

## Fast Computation of Math Constants in Arbitrary Precision.

By Henrik Vestermarck (hve@hvks.com)

### Abstract

This is a continuation of the series of papers written dealing with the practical aspect of implementing an arbitrary precision math package. The paper describes the many constants, like the Euler-Mascheroni, Catalan, Apéry (zeta(3)), and the Lemniscate constant, needed for making a complete Arbitrary precision Math package. These constants arise in integrals, series expansions, and special functions, and efficient routines for computing them with arbitrary precision enhance the versatility of mathematical software libraries.

### Introduction

We begin by examining the various mathematical constants and the different methods available to compute these constants:

- The Euler-Mascheroni constant
- The Catalan Constant
- The Apéry Constant  $\zeta(3)$
- The Lemniscate constant  $\varpi$

We will show the actual C++ source for the calculation using the author's arbitrary precision Math library. See [1].

This paper is part of a series of arbitrary precision papers describing methods, implementation details, and optimization techniques. Earlier work, accessible at [www.hvks.com/Numerical/papers.html](http://www.hvks.com/Numerical/papers.html), examines additional details on arbitrary precision methods. These are listed below:

1. Fast Gamma, Beta, Error, and Zeta functions for arbitrary precision. [HVE Fast Gamma, Beta, Error, and Zeta functions for arbitrary precision.](#)
2. Fast Square Root & Inverse calculation for arbitrary precision math. [HVE Fast Square Root & inverse calculation for arbitrary precision](#)
3. Fast Exponential calculation for arbitrary precision math. [HVE Fast Exp\(\) calculation for arbitrary precision](#)
4. Fast logarithm calculation for arbitrary precision math. [HVE Fast Log\(\) calculation for arbitrary precision](#)
5. Practical implementation of Spigot Algorithms for Transcendental Constants. [Practical implementation of Spigot Algorithms for transcendental constants](#)
6. Practical implementation of Spigot Algorithms for transcendental constants. [Practical implementation of  \$\pi\$  algorithms.](#) [HVE Practical implementation of PI Algorithms](#)
7. Fast Trigonometric function for arbitrary precision. [Fast Trigonometric function for arbitrary precision.](#) [HVE Fast Trigonometric calculation for arbitrary precision](#)

# Fast Computation of Math Constants in Arbitrary Precision

---

8. Fast Hyperbolic functions for arbitrary precision. [HVE Fast Hyperbolic calculation for arbitrary precision](#)
9. Fast conversion from arbitrary precision number to a string. [HVE Fast conversion from arbitrary precision to string](#)
10. Fast conversion from a decimal string to an arbitrary precision number. [HVE Fast conversion from string to arbitrary precision](#)

## Change Log

6-March-2025 Added the Binary Splitting method for Zuniga Catalan constant.

10-March-2025 Added the four-variable version of the Binary Splitting method for the Euler-Mascheroni constant.

17-March-2025 Added the Binary Splitting method for the Lemniscate constant.

# Fast Computation of Math Constants in Arbitrary Precision

---

## Contents

Abstract .....	1
Introduction.....	1
Change Log.....	2
The Arbitrary Precision Library.....	5
Internal format for float_precision variables .....	6
Normalized numbers.....	6
Euler-Mascheroni constant $\gamma$ .....	7
Brent-McMillan method .....	7
Source Euler Brent-McMillan .....	8
Brent enhancement.....	9
Source Euler Brent summation trick.....	9
The binary splitting method for $\gamma$ .....	10
Source of the five-variable Binary splitting method.....	12
Source of the threaded five-variable Binary splitting method.....	13
Source of the four-variable Binary splitting method .....	14
Source of the threaded four-variable Binary splitting method .....	15
Performance of Euler-Mascheroni constant.....	16
Recommendation for the Euler-Mascheroni constant.....	17
Catalan's constant G .....	18
Ramanujan's method I.....	18
Source Ramanujan's I.....	19
Ramanujan's method II.....	19
Source Ramanujan's II.....	20
Broadhurst series.....	20
Source Broadhurst.....	20
The Binary Splitting method for the Catalan constant.....	21
Lupas Binary Splitting method .....	22
Source Catalan-Lupas binary splitting.....	23
Guillera Binary Splitting method.....	24
Source Catalan-Guillera binary splitting (2008).....	25
Source Catalan-Guillera binary splitting (2019).....	26
Pilehrood binary splitting method.....	27
Source Catalan-Pilehrood binary splitting (2010) .....	29
Source Catalan-Pilehrood long binary splitting (2010) .....	30
Zuniga binary splitting method.....	31
Source Catalan-Zuniga binary splitting (2023) .....	32
Comparison of the Catalan Methods.....	33
Catalan Constant Performance.....	33
Recommendation for the Catalan constant .....	35
Apéry's constant $\zeta(3)$ .....	36
Amdeberhan-Zeilberger series.....	36
Source Amdeberhan-Zeilberger Binary splitting method.....	37
Source Threaded Amdeberhan-Zeilberger Binary splitting method.....	38
Wedeniwski series .....	39
Source Wedeniwski binary splitting method .....	39

# Fast Computation of Math Constants in Arbitrary Precision

---

Zuniga series (v) .....	40
Source Zuniga (v) binary splitting method .....	41
Zuniga series (vi) .....	43
Source Zuniga (vi) binary splitting method .....	44
Comparison of the Apéry's Methods .....	45
Apéry Constant $\zeta(3)$ performance .....	46
Recommendation for the constant $\zeta(3)$ .....	47
The Lemniscate Constant $\varpi$ .....	48
Guillera Binary Splitting method .....	51
Comparison of the Lemniscate methods .....	51
Recommendation for the Lemniscate constant $\varpi$ .....	53
Overall conclusion .....	53
Reference .....	53

---

# Fast Computation of Math Constants in Arbitrary Precision

---

## The Arbitrary Precision Library

You can skip this section if you are already familiar with the arbitrary precision library.

To understand the C++ code and text, we must highlight a few features of the arbitrary precision library, such as the class name *float\_precision*. Instead of declaring a variable with a float or double, you 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 an optional second parameter is a floating-point precision. The native float type has a fixed size of 4 bytes and 8 bytes for *double*. However, since this precision can be arbitrary, we can declare the desired precision as the number of **decimal digits** we want 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 to manipulate the variable. To change or set the precision, you can call the method `.precision()`, for example,

```
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 to manipulate the exponent of the variables. The method is called `.exponent()` and returns or sets the exponent as a power of two exponents (same as for our regular build-in types *float* and *double*), E.g.

```
f.exponent(); // Return the exponent as 2e
f.exponent(0) // Remove the exponent
f.exponent(16) // Set the exponent to 216
```

There is a second way to manipulate the exponent: the class method `.adjustexponent()`. This method 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 with 2.
f.adjustExponent(-1); // Subtract 1 from the exponent, the same as dividing the number with 2.
```

This allows very fast division multiplication with a number that is any power of two.

The method `.iszero()` returns true if the *float\_precision* number is zero otherwise, false. There are additional methods, but I will refer to the reference for the user manual and the arbitrary precision math package for details.

# Fast Computation of Math Constants in Arbitrary Precision

---

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

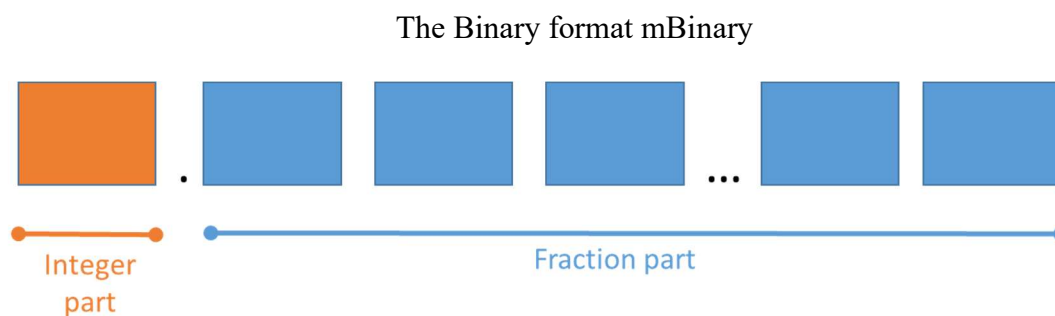
## Internal format for float\_precision variables

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. 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  $\geq 1$
- 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.

Other internal class variables like the sign, exponent, precision, and rounding mode 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].

## Euler-Mascheroni constant $\gamma$

The Euler–Mascheroni constant, often denoted by  $\gamma$ , is approximately 0.5772156649. It appears when comparing the harmonic series to the natural logarithm. Specifically, if you take the sum of the harmonic series up to  $1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n}$  When subtracting  $\ln(n)$ , the difference approaches  $\gamma$  as  $n$  grows large. This constant also appears in integrals and special functions, making it a recurring element in various branches of analysis.

Leonhard Euler introduced the constant while studying infinite series in the 18th century. Later, the Italian mathematician Lorenzo Mascheroni took an interest in it, leading to deeper investigations and the name we use today. Over time, many mathematicians attempted to understand its algebraic nature. Whether  $\gamma$  is rational or irrational remains unknown, and it continues to spark interest due to its ties to the harmonic series, logarithms, and other core areas of mathematics.

The Euler-Mascheroni constant  $\gamma$  is defined as:

$$\gamma = \lim_{n \rightarrow \infty} \left( \sum_{k=1}^n \frac{1}{k} - \ln(n) \right) \approx 0.577215664 \quad (1)$$

The above equation (1) converges slowly and is useless for arbitrary precision arithmetic. Instead, there are a few other interesting methods to compute this constant.

- Brent-McMillan method
- Brent enhancement
- The binary splitting version of the Brent-McMillan method

### ***Brent-McMillan method***

To compute  $\gamma$  you can use the Brent-McMillan decomposition [3].

$$\gamma \approx \frac{S(n)}{V(n)} - \ln(n) \quad (2)$$

$S(n)$  and  $V(n)$  are some auxiliary functions, and  $n$  is chosen to ensure high enough precision for the result.

Furthermore, the sequence  $S(n)$  and  $V(n)$  is defined as:

# Fast Computation of Math Constants in Arbitrary Precision

$$S(n) = \sum_{k=0}^{\infty} \left(\frac{n^k}{k!}\right)^2 \cdot H_k \quad (3)$$

$$V(n) = \sum_{k=0}^{\infty} \left(\frac{n^k}{k!}\right)^2 \quad (4)$$

The sequence  $H_n$  is defined as the partial sum of the Harmonic series:

$$H_n = \sum_{k=1}^n \frac{1}{k} \quad (5)$$

Two questions arise. What is an appropriate value for  $n$ , and when should the  $k$  summation stop? Brent estimated the minimum value for  $n$  as a function for the required precision  $P$  to be:

$$n = \left\lceil \frac{P \cdot \ln(10) + \ln(\pi)}{4} \right\rceil \quad (6)$$

The required number of terms  $k_{max}$  in the summation as a function of the precision  $P$  to be:

$$k_{max} \approx 2.07 \cdot P \quad (7)$$

Technically, we don't need the  $k_{max}$  a priori since we can terminate the  $S(n)$  and  $V(n)$  sequence when the individual term value becomes less than the required precision dictate. (Usually, that will require more iterations than just using  $k_{max}$ ).

## Source Euler Brent-McMillan

```
// Euler-Mascheroni constant. Classic method.
//
static float_precision EulerConstant1(const size_t precision)
{
    const uintmax_t kmax=(uintmax_t)ceil(2.07*precision);
    const uintmax_t n = (uintmax_t)ceil((precision * log(10) + log(3.14159265)) / 4);
    const size_t workprec = (size_t)ceil(precision+4*log(kmax)/log(10)+1);
    const float_precision c1(1);
    float_precision res(0,workprec), vsq(0,workprec);
    float_precision vsum(1, workprec), ssum(0,workprec), hsum(0,workprec);
    int_precision np(1), kfac(1);

    for (uintmax_t k = 1;kmax>=k;++k)
    {
        np *= n;           // np=n^k
        kfac *= k;         // kfac=k!
        hsum += c1 / float_precision(k, workprec); // +1/k
        vsq=float_precision(np,workprec) / float_precision(kfac, workprec);
        vsq = vsq.square();
        vsum += vsq;
        vsq *= hsum;
        ssum += vsq;
    }

    hsum = vsum * log(float_precision(n, workprec));
    res = (ssum - hsum) / vsum;
    res.precision(precision);
    return res;
}
```

## ***Brent enhancement***

Brent further improves the above formula by using a clever summation trick. Brent defined  $U(n)$  as:

$$U(n) = S(n) - V(n) \cdot \ln(n) \quad (8)$$

Then

$$\gamma \approx \frac{U(n)}{V(n)} \quad (9)$$

And

$$U(n) = \sum_{k=0}^{\infty} A_k \quad (10)$$

Where  $A_k$  is:

$$A_k = \left(\frac{n^k}{k!}\right)^2 (H_k - \ln(n)) \quad (11)$$

Furthermore, we use  $B_k$  as the  $n^{\text{th}}$  term of the  $V(n)$  series.

$$B_k = \left(\frac{n^k}{k!}\right)^2 \quad (12)$$

We can then compute the  $A_k$  and  $B_k$  simultaneously using the below recurrence.

