 after one term. Normally, Ramanujan π delivers approx. eight decimal digits per term. For an IEEE754 64-bit float, you only need a maximum of two splits to get an accurate answer.

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.

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

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

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

Given the first k terms of the series, you get (see [3]):

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

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

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:

Algorithm for Chudnovsky π .

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

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

$$Q(a,b) = Q(a,m)Q(m,b)$$

$$R(a,b) = R(a,m)R(m,b)$$

Where:

$$P(b-1,b) = (13591409 + 545140134b)(2b-1)(6b-5)(6b-1)(-1)^b$$

$$Q(b-1,b) = 10939058860032000b^3$$

$$R(b-1,b) = (2b-1)(6b-5)(6b-1)$$

Source Binary splitting for Chudnovsky π

```
function binarySplitChudnovsky(a, b) {
  // Base case: interval [a..b) of length 1
  if (b - a === 1) {
    const r = BigInt((2 * b - 1) * (6 * b - 5) * (6 * b - 1));
    const p = BigInt(13591409 + 545140134 * b) * r *
    BigInt(Math.pow(-1, b));
    const q = BigInt(b ** 3) * 10939058860032000n;
    return { p, q, r };
  }
}
```

```

    }

    const mid = Math.floor((a + b) / 2);

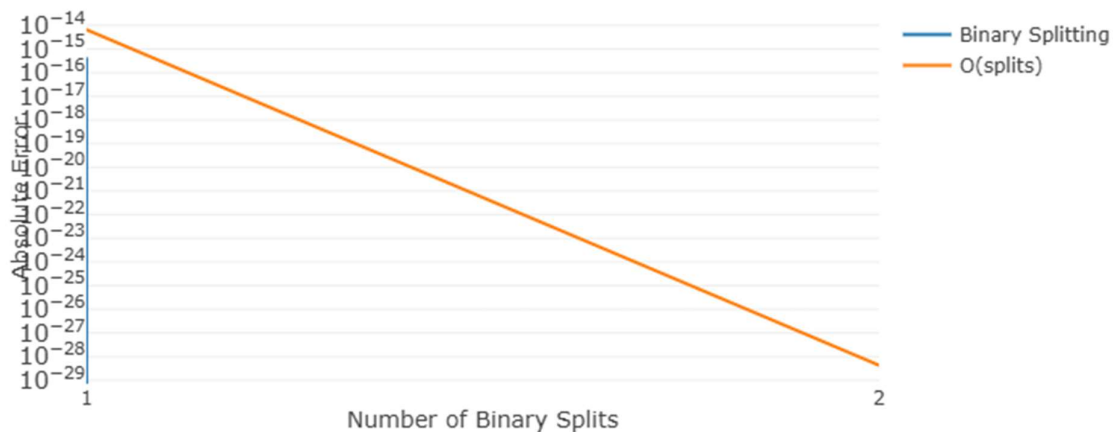
    // Recursively compute left and right intervals
    const left = binarySplitChudnovsky(a, mid);
    const right = binarySplitChudnovsky(mid, b);

    // Merge results
    const p = left.p * right.q + right.p * left.r;
    const q = left.q * right.q;
    const r = left.r * right.r;
    return { p, q, r };
  }

  // Example:
  const result = binarySplitChudnovsky(0, 1);
  const piValue = Number(4270934400n * result.q) /
    Number(result.p + 13591409n * result.q) /
    Math.sqrt(10005);

```

Binary Splitting Error of pi-Chudnovsky



Binary Splitting of pi-Chudnovsky for:

```

Value: 3.1415926535897936
Error: 4.44e-16
O(1) : 6.58e-15
Requested Splitting Terms: 1
Actual Splitting Terms: 1

```

After one split, the error is 4e-16. Chudnovsky generated approx. 14 correct decimals per split.

Catalan's constant G

The Catalan constant G is defined as:

$$G = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)^2}$$

The Catalan constant is $\sim 0.9159655941772\dots$

This series also converges slowly. However, there are several alternative Binary Splitting methods to consider (ref. [3]).

- Lucas(2000)
- Guillera (2008)
- Guillera (2019)
- Pilehrood (2010)
- Zuniga (2023)

Lucas (2000) examined how binary splitting could be combined with particular series expansions to handle certain special functions. This work highlighted how breaking large sums into smaller intervals and carefully managing partial products can significantly reduce the necessary computational steps. By systematically merging intermediate outcomes, Lucas provided a framework that reduced round-off errors and the time required to attain a specified number of correct digits.

Guillera (2008) built on these ideas by unveiling a new series that benefited from the efficiency of binary splitting. Guillera's approach often produced rapid convergence for constants of interest, and it demonstrated how the method could exploit algebraic factorizations that may not be obvious at first glance. In a later publication, Guillera (2019) refined these techniques by introducing a transformed series that further simplified intermediate products. This made the method more practical for modern high-precision arithmetic libraries and shed light on creative ways to manipulate series for faster convergence.

Pilehrood (2010) studied binary splitting from the perspective of double series and convolution structures. This angle expanded possibilities for summing constancy expansions involving multiple summation indices. By focusing on how these multiple indices interact, the Pilehrood work revealed efficient factorization strategies, keeping the core principle of partial-product pairing intact while broadening the range of constants that can be computed. Zuniga (2023) extended the method through alternative representations well-suited to parallel computation, which is valuable in today's multiprocessor environments. The focus here was on creating formulas that minimize dependency between partial sums, allowing computation to be distributed among multiple processing units. This can deliver even faster results for large-scale, high-precision calculations while remaining faithful to the core principles that make binary splitting appealing.

Lupas Binary Splitting method

Lupas series for the Catalan constant is:

$$G = \frac{1}{64} \sum_{k=1}^{\infty} \frac{(-1)^{k-1} 2^{8k} (40k^2 - 24k + 3) (2k)!^3 k!^2}{k^3 (2k-1) (4k)!^2}$$

As we have seen many times before, we can transform this series into a binary Splitting method using the below algorithm:

Algorithm: Binary splitting method for Catalan – Lupas (2000)

```

set  $m = \frac{a+b}{2}$  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)

```

And:

```

P(b-1,b)=(32(2b-1)b3)(40b2+56b+19)(-1)b
Q(b-1,b)=(4b+1)2(4b+3)2
R(b-1,b)=32(2b-1)b3

```

You continue this recursive breakdown until $a+1=b$ and let the formula reverse bottom up. In the end, you find G by:

$$G = \frac{P(0, n) + 19Q(0, n)}{18Q(0, n)} + O(4^{-n})$$

For n terms, the error is $O(4^{-n})$ and for a given precision, P you need:

$$n = \left\lceil P \frac{\ln(10)}{\ln(4)} \right\rceil$$

In [3], they found the linearly convergent cost to be ~ 11.5 , which is not as fast as the Guillera or Pilehrood methods.

Source Binary splitting for Lupas Catalan constant

```

function binarySplitLupas(a, b) {
  // Base case: interval [a..b) of length 1
  if (b - a === 1) {
    const bb3=BigInt(b*b*b);
    const b4p1 = 4 * b + 1;
    const b4p3 = b4p1 + 2;

    // Compute q
    const q = BigInt(b4p1 * b4p1) * BigInt(b4p3 * b4p3); //
(4b+1)^2(4b+3)^2
    // Compute r
    const r = bb3 * BigInt(32 * (2 * b - 1)); // 32*(2b-1)*b^3
    // Compute p
    const p = BigInt((40 * b+56)*b+19) * r * BigInt(Math.pow(-1, b));
// (32*(2b-1)*b^3)(40b^2+56b+19)(-1)^b
    return { p, q, r };
  }

  const mid = Math.floor((a + b) / 2);
  // Recursively compute left and right intervals
  const left = binarySplitLupas(a, mid);
  const right = binarySplitLupas(mid, b);
  // Merge results
  const p = left.p * right.q + right.p * left.r;
}

```

```

const q = left.q * right.q;
const r = left.r * right.r;
return { p, q, r };
}

```

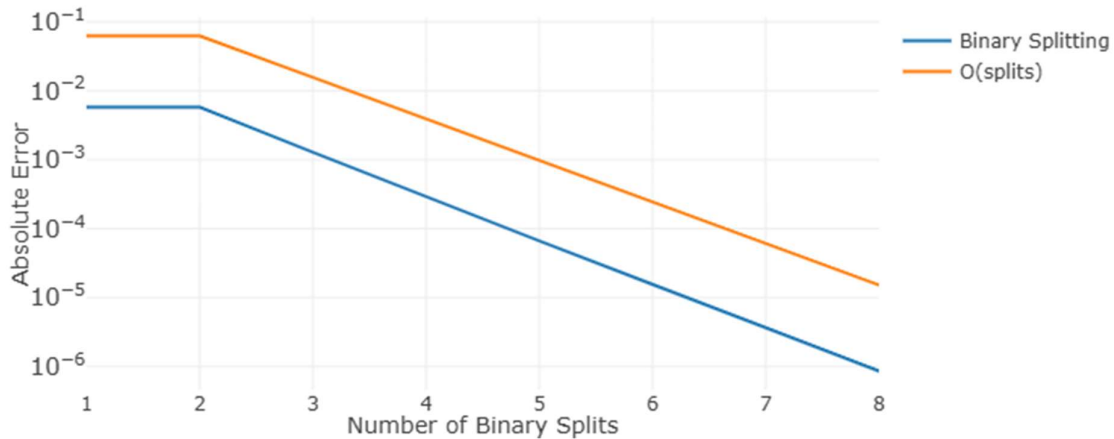
```
// Example:
```

```

const result = binarySplitLupas(0, 5);
const gValue = Number(result.p + 19n * result.q) / Number(18n * result.q);