Algorithm for Brent summation trick

$$\begin{aligned} A_0 &= -\ln(n), B_0 = 1 \\ B_k &= B_{k-1} \cdot \frac{n^2}{k^2} \\ A_k &= \frac{1}{k} \left( A_{k-1} \cdot \frac{n^2}{k} + B_k \right) = A_{k-1} \cdot \frac{n^2}{k^2} + \frac{B_k}{k} \end{aligned}$$

Algorithm 1

Source Euler Brent summation trick

Although the algorithm above called for two divisions per term, we can reduce it to one division in the actual source code.

```
// Euler-Mascheroni constant with Brent enhancement
// This is faster than the previous EulerConstant1
//
static float_precision EulerConstant2(const size_t precision)
{
    const uintmax_t kmax = (uintmax_t)ceil(2.07 * precision);
```

# Fast Computation of Math Constants in Arbitrary Precision

```
const uintmax_t n = (uintmax_t)ceil((precision * log(10) + log(3.14159265)) / 4);
const size_t workprec = (size_t)ceil(precision + 3* log(kmax))+1;
uintmax_t k;
float_precision res(0, workprec);
float_precision ak(0, workprec), bk(1, workprec), nsq(n * n, workprec), fpk(0,
workprec), tmp(0,workprec);

ak = -log(float_precision(n, workprec)); // initialize ak=-ln(n)
for (k = 1; kmax >= k; ++k)
{
    fpk = float_precision(k, workprec);
    fpk = fpk.inverse(); // 1/k
    tmp = nsq * fpk.square(); // n^2/k^2
    bk *= tmp; // Bk=Bk-1*n^2/k^2
    ak *= tmp; // Ak=Ak-1*n^2/k^2
    ak += bk*fpk; // Ak+=Bk/k
}

res = ak / bk;
res.precision(precision);
return res;
}
```

## The binary splitting method for $\gamma$

Lastly, you can use the Binary splitting technique as outlined in [4]. The method is derived from the following series (Eq. 2), which leads to the recursive procedure we implement below.

Algorithm: Binary splitting method for  $\gamma$  (7 variables)

$$\begin{aligned} \text{set } m &= \frac{a+b}{2} \text{ integer division} \\ P(a,b) &= P(a,m)Q(m,b) + Q(a,m)P(m,b) \\ Q(a,b) &= Q(a,m)Q(m,b) \\ R(a,b) &= R(a,m)S(m,b) + T(a,m)R(m,b) \\ S(a,b) &= S(a,m)S(m,b) \\ T(a,b) &= T(a,m)T(m,b) \\ U(a,b) &= U(a,m)V(m,b) + P(a,m)T(a,m)Q(m,b)R(m,b) + Q(a,m)T(a,m)U(m,b) \\ V(a,b) &= V(a,m)V(m,b) \end{aligned}$$
$$\begin{aligned} \text{And } P(b-1,b) &= 1; \quad Q(b-1,b) = b; \quad R(b-1,b) = n^2; \quad S(b-1,b) = b^2; \quad T(b-1,b) = n^2; \\ U(b-1,b) &= n^2; \quad V(b-1,b) = b^3; \end{aligned}$$

Algorithm 2. The 7-variable Binary Splitting for the Euler-Mascheroni constant

You continue this recursive breakdown until  $a+1=b$ , and you set:

$$P(a,b)=1 \quad Q(a,b)=b \quad R(a,b)=n^2 \quad S(a,b)=b^2 \quad T(a,b)=n^2 \quad U(a,b)=n^2 \quad V(a,b)=b^3$$

And let the formula reverse bottom up.

In the end, you find  $\gamma$  by:

$$\gamma = \frac{U(0,i)}{Q(0,i)(R(0,i)+S(0,i))} - \ln(n) \quad (13)$$

## Fast Computation of Math Constants in Arbitrary Precision

---

In [4], they found that  $i=3.5911214766686221366n$  is the number of needed terms as a function of  $n$ . And  $n$  can be chosen as in (6).

Now, the binary splitting algorithm requires seven variables. You can quickly reduce the number of variables from 7 to 5 by noting that  $S(m,b)=Q(m,b)^2$  and  $V(m,b)=Q(m,b)^3$ , and you get the reduced variable version below. This reduction cuts computational overhead and simplifies the implementation by minimizing data dependencies.

Algorithm: Binary splitting method for  $\gamma$  (5 variables)

$$\begin{aligned}
 & \text{set } m = \frac{a+b}{2} \text{ integer division} \\
 & P(a,b)=P(a,m)Q(m,b)+Q(a,m)P(m,b) \\
 & Q(a,b)=Q(a,m)Q(m,b) \\
 & R(a,b)=R(a,m)Q(m,b)^2+T(a,m)R(m,b) \\
 & T(a,b)=T(a,m)T(m,b) \\
 & U(a,b)=U(a,m)Q(m,b)^3+P(a,m)T(a,m)Q(m,b)R(m,b)+Q(a,m)T(a,m)U(m,b) \\
 \\
 & \text{And } P(b-1,b)=1; \quad Q(b-1,b)=b; \quad R(b-1,b)=n^2; \quad T(b-1,b)=n^2; \quad U(b-1,b)=n^2;
 \end{aligned}$$

Algorithm 3. The 5-variable Binary Splitting for the Euler-Mascheroni constant

In the end, you find  $\gamma$  by (now that  $S(0,i)$  has been replaced by  $Q(0,i)^2$ ):

$$\gamma = \frac{U(0,i)}{Q(0,i)(R(0,i)+Q(0,i)^2)} - \ln(n) \tag{14}$$

The five variables version does perform better but not impressively better. In [4], they have further reduced it down to 4 variables.

The reduction to a 4-variable version can be done by eliminating the  $T(a,b)$ , noting that  $T(a,b)=(n^2)^{b-a}$ , and substituting it into the four other functions,  $Q, P, R, U$ .

Algorithm: Binary splitting method for  $\gamma$  (4 variables)

$$\begin{aligned}
 & \text{set } m = \frac{a+b}{2} \text{ integer division} \\
 & P(a,b)=P(a,m)Q(m,b)+Q(a,m)P(m,b) \\
 & Q(a,b)=Q(a,m)Q(m,b) \\
 & R(a,b)=R(a,m)Q(m,b)^2+(n^2)^{m-a} R(m,b) \\
 & U(a,b)=U(a,m)Q(m,b)^3+P(a,m)(n^2)^{m-a} Q(m,b)R(m,b)+Q(a,m)(n^2)^{m-a} U(m,b) \\
 \\
 & \text{And } P(b-1,b)=1; \quad Q(b-1,b)=b; \quad R(b-1,b)=n^2; \quad U(b-1,b)=n^2;
 \end{aligned}$$

Algorithm 4. The 4-variable Binary Splitting for the Euler-Mascheroni constant

The method is faster than the five-variable version with only four variables. We use memorization to calculate each  $(n^2)^{m-a}$  once. It is interesting to find out how many different  $(n^2)^{m-a}$  you must evaluate between  $a$  and  $b$ . It grows slowly with the size of  $b$ . E.g., if you plan to perform 9,999 splits, you need to assess only 25 different  $(n^2)^{m-a}$ . For 999,999 splits, you need 38 unique values. This makes using memorization highly attractive for implementation.

# Fast Computation of Math Constants in Arbitrary Precision

Source of the five-variable Binary splitting method

The main call is to computeEulerDigits(), which initializes and calls the recursive binary splitting function binarysplittingEuler(). This is the five-variable version.

```
// From the original 7 variables recursion. s and v has been eliminated and replaced with
// ss=Q(m,b)^2 and vv=Q(m,b)^3
// for the 5 variables recursions
//
static void binarysplittingEuler5(const uintmax_t a, const uintmax_t b, const uintmax_t nsq,
int_precision& p, int_precision& q, int_precision& r, int_precision& t, int_precision& u)
{
    uintmax_t mid;
    int_precision pp, qq, rr, tt, uu, tmp;
    if (a + 1 == b)
    {
        p = int_precision(1); //p=1
        q = int_precision(b); //q=b
        r = int_precision(nsq); //r=n^2
        t = r; //t=n^2
        u = r; //u=n^2
        return;
    }

    mid = (a + b) / 2;
    binarysplittingEuler5(a, mid, nsq, p, q, r, t, u); // interval [a..mid]
    binarysplittingEuler5(mid, b, nsq, pp, qq, rr, tt, uu ); // interval [mid..b]

    // Reconstruct interval [a..b] and return updated p,q,r,t,u
    tmp = qq; tmp *= tmp; // qq^2
    // Update r
    r *= tmp; r += t * rr;
    tmp *= qq; // qq^3
    // Update u
    u = u * tmp + p * t * qq * rr + q * t * uu;
    // Update p
    p *= qq; p+=q * pp;
    // Update q
    q *= qq;
    // Update t
    t *= tt;
    return;
}

// Driver function for the Euler-Mascheroni 5 variable Binary Splitting method
//
static float_precision computeEulerdigits(const uintmax_t precision)
{
    const uintmax_t EXTRA = 1;
    const uintmax_t n = (uintmax_t)ceil((precision * log(10) +
log(3.14159265358979323846)) / 4);
    const uintmax_t k = uintmax_t(ceil(n * 3.5911214766686221366));
    const size_t workprec = (size_t)ceil(precision + EXTRA + log(k));
    int_precision p, q, r, t, u;
    float_precision fp, fq;

    binarysplittingEuler5(0, k, n * n, p, q, r, t, u);

    q *= r + q * q;
    fp.precision(workprec);
    fq.precision(workprec);
    fp = float_precision(u, workprec);
    fq = float_precision(q, workprec);
    fp -= fq * log(float_precision(n, workprec));
    fp /= fq;
    fp.precision(precision);
    return fp;
}
```

---

# Fast Computation of Math Constants in Arbitrary Precision

---

Creating a threaded version of the binary splitting algorithm is relatively easy, and threading the computational task has proven advantageous in increasing performance.

The below-threaded version only divides the computation into two parallel threads. Expanding the source into a four-threaded version or higher is relatively easy if needed.

## Source of the threaded five-variable Binary splitting method

The recursive function `binarysplittingEuler()` is the same for threaded and the non-threaded versions.

```
// Driver function for the Euler-Mascheroni 5 variable Binary Splitting method
// using two threads.
//
static float_precision computeEulerdigitsThread(const uintmax_t precision)
{
    const uintmax_t EXTRA = 1;
    const uintmax_t n = (uintmax_t)ceil((precision * log(10) +
log(3.14159265358979323846)) / 4);
    const uintmax_t k = uintmax_t(ceil(n * 3.5911214766686221366));
    const size_t workprec = (size_t)ceil(precision + EXTRA + log(k));
    int_precision p, q, r, t, u;
    int_precision pp, qq, rr, tt, uu, tmp;
    float_precision fp, fq;

    std::thread first([=, &p, &q, &r, &t, &u]()
        {binarysplittingEuler5(0, k / 2, n*n, p, q, r, t, u); }); // interval
[a..k/2]

    std::thread second([=, &pp, &qq, &rr, &tt, &uu]()
        {binarysplittingEuler5(k / 2, k, n*n, pp, qq, rr, tt, uu ); }); //
interval [k/2..k]

    first.join();
    second.join();

    // Reconstruct interval [0..k] and return updated p,q,r,t,u
    tmp = qq; tmp *= tmp;
    r *= tmp; r += t * rr;
    tmp *= qq;
    u = u * tmp + p * t * qq * rr + q * t * uu;
    p *= qq; p += q * pp;
    q *= qq;
    t *= tt;

    // Finalize Calculation
    q *= r + q * q;
    fp.precision(workprec);
    fq.precision(workprec);
    fp = float_precision(u, workprec);
    fq = float_precision(q, workprec);
    fp -= fq * log(float_precision(n, workprec));
    fp /= fq;
    fp.precision(precision);
    return fp;
}
```

And finally, we have the four-variable version of the binary splitting method for the Euler-Mascheroni constant where the `t` variable has been eliminated.

# Fast Computation of Math Constants in Arbitrary Precision

## Source of the four-variable Binary splitting method

The main call is to compute `EulerDigits4()`, which initializes and calls the recursive binary splitting function `binarysplittingEuler4()`.

```
// From the original 7 variables recursion. s and v has been eliminated and replaced with
// ss=Q(m,b)^2 and vv=Q(m,b)^3
// Further reduce by eliminating t, to a 4 variable recursion.
//
static void binarysplittingEuler4(const uintmax_t a, const uintmax_t b, const
std::vector<int_precision>& powsq, int_precision& p, int_precision& q, int_precision& r,
int_precision& u)
{
    uintmax_t mid;
    int_precision pp, qq, rr, uu, qmb2, qmb3;
    if (a + 1 == b)
    {
        p = int_precision(1); //p=1
        q = int_precision(b); //q=b
        r = powsq[1]; //r=n^2
        u = r; //u=n^2
        return;
    }

    mid = (a + b) / 2;
    binarysplittingEuler4(a, mid, powsq, p, q, r,u); // interval [a..mid]
    binarysplittingEuler4(mid, b, powsq, pp, qq, rr, uu); // interval [mid..b]

    // Reconstruct interval [a..b] and return updated p,q,t,u
    const int_precision ip((powsq[mid-
a]==int_precision(0)?ipow(powsq[1],int_precision(mid-a)):powsq[mid - a]));

    // Build qq^3
    qmb2 = qq; // qq
    qmb2 *= qmb2; // qq^2
    qmb3 = qmb2 * qq; // qq^3
    // Update u
    u = u * qmb3 + ip*(p * qq * rr + q * uu);
    // Update p
    p *= qq; p += q * pp;
    // Update q
    q *= qq;
    // Update r
    r *= qq * qq; r += ip * rr;
    return;
}

// Driver function for the Euler-Mascheroni 4 variable Binary Splitting method
//
static float_precision computeEulerdigits4(const uintmax_t precision)
{
    const uintmax_t EXTRA = 1;
    const uintmax_t n = (uintmax_t)ceil((precision * log(10) + log(3.14159265)) / 4);
    const double nd = ((precision * log(10) + log(3.14159265358979323846)) / 4);
    const uintmax_t k = uintmax_t(ceil(n * 3.5911214766686221366));
    const size_t workprec = (size_t)ceil(precision + EXTRA + log(k));
    const int_precision nsq(n * n);
    int_precision p, q, r, u;
    float_precision fp, fq;

    std::vector<int_precision> powsq(k / 2 + 2);
    powsq[0] = int_precision(1);
    powsq[1] = nsq;

    binarysplittingEuler4(0, k, powsq, p, q, r, u);

    // Finalize Calculation
    q *= r + q * q;
    fp.precision(workprec);
}
```

# Fast Computation of Math Constants in Arbitrary Precision

```
fq.precision(workprec);
fp = float_precision(u, workprec);
fq = float_precision(q, workprec);
fp -= fq * log(float_precision(n, workprec));
fp /= fq;
fp.precision(precision);

return fp;
}
```

And the threaded four-variable version. That is calling the standard four-variable version from each thread. Notice that the two threads don't share the powers variable to avoid hazard conditions arising.

## Source of the threaded four-variable Binary splitting method

```
// Driver function for the Euler-Mascheroni 4 variable Binary Splitting method
// using two threads.
//
static float_precision computeEulerdigitsThread4(const uintmax_t precision)
{
    const uintmax_t EXTRA = 1;
    const uintmax_t n = (uintmax_t)ceil((precision * log(10) +
log(3.14159265358979323846)) / 4);
    const uintmax_t k = uintmax_t(ceil(n * 3.5911214766686221366));
    const size_t workprec = (size_t)ceil(precision + EXTRA + 4 * log(k));
    const int_precision nsq(n * n);
    int_precision p, q, r, u;
    int_precision pp, qq, rr, uu, tmp;
    float_precision fp, fq;

    // Inialize the powsq array for the first hthread
    std::vector<int_precision> powsq(k / 2 + 2);
    powsq[0] = int_precision(1);
    powsq[1] = nsq;

    // Initialize the powsq array for the second thread
    std::vector<int_precision> powsq2(k / 2 + 2);
    powsq2[0] = int_precision(1);
    powsq2[1] = nsq;

    std::thread first([=, &p, &q, &r, &u, &powsq](){
        [a..k/2]
        binariesplittingEuler4(0, k / 2, powsq, p, q, r, u); }); // interval

    std::thread second([=, &pp, &qq, &rr, &uu, &powsq2](){
        [k/2..k]
        binariesplittingEuler4(k / 2, k, powsq2, pp, qq, rr, uu); }); // interval

    first.join();
    second.join();

    // Reconstruct interval [a..b] and return updated p,q,t,u
    const int_precision ip(ipow(powsq[1], int_precision(k/2 - 0)));
    int_precision qmb2 = qq; // qq
    qmb2 *= qmb2; // qq^2
    int_precision qmb3 = qmb2 * qq; // qq^3
    // Update u
    u = u * qmb3 + ip * (p * qq * rr + q * uu);
    // Update p
    p *= qq; p += q * pp;
    // Update q
    q *= qq;
    // Update r
    r *= qq * qq; r += ip * rr;

    // Finalize Calculation
    q *= r + q * q;
    fp.precision(workprec);
}
```

# Fast Computation of Math Constants in Arbitrary Precision

```
fq.precision(workprec);  
fp = float_precision(u, workprec);  
fq = float_precision(q, workprec);  
fp -= fq * log(float_precision(n, workprec));  
fp /= fq;  
fp.precision(precision);  
return fp;  
}
```

## Performance of Euler-Mascheroni constant

The performance chart clearly shows that the binary splitting method is superior for computing the Euler-Mascheroni constant. The traditional Brent-McMillan (Euler) and Brent Summation trick (Euler Brent) methods can't be recommended for fast computation. However, the Brent Summarization trick is a significant improvement over the standard Brent-McMillan formula.

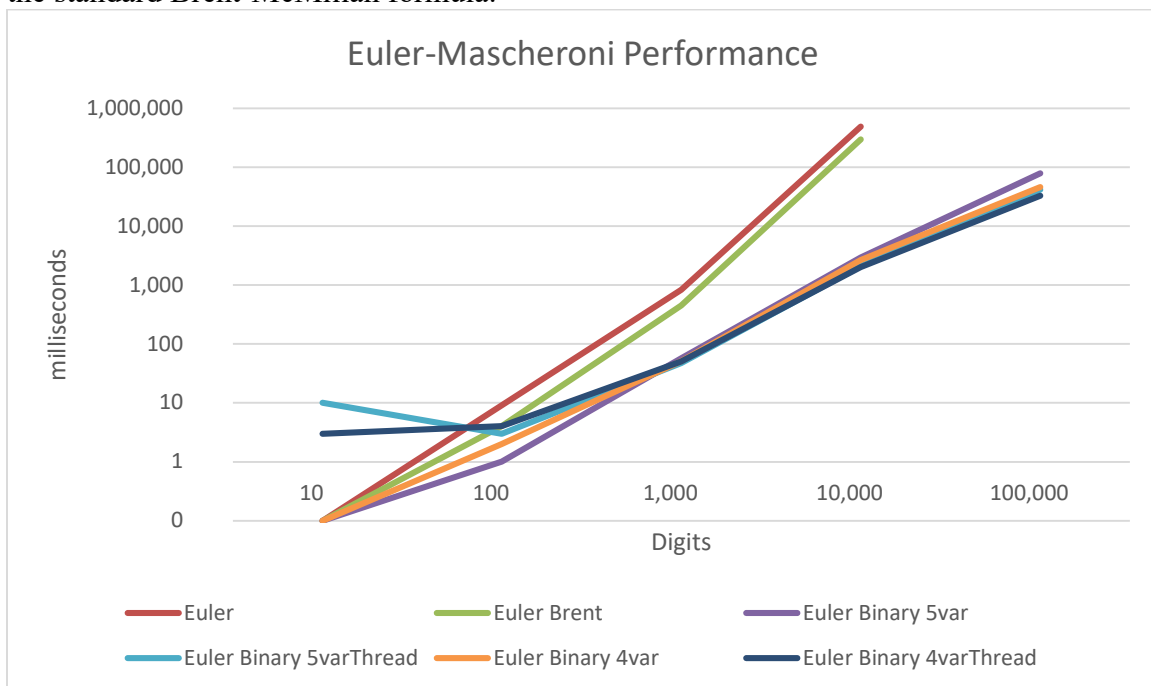


Figure 1. Euler-Mascheroni Performance

The Binary Splitting method, as recommended in [4], is superior to the other methods, and the reduced four variables ((Euler Binary 4var) is faster than the five-variable method (Euler Binary 5var). When applying a simple 2-way threading, the performance increases again.

See the table of actual performance, which is 10 to 100,000 digits.

Euler-Mascheroni Performance Result. All times are in msec					
Digits	10	100	1,000	10,000	100,000
Euler	-	7	830	490,784	-
Euler Brent	-	4	451	298,059	-
Euler Binary 5var	-	2	57	2,928	79,085

# Fast Computation of Math Constants in Arbitrary Precision

---

<b>Euler Binary 5varThread</b>	10	4	47	2,091	41,825
<b>Euler Binary 4var</b>	-	2	51	2,648	46,082
<b>Euler Binary 4varThread</b>	3	3	50	2,024	32,775

Table 1. Euler-Mascheroni Performance results.

Table 1 confirms that binary splitting outperforms traditional Brent–McMillan approaches, especially for higher digit counts.

## ***Recommendation for the Euler-Mascheroni constant***

As shown in [4], the four-variable version enhances speed by reducing memory operations. Based on the performance chart above, I recommend the Binary splitting method (with four variables). For digits above 1,000 digits, the threaded binary splitting version outperforms the non-threaded version.

## Catalan's constant G

The Catalan constant G is defined as:

$$G = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)^2} \quad (15)$$

The Catalan constant is  $\sim 0.9159655941772\dots$

This series also converges slowly. However, there are several alternative methods to consider.

- Ramanujan method I
- Ramanujan method II
- Broadhurst series
- Binary splitting method (ref. [4])
  - Lucas(2000)
  - Guillera (2008)
  - Guillera (2019)
  - Pilehrood (2010)
  - Zuniga (2023)

## Ramanujan's method I

This is one of the many Ramanujan series for fast calculation of the Catalan constant.

$$G = \frac{\pi}{8} \ln(2 + \sqrt{3}) + \frac{3}{8} \sum_{n=0}^{\infty} \frac{(n!)^2}{(2n)!(2n+1)^2} \quad (16)$$

To achieve  $P$ , decimal precision, we need to take  $P \frac{\ln(10)}{\ln(4)}$  terms of the series, and we can use Horner's schema to summarize the series efficiently. One of the drawbacks of this method is that we need to calculate  $\pi$ ,  $\ln(2+\sqrt{3})$ , and  $\sqrt{3}$  to arbitrary precision. Horner's schema looks like this:

$$1 + \frac{1}{2 \cdot 3} \left( \frac{1}{3} + \frac{2}{2 \cdot 5} \left( \frac{1}{5} + \frac{3}{2 \cdot 7} \left( \frac{1}{7} + \dots \right) \right) \right) \quad (17)$$

And the algorithm for the Horner schema sum.

```
Sum=0
For(k=1;k<=n;++k)
    Sum+=1/(2k+1)
```

# Fast Computation of Math Constants in Arbitrary Precision

$$\text{Sum} *= k / (2(2k+1))$$

Algorithm 5. Horner's evaluation of the fractional sum

Source Ramanujan's I

```
static float_precision RamanujanCatalan1(const size_t precision)
{
    const uintmax_t n = (uintmax_t)ceil((precision * log(10) ) / log(4));
    const size_t workprec = (size_t)ceil(precision+1);
    const float_precision c1(1), c2(2);
    uintmax_t k;
    float_precision res(0, workprec);
    float_precision sum(0, workprec), fpk(0, workprec);

    for (k = n; k>0; --k)
    {
        fpk = float_precision(2 * k + 1, workprec); // (2k+1)
        sum += c1 / fpk; // 1/(2k+1)+sum
        fpk.adjustExponent(+1); // 2(2k+1)
        fpk = float_precision(k, workprec) / fpk; // k/(2(2k+1))
        sum *= fpk;
    }
    sum += c1;
    sum *= float_precision(3); // 3*sum
    sum.adjustExponent(-3); // 3*sum/8

    res = _float_table(_SQRT3, precision); // sqrt(3)
    res += c2; // 2+sqrt(3)
    res = log(res); // log(2+sqrt(3))
    res *= _float_table(_PI, precision); // pi*log(2+sqrt(3))
    res.adjustExponent(-3); // pi*log(2+sqrt(3))/8

    res += sum;
    res.precision(precision);
    return res;
}
```

## Ramanujan's method II

Another of Ramanujan's methods is based on this formula:

$$G = \sum_{k=0}^{\infty} \frac{(k!)^2}{(2k+1)!} \cdot 2^{k-1} \sum_{j=0}^k \frac{1}{2j+1} \quad (18)$$

In [5], Free uses this formula and Brent summation trick to obtain the following algorithm:

Algorithm for Ramanujan's II

$B_0=0.5, C_0=0.5, G_0=0.5$

For( $k=1; k \leq n; ++k$ )

$\text{tmp}=k/(2k+1)$

$B_k=B_{k-1}*\text{tmp}$

$C_k=C_{k-1}*\text{tmp}+B_k/(2k+1)$

$G_k=G_{k-1}+C_k$

Algorithm 6. Ramanujan II method.

# Fast Computation of Math Constants in Arbitrary Precision

We need a little bit more to achieve  $P$ , decimal precision compare to the first method. In [5], they find that we need to take  $P \frac{\ln(10)}{\ln(2)}$  terms to reach the desired precision.

## Source Ramanujan's II

```
static float_precision RamanujanCatalan2(const size_t precision)
{
    const uintmax_t n = (uintmax_t)ceil((precision * log(10)) / log(2));
    const size_t workprec = (size_t)ceil(precision+1);
    uintmax_t k;
    float_precision res(0.5, workprec);
    float_precision fpk(0, workprec), bk(0.5, workprec), ck(0.5, workprec),
    tmp(0, workprec);

    for (k = 1; k <= n; ++k)
    {
        fpk = float_precision(2 * k + 1, workprec); // (2k+1)
        fpk = _float_precision_inverse(fpk); // 1/(2k+1)
        tmp = float_precision(k) * fpk; // k/(2k+1)
        bk *= tmp; // bk=bk*(k/(2k+1))
        ck *= tmp; // ck=ck*(k/(2k+1))
        ck += bk*fpk; // ck+=bk/(2k+1)
        res += ck; // res+=ck
    }

    res.precision(precision);
    return res;
}
```

## Broadhurst series

Broadhurst series has a faster convergence rate than Ramanujan's series at the expense of higher complexity.

$$G = 3 \sum_{k=0}^{\infty} \frac{1}{16^k} \sum_{i=0}^7 \frac{a_i}{(8k+i)^2} - 2 \sum_{k=0}^{\infty} \frac{1}{4096^k} \sum_{i=0}^7 \frac{b_i}{(8k+i)^2} \quad (19)$$

Where:

$$a_i=(0,1/2,-1/2,1/4,0,-1/8,1/8,-1/16) \text{ and} \\ b_i=(0,1/8,1/16,1/64,0,-1/512,-1/1024,-1/4096).$$

For a precision  $P$ , we need only to take  $\left\lceil P \frac{\ln(10)}{\ln(16)} \right\rceil$  terms to reach the desired precision of the first series and  $\left\lceil P \frac{\ln(10)}{\ln(4096)} \right\rceil$  for the second series. However, each term is also 6 times more complicated than the Ramanujan I series. In [3], they state that due to the extra complexity, it is not worth implementing it. However, I found that an efficient implementation of the Broadhurst series results in higher performance than the Ramanujan series for the Catalan constant.