```

Binary Splitting Error of Catalan-Lupas



Binary Splitting of Catalan-Lupas for:

Value: 0.9158990309461097

Error: -6.66e-5

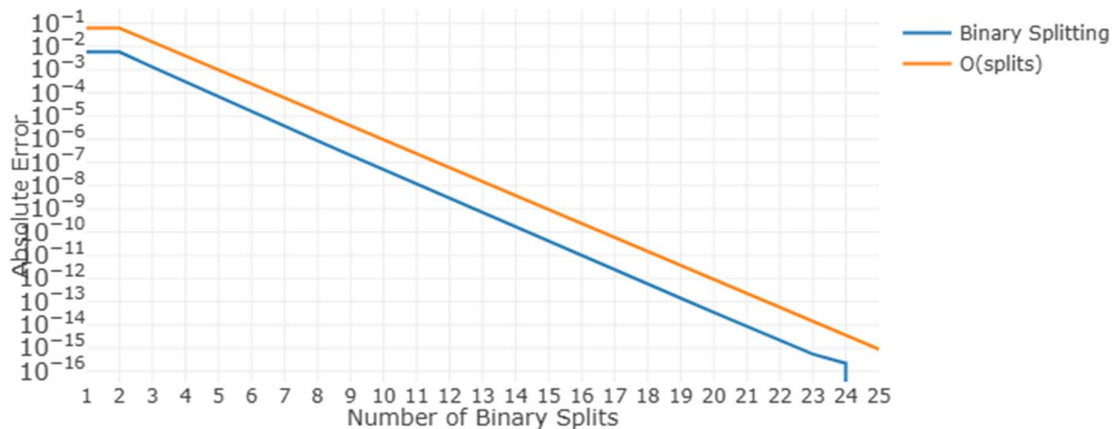
O(5) : 9.77e-4

Requested Splitting Terms: 5

Actual Splitting Terms: 5

As can be seen from the plot, the number of splits needed is a lot higher than for the binary splitting of π . After five splits, our accuracy is around $\sim 6e-5$

Binary Splitting Error of Catalan-Lupas



Binary Splitting of Catalan-Lupas for:

Value: 0.915965594177219

Error: 0.00e+0

O(25) : 8.88e-16

Requested Splitting Terms: auto

Actual Splitting Terms: 25

We need 25 times splits to get the highest accuracy using 64-bit floats.

Guillera Binary Splitting method

Guillera published two methods in 2008 and 2019. The first method used:

$$G = \frac{1}{2} \sum_{k=0}^{\infty} \frac{(-8)^k (3k + 2)}{(2k + 1)^3 \binom{2k}{k}^3}$$

Converting into a binary splitting method, they found in [3] that the linearly convergent cost is ~ 11.5 , around the same as for the Lupas binary splitting method. In [3] they rewrote the formula to:

$$G = \frac{1}{2} \sum_{k=0}^{\infty} \frac{(-8)^k (3k + 2) k!^6}{(2k + 1)!^3}$$

And archive a linearly convergent cost of ~ 5.7 , making it faster than the Lupas method.

Algorithm: Binary splitting method for Catalan – Guillera (2008)

```
set 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)
```

And:

```
P(b-1,b)= b3(3b+2)
Q(b-1,b)=- (2b+1)3(2b-1)3
R(b-1,b)=b3(2b+1)3
```

You continue this recursive breakdown until $a+1=b$ and let the formula reverse bottom up.

In the end, you find G by:

$$G = 1 + \frac{1}{2} \frac{P(0, n)}{Q(0, n)} + O(8^{-n})$$

For n terms, the error is $O(8^{-n})$ And for a given precision, P you need:

$$n = \left\lceil P \frac{\ln(10)}{\ln(8)} \right\rceil$$

Source Binary splitting for Guillera 2008 Catalan constant

```
function binarySplitGuillera(a, b) {
  // Base case: interval [a..b) of length 1
  if (b - a === 1) {
    const b3 = b * b * b;
    // Build p
    const p = BigInt(b3) * BigInt(3 * b + 2); // b^3(3b+2)
```

```

    // Build q
    const q = -BigInt(Math.pow(2 * b + 1, 3)); // -(2b+1)^3
    // Build r
    const r = BigInt(b3); // b^3
    return { p, q, r };
  }

  const mid = Math.floor((a + b) / 2);

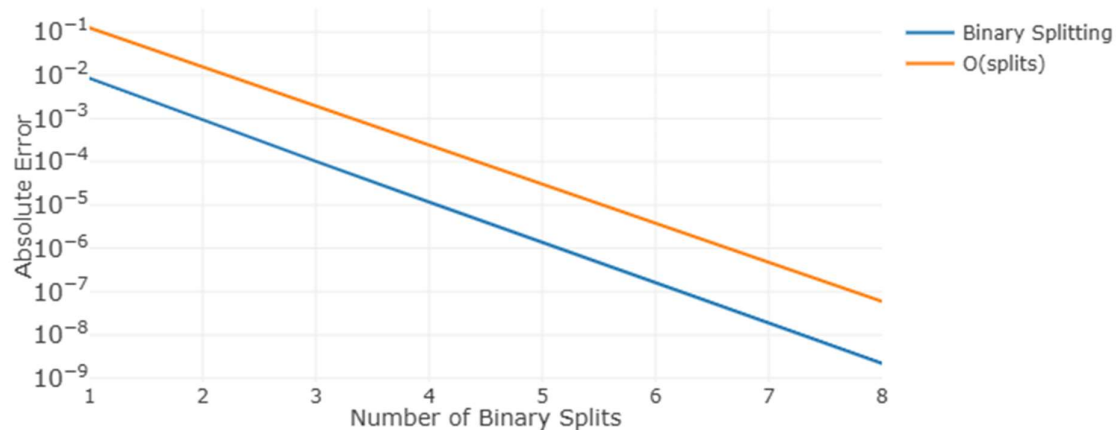
  // Recursively compute left and right intervals
  const left = binarySplitGuillera(a, mid);
  const right = binarySplitGuillera(mid, b);

  // Merge results
  const p = left.p * right.q + right.p * left.r;
  const q = left.q * right.q;
  const r = left.r * right.r;
  return { p, q, r };
}

// Example:
const result = binarySplitGuillera(0, 5);
const gValue = 1 + 0.5 * Number(result.p) / Number(result.q);

```

Binary Splitting Error of Catalan-Guillera2008



Binary Splitting of Catalan-Guillera2008 for:

```

Value: 0.915964239834857
Error: -1.35e-6
O(5) : 3.05e-5
Requested Splitting Terms: 5
Actual Splitting Terms: 5

```

If you select auto for the splitting. It requires 17 Splits to reach an accuracy with an error of less than $1e-16$.

In 2019, Guillera published another formula with a higher convergence rate. The formula looks intimidating at first glance:

$$G = -\frac{1}{1024} \sum_{k=1}^{\infty} \frac{(-4096)^k (45136k^4 - 57184k^3 + 21240k^2 - 3160k + 165)}{k^3(2k-1)^3} \left(\frac{(2k)!^6 (3k)!^3}{k!^3 (6k)!^3} \right)$$

But has a linearly convergent cost of only ~ 4.2 , which is lower than Guillera's formula from 2008.

Algorithm: Binary splitting method for Catalan – Guillera (2019)

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

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

$Q(a,b) = Q(a,m)Q(m,b)$

$R(a,b) = R(a,m)R(m,b)$

And:

$P(b-1,b) = 45136b^4 - 57184b^3 + 21240b^2 - 3160b + 165$

$Q(b-1,b) = -27(6b-1)^3(6b-5)^3$

$R(b-1,b) = 512b^3(2b-1)^3$

You continue this recursive breakdown until $a+1=b$ and let the formula reverse bottom up.