## Source Broadhurst

```
static float_precision BroadhurstCatalan(const size_t precision)
{
    const uintmax_t n1 = (uintmax_t)ceil((precision * log(10)) / log(16));
    const uintmax_t n2 = (uintmax_t)ceil((precision * log(10)) / log(4096));
    const size_t workprec = (size_t)ceil(precision);
```

# Fast Computation of Math Constants in Arbitrary Precision

```
const float_precision c0(0), c1(1), c3(3);
static const float_precision ai[] = {0,1.0/2,-1.0/2,1.0/4,0,-1.0/8,1.0/8,-1.0/16};
static const float_precision bi[] = {0,1.0/8,1.0/16,1.0/64,0,-1.0/512,-1.0/1024,-
1.0/4096};
uintmax_t k;
float_precision res(0, workprec);
float_precision fpk(0, workprec), tmp(0, workprec);
float_precision sum1(0,workprec), sum2(0,workprec), suma(0, workprec), sumb(0,
workprec);

for (k = 0; k <= n1; ++k)
{
    suma = c0; sumb = c0; // zero inner suma and sumb
    for (int i = 1; i <= 7; ++i)
    {
        if (ai[i].iszero())
            continue;
        fpk = float_precision(8 * k + i); // (8k+i)
        fpk = fpk.square(); // (8k+i)^2
        fpk = _float_precision_inverse(fpk); // 1/(8k+i)^2
        suma += ai[i] * fpk; // ai/(8k+i)^2
        if(k<=n2)
            sumb += bi[i] * fpk; // bi/(8k+i)^2
    }
    tmp = c1;
    tmp.adjustExponent(-4 * k); // Divide with 16^k
    sum1 += tmp * suma; // Add to outer sum1
    if (k <= n2) // is precision reach for sum2
    { // continue adding to sum2
        tmp = c1;
        tmp.adjustExponent(-12 * k); // Divide with 4096^k
        sum2 += tmp * sumb; // Add to outer sum2
    }
}
sum1 *= c3; // sum1*=3
sum2.adjustExponent(+1); // sum2*2
res = sum1 - sum2;
res.precision(precision);
return res;
}
```

## *The Binary Splitting method for the Catalan constant.*

Several methods are to be considered for the Binary splitting method for the Catalan Constant (ref. [3] & [4]).

- 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.

# Fast Computation of Math Constants in Arbitrary Precision

---

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.

Pilehood (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 Pilehood 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:

$$G = \frac{1}{64} \sum_{k=1}^{\infty} \frac{(-1)^{k-1} 2^{8k} (40k^2 - 24k + 3) (2k)!^3 k!^2}{k^3 (2k-1) (4k)!^2} \quad (20)$$

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
    
```

Algorithm 7. Lupas 2000 binary splitting method for the Catalan constant.

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

# Fast Computation of Math Constants in Arbitrary Precision

In the end, you find  $G$  by:

$$G = \frac{P(0,n)+19Q(0,n)}{18Q(0,n)} + O(4^{-n}) \quad (21)$$

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

$$n = \left\lceil P \frac{\ln(10)}{\ln(4)} \right\rceil \quad (22)$$

In [4], they found the linearly convergent cost to be  $\sim 11.5$ , which is not as fast as the Guillera or Pilehrood methods.

## Source Catalan-Lupas binary splitting

```
static void binarysplittingLupas(const uintmax_t a, const uintmax_t b, int_precision& p,
int_precision& q, int_precision& r )
{
    uintmax_t mid;
    int_precision pp, qq, rr;

    if (a + 1 == b)
    {
        // No 64-bit overflow checking
        const uintmax_t b2 = b * b;
        const uintmax_t b4p1 = 4 * b + 1;
        const uintmax_t b4p3 = 4 * b + 3;

        // Build q
        q = int_precision(b4p1*b4p1); // (4b+1)^2b
        q *= int_precision(b4p3*b4p3); // (4b+1)^2(4b+3)^2
        // Build r
        r = int_precision(b2); // b^2
        r *= int_precision(32*b*(2*b-1)); // 32*(2b-1)*b^3
        //Build p
        p = r; // 32*(2b-1)*b^3
        p *= int_precision(40*b2+56*b+19); // (32*(2b-1)*b^3)(40b^2+56b+19)
        if (b & 0x1)
            p.sign(-1); // (32*(2b-1)*b^3)(40b^2+56b+19)(-1)^b
        return;
    }

    mid = (a + b) / 2;
    binarysplittingLupas(a, mid, p, q, r); // interval [a..mid]
    binarysplittingLupas(mid, b, pp, qq, rr ); // interval [mid..b]
    // Reconstruct interval [a..b] and return updated p,q,r
    p = p * qq + pp * r;
    q *= qq;
    r *= rr;
    return;
}

// Lupas 2000
static float_precision LupasCatalanConstant(const uintmax_t precision)
{
    const uintmax_t n = (uintmax_t)ceil((precision * log(10) ) / log(4));
    int_precision p, q, r;
    float_precision fp, fq;

    fp.precision(precision + 1);
    fq.precision(precision + 1);

    binarysplittingLupas(0, n, p, q, r);
}
```

# Fast Computation of Math Constants in Arbitrary Precision

```
p += int_precision(19) * q;  
q *= int_precision(18);  
fp = float_precision(p, precision + 1);  
fq = float_precision(q, precision + 1);  
fp /= fq;  
fp.precision(precision);  
return fp;  
}
```

## Guillera Binary Splitting method

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

$$G = \frac{1}{2} \sum_{k=0}^{\infty} \frac{(-8)^k (3k+2)}{(2k+1)^3 \binom{2k}{k}^3} \quad (23)$$

Converting into a binary splitting method, they found in [4] that the linearly convergent cost is  $\sim 11.5$ , around the same as for the Lupas binary splitting method. In [4], they rewrote the formula to:

$$G = \frac{1}{2} \sum_{k=0}^{\infty} \frac{(-8)^k (3k+2) k!^6}{(2k+1)!^3} \quad (24)$$

And archive a linearly convergent cost of  $\sim 5.7$ , 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
```

Algorithm 8. Guillera 2008 binary splitting method for the Catalan constant.

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

In the end, you find  $G$  by:

$$G = 1 + \frac{1}{2} \frac{P(0,n)}{Q(0,n)} + O(8^{-n}) \quad (25)$$

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

$$n = \left\lceil P \frac{\ln(10)}{\ln(8)} \right\rceil \quad (26)$$

# Fast Computation of Math Constants in Arbitrary Precision

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

$$G = -\frac{1}{1024} \sum_{k=1}^{\infty} \frac{(-4096)^k (45136k^4 - 57184k^3 + 21240k^2 - 3160k + 165)}{k^3 (2k-1)^3} \left( \frac{(2k)!^6 (3k)!^3}{k!^3 (6k)!^3} \right) \quad (27)$$

This method has a linearly convergent cost of only  $\sim 4.2$ , 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)= 45136b4-57184b3+21240b2-3160b+165
Q(b-1,b)=-27(6b-1)3(6b-5)3
R(b-1,b)=512b3(2b-1)3
    
```

Algorithm 9. Guillera 2019 binary splitting method for the Catalan constant.

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

In the end, you find  $G$  by:

$$G = -\frac{1}{2} \frac{P(0,n)}{Q(0,n)} + O\left(\left(\frac{19683}{64}\right)^{-n}\right) \quad (28)$$

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

$$n = \left\lceil P \frac{\ln(10)}{\ln\left(\frac{19683}{64}\right)} \right\rceil \quad (29)$$

Source Catalan-Guillera binary splitting (2008)

```

static void binarysplittingGuillera2008(const uintmax_t a, const uintmax_t b, int_precision&
p, int_precision& q, int_precision& r)
{
    uintmax_t mid;
    int_precision pp, qq, rr;
    if (a + 1 == b)
    {
        // No overflow detection
        const uintmax_t b2 = b * b;
        const uintmax_t b2p1 = 2 * b + 1;
        r = int_precision(b2); // b^2
        r *= int_precision(b); // b^3
        p = r; // b^3
        p *= int_precision(3*b+2); // b^3(3b+2)
    }
}
    
```

# Fast Computation of Math Constants in Arbitrary Precision

```
        q = int_precision(b2p1*b2p1); // (2b+1)^2
        q *= int_precision(b2p1);    // (2b+1)^3
        q.sign(-1);                  // -(2b+1)^3
        return;
    }

    mid = (a + b) / 2;
    binarysplittingGuillera2008(a, mid, p, q, r); // interval [a..mid]
    binarysplittingGuillera2008(mid, b, pp, qq, rr); // interval [mid..b]
    // Reconstruct interval [a..b] and return updated p,q,r
    p = p * qq + pp * r;
    q *= qq;
    r *= rr;
    return;
}

static float_precision GuilleraCatalan2008(const uintmax_t precision)
{
    const uintmax_t n = (uintmax_t)ceil((precision * log(10)) / log(8));
    int_precision p, q, r;
    float_precision fp, fq;

    fp.precision(precision + 1);
    fq.precision(precision + 1);

    binarysplittingGuillera2008(0, n, p, q, r);

    fp = float_precision(p, precision + 1);
    fq = float_precision(q, precision + 1);
    fp /= fq; // P(0,n)/Q(0,n)
    fp.adjustExponent(-1); // 0.5P(0,n)/Q(0,n)
    fp += float_precision(1); // 1+0.5P(0,n)/Q(0,n)
    fp.precision(precision);
    return fp;
}
```

## Source Catalan-Guillera binary splitting (2019)

```
static void binarysplittingGuillera2019(const uintmax_t a, const uintmax_t b, int_precision&
p, int_precision& q, int_precision& r)
{
    uintmax_t mid;
    int_precision pp, qq, rr;
    if (a + 1 == b)
    {
        const uintmax_t b2 = b * b; // b^2
        const uintmax_t b2m1 = (2 * b - 1); // (2b-1)
        const uintmax_t b6m1 = (6 * b - 1); // (6b-1)
        const uintmax_t b6m5 = (6 * b - 5); // (6b-5)

        // Build p
        p = int_precision(b2); // p=b^2
        // 45'136b^4-57'184b^3+21'240b^2-3'160b+165
        p = p * p * int_precision(45'136)
            - p * int_precision(57'184 * b)
            + p * int_precision(21'240)
            - int_precision(3'160 * b-165);

        // Build q
        q = int_precision(b6m5*b6m5); // (6b-5)^2
        q *= int_precision(b6m5); // (6b-5)^3
        q *= int_precision(b6m1 * b6m1); // (6b-5)^3(6b-1)^2
        q *= int_precision(b6m1 * 27); // 27(6b-5)^3(6b-1)^3
        q.sign(-1); // -27(6b-5)^3(6b-1)^3

        // Build r
        r = int_precision(b2); // b^2
        r *= int_precision(512 * b); // 512b^3
        r *= int_precision(b2m1 * b2m1); // 512b^3(2b-1)^2
    }
}
```

# Fast Computation of Math Constants in Arbitrary Precision

```
        r *= int_precision(b2m1);           // 512b^3(2b-1)^3
        return;
    }

    mid = (a + b) / 2;
    binarysplittingGuillera2019(a, mid, p, q, r);           // interval [a..mid]
    binarysplittingGuillera2019(mid, b, pp, qq, rr);       // interval [mid..b]
    // Reconstruct interval [a..b] and return updated p,q,r
    p = p * qq + pp * r;
    q *= qq;
    r *= rr;
    return;
}

static float_precision GuilleraCatalan2019(const uintmax_t precision)
{
    const uintmax_t n = (uintmax_t)ceil((precision * log(10)) / log(19'683.0/64.0));
    int_precision p, q, r;
    float_precision fp, fq;

    fp.precision(precision + 1);
    fq.precision(precision + 1);

    binarysplittingGuillera2019(0, n, p, q, r);

    fp = float_precision(p, precision + 1);
    fq = float_precision(q, precision + 1);
    fp /= fq;           // P(0,n)/Q(0,n)
    fp.adjustExponent(-1); // 0.5P(0,n)Q(0,n)
    fp.change_sign();   // -0.5P(0,n)Q(0,n)
    fp.precision(precision);
    return fp;
}
```

## ***Pilehrood binary splitting method***

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

$$G = \frac{1}{64} \sum_{k=1}^{\infty} \frac{256^k (580k^2 - 184k + 15)}{k^3 (2k-1) \binom{6k}{3k} \binom{6k}{4k} \binom{4k}{2k}} \quad (30)$$

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.

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)= 580b2-184b+15
Q(b-1,b)=9(6b-1)2(6b-5)2
R(b-1,b)=32b3(2b-1)
```

Algorithm 10. Pilehrood 2010 short binary splitting method for the Catalan constant.