In the end, you find G by:

$$G = -\frac{1}{2} \frac{P(0,n)}{Q(0,n)} + O\left(\left(\frac{19683}{64}\right)^{-n}\right)$$

For n terms, the error is $O\left(\left(\frac{19683}{64}\right)^{-n}\right)$ and for a given precision, P you need:

$$n = \left\lceil P \frac{\ln(10)}{\ln\left(\frac{19683}{64}\right)} \right\rceil$$

Source [Binary splitting for Guillera 2019 Catalan constant](#)

```
function binarySplitGuillera(a, b) {
  // Base case: interval [a..b] of length 1
  if (b - a === 1) {
    const bb = BigInt(b);
    // Build p
    const p = (
      ((45136n * bb - 57184n) * bb + 21240n) * bb - 3160n
    ) * bb + 165n;
    // Build q
    const q = -27n
      * BigInt(Math.pow(6 * b - 1, 3))
      * BigInt(Math.pow(6 * b - 5, 3));
    // Build r
    const r = 512n
      * BigInt(b * b * b)
      * BigInt(Math.pow(2 * b - 1, 3));
    return { p, q, r };
  }
}
```

```

const mid = Math.floor((a + b) / 2);

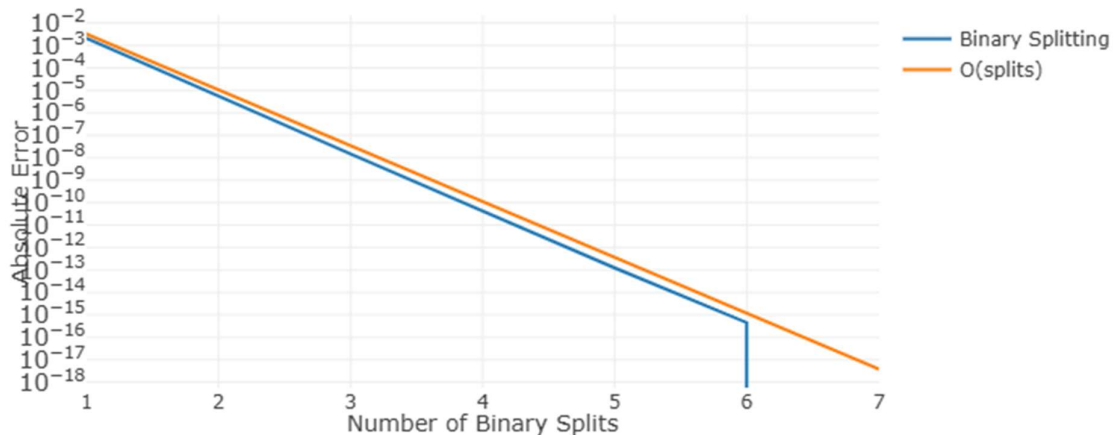
// Recursively compute left and right intervals
const left = binarySplitGuillera(a, mid);
const right = binarySplitGuillera(mid, b);

// Merge results
const p = left.p * right.q + right.p * left.r;
const q = left.q * right.q;
const r = left.r * right.r;
return { p, q, r };
}

// Example:
const result = binarySplitGuillera(0, 5);
const gValue = -Number(result.p) / Number(2n * result.q);

```

Binary Splitting Error of Catalan-Guillera2019



Binary Splitting of Catalan-Guillera2019 for:
 Value: 0.9159655941773428
 Error: 1.24e-13
 O(5) : 3.63e-13
 Requested Splitting Terms: 5
 Actual Splitting Terms: 5

The Guillera 2019 method converges much faster than the 2008 version. After five terms, the error is $\sim 1e-13$. If you choose auto-splitting, the most accurate solution requires seven splits.

Pilehrood binary splitting method

Pilehrood published two formulas in 2010. The short and long formula.

$$G = \frac{1}{64} \sum_{k=1}^{\infty} \frac{256^k (580k^2 - 184k + 15)}{k^3 (2k-1) \binom{6k}{3k} \binom{6k}{4k} \binom{4k}{2k}}$$

When applying the binary splitting method, you get a linearly convergent cost of only ~ 3.1 , which is the lowest of all the Catalan binary splitting methods so far.

Algorithm: Binary splitting method for Catalan – Pilehrood (2010-short)

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

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

$$Q(a,b)=Q(a,m)Q(m,b)$$

$$R(a,b)=R(a,m)R(m,b)$$

And:

$$P(b-1,b)=580b^2-184b+15$$

$$Q(b-1,b)=9(6b-1)^2(6b-5)^2$$

$$R(b-1,b)=32b^3(2b-1)$$

You continue this recursive breakdown until $a+1=b$ and let the formula reverse bottom up.

In the end, you find G by:

$$G = \frac{1}{2} \frac{P(0,n)}{Q(0,n)} + O\left(\left(\frac{729}{4}\right)^{-n}\right)$$

For n terms, the error is $O\left(\left(\frac{729}{4}\right)^{-n}\right)$ and for a given precision, P you need:

$$n = \left\lceil P \frac{\ln(10)}{\ln\left(\frac{729}{4}\right)} \right\rceil$$

Source Binary splitting for Pilehrood-short Catalan constant

```
function binarySplitPilehrood(a, b) {
  // Base case: interval [a..b) of length 1
  if (b - a === 1) {
    // Build p
    const p = BigInt((580 * b - 184) * b + 15);

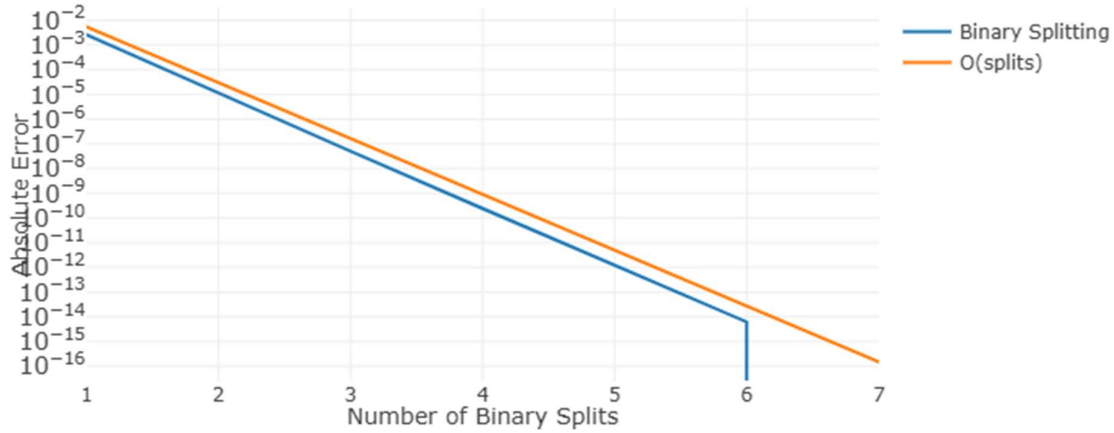
    // Build q
    const q = 9n
      * BigInt(Math.pow(6 * b - 1, 2))
      * BigInt(Math.pow(6 * b - 5, 2));

    // Build r
    const r = 32n
      * BigInt(b * b * b)
      * BigInt(2 * b - 1);
    return { p, q, r };
  }

  const mid = Math.floor((a + b) / 2);
  // Recursively compute left and right intervals
  const left = binarySplitPilehrood(a, mid);
  const right = binarySplitPilehrood(mid, b);
  // Merge results
  const p = left.p * right.q + right.p * left.r;
  const q = left.q * right.q;
  const r = left.r * right.r;
  return { p, q, r };
}
```

```
// Example:
const result = binarySplitPilehrood(0, 5);
const gValue = Number(result.p) / Number(2n * result.q);
```

Binary Splitting Error of Catalan-PilehroodShort



Binary Splitting of Catalan-PilehroodShort for:

Value: 0.9159655941760203

Error: -1.20e-12

O(5) : 4.97e-12

Requested Splitting Terms: 5

Actual Splitting Terms: 5

Pilehrood also has a long version formula:

$$G = -\frac{1}{64} \sum_{k=1}^{\infty} \frac{(-256)^k (419840k^6 - 915456k^5 + 782848k^4 - 332800k^3 + 73256k^2 - 7800k + 315)}{k^3 (2k-1)(4k-1)^2 (4k-3)^2 \binom{8k}{4k}^2 \binom{2k}{k}}$$

Which have a linearly convergent cost of ~4.6, which is at little higher than the Pilehrood short version.

Algorithm: Binary splitting method for Catalan – Pilehrood (2010-long)

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

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

$Q(a,b) = Q(a,m)Q(m,b)$

$R(a,b) = R(a,m)R(m,b)$

And:

$P(b-1,b) = (-1)^b (419840b^6 - 915456b^5 + 782848b^4 - 332800b^3 + 73256b^2 - 7800b + 315)$

$Q(b-1,b) = (8b-1)^2 (8b-3)^2 (8b-5)^2 (8b-7)^2$

$R(b-1,b) = 32b^3 (2b-1)(4b-1)^2 (4b-3)^2$

You continue this recursive breakdown until $a+1=b$ and let the formula reverse bottom up.

In the end, you find G by:

$$G = -\frac{1}{2} \frac{P(0, n)}{Q(0, n)} + O(1024^{-n})$$

For n terms, the error is $O(1024^{-n})$ And for a given precision, P you need:

$$n = \left\lceil P \frac{\ln(10)}{\ln(1024)} \right\rceil$$

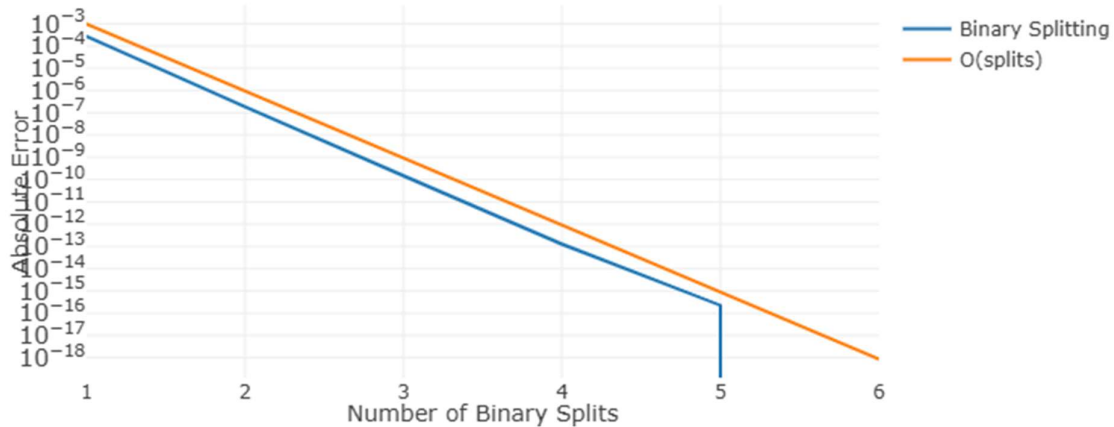
Source Binary splitting for Pilehrood-long Catalan constant

```
function binarySplitPilehrood(a, b) {
  // Base case: interval [a..b) of length 1
  if (b - a === 1)
    // Single term
    return (function () {
      const bb = BigInt(b)
      // p
      const p = BigInt(Math.pow(-1, b))
      * BigInt(
        (((((419840n * bb - 915456n) * bb + 782848n)
          * bb - 332800n) * bb + 73256n) * bb - 7800n) * bb +
315n
      )
      // q
      const q = BigInt(Math.pow(8 * b - 1, 2))
      * BigInt(Math.pow(8 * b - 3, 2))
      * BigInt(Math.pow(8 * b - 5, 2))
      * BigInt(Math.pow(8 * b - 7, 2))
      // r
      const r = 32n
      * BigInt(b * b * b)
      * BigInt(
        (2 * b - 1)
        * Math.pow(4 * b - 1, 2)
      )
      * BigInt(Math.pow(4 * b - 3, 2))
      return { p, q, r }
    })()

  const mid = Math.floor((a + b) / 2)
  // Recursively compute left and right intervals
  const left = binarySplitPilehrood(a, mid)
  const right = binarySplitPilehrood(mid, b)
  // Merge results
  const p = left.p * right.q + right.p * left.r
  const q = left.q * right.q
  const r = left.r * right.r
  return { p, q, r }
}

// Example:
const result = binarySplitPilehrood(0, terms)
const gValue = -Number(result.p) / Number(2n * result.q)
```

Binary Splitting Error of Catalan-PilehroodLong



Binary Splitting of Catalan-PilehroodLong for:
 Value: 0.9159655941772192
 Error: 2.22e-16
 O(5) : 8.88e-16
 Requested Splitting Terms: 5
 Actual Splitting Terms: 5

Zuniga binary splitting method

A freshly new method from 2023 by Zuniga. [10]

$$G = \frac{1}{768} \sum_{k=1}^{\infty} \frac{(-4096)^k (-43203456k^6 + 92809152k^5 - 76613904k^4 + 30494304k^3 - 6004944k^2 + 536620k - 17325)}{k^3(2k-1)(3k-1)(3k-2)(6k-1)(6k-5) \binom{5k}{k} \binom{10k}{5k} \binom{12k}{6k}}$$

Which have a linearly convergent cost of ~3.4, which is lower than the Pilehrood versions.

Algorithm: Binary splitting method for Catalan – Zuniga 2023.

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

And:

$$P(b-1,b) = 43203456b^6 - 92809152b^5 + 76613904b^4 - 30494304b^3 + 6004944b^2 - 536620b + 17325$$

$$Q(b-1,b) = 5(10b-9)(10b-7)(10b-3)(12b-11)(12b-7)(12b-1)$$

$$R(b-1,b) = -128b^3(2b-1)(3b-2)(3b-1)(6b-5)(6b-1)$$

You continue this recursive breakdown until $a+1=b$ and let the formula reverse bottom up.

In the end, you find G by:

$$G = \frac{1}{6} \frac{P(0,n)}{Q(0,n)} + O(12500^{-n})$$

For n terms, the error is $O(12500^{-n})$ And for a given precision, P you need:

$$n = \left\lceil P \frac{\ln(10)}{\ln(12500)} \right\rceil$$

Source Binary splitting for Zuniga Catalan constant

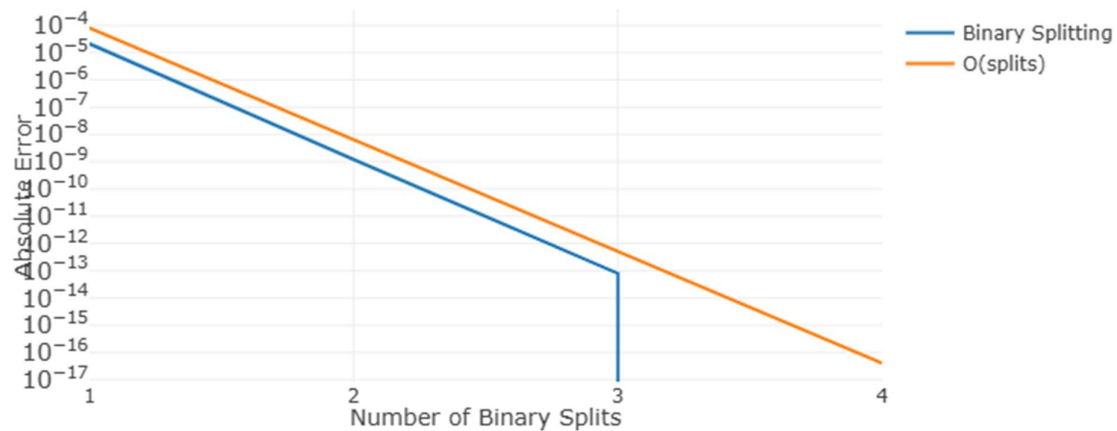
```
function binarySplitZuniga(a, b) {
  // Base case: interval [a..b) of length 1
  if (b - a === 1)
    return (function() {
      const bb = BigInt(b)
      const b6 = b * 6
      const b10 = b * 10
      const b12 = b * 12

      // p
      const p = BigInt(
        (((43203456n * bb - 92809152n) * bb + 76613904n)
          * bb - 30494304n) * bb + 6004944n)
          * bb - 536620n) * bb + 17325n
        )
      // q
      const q = BigInt(5 * (b10 - 9) * (b10 - 7) * (b10 - 3))
        * BigInt((b10 - 1) * (b12 - 11) * (b12 - 7))
        * BigInt((b12 - 5) * (b12 - 1))
      // r
      const r = -128n
        * BigInt(b * b * b)
        * BigInt((2 * b - 1) * (3 * b - 2) * (3 * b - 1))
        * BigInt((b6 - 5) * (b6 - 1))
      return { p, q, r }
    })()

  const mid = Math.floor((a + b) / 2)
  const left = binarySplitZuniga(a, mid)
  const right = binarySplitZuniga(mid, b)
  // Merge results
  const p = left.p * right.q + right.p * left.r
  const q = left.q * right.q
  const r = left.r * right.r
  return { p, q, r }
}

// Example:
const result = binarySplitZuniga(0, terms)
const gValue = Number(result.p) / Number(result.q) / 6;
```

Binary Splitting Error of Catalan-Zuniga



Binary Splitting of Catalan-Zuniga for:

Value: 0.915965594177219

Error: 0.00e+0

O(5) : 3.28e-21

Requested Splitting Terms: 5

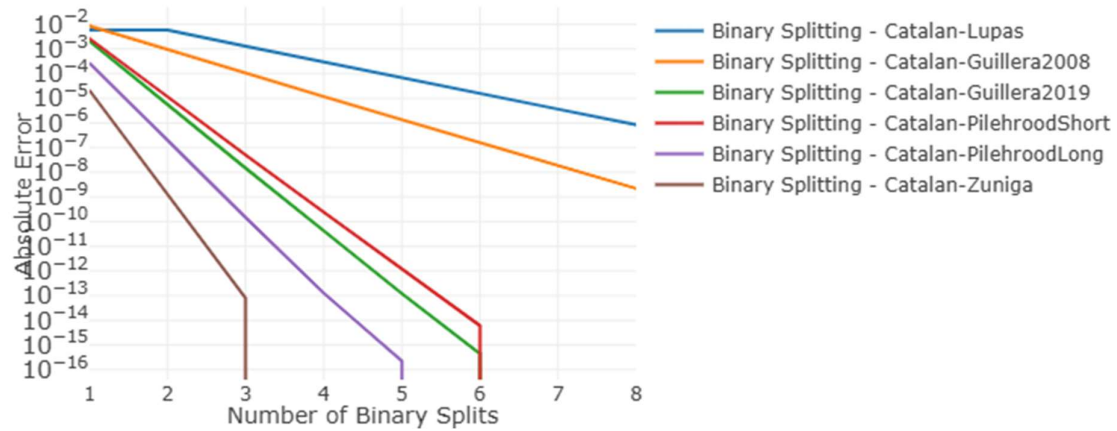
Actual Splitting Terms: 5

Comparison of the Catalan Methods

We have outlined quite a few methods for calculating the Catalan constant. To get an overview of the different methods, see the table below, which outlines the name, implementation type, error, and precision requirement. $N(P)$ is the number of splits for a given decimal precision P .