## Fast Computation of Math Constants in Arbitrary Precision

---

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

In the end, you find  $G$  by:

$$G = \frac{1}{2} \frac{P(0,n)}{Q(0,n)} + O\left(\left(\frac{729}{4}\right)^{-n}\right) \quad (31)$$

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

$$n = \left\lceil P \frac{\ln(10)}{\ln\left(\frac{729}{4}\right)} \right\rceil \quad (32)$$

Pilehrood also has a long version formula:

$$G = -\frac{1}{64} \sum_{k=1}^{\infty} \frac{(-256)^k (419840k^6 - 915456k^5 + 782848k^4 - 332800k^3 + 73256k^2 - 7800k + 315)}{k^3(2k-1)(4k-1)^2(4k-3)^2 \binom{8k}{4k}^2 \binom{2k}{k}} \quad (33)$$

This method has a linearly convergent cost of  $\sim 4.6$ , 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(419840b6 - 915456b5 + 782848b4 - 332800b3 + 73256b2 - 7800b + 315)
Q(b-1,b)=(8b-1)2(8b-3)2(8b-5)2(8b-7)2
R(b-1,b)=32b3(2b-1)(4b-1)2(4b-3)2
    
```

Algorithm 11. Pilehrood 2010 long binary splitting method for the Catalan constant.

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

In the end, you find  $G$  by:

$$G = -\frac{1}{2} \frac{P(0,n)}{Q(0,n)} + O(1024^{-n}) \quad (34)$$

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

$$n = \left\lceil P \frac{\ln(10)}{\ln(1024)} \right\rceil \quad (35)$$

# Fast Computation of Math Constants in Arbitrary Precision

## Source Catalan-Pilehrood binary splitting (2010)

```
static void binarysplittingPilehrood2010(const uintmax_t a, const uintmax_t b,
int_precision& p, int_precision& q, int_precision& r)
{
    uintmax_t mid;
    int_precision pp, qq, rr;
    if (a + 1 == b)
    {
        const uintmax_t b64m32 = (64 * b - 32); // (64b-32)
        const uintmax_t b6m1 = (6 * b - 1); // (6b-1)
        const uintmax_t b6m5 = (6 * b - 5); // (6b-5)

        // Calculate p
        if(b<=178'338'809) // Check for overflow
            p = int_precision((580*b-184)*b+15); // 580b^2-184b+15
        else
        {
            // Handle b >178'338'809.
            p = int_precision(580*b-184); // 580b-184
            p *= int_precision(b); // (580b-184)b
            p += int_precision(15); // (580b-184)b+15
        }

        // Calculate q
        if (b <= 6'306) // Check for overflow
            q = int_precision(b6m1*b6m1*b6m5*b6m5*9);
        else
        {
            if (b <= 238'609'294)
            {
                q = int_precision(b6m5*b6m5*9); // 9(6b-5)^2
                q *= int_precision(b6m1*b6m1); // 9(6b-1)^2(6b-5)^2
            }
            else
            {
                // no overflow
                q = int_precision(b6m5*9); // 9(6b-5)^2
                q *= int_precision(b6m5); // 9(6b-5)^2
                q *= int_precision(b6m1); // 9(6b-5)^2(6b-1)
                q *= int_precision(b6m1); // 9(6b-1)^2(6b-5)^2
            }
        }

        // Calculate r
        if (b <= 23'170)
            r = int_precision(b*b*b*b64m32); // 32b^3(2b-1)
        else
        {
            if (b <= 2'642'245)
            {
                r = int_precision(b*b*b); // b^3
                r *= int_precision(b64m32); // 32b^3(2b-1)
            }
            else
            {
                // b<536'870'912 -- 4'294'967'296 otherwise undetected overflow
                r = int_precision(b*b); // b^2
                r *= int_precision(b*b64m32); // 32b^3(2b-1)
            }
        }
    }
    return;
}

mid = (a + b) / 2;
binarysplittingPilehrood2010(a, mid, p, q, r); // interval [a..mid]
binarysplittingPilehrood2010(mid, b, pp, qq, rr); // interval [mid..b]
// Reconstruct interval [a..b] and return updated p,q,r
p = p * qq + pp * r;
q *= qq;
r *= rr;
```

# Fast Computation of Math Constants in Arbitrary Precision

```
    return;
}

static float_precision PilehroodCatalan2010(const uintmax_t precision)
{
    const uintmax_t n = (uintmax_t)ceil((precision * log(10)) / log(729.0 / 4.0));
    int_precision p, q, r;
    float_precision fp, fq;

    fp.precision(precision + 1);
    fq.precision(precision + 1);

    binarysplittingPilehrood2010(0, n, p, q, r);

    fp = float_precision(p, precision + 1);
    fq = float_precision(q, precision + 1);
    fp /= fq; // P(0,n)/Q(0,n)
    fp.adjustExponent(-1); //0.5P(0,n)/Q(0,n)
    fp.precision(precision);
    return fp;
}
```

## Source Catalan-Pilehrood long binary splitting (2010)

```
static void binarysplittingPilehrood2010long(const uintmax_t a, const uintmax_t b,
int_precision& p, int_precision& q, int_precision& r)
{
    uintmax_t mid;
    int_precision pp, qq, rr;
    if (a + 1 == b)
    {
        const uintmax_t b2m1 = (2 * b - 1); // (2b-1)
        const uintmax_t b4m1 = (4 * b - 1); // (4b-1)
        const uintmax_t b4m3 = (4 * b - 3); // (4b-3)
        const uintmax_t b8m1 = (8 * b - 1); // (8b-1)
        const uintmax_t b8m3 = (8 * b - 3); // (8b-3)
        const uintmax_t b8m5 = (8 * b - 5); // (8b-5)
        const uintmax_t b8m7 = (8 * b - 7); // (8b-7)
        const int_precision bip(b);

        // Build p using Horner schema
        p = int_precision(419'840 * intmax_t(b) - 915'456);
        p *= bip; p += int_precision(782'848);
        p *= bip; p -= int_precision(332'800);
        p *= bip; p += int_precision(73'256);
        p *= bip; p -= int_precision(7'800);
        p *= bip; p += int_precision(315);
        if (b & 0x1)
            p.change_sign(); // p*=(-1)^b

        q = int_precision(b8m1*b8m1); // (8b-1)^2
        q *= int_precision(b8m3*b8m3); // (8b-1)^2(8b-3)^2
        q *= int_precision(b8m5*b8m5); // (8b-1)^2(8b-3)^2(8b-5)^2
        q *= int_precision(b8m7*b8m7); // (8b-1)^2(8b-3)^2(8b-5)^2(8b-7)^2

        r = int_precision(b*b); // b^2
        r *= int_precision(b*b2m1*32); // 32b^3(2b-1)
        r *= int_precision(b4m1 * b4m1); // 32b^3(2b-1)(4b-1)^2
        r *= int_precision(b4m3 * b4m3); // 32b^3(2b-1)(4b-1)^2(4b-3)^2
        return;
    }

    mid = (a + b) / 2;
    binarysplittingPilehrood2010long(a, mid, p, q, r); // interval [a..mid]
    binarysplittingPilehrood2010long(mid, b, pp, qq, rr); // interval [mid..b]
    // Reconstruct interval [a..b] and return updated p,q,r
    p = p * qq + pp * r;
    q *= qq;
    r *= rr;
}
```

# Fast Computation of Math Constants in Arbitrary Precision

```

    return;
}

static float_precision PilehroodCatalan2010long(const uintmax_t precision)
{
    const uintmax_t n = (uintmax_t)ceil((precision * log(10)) / log(1024));
    int_precision p, q, r;
    float_precision fp, fq;

    fp.precision(precision + 1);
    fq.precision(precision + 1);

    binarysplittingPilehrood2010long(0, n, p, q, r);

    fp = float_precision(p, precision + 1);
    fq = float_precision(q, precision + 1);
    fp /= fq; // P(0,n)/Q(0,n)
    fp.adjustExponent(-1); // 0.5P(0,n)/Q(0,n)
    fp.change_sign(); // -0.5P(0,n)/Q(0,n)
    fp.precision(precision);
    return fp;
}

```

## Zuniga binary splitting method

A freshly new method from 2023 by Zuniga. [4]

$$G = \frac{1}{768} \sum_{k=1}^{\infty} \frac{(-40)^k (-43203456^6 + 92809152k^5 - 76613904^4 + 30494304^3 - 6004944k^2 + 536620k - 17325)}{k^3 (2k-1)(3k-1)(3k-2)(6k-1)(6k-5) \binom{5k}{k} \binom{10}{5k} \binom{12k}{6k}} \quad (36)$$

Which have a linearly convergent cost of  $\sim 3.4$ , which is slightly higher than the Pilehrood short versions but lower than the Pilehrood long version.

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:

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

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

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

Algorithm 12. Zuniga 2023 binary splitting method for the Catalan constant.

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

In the end, you find  $G$  by:

$$G = \frac{1}{6} \frac{P(0,n)}{Q(0,n)} + O(12500^{-n}) \quad (37)$$

# Fast Computation of Math Constants in Arbitrary Precision

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

$$n = \left\lceil P \frac{\ln(10)}{\ln(12500)} \right\rceil \quad (38)$$

## Source Catalan-Zuniga binary splitting (2023)

```
// Zuniga 2023 Binary Splitting for the Catalan Constant
static void binarysplittingZuniga2023(const uintmax_t a, const uintmax_t b, int_precision&
p, int_precision& q, int_precision& r)
{
    uintmax_t mid;
    int_precision pp, qq, rr;
    if (a + 1 == b)
    {
        const int_precision bip(b);
        const uintmax_t b3 = b * 3;
        const uintmax_t b6 = b * 6;
        const uintmax_t b10 = b * 10;
        const uintmax_t b12 = b * 12;

        // p
        p = (((((int_precision(43203456) * bip - int_precision(92809152))
            * bip + int_precision(76613904))
            * bip - int_precision(30494304))
            * bip + int_precision(6004944))
            * bip - int_precision(536620))
            * bip + int_precision(17325));
        // q
        q = int_precision(5 * (b10 - 9))
            * int_precision((b10 - 7) * (b10 - 3))
            * int_precision((b10 - 1) * (b12 - 11))
            * int_precision((b12 - 7) * (b12 - 5))
            * int_precision(b12 - 1);
        // r
        r = int_precision(b * b )
            * int_precision(b*(2*b-1))
            * int_precision((b3 - 2) * (b3 - 1))
            * int_precision((b6 - 5) * (b6 - 1))
            * int_precision(-128);

        return;
    }

    mid = (a + b) / 2;
    binarysplittingZuniga2023(a, mid, p, q, r); // interval [a..mid]
    binarysplittingZuniga2023(mid, b, pp, qq, rr); // interval [mid..b]
    // Reconstruct interval [a..b] and return updated p,q,r
    p = p * qq + pp * r;
    q *= qq;
    r *= rr;
    return;
}

static float_precision ZunigaCatalan2023(const uintmax_t precision)
{
    const uintmax_t n = (uintmax_t)ceil((precision * log(10)) / log(12500));
    int_precision p, q, r;
    float_precision fp, fq;

    fp.precision(precision + 1);
    fq.precision(precision + 1);

    binarysplittingZuniga2023(0, n, p, q, r);

    fp = float_precision(p, precision + 1);
    fq = float_precision(q*6, precision + 1);
    fp /= fq; // P(0,n)/(6*Q(0,n))
    fp.precision(precision);
    return fp;
}
```

}

## 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.

Method	Implementation	Error	N(P), <i>P=precision</i>
<b>Ramanujan-I</b>	Series	$O(4^{-n})$	1.661P
<b>Ramanujan-II</b>	Series	$O(2^{-4})$	3.322P
<b>Broadhurst</b>	Series	$O(16^{-n})$	0.830P
<b>Lupas</b>	Binary Splitting	$O(4^{-n})$	1.661P
<b>Guillera-2008</b>	Binary-Splitting	$O(8^{-n})$	1.107P
<b>Guillera-2019</b>	Binary-Splitting	$O\left(\left(\frac{19683}{64}\right)^{-n}\right)$	0.402P
<b>Pilehrood-short</b>	Binary-Splitting	$O\left(\left(\frac{729}{4}\right)^{-n}\right)$	0.442P
<b>Pilehrood-long</b>	Binary-Splitting	$O(1024^{-n})$	0.332P
<b>Zuniga</b>	Binary-Splitting	$O(12500^{-n})$	0.244P

Table 2. Comparison of the Catalan Methods.

Not surprisingly, performance depends heavily on the convergence speed and implementation type, e.g., a Series or binary splitting method, as shown in the next section.

## Catalan Constant Performance

Not surprisingly, the linearly convergent cost predicts the performance of the method. The clear winner is the Pilehrood binary splitting method from 2010. It outperforms the others significantly. Furthermore, a two-way multi-threaded version further improves the performance by 30-40%. The Pilehrood method is 40-50% faster than Guillera 2019 method and 90-100% faster than Guillera 2008 method. Comparing Pilehrood and Lupas, Pilehrood is more than 5 times faster. If we compare the Binary splitting method against the classical series formula, the binary splitting version is several magnitudes faster. Among the classical series, the Broadhurst method is by far the fastest.

# Fast Computation of Math Constants in Arbitrary Precision

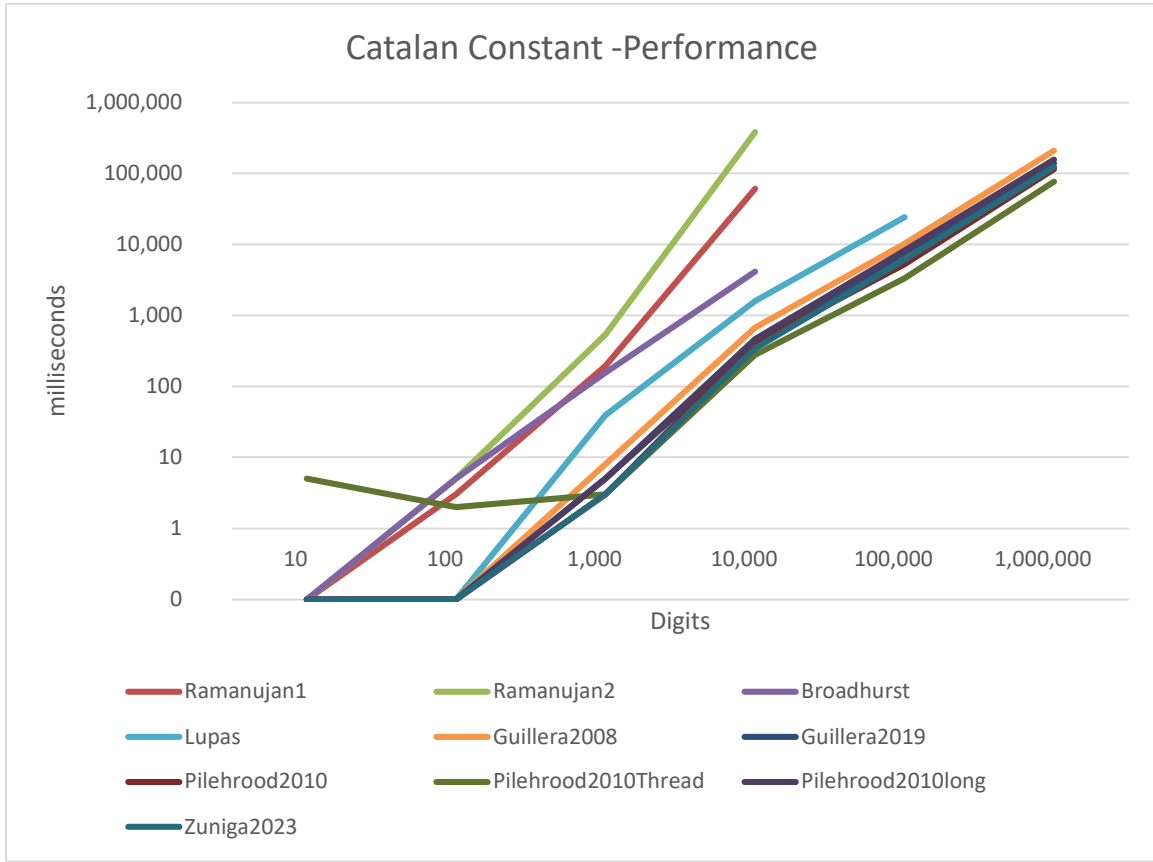


Figure 2. Catalan Constant Performance chart

Digits	10	100	1,000	10,000	100,000	1,000,000
<b>Ramanujan1</b>	0	3	194	61,097		
<b>Ramanujan2</b>	0	5	534	385,528		
<b>Broadhurst</b>	0	5	155	4,142		
<b>Lupas</b>	0	0	39	1,592	24,183	
<b>Guillera2008</b>	0	0	8	673	10,198	210,692
<b>Guillera2019</b>	0	0	5	388	7,618	138,037
<b>Pilehrood2010</b>	0	0	3	399	5,225	114,603
<b>Pilehrood2010Thread</b>	5	2	3	279	3,314	76,399
<b>Pilehrood2010long</b>	0	0	5	463	8,220	157,580

# Fast Computation of Math Constants in Arbitrary Precision

---

Zuniga2023	0	0	3	341	6,043	123,388
------------	---	---	---	-----	-------	---------

Table 3. Table of performance of the Catalan Methods.

Notice that even when Zuniga2023 needs fewer splitting compared to Pilehrood 2010, it is not faster, mainly due to the more complex variable for P and Q. That growth to approximately double the number of digits compared to the Pilehrood 2010 method P and Q variable. Notice that the Pilehrood 2010 threaded version is 30-40% faster than the non-threaded version.

## ***Recommendation for the Catalan constant***

Based on the performance chart and ease of implementation, I recommend the Pilehrood 2010 short version as the preferred binary splitting method. Use or implement a threaded version of the Pilehrood method if performance is required. Creating a 2, 3, 4, or more core-threaded version of the binary splitting method is easy. If we only want a classical method, I recommend the Broadhurst method.

## Apéry's constant $\zeta(3)$

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 methods 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 is given by Amdeberhan-Zeilberger back in 1997.

$$\zeta(3) = \frac{1}{64} \sum_{k=0}^{\infty} \frac{(-1)^k (205^2 + 250k + 77)(k!)^{10}}{(2k+1)^5} \quad (39)$$

By now, we should have learned that the most efficient computation is using the binary splitting method. Amdeberhan-Zeilberger algorithm is layout below.

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)
```

# Fast Computation of Math Constants in Arbitrary Precision

And:  
 $P(b-1,b)=(-1)^b(205b^2+250b+77)b^5$   
 $Q(b-1,b)=32(2b+1)^5$   
 $R(b-1,b)=b^5$

Algorithm 13. Amdeberhan-Zeilberger binary splitting algorithm for zeta(3).

And then

$$\zeta(3) = \frac{P(0,n)+77Q(0,n)}{64Q(0,n)} + O(1024^{-n}) \quad (40)$$

Which have a linearly convergent cost of  $\sim 2.89$ , slightly higher than the next Wedeniwski method.

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

$$n = \left\lceil P \frac{\ln(10)}{\ln(1024)} \right\rceil \quad (41)$$

Source Amdeberhan-Zeilberger Binary splitting method

Notice that we try to use as much native calculation as possible, assuming a 64-bit environment.

```
// Zeta(3) Amdeberhan - Zeilberger(1997)
static void binariesplittingAperyZeilberger(const uintmax_t a, const uintmax_t b,
int_precision & p, int_precision & q, int_precision & r)
{
    uintmax_t mid;
    int_precision pp, qq, rr;

    if (a + 1 == b)
    {
        const uintmax_t b2p1 = 2 * b + 1;
        const uintmax_t bsq = b * b;

        // Compute r
        if (b <= 7'131)
            r = b * b * b * b * b; // No overflow if b<=7'131
        else
            if (b <= 2'642'245)
            {
                // Max b^3
                r = b * b * b;
                r *= b * b;
            }
            else
            {
                // Max b^2
                r = b * b; r *= r; r *= b;
            }

        // Compute q
        if (b < 1'782)
            q = b2p1 * b2p1 * b2p1 * b2p1 * b2p1 * 32; // No overflow if
b<=1'782
        else
            if (b <= 1'321'122)
            {
                q = b2p1*b2p1*b2p1; q *= 32 *b2p1* b2p1;// 32(2b+1)^5
            }
    }
}
```

# Fast Computation of Math Constants in Arbitrary Precision

```
        else
        {
            q = b2p1 * b2p1; q *= q; q *= 32 * b2p1; // 32(2b+1)^5
        }

        // Compute p
        p = r;
        if(b<=299'973'527)
            p *= 205 * bsq + 250 * b + 77; // (205b^2+250b+77)b^5
        else
        {
            rr = 205 * b + 250; rr *= b; rr += 77; p *= rr;
        }
        if (b & 0x1)
            p.change_sign() // (205b^2+250b+77)b^5*(-1)^b
        return;
    }

    mid = (a + b) / 2;
    binarysplittingAperyZeilberger(a, mid, p, q, r); // interval [a..mid]
    binarysplittingAperyZeilberger(mid, b, pp, qq, rr); // interval [mid..b]
    // Reconstruct interval [a..b] and return updated p,q,r,t,u
    p = p * qq + pp * r;
    q *= qq;
    r *= rr;
    return;
}

static float_precision computeAperydigitsZeilberger(const uintmax_t precision)
{
    intmax_t kmax = intmax_t(ceil(precision * log(10) / log(1024)));
    const size_t workprec = (size_t)ceil(precision + 1 + log(kmax));
    int_precision p, q, r;
    float_precision fp, fq;

    binarysplittingAperyZeilberger(0, kmax, p, q, r);
    p += int_precision(77) * q;
    q *= int_precision(64);
    fp.precision(workprec);
    fq.precision(workprec);
    fp = float_precision(p, workprec);
    fq = float_precision(q, workprec);
    fp /= fq;
    fp.precision(precision);
    return fp;
}
```

## Source Threaded Amdeberhan-Zeilberger Binary splitting method

Only the driving function needs to be changed to a threaded version. This can be continued to create a three-way, four-way, or even higher-threaded version.

```
static float_precision computeAperydigitsZeilbergerThread(const uintmax_t precision)
{
    intmax_t kmax = intmax_t(ceil(precision * log(10) / log(1024)));
    const size_t workprec = (size_t)ceil(precision + 1 + log(kmax));
    int_precision p, q, r, pp, qq, rr;
    float_precision fp, fq;

    std::thread first([=, &p, &q, &r](){
        binarysplittingAperyZeilberger(0, kmax/2, p, q, r); }); // interval [a..k/2]

    std::thread second([=, &pp, &qq, &rr](){
        binarysplittingAperyZeilberger(kmax/2, kmax, pp, qq, rr); }); // interval
[k/2..k]

    first.join();
    second.join();
}
```

# Fast Computation of Math Constants in Arbitrary Precision

```
// Reconstruct interval [a..b] and return updated p,q,r
p = p * qq + pp * r;
q *= qq;
//r *= rr;      // not used in final calculation below

p += int_precision(77) * q;
q *= int_precision(64);
fp.precision(workprec);
fq.precision(workprec);
fp = float_precision(p, workprec);
fq = float_precision(q, workprec);
fp /= fq;
fp.precision(precision);
return fp;
}
```

## Wedeniowski series

This series was given by Amdeberhan-Zeilberger in 1997.

$$\zeta(3) = \frac{1}{24} \sum_{k=0}^{\infty} \frac{(-1)^k (126392k^5 + 412708k^4 + 531578k^3 + 336367k^2 + 104000k + 12)}{(3k+2)!(4k+3)!^3} \quad (42)$$

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:

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

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

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

Algorithm 14. Wedeniowski binary splitting algorithm for zeta(3).

And then

$$\zeta(3) = \frac{P(0,n) + 12463Q(0,n)}{10368Q(0,n)} + O(110592^{-n}) \quad (43)$$

It has a linearly convergent cost of  $\sim 2.78$  which is 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:

$$n = \left\lceil P \frac{\ln(10)}{\ln(110592)} \right\rceil \quad (44)$$

Source Wedeniowski binary splitting method

```
static void binariesplittingAperyWedeniowski(const uintmax_t a, const uintmax_t b,
int_precision& p, int_precision& q, int_precision& r)
{
    uintmax_t mid;
```

# Fast Computation of Math Constants in Arbitrary Precision

```
int_precision pp, qq, rr;

if (a + 1 == b)
{
    const uintmax_t b3p1 = 3 * b + 1;
    const uintmax_t b4p1 = 4 * b + 1;
    const uintmax_t b2m1 = 2 * b - 1;

    r = int_precision(b*b); // b^2
    r *= r; r *= int_precision(b); // b^5
    r *= b2m1 * b2m1; r *= b2m1; // b^5*(2b-1)^3
    q = b4p1 * (b4p1+2); q *= q * q; q *= 24 * b3p1*(b3p1+1);
    //24(3b+1)(3b+2)(4b+1)^3(4b+3)^3
    p = 126392 * b;
    p += 412708; p *= b;
    p += 531578; p *= b;
    p += 336367; p *= b;
    p += 104000; p *= b;
    p += 12463;
    p *= r; // b^5(2b-1)^3
    if (b & 0x1)
        p.change_sign();

    return;
}

mid = (a + b) / 2;
binarysplittingAperyWedeniowski(a, mid, p, q, r); // interval [a..mid]
binarysplittingAperyWedeniowski(mid, b, pp, qq, rr); // interval [mid..b]
// Reconstruct interval [a..b] and return updated p,q,r,t,u
p = p * qq + r * pp;
q *= qq;
r *= rr;
return;
}

static float_precision computeAperydigitsWedeniowski(const uintmax_t precision)
{
    intmax_t kmax = intmax_t(ceil(precision * log(10) / log(110592)));
    const size_t workprec = (size_t)ceil(precision + 1 + log(kmax));
    int_precision p, q, r;
    float_precision fp, fq;

    binarysplittingAperyWedeniowski(0, kmax, p, q, r);
    fp.precision(workprec);
    fq.precision(workprec);
    p += int_precision(12463) * q;
    q *= int_precision(10368);
    fp = float_precision(p, workprec);
    fq = float_precision(q, workprec);
    fp /= fq;
    fp.precision(precision);
    return fp;
}
```

## Zuniga series (v)

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

$$\zeta(3) = \frac{1}{24} \sum_{k=0}^{\infty} \frac{P(k)}{k^5(2k-1)(3k-1)(3k-2)(4k-1)(4k-3)(5k-1)(5k-2)(5k-3)(5k-4) \binom{3k}{k} \binom{6k}{3k} \binom{8k}{4k} \binom{9k}{3k} \binom{10}{5k}} \quad (45)$$

# Fast Computation of Math Constants in Arbitrary Precision

---

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 (v) 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)=250765325100000b11-1087318449630000b10+2067749814046250b9-
    2269551612681475b8+1592180015776565b7-746938801646725b6
    +238210943593421b5-51452348050672b4+7352050259484b3-660416507568b2
    +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)=b5(2b-1)(3b-2)(3b-1)(4b-3)(4b-1)(5b-4)(5b-3)(5b-2)(5b-1)
    
```

Algorithm 15. Zuniga (v) binary splitting algorithm for zeta(3).

And then

$$\zeta(3) = \frac{1}{24} \frac{P(0,n)}{Q(0,n)} + O(34828517376^{-n}) \quad (46)$$

It has a linearly convergent cost of  $\sim 2.3$ , which is lower than the Wedeniwski method.

For  $n$  terms, the error is  $O(34828517376^{-n})$  And for a given precision,  $P$  you need:

$$n = \left\lceil P \frac{\ln(10)}{\ln(34828517376)} \right\rceil \quad (47)$$

Source Zuniga (v) binary splitting method

```

static void binarySplittingAperyZunigaV(const uintmax_t a, const uintmax_t b, int_precision&
p, int_precision& q, int_precision& r)
{
    uintmax_t mid;
    int_precision pp, qq, rr;

    // Base case: interval [a..b) of length 1
    // Notice bmax is ~ 0.1*precision. 1G precision => bmax ~ 10^8. so b^2 is max using
    uintmax_t
    if (a + 1 == b) {
        const uintmax_t b3(3 * b);
        const uintmax_t b4(4 * b);
        const uintmax_t b5(5 * b);
        const uintmax_t b8(8 * b);
        const uintmax_t b9(9 * b);
        const uintmax_t b10(10 * b);
        const int_precision bb(b);
        const int_precision bp5(bb);
    }
}
    
```

## Fast Computation of Math Constants in Arbitrary Precision

```
// Compute q
q = int_precision(288 * (b8 - 7) * (b8 - 5) * (b8 - 3) * (b8 - 1)) *
    int_precision((b9 - 8) * (b9 - 7) * (b9 - 5)) *
    int_precision((b9 - 4) * (b9 - 2) * (b9 - 1)) *
    int_precision((b10 - 9) * (b10 - 7) * (b10 - 3) * (b10 - 1));

// Compute r
if (b <= 7'131)
    r = int_precision(b * b * b * b * b); // No overflow if b<=7'131
else
    if (b <= 2'642'245)
    {
        // Max b^3
        r = int_precision(b * b * b);
        r *= int_precision(b * b);
    }
    else
    {
        // Max b^2
        r = int_precision(b * b);
        r *= r;
        r *= bb;
    }
r *= int_precision(2 * b - 1) *
    int_precision((b3 - 2) * (b3 - 1)) *
    int_precision((b4 - 3) * (b4 - 1)) *
    int_precision((b5 - 4) * (b5 - 3)) *
    int_precision((b5 - 2) * (b5 - 1));

// Compute p
p = ((((((((((int_precision(250765325100000ull) *
    bb - int_precision(1087318449630000ull)) *
    bb + int_precision(2067749814046250ull)) *
    bb - int_precision(2269551612681475ull)) *
    bb + int_precision(1592180015776565ull)) *
    bb - int_precision(746938801646725ull)) *
    bb + int_precision(238210943593421ull)) *
    bb - int_precision(51452348050672ull)) *
    bb + int_precision(7352050259484ull)) *
    bb - int_precision(660416507568ull)) *
    bb + int_precision(33552610560ull)) *
    bb - int_precision(731566080ull);

    return;
}

mid = (a + b) / 2;
// Recursively compute left and right intervals
binarySplittingAperyZunigaV(a, mid, p, q, r);
binarySplittingAperyZunigaV(mid, b, pp, qq, rr);
// Merge results
p = p * qq + pp * r;
q *= qq;
r *= rr;

return;
}

static float_precision computeAperydigitsZunigaV(const uintmax_t precision)
{
    // Determine the number of terms to compute
    intmax_t kmax = intmax_t(ceil(precision * log(10) / log(34828517376)));
    const size_t workprec = (size_t)ceil(precision + 1 + log(kmax));
    int_precision p, q, r;
    float_precision fp, fq;

    // Compute the result
    binarySplittingAperyZunigaV(0, kmax, p, q, r);
    // Finalize the result
    q *= int_precision(24);
    fp.precision(workprec);
    fq.precision(workprec);
}
```

# Fast Computation of Math Constants in Arbitrary Precision

---

```

fp = float_precision(p, workprec);
fq = float_precision(q, workprec);
fp /= fq;
fp.precision(precision);
return fp;
}

```

## Zuniga series (vi)

Also, in 2023, Zuniga revealed another method with a computation cost of  $\sim 2$ .

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

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)=1565994397644288b11-6719460725627136b10+12632254526031264b9-
13684352515879536b8+9451223531851808b7-4348596587040104b6
+1352700034136826b5-282805786014979b4+38721705264979b3-3292502315430b2
+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)=-b5(2b-1)3(3b-2)(3b-1)(4b-3)(4b-1)(6b-5)(6b-1)

```

Algorithm 16. Zuniga (vi) binary splitting algorithm for zeta(3).