Method	Implementation	Error	$N(P)$, P =precision
Lupas	Binary Splitting	$O(4^{-n})$	1.661P
Guillera-2008	Binary-Splitting	$O(8^{-n})$	1.107P
Guillera-2019	Binary-Splitting	$O\left(\left(\frac{19683}{64}\right)^{-n}\right)$	0.402P
Pilehrood-short	Binary-Splitting	$O\left(\left(\frac{729}{4}\right)^{-n}\right)$	0.442P
Pilehrood-long	Binary-Splitting	$O(1024^{-n})$	0.332P
Zuniga	Binary-Splitting	$O(12500^{-n})$	0.244P

Binary Splitting Error Comparison



Above, we have all Catalan binary splitting methods at a glance. The Zuniga and Pilehrood-long methods are superior to the others. Pilehrood-short and Guillera 2019 are in the middle, while Lupas and Guillera 2008 trail the others. Zuniga is the clear winner here, and it is no surprise that it is implemented in the y-cruncher [3] as the preferred method.

Apéry's constant $\zeta(3)$ (Zeta(3))

It is the common short name for the $\zeta(3)$ value. This is a specialized formula for the $\zeta(3)$ instead of using the more general computation of $\zeta(s)$. There has been research into finding a formula, series, etc., for the odd integer's values of the zeta function. One of them is the value of $\zeta(3)$. Three formulas come to mind, and these are [3]:

- Amdeberhan-Zeilberger series (1997)
- Wedeniwski series (1998)
- And the newer Zuniga (2023)

Amdeberhan and Zeilberger introduced a technique in 1997 that relies on hypergeometric series and symbolic summation. They developed formulas allowing zeta(3) to be expressed in sums that converge at practical rates, using a careful arrangement of terms to reduce numerical error. Their work is notable for combining combinatorial arguments and computer algebra tools, enabling reliable calculations for higher precision.

In 1998, Wedeniwski presented an alternative method that used a series of transformations and computational optimizations. By reorganizing known expansions for zeta(3) and using error bounds to accelerate convergence, Wedeniwski's approach delivered improved efficiency. This was important for applications where high-precision zeta(3) values were needed, such as in certain number-theoretic or physical computations.

Zuniga's contribution in 2023 offered a newer summation framework. The main idea was to derive specialized variants of known series representations for zeta(3) and then apply a combination of convergence accelerators. These refinements further reduced computational complexity and provided another path to reaching reliable decimal approximations with fewer terms, reflecting ongoing progress in the theoretical and practical computing of zeta(3).

Amdeberhan-Zeilberger series

This series was given by Amdeberhan-Zeilberger in 1997.

$$\zeta(3) = \frac{1}{64} \sum_{k=0}^{\infty} \frac{(-1)^k (205k^2 + 250k + 77)(k!)^{10}}{(2k+1)^5}$$

We should have learned that the binary splitting method is the most efficient computation by now.

Algorithm: Binary splitting method for $\zeta(3)$ (1997)

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

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

$Q(a,b) = Q(a,m)Q(m,b)$

$R(a,b) = R(a,m)R(m,b)$

And:

$P(b-1,b) = (-1)^b (205b^2 + 250b + 77)b^5$

$$Q(b-1,b)=32(2b+1)^5$$

$$R(b-1,b)=b^5$$

And then

$$\zeta(3) = \frac{P(0,n) + 77Q(0,n)}{64Q(0,n)} + O(1024^{-n})$$

Which have a linearly convergent cost of ~ 2.89 , slightly higher than the following Wedeniwski method.

For n terms, the error is $O(1024^{-n})$ And for a given precision, P you need:

$$n = \left\lceil P \frac{\ln(10)}{\ln(1024)} \right\rceil$$

Source Binary splitting for Amdeberhan-Zeilberger Zeta(3)

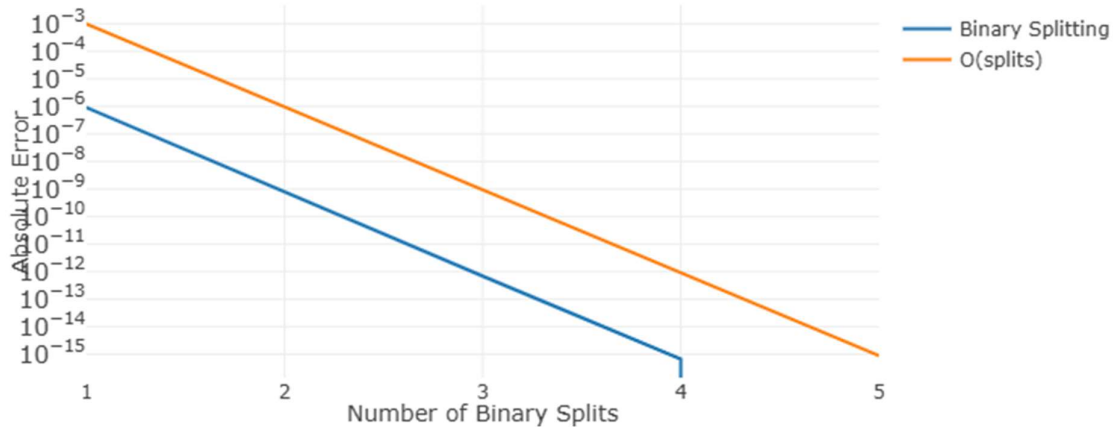
```
function binarySplitAmdeberhan(a, b) {
  // Base case: interval [a..b) of length 1
  if (b - a === 1) {
    // Compute q
    const q = BigInt(32*Math.pow(2*b+1,5));
    // Compute r
    const r = BigInt(Math.pow(b,5));
    // Compute p
    const p = BigInt((205*b+250)*b+77)*r*BigInt(Math.pow(-1, b));

    return { p, q, r };
  }

  const mid = Math.floor((a + b) / 2);
  // Recursively compute left and right intervals
  const left = binarySplitAmdeberhan(a, mid);
  const right = binarySplitAmdeberhan(mid, b);
  // Merge results
  const p = left.p * right.q + right.p * left.r;
  const q = left.q * right.q;
  const r = left.r * right.r;
  return { p, q, r };
}

// Example
const result = binarySplitAmdeberhan(0, terms);
const zeta3Value = Number(result.p+77n*result.q) / Number(64n *
result.q);
```

Binary Splitting Error of Zeta3-Amdeberhan-Zeilberger



Binary Splitting of Zeta3-Amdeberhan-Zeilberger for:

Value: 1.2020569031595942

Error: 0.00e+0

O(5) : 8.88e-16

Requested Splitting Terms: 5

Actual Splitting Terms: 5

It has a fast convergence rate, reaching the highest accuracy in just five splits.

Wedeniowski series

This series was given by Wedeniowski in 1998.

$$\zeta(3) = \frac{1}{24} \sum_{k=0}^{\infty} \frac{(-1)^k (126392k^5 + 412708k^4 + 531578k^3 + 336367k^2 + 104000k + 12643) ((2k+1)!(2k)!k!)^3}{(3k+2)!(4k+3)!^3}$$

Algorithm: Wedeniowski Binary splitting method for $\zeta(3)$ (1998)

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

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

$Q(a,b) = Q(a,m)Q(m,b)$

$R(a,b) = R(a,m)R(m,b)$

And:

$P(b-1,b) = (-1)^b (126392b^5 + 412708b^4 + 531578b^3 + 336367b^2 + 104000b + 12463) b^5 (2b-1)^3$

$Q(b-1,b) = 24(3b+1)(3b+2)(4b+1)^3(4b+3)^3$

$R(b-1,b) = b^5(2b-1)^3$

And then

$$\zeta(3) = \frac{P(0,n) + 12463Q(0,n)}{10368Q(0,n)} + O(110592^{-n})$$

These methods have a linearly convergent cost of ~ 2.78 , slightly lower than the Amdeberhan-Zeilberger method. You should expect close to the same performance for both methods.

For n terms, the error is $O(110592^{-n})$ And for a given precision, P you need:

$$n = \left\lceil P \frac{\ln(10)}{\ln(110592)} \right\rceil$$

Source Binary splitting for Wedeniwski Zeta(3)

```
function binarySplitWedeniwski(a, b) {
  // Base case: interval [a..b) of length 1
  if (b - a === 1) {
    const bb=BigInt(b);
    // Compute q
    const q = BigInt(24*(3*b+1)*(3*b+2)) *BigInt(Math.pow(4*b+1,3))
*BigInt(Math.pow(4*b+3,3));
    // Compute r
    const r = BigInt(Math.pow(b,5))*BigInt(Math.pow(2*b-1,3));
    // Compute p
    const p = (((((126392n*bb+412708n) *bb+531578n) *bb+336367n)
*bb+104000n) *bb+12463n) *r*BigInt(Math.pow(-1, b));

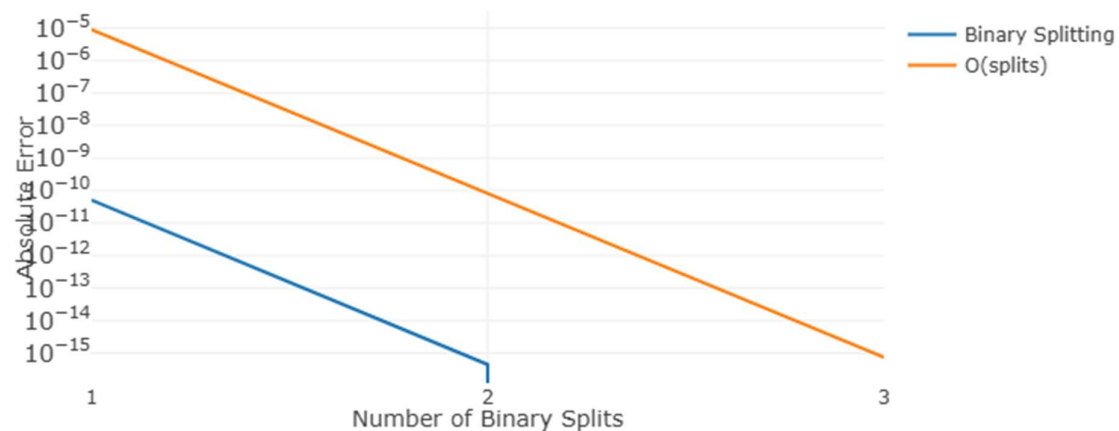
    return { p, q, r };
  }

  const mid = Math.floor((a + b) / 2);
  // Recursively compute left and right intervals
  const left = binarySplitWedeniwski(a, mid);
  const right = binarySplitWedeniwski(mid, b);
  // Merge results
  const p = left.p * right.q + right.p * left.r;
  const q = left.q * right.q;
  const r = left.r * right.r;

  return { p, q, r };
}

// Example
const result = binarySplitWedeniwski(0, terms);
const zeta3Value = Number(result.p+12463n*result.q) / Number(10368n *
result.q);
```

Binary Splitting Error of Zeta3-Wedeniwski



Binary Splitting of Zeta3-Wedeniwski for:

Value: 1.2020569031595942

Error: 0.00e+0

$O(5) : 6.04e-26$
 Requested Splitting Terms: 5
 Actual Splitting Terms: 5

Here, we reach the maximum accuracy after just three splits.

Zuniga series (ζ)

The 2023 Zuniga series is fresh and new; however, it is also a monster series.

$$\zeta(3) = \frac{1}{24} \sum_{k=0}^{\infty} \frac{P(k)}{k^5(2k-1)(3k-1)(3k-2)(4k-1)(4k-3)(5k-1)(5k-2)(5k-3)(5k-4) \binom{3k}{k} \binom{6k}{3k} \binom{8k}{4k} \binom{9k}{3k} \binom{10k}{5k}}$$

where $P(k) = 250765325100000k^{11} - 1087318449630000k^{10} + 2067749814046250k^9 - 2269551612681475k^8 + 1592180015776565k^7 - 746938801646725k^6 + 238210943593421k^5 - 51452348050672k^4 + 7352050259484k^3 - 660416507568k^2 + 33552610560k - 731566080$

Algorithm: Zuniga (ζ) Binary splitting method for $\zeta(3)$ (2023)

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

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

$Q(a,b) = Q(a,m)Q(m,b)$

$R(a,b) = R(a,m)R(m,b)$

And:

$P(b-1,b) = 250765325100000b^{11} - 1087318449630000b^{10} + 2067749814046250b^9 - 2269551612681475b^8 + 1592180015776565b^7 - 746938801646725b^6 + 238210943593421b^5 - 51452348050672b^4 + 7352050259484b^3 - 660416507568b^2 + 33552610560b - 731566080$

$Q(b-1,b) = 288(8b-7)(8b-5)(8b-3)(8b-1)(9b-8)(9b-7)(9b-5)(9b-4)(9b-2)(9b-1)(10b-9)(10b-7)(10b-3)(10b-1)$

$R(b-1,b) = b^5(2b-1)(3b-2)(3b-1)(4b-3)(4b-1)(5b-4)(5b-3)(5b-2)(5b-1)$

And then

$$\zeta(3) = \frac{1}{24} \frac{P(0,n)}{Q(0,n)} + O(34828517376^{-n})$$

Which have a linearly convergent cost of ~ 2.3 , which is lower than the Wedeniwski versions.

For n terms, the error is $O(110592^{-n})$ And for a given precision, P you need:

$$n = \left\lceil P \frac{\ln(10)}{\ln(34828517376)} \right\rceil$$

Source Binary splitting for Zuniga Zeta(3)

```
function binarySplitZuniga(a, b) {
  // Base case: interval [a..b] of length 1
  if (b - a === 1) {
```

```

    const bb=BigInt(b);
    // Compute q
    const q = BigInt(288*(8*b-7)*(8*b-5)*(8*b-3)*(8*b-1)) *
    BigInt((9*b-8)*(9*b-7)*(9*b-5)) * BigInt((9*b-4)*(9*b-2)*(9*b-1)) *
    BigInt((10*b-9)*(10*b-7)*(10*b-3)*(10*b-1));
    // Compute r
    const r = bb**5n * BigInt((2*b-1)*(3*b-2)*(3*b-1)) * BigInt((4*b-
    3)*(4*b-1)) * BigInt((5*b-4)*(5*b-3)*(5*b-2)*(5*b-1));
    // Compute p
    const p = (((((((((250765325100000n*bb-1087318449630000n)
    *bb+2067749814046250n) *bb-2269551612681475n) *bb+1592180015776565n) *bb-
    746938801646725n) * bb+238210943593421n)*bb-51452348050672n) * bb +
    7352050259484n) * bb -660416507568n)*bb+33552610560n)*bb-731566080n;

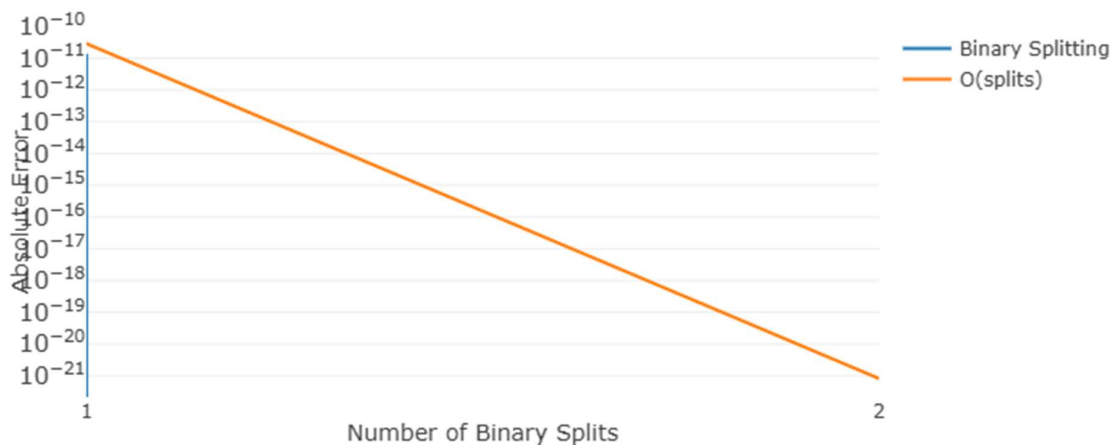
    return { p, q, r };
}

const mid = Math.floor((a + b) / 2);
// Recursively compute left and right intervals
const left = binarySplitZuniga(a, mid);
const right = binarySplitZuniga(mid, b);
// Merge results
const p = left.p * right.q + right.p * left.r;
const q = left.q * right.q;
const r = left.r * right.r;
return { p, q, r };
}

// Example
const result = binarySplitZuniga(0, terms);
const zeta3Value = Number(result.p) / Number(24n * result.q);

```

Binary Splitting Error of Zeta3-Zuniga



Binary Splitting of Zeta3-Zuniga for:

```

Value: 1.2020569031595942
Error: 0.00e+0
O(5) : 1.95e-53
Requested Splitting Terms: 5
Actual Splitting Terms: 5

```

Zuniga series (vi)

Also, in 2023, Zuniga revealed another method with a computation cost of ~ 2 .

$$\zeta(3) = \frac{1}{48} \sum_{k=0}^{\infty} \frac{-(-1)^k P(k)}{k^5 (2k-1)^3 (3k-1)(3k-2)(4k-1)(4k-3)(6k-1)(6k-5) \binom{5k}{k} \binom{5k}{2k} \binom{9k}{4k} \binom{10k}{5k} \binom{12k}{6k}}$$

where $P(k) = 1565994397644288k^{11} - 6719460725627136k^{10} + 12632254526031264k^9 - 13684352515879536k^8 + 9451223531851808k^7 - 4348596587040104k^6 + 1352700034136826k^5 - 282805786014979k^4 + 38721705264979k^3 - 3292502315430k^2 + 156286859400k - 3143448000$

Algorithm: Zuniga (vi) Binary splitting method for $\zeta(3)$ (2023)

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

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

$Q(a,b) = Q(a,m)Q(m,b)$

$R(a,b) = R(a,m)R(m,b)$

And:

$P(b-1,b) = 1565994397644288b^{11} - 6719460725627136b^{10} + 12632254526031264b^9 - 13684352515879536b^8 + 9451223531851808b^7 - 4348596587040104b^6 + 1352700034136826b^5 - 282805786014979b^4 + 38721705264979b^3 - 3292502315430b^2 + 156286859400b - 3143448000$