And then

$$\zeta(3) = \frac{1}{48} \frac{P(0,n)}{Q(0,n)} + O(717445350000^{-n}) \quad (49)$$

It has a linearly convergent cost of  $\sim 2.1$ , which is lower than Zuniga's version V.

For  $n$  terms, the error is  $O(717445350000^{-n})$  And for a given precision,  $P$  you need:

$$n = \left\lceil P \frac{\ln(10)}{\ln(717445350000)} \right\rceil \quad (50)$$

# Fast Computation of Math Constants in Arbitrary Precision

## Source Zuniga (vi) binary splitting method

```
static void binarySplittingAperyZunigaVI(const uintmax_t a, const uintmax_t b,
int_precision& p, int_precision& q, int_precision& r)
{
    uintmax_t mid;
    int_precision pp, qq, rr;

    // Base case: interval [a..b) of length 1
    // Notice bmax is ~ 0.1*precision. 1G precision => bmax ~ 10^8. so b^2 is max using
uintmax_t
    // For lower precision this could be improved by relying more on the native 64-bit
integers when calculating
    // p, q, and r
    if (a + 1 == b) {
        const uintmax_t b2m1(2 * b - 1);
        const uintmax_t b3(3 * b);
        const uintmax_t b4(4 * b);
        const uintmax_t b6(6 * b);
        const uintmax_t b8(8 * b);
        const uintmax_t b9(9 * b);
        const uintmax_t b10(10 * b);
        const uintmax_t b12(12 * b);
        const int_precision bb(b);
        const int_precision bp5(bb);

        // Compute q
        q = int_precision(270 * (b9 - 8) * (b9 - 7) ) *
            int_precision((b9 - 5) * (b9 - 4)) *
            int_precision((b9 - 2) * (b9 - 1)) *
            int_precision((b10 - 9) * (b10 - 7)) *
            int_precision((b10 - 3) * (b10 - 1)) *
            int_precision((b12 - 11) * (b12 - 7)) *
            int_precision((b12 - 5) * (b12 - 1));

        q = int_precision(288 * (b8 - 7) * (b8 - 5) * (b8 - 3) * (b8 - 1)) *
            int_precision((b9 - 8) * (b9 - 7) * (b9 - 5)) *
            int_precision((b9 - 4) * (b9 - 2) * (b9 - 1)) *
            int_precision((b10 - 9) * (b10 - 7) * (b10 - 3) * (b10 - 1));

        // Compute r
        if (b <= 7'131)
            r = int_precision(b * b * b * b * b); // No overflow if
b<=7'131
        else
            if (b <= 2'642'245)
            {
                // Max b^3
                r = int_precision(b * b * b);
                r *= int_precision(b * b);
            }
            else
            {
                // Max b^2
                r = int_precision(b * b);
                r *= r;
                r *= bb;
            }

        r *= -int_precision(b2m1*b2m1) * int_precision(b2m1) *
            int_precision((b3 - 2) * (b3 - 1)) *
            int_precision((b4 - 3) * (b4 - 1)) *
            int_precision((b6 - 5) * (b6 - 1));

        // Compute p
        p = ((((((((((int_precision(1565994397644288ull) *
            bb - int_precision(6719460725627136ull)) *
            bb + int_precision(12632254526031264ull)) *
            bb - int_precision(13684352515879536ull)) *
            bb + int_precision(9451223531851808ull)) *
            bb - int_precision(4348596587040104ull)) *
            bb + int_precision(1352700034136826ull)) *
            bb - int_precision(282805786014979ull)) *
            bb + int_precision(38721705264979ull)) *
```

# Fast Computation of Math Constants in Arbitrary Precision

```

        bb - int_precision(3292502315430ull)) *
        bb + int_precision(156286859400ull)) *
        bb - int_precision(3143448000ull);

        return;
    }

    mid = (a + b) / 2;
    // Recursively compute left and right intervals
    binarySplittingAperyZunigaVI(a, mid, p, q, r);
    binarySplittingAperyZunigaVI(mid, b, pp, qq, rr);
    // Merge results
    p = p * qq + pp * r;
    q *= qq;
    r *= rr;

    return;
}

static float_precision computeAperydigitsZunigaVI(const uintmax_t precision)
{
    // Determine the number of terms to compute
    intmax_t kmax = intmax_t(ceil(precision * log(10) / log(717445350000)));
    const size_t workprec = (size_t)ceil(precision + 1 + log(kmax));
    int_precision p, q, r;
    float_precision fp, fq;

    // Compute the result
    binarySplittingAperyZunigaVI(0, kmax, p, q, r);
    // Finalize the result
    q *= int_precision(48);
    fp.precision(workprec);
    fq.precision(workprec);
    fp = float_precision(p, workprec);
    fq = float_precision(q, workprec);
    fp /= fq;
    fp.precision(precision);
    return fp;
}

```

Both Zuniga series V and VI can quickly be turned into a simple two-way threading as outlined for the Amdeberhan-Zeilberger method,

## 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
<b>Amdeberhan-Zeilberger</b>	Binary Splitting	$O(1024^{-n})$	1.661P
<b>Wedeniwski</b>	Binary-Splitting	$O(110592^{-n})$	1.107P
<b>Zuniga (v)</b>	Binary-Splitting	$O(34828517376^{-n})$	0.095P
<b>Zuniga (vi)</b>	Binary-Splitting	$O(717445350000^{-n})$	0.084P

Table 4 Comparison of the Apéry methods.

Due to its much faster convergence rate, the Zuniga two series requires more than 10 times fewer splits than the Amdeberhan-Zeilberger and Wedeniwski method.

## Apéry Constant $\zeta(3)$ performance

Both Binary splitting methods outperform the general zeta function implementation with several magnitudes (not shown in the figure below). It seems that the Wedeniwski method has a slight edge over the Amdeberhan. This was expected since the linearly convergent cost is 2.78 for Wedeniwski versus 2.89 for Amdeberhan-Zeilberger. Furthermore, Wedeniwski only needs  $\sim 0.198 \cdot \text{Precision}$  terms versus  $\sim 0.332 \cdot \text{Precision}$  terms for Amdeberhan-Zeilberger. However, the computation of the  $P(b-1,b)$ ,  $Q(b-1,b)$ , and  $R(b-1,b)$  is more complicated for the Wedeniwski method. Furthermore, none of them can match the performance of the two Zuniga series, even though  $P(b-1,b)$ ,  $Q(b-1,b)$ , and  $R(b-1,b)$  are way more complicated to calculate.

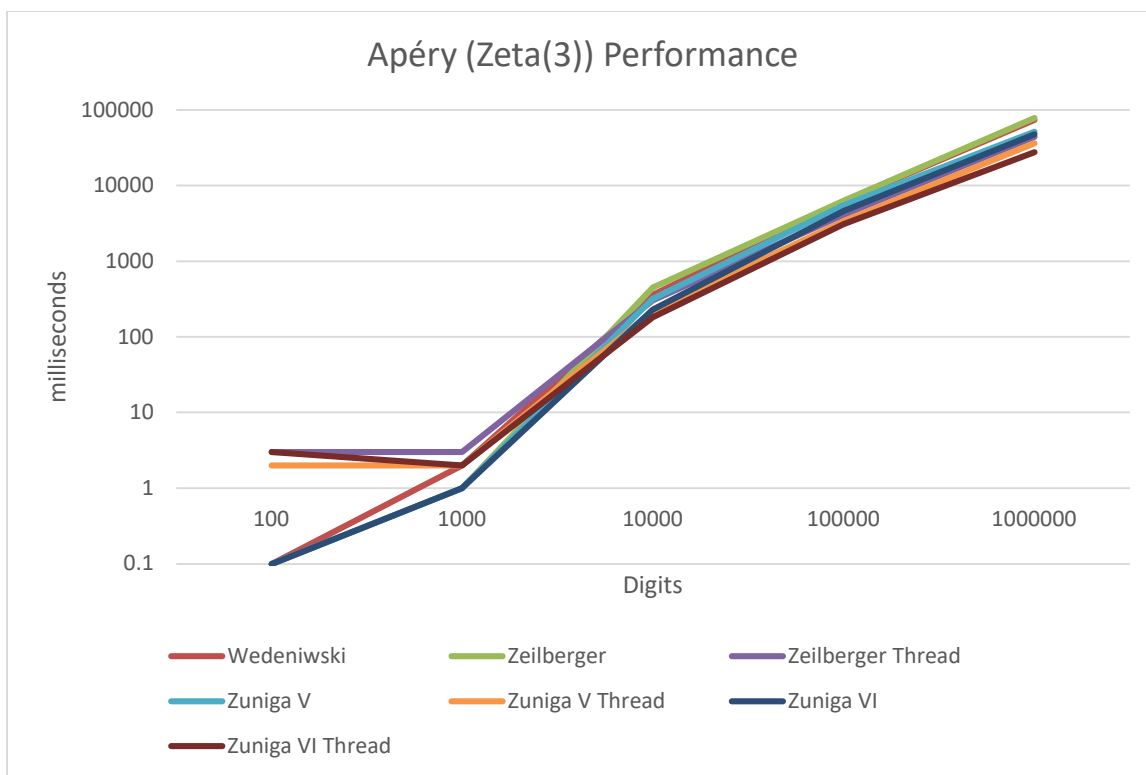


Figure 3 Apéry Constant Performance

It is sometimes clearer to look at the tables below, which show the time required to calculate the Apéry constant from 10 to 1M digits. All times in milliseconds.

Digits	10	100	1,000	10,000	100,000	1,000,000
Wedeniwski	0	0	2	359	6,134	73,926
Zeilberger	0	0	1	447	6,284	78,453
Zeilberger Thread	5	3	3	304	4,098	44,105
Zuniga V	0	0	1	318	5,467	51,975
Zuniga V Thread	0	2	2	217	3,351	36,539
Zuniga VI	0	0	1	228	4,598	47,941

# Fast Computation of Math Constants in Arbitrary Precision

---

<b>Zuniga VI Thread</b>	0	3	2	181	3,086	27,788
-------------------------	---	---	---	-----	-------	--------

Table 5. Time in milliseconds to compute the Apéry constant from 10 to 1M digits.

It is worth mentioning that simple two-way threading improves performance significantly for all the methods presented.

## ***Recommendation for the constant $\zeta(3)$***

I recommend the following:

- 1) It is clear that if you are serious, you would implement one of the binary splitting methods.
- 2) The general zeta(s) function is not recommended for the computation of the Apéry constant.
- 3) Wedeniwski is slightly faster but Amdeberhan-Zeilberger is more straightforward to implement.
- 4) However, the fastest method is the Zuniga (vi) version, which performs the fastest in both the non-threaded and threaded versions.
- 5) Implement the threaded version for the binary splitting method if speed is of the essence.

## The Lemniscate Constant $\varpi$

The lemniscate constant ( $\varpi$ ) is the ratio of the perimeter of Bernoulli's lemniscate to its diameter. The  $\varpi \sim 2.622057554292$ . In literature, you sometimes see that  $2\varpi$  or  $\varpi/2$  is also referred to as the Lemniscate constant.

There are several definitions of the Lemniscate constant. One of them is:

$$\varpi = 2 \int_0^1 \frac{1}{\sqrt{1-x^4}} dx \quad (51)$$

There are also many available series or continuous fractions to compute the lemniscate constant. As we have seen many times before, you can use the Binary Splitting method to get a much more efficient computation of the Lemniscate constant. Particularly the many binary splitting methods from Zuniga 2023 that compute  $2\varpi$  as the Lemniscate constant. Other methods have seen the light. One of them is the method used by Guillera.