$Q(b-1,b) = 270(9b-8)(9b-7)(9b-5)(9b-4)(9b-2)(9b-1)(10b-9)(10b-7)(10b-3)(10b-1)(12b-11)(12b-7)(12b-5)(12b-1)$

$R(b-1,b) = -b^5(2b-1)^3(3b-2)(3b-1)(4b-3)(4b-1)(6b-5)(6b-1)$

And then

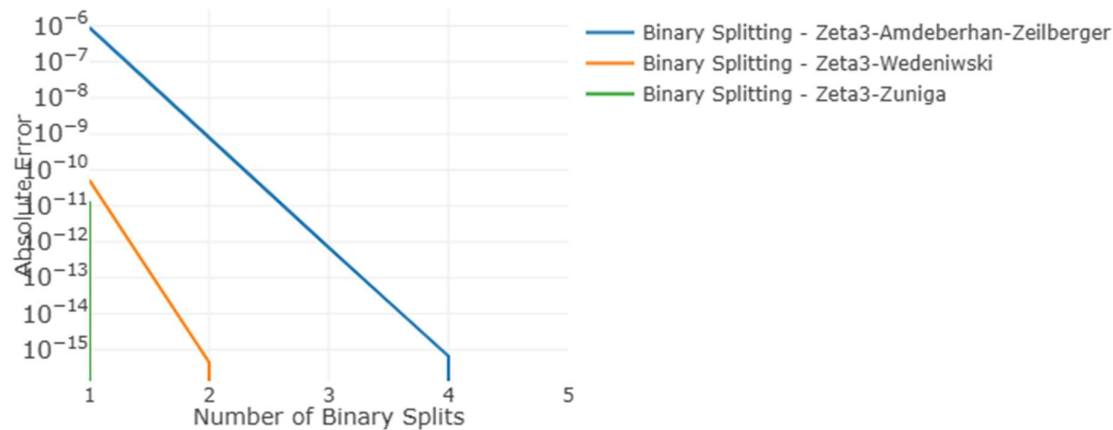
$$\zeta(3) = \frac{1}{48} \frac{P(0,n)}{Q(0,n)} + O(717445350000^{-n})$$

Comparison of the Apéry's Methods

We have outlined quite a few methods for calculating the Apéry's constant. To get an overview of the different methods, see the table below, which outlines the name, implementation type, error, and precision requirement.

Method	Implementation	Error	N(P), P=precision
Amdeberhan-Zeilberger	Binary Splitting	$O(1024^{-n})$	1.661P
Wedeniwski	Binary-Splitting	$O(110592^{-n})$	1.107P
Zuniga (v)	Binary-Splitting	$O(34828517376^{-n})$	0.095P
Zuniga (vi)	Binary-Splitting	$O(717445350000^{-n})$	0.084P

Binary Splitting Error Comparison



As the above graph shows, Zuniga requires the fewest splits (1) to reach its most accurate computation with a 64-bit float, while Wedeniowski requires two splits, and Amdeberhan-Zeilberger needs four splits.

Lemniscate constant ϖ

The lemniscate constant (ϖ) is the ratio of the perimeter of Bernoulli's lemniscate to its diameter. The $\varpi=2.62205755429211981046483$. In literature, you sometimes see that 2ϖ or $\varpi/2$ is also referred to as the Lemniscate constant, and particularly, the binary method from Zuniga 2023 below computes the 2ϖ as the Lemniscate constant. Two methods have seen the light. Both were published in 2023. One is the method from Zuniga, and the other is the method used by Guillera.

Zuniga many series

Zuniga published over 10 variations of the formula, the following of which is interesting.

Zuniga version vii.

$$\frac{1}{2\varpi} = \frac{214326}{\sqrt[6]{11816941917501}} \sum_{k=1}^{\infty} \left(\frac{-512}{2315685267}\right)^k k^2 \frac{P(k)}{R(k)} \left(\frac{\frac{1}{36}^k \frac{7}{36}^k \frac{13}{36}^k \frac{19}{36}^k \frac{25}{36}^k \frac{31}{36}^k}{\left(\frac{1}{3}\right)^k \left(\frac{1}{3}\right)^k \left(\frac{2}{3}\right)^k \left(\frac{2}{3}\right)^k k!^2}\right)$$

After some manipulation and simplification, the algorithm below is yielded.

Algorithm: Zuniga (vii) Binary splitting method for ϖ (2023)

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

And:

$$\begin{aligned}
 P(b-1,b) &= 99446494228488b^7 - 296948949253092b^6 + 339735211540956b^5 - \\
 & 185806427026662b^4 + 48479683290426b^3 - 4840729282291b^2 \\
 Q(b-1,b) &= 121545688294296b^2(3b-1)^2(3b-2)^2 \\
 R(b-1,b) &= -(36b-5)(36b-11)(36b-17)(36b-23)(36b-29)(36b-35)
 \end{aligned}$$

And then

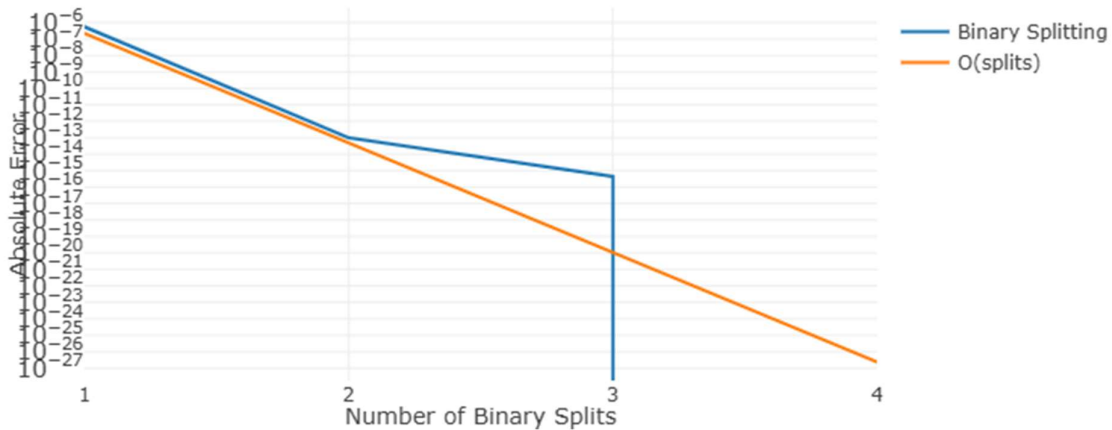
$$\varpi = \frac{1}{324\sqrt[6]{453789}} \frac{Q(0,n)}{P(0,n)} + O\left(\left(\frac{512}{2315685267}\right)^n\right)$$

These methods have a linearly convergent cost of ~ 1.566 ,

For n terms, the error is $O\left(\left(\frac{512}{2315685267}\right)^n\right)$ And for a given precision, P you need:

$$n = \left\lceil P \frac{\ln(10)}{\ln\left(\frac{512}{2315685267}\right)} \right\rceil$$

Binary Splitting Error of Lemniscate-Zunigavii



Binary Splitting of Lemniscate-Zuniga-vii for:

Value: 2.6220575542921196

Error: 0.00e+0

O(5) : 5.28e-34

Requested Splitting Terms: 5

Actual Splitting Terms: 5

Notice that after three splits, we reached an error of 0.

Zuniga version viii, which is similar.

$$\frac{1}{2\omega} = \frac{215622}{\sqrt[4]{483153}} \sum_{k=1}^{\infty} \left(\frac{512}{2357947691}\right)^k k^2 \frac{P(k)}{R(k)} \left(\frac{1}{36}\right)_k \left(\frac{5}{36}\right)_k \left(\frac{13}{36}\right)_k \left(\frac{17}{36}\right)_k \left(\frac{25}{36}\right)_k \left(\frac{29}{36}\right)_k \left(\frac{1}{3}\right)_k \left(\frac{1}{3}\right)_k \left(\frac{2}{3}\right)_k \left(\frac{2}{3}\right)_k k!^2$$

After some manipulation and simplification, the algorithm below is yielded.

Algorithm: Zuniga (viii) Binary splitting method for ϖ (2023)

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

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

$Q(a,b)=Q(a,m)Q(m,b)$

$R(a,b)=R(a,m)R(m,b)$

And:

$P(b-1,b)=1768056164733b^7-52825631815620b^6+60473303319276b^5-33092086224942b^4+8638260598818b^3-862864755643b^2$

$Q(b-1,b)=123763958405208b^2(3b-1)^2(3b-2)^2$

$R(b-1,b)=(36b-7)(36b-11)(36b-19)(36b-23)(36b-31)(36b-35)$

And then

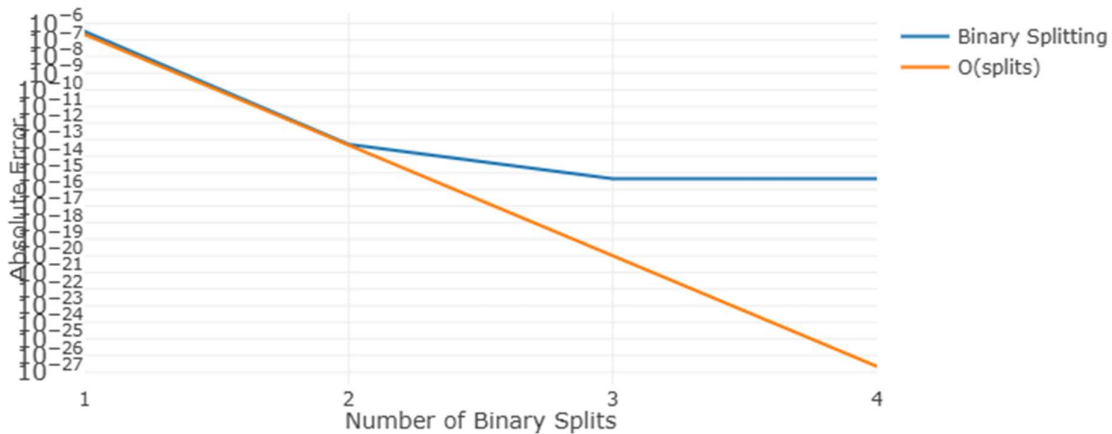
$$\varpi = \frac{1}{1188\sqrt[4]{35937}} \frac{Q(0,n)}{P(0,n)} + O\left(\left(\frac{512}{2357947691}\right)^n\right)$$

These methods have a linearly convergent cost of ~ 1.564

For n terms, the error is $O\left(\left(\frac{512}{2357947691}\right)^n\right)$ And for a given precision, P you need:

$$n = \left\lceil P \frac{\ln(10)}{\ln\left(\frac{512}{2357947691}\right)} \right\rceil$$

Binary Splitting Error of Lemniscate-Zunigaviii



Binary Splitting of Lemniscate-Zunigaviii for:

Value: 2.62205755429212

Error: 4.44e-16

O(3) : 1.02e-20

Requested Splitting Terms: 3

Actual Splitting Terms: 3

Notice that computational rounding errors sneak in after two splits. We have the highest accuracy we can expect.

Guillera series

Guillera's series in the form proper for the Binary splitting method is:

$$\frac{1}{2\varpi} = \frac{34560}{\sqrt[8]{162000}} \sum_{k=1}^n \frac{k^2(1288k - 1247)}{-(8k - 5)(8k - 7)} \prod_{i=1}^k \frac{-(8i - 5)(8i - 7)}{1658880i^2}$$

Not as voluminous as some of the previous series.

Algorithm: Guillera Binary splitting method for ϖ (2023)

```

set 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)

And:
P(b-1,b)=b2(1288b-1247)
Q(b-1,b)=1658880b2
R(b-1,b)=-(8b-5)(8b-7)

```

And then

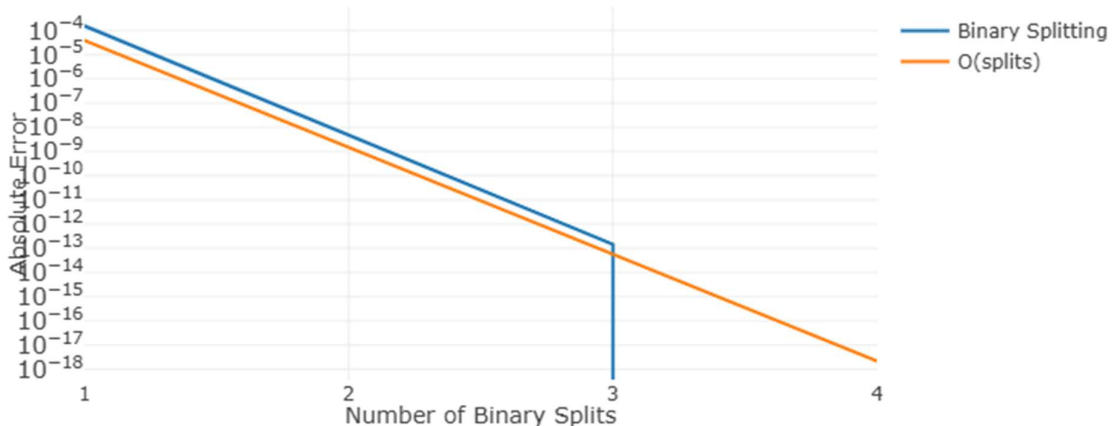
$$\varpi = \frac{\sqrt[8]{162000} Q(0, n)}{69120 P(0, n)} + O(25920^{-n})$$

These methods have a linearly convergent cost of ~ 0.7872 , which is good and lower than the Zuniga series. However it requires more splits that the Zuniga series.

For n terms, the error is $O(25920^{-n})$ And for a given precision, P you need:

$$n = \left\lceil P \frac{\ln(10)}{\ln(25920)} \right\rceil$$

Binary Splitting Error of Lemniscate-Guillera



Binary Splitting of Lemniscate-Guillera for:

Value: 2.622057554291974

Error: -1.46e-13

$O(3) : 5.74e-14$

Requested Splitting Terms: 3

Actual Splitting Terms: 3

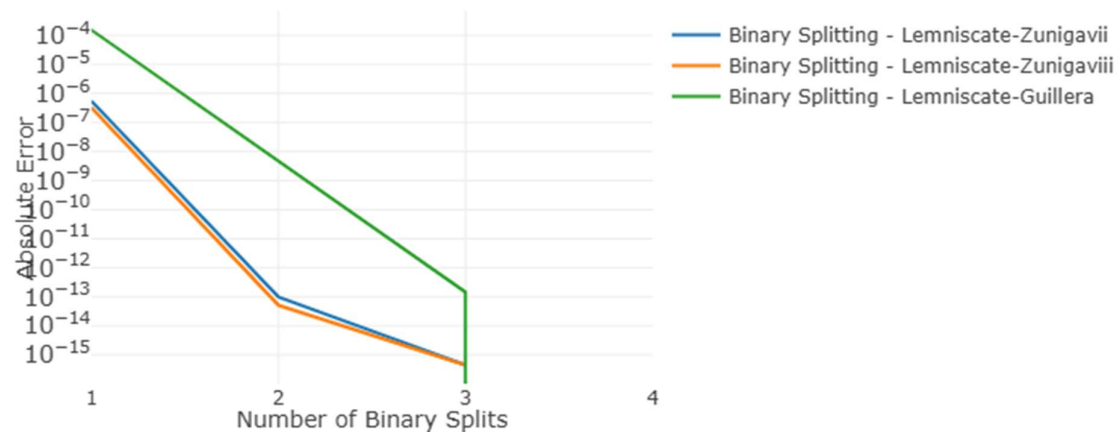
After 3 splits, we get an error of $\sim 1e-13$, and after 4 splits, we get an error of 0.

Comparison of the Lemniscate methods.

The three Lemniscate methods are listed below. The two Zuniga versions have similar characteristics and are more efficient than the Guillera method.

Method	Implementation	Error	$N(P), P=precision$
Zuniga vii	Binary Splitting	$O\left(\left(\frac{512}{2315685267}\right)^n\right)$	0.15P
Zuniga viii	Binary-Splitting	$O\left(\left(\frac{512}{2357947691}\right)^n\right)$	0.15P
Guillera	Binary-Splitting	$O(25920^{-n})$	0.2266P

Binary Splitting Error Comparison



This plot shows a picture similar to the comparison table. The Two Zuniga methods have identical characteristics and performance, while Guillera has slower convergence but is

Conclusion

The binary splitting method is robust for high-precision computation of mathematical constants. By organizing series expansions around carefully defined products and sums, often expressed in two- or three-variable recursive forms, this technique minimizes the size of intermediate values and reduces the number of large multiplications. Such advantages become increasingly important when aiming for tens of millions or even trillions of digits.

Through concrete examples, we have shown how various constants—ranging from e to π , $\zeta(3)$ (zeta(3), Catalan's constant G , and the lemniscate constant ϖ , can be calculated efficiently using different binary splitting formulas. These examples illustrate that seemingly small differences in series representations can lead to dramatic variations in convergence speed, implementation complexity, and memory usage. In particular, modern formulas (e.g.,

Zuniga's methods for $\zeta(3)$ and π often outperform earlier approaches, reflecting the ongoing search for ever-faster series expansions.

Crucially, binary splitting's divide-and-conquer structure facilitates parallelization, letting large problems be tackled on multi-core CPUs or distributed systems. At the same time, the method's reliance on exact rational arithmetic ensures that only one final conversion to floating point occurs, preserving precision throughout most of the process. The examples show that combining these factors with suitable big-integer libraries (such as JavaScript's BigInt) provides a reliable pipeline for computing constants to millions of digits with manageable runtime and memory demands.

Further research may develop new hypergeometric transformations or alternative factorization schemes that reduce the cost of partial products or improve convergence rates. Yet, due to its flexibility, performance, and parallel potential, binary splitting remains important for numerical computations of special constants. Researchers and enthusiasts can benefit from this robust approach, whether exploring fundamental constants at unprecedented depths or integrating high-precision arithmetic into next-generation computational tools.

For those wishing to experiment directly, the accompanying web interface (available at www.hvks.com/Numerical/webbinarySplitting.html) provides an accessible starting point. It allows users to test different formulas, visualize convergence, and appreciate the efficiency gains afforded by binary splitting in practice.

References

1. IEEE Standard for Floating-Point Arithmetic (IEEE 754)
2. David Goldberg, "What Every Computer Scientist Should Know About Floating-Point Arithmetic"
3. Numberworld.org by Alexander Yee, numberworld.org