### Zuniga many binary splitting methods

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

#### Zuniga version vii.

$$\frac{1}{2\varpi} = \frac{214326}{\sqrt[6]{11816941917501}} \sum_{k=1}^{\infty} \left(\frac{-512}{2315685267}\right)^k k^2 \frac{P(k)}{R(k)} \left(\frac{\left(\frac{1}{36}\right)_k \left(\frac{7}{36}\right)_k \left(\frac{13}{36}\right)_k \left(\frac{19}{36}\right)_k \left(\frac{25}{36}\right)_k \left(\frac{31}{36}\right)_k}{\left(\frac{1}{3}\right)_k \left(\frac{1}{3}\right)_k \left(\frac{2}{3}\right)_k \left(\frac{2}{3}\right)_k k^{12}}\right) \quad (52)$$

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

Algorithm: Zuniga (vii) Binary splitting method for  $\varpi$  (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)=99446494228488b7-296948949253092b6+339735211540956b5-
185806427026662b4+48479683290426b3-4840729282291b2
Q(b-1,b)=121545688294296b2(3b-1)2(3b-2)2
R(b-1,b)=-(36b-5)(36b-11)(36b-17)(36b-23)(36b-29)(36b-35)
    
```

Algorithm 17. Zuniga 2023 version vii. Notice the complexity in computing P, Q and R

And then

$$\varpi = \frac{1}{324 \sqrt[6]{453789}} \frac{Q(0,n)}{P(0,n)} + O\left(\left(\frac{512}{2315685267}\right)^n\right) \quad (53)$$

This method has a linearly convergent cost of  $\sim 1.566$ .

# Fast Computation of Math Constants in Arbitrary Precision

---

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

$$n = \left\lceil P \frac{\ln(10)}{\ln\left(\frac{2315685267}{512}\right)} \right\rceil \quad (54)$$

**Zuniga version viii**, which is similar.

$$\frac{1}{2\omega} = \frac{215622}{\sqrt[4]{483153}} \sum_{k=1}^{\infty} \left(\frac{512}{2357947691}\right)^k k^2 \frac{P(k)}{R(k)} \left(\frac{\left(\frac{1}{36}\right)_k \left(\frac{5}{36}\right)_k \left(\frac{13}{36}\right)_k \left(\frac{17}{36}\right)_k \left(\frac{25}{36}\right)_k \left(\frac{29}{36}\right)_k}{\left(\frac{1}{3}\right)_k \left(\frac{1}{3}\right)_k \left(\frac{2}{3}\right)_k \left(\frac{2}{3}\right)_k k!^2}\right) \quad (55)$$

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

Algorithm: Zuniga (viii) Binary splitting method for  $\omega$  (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)=1768056164733b7-52825631815620b6+60473303319276b5-
33092086224942b4+8638260598818b3-862864755643b2
Q(b-1,b)=123763958405208b2(3b-1)2(3b-2)2
R(b-1,b)=(36b-7)(36b-11)(36b-19)(36b-23)(36b-31)(36b-35)
    
```

Algorithm 18. Zuniga 2023 version viii.

And then

$$\omega = \frac{1}{1188 \sqrt[4]{35937}} \frac{Q(0,n)}{P(0,n)} + O\left(\left(\frac{512}{2357947691}\right)^n\right) \quad (56)$$

This method has a linearly convergent cost of  $\sim 1.564$ .

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

$$n = \left\lceil P \frac{\ln(10)}{\ln\left(\frac{2357947691}{512}\right)} \right\rceil \quad (57)$$

**Zuniga version x**, which is similar.

$$\frac{1}{2\omega} = 128 \sqrt[4]{48315310433320250} \sum_{k=1}^{\infty} \frac{k^2 P(k)}{(16k-3)(16k-7)(16k-11)(16k-15)} \prod_{i=1}^k \frac{(16k-3)(16k-7)(16k-11)(16k-15)}{11008380780544 \cdot 2^{2(i-1)}} \quad (58)$$

Algorithm: Zuniga (x) Binary splitting method for  $\omega$  (2023)

```

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

## Fast Computation of Math Constants in Arbitrary Precision

---

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

And:

$$\begin{aligned} P(b-1,b) &= 22934126592b^3 - 45503382016b^2 + 28302850848b - 5642344589 \\ Q(b-1,b) &= 11008380780544b^2(2b-1)^2 \\ R(b-1,b) &= (16b-3)(16b-7)(16b-11)(16b-15) \end{aligned}$$

Algorithm 19. Zuniga 2023 version x.

And then

$$\varpi = \frac{1}{128^4 \sqrt[4]{1043320250}} \frac{Q(0,n)}{P(0,n)} + O(671898241^{-n}) \quad (59)$$

This method has a linearly convergent cost of  $\sim 0.78$

For  $n$  terms, the error is  $O(671898241^{-n})$  And for a given precision,  $P$  you need:

$$n = \left\lceil P \frac{\ln(10)}{\ln(671898241)} \right\rceil \quad (60)$$

**Zuniga version x(2)**, which is a more straightforward version of it.

$$\frac{1}{2\varpi} = \frac{10304}{\sqrt[4]{6440}} \sum_{k=1}^{\infty} \frac{k^2(8640n-8365)}{(8k-3)(8k-7)} \prod_{i=1}^k \frac{(8k-3)(8k-7)}{1658944i^2} \quad (61)$$

Algorithm: Zuniga (x(2)) Binary splitting method for  $\varpi$  (2023)

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

And:

$$\begin{aligned} P(b-1,b) &= b^2(8640b-8365) \\ Q(b-1,b) &= 1658944b^2 \\ R(b-1,b) &= (16b-3)(16b-7) \end{aligned}$$

Algorithm 20. Zuniga version x(2).

And then

$$\varpi = \frac{\sqrt[4]{6440}}{20608} \frac{Q(0,n)}{P(0,n)} + O((25491)^{-n}) \quad (62)$$

These methods have a linearly convergent cost of  $\sim 0.78$

For  $n$  terms, the error is  $O(25491^{-n})$  And for a given precision,  $P$  you need:

$$n = \left\lceil P \frac{\ln(10)}{\ln(25491)} \right\rceil \quad (63)$$

## ***Guillera Binary Splitting method.***

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

$$\frac{1}{2\varpi} = \frac{34560}{\sqrt[8]{162000}} \sum_{k=1}^n \frac{k^2(1288k-124)}{-(8k-5)(8k-7)} \prod_{i=1}^k \frac{-(8i-5)(8i-7)}{1658880i^2} \quad (64)$$

Not as voluminous as some of the previous Zuniga series.

Algorithm: Guillera Binary splitting method for  $\varpi$  (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)
    
```

Algorithm 21. Guillera 2023 version.

And then

$$\varpi = \frac{\sqrt[8]{162000} Q(0,n)}{69120 P(0,n)} + O(25920^{-n}) \quad (65)$$

These methods have a linearly convergent cost of  $\sim 0.78$ , which is good and lower than the Zuniga series (except version x and x(2)). However, they require more splits than the Zuniga's series. However, a lower linearly convergent cost looks promising.

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

$$n = \left\lceil P \frac{\ln(10)}{\ln(25920)} \right\rceil \quad (66)$$

## **Comparison of the Lemniscate methods.**

The five Lemniscate methods are listed below. The Zuniga versions (vii and viii) have similar characteristics and are more efficient than the Guillera method.

Method	Implementation	Error	N(P), P=precision
<b>Zuniga vii</b>	Binary Splitting	$O\left(\left(\frac{512}{2315685267}\right)^n\right)$	0.15P

# Fast Computation of Math Constants in Arbitrary Precision

<b>Zuniga viii</b>	Binary-Splitting	$O\left(\left(\frac{512}{2357947691}\right)^n\right)$	0.15P
<b>Zuniga x</b>	Binary-Splitting	$O(671898241^{-n})$	0.11P
<b>Zuniga x(2)</b>	Binary-Splitting	$O(25491^{-n})$	0.23P
<b>Guillera</b>	Binary-Splitting	$O(25920^{-n})$	0.23P

Table 6. Comparison of key characteristics of the three methods.

The Zuniga methods (vii, viii, and x) require fewer splits to achieve a given precision of the result. However, each split is more time-consuming than the Guillera version.

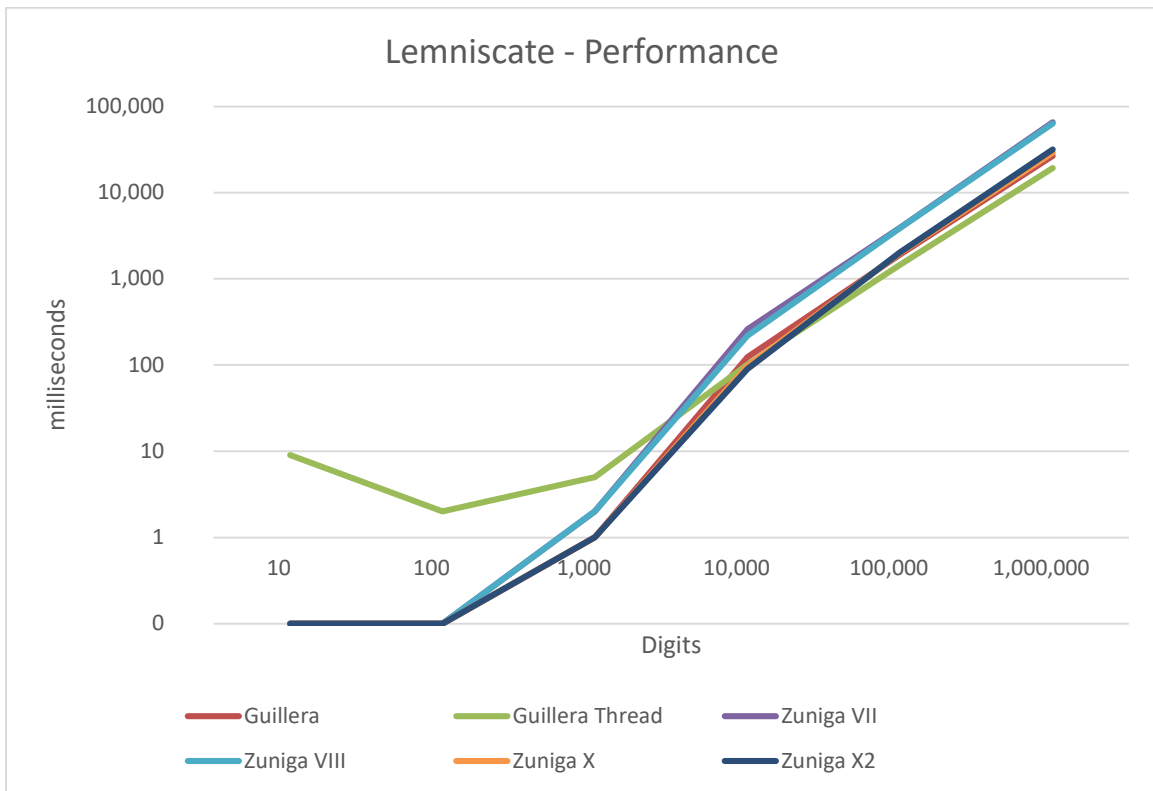


Figure 4. Time to compute the Lemniscate constant up to 1M digits. Notice that Guillermo is considerably faster than the two Zuniga methods. A threaded version of the Guillermo method is advantageous over 10,000 digits precision.

Lemniscate constant. Time in milliseconds							
Digits	10	100	1,000	10,000	100,000	1,000,000	
<b>Guillera</b>	0	0	1	127	2,023	27,658	
<b>Guillera Thread</b>	7	3	3	101	1,671	20,015	
<b>Zuniga VII</b>	0	0	3	266	4,210	64,671	
<b>Zuniga VIII</b>	0	0	3	237	4,127	72,702	
<b>Zuniga X</b>	0	0	1	97	1,975	29,160	
<b>Zuniga X(2)</b>	0	0	1	90	2,013	31,823	

# Fast Computation of Math Constants in Arbitrary Precision

---

Table 7. Performance of the Lemniscate constant. As you can see, the Guillera method is more than twice as fast as the Zuniga versions and a threaded version of the Guillera method is approx. 30% faster in line with expectations.

## ***Recommendation for the Lemniscate constant $\varpi$***

I recommend the following:

1. It is clear that if you are serious, you would implement one of the binary splitting methods.
2. The Guillera method is faster than the Zuniga four versions and more straightforward to implement. Therefore, it is recommended.
3. Implement the threaded version for the binary splitting method if speed is of the essence (a 30% increase in performance above 10,000 digits).

## **Overall conclusion**

This paper demonstrates that efficiently computing a range of special constants—Euler–Mascheroni, Catalan’s, Apéry’s  $\zeta(3)$ , and the Lemniscate constant—can be significantly accelerated by combining well-chosen series expansions with binary splitting implementations. While traditional methods such as the Brent–McMillan formula or straightforward summation may suffice for lower precision, the binary splitting approach dominates as the digit count grows, particularly when extended with simple multithreading. Each constant has multiple expansions, but the performance data and comparative charts underline that a careful balance of convergence speed, arithmetic complexity, and memory operations is essential. This work underscores the benefits of modular, high-performance routines by illustrating how to incorporate these algorithms into a practical C++ arbitrary precision library. The results and code samples also serve as a blueprint for further refinements, pointing toward broader applications where high-precision arithmetic is central—whether in advanced numerical analysis, special function evaluations, or computational mathematics research.

## **Reference**

- 1) Arbitrary precision library package. [Arbitrary Precision C++ Packages \(hvks.com\)](http://hvks.com)
- 2) Numerical recipes in C++, 3<sup>rd</sup> edition, Cambridge University Press, New York, NY 2007
- 3) The Yacas book of algorithm, Version 1.3.3, April 1, 2013, by the Yacas team
- 4) Alexander Yee, Binary Splitting Recursion Library. [Binary Splitting Recursion Library](#)
- 5) G.Free, Computation of Catalan’s Constant using Ramanujan’s formula. 1990 ACM
- 6) Fast Gamma, Beta, Error, and Zeta functions for arbitrary precision. [HVE Fast Gamma, Beta, Error, and Zeta functions for arbitrary precision.](#)

## Fast Computation of Math Constants in Arbitrary Precision

---

- 7) Exploring the Binary Splitting method. [HVE Exploring Binary Splitting Method.](#)