Fast Computation of Math Constants in Arbitrary Precision.

By Henrik Vestermark (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, and the Apéry (zeta(3) constant, needed for making a complete Arbitrary precision Math package.

Introduction:

We start up by looking at the various mathematical constants and the different methods available to compute these constants:

- The Euler-Mascheroni constant
- The Catalan Constant
- The Apery Constant $\zeta(3)$

As usual, we will show the actual C++ source for the calculation using the author's own arbitrary precision Math library, see [1].
This paper is part of a series of arbitrary precision papers describing methods, implementation details, and optimization techniques. These papers can be found on my website at www.hvks.com/Numerical/papers.html and are listed below:

1. Fast 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 π 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
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

Contents

The Arbitrary precision library

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

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

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

You can add a few parameters to the declaration. The first is the optional initial value, and an optional second parameter is a floating-point precision. The native type of a *float* has a fixed size of 4 bytes and 8 bytes for *double*, however since this precision can be arbitrary we can declare the wanted 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() E.g.

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

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

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

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

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

This allows very fast multiplication of division 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 to the arbitrary precision math package for details.

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

Internal format for float_precision variables

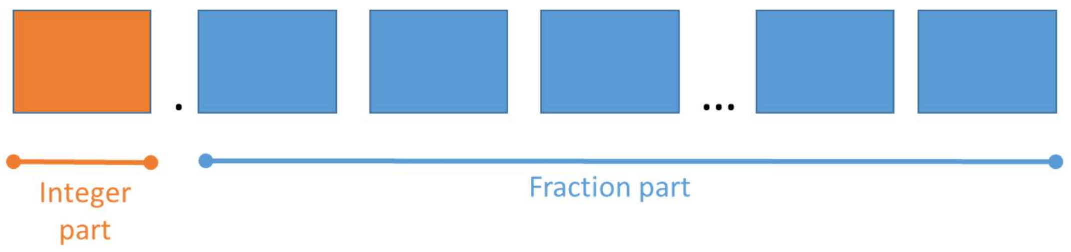
For the internal layout of the arbitrary precision number, we are using the STL vector library declared as:

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

uintmax_t is mostly a 64-bit quantity on most systems. 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 mBinary



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

There are other internal class variables like the sign, exponent, precision, and rounding mode but these are not important to understand 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

The Euler-Mascheroni constant γ is defined as:

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

The above equation (1) converts only slowly and it is not useful for arbitrary precision arithmetic. Instead, there are a few other interesting methods

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

Brent-McMillan method

To compute γ you can use the Brent-McMillan decomposition [3].

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

Where 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:

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

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

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

$$H_n = \sum_{k=1}^{n}\frac{1}{k} \tag{5}$$

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:

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

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

$$k_{max} \approx 2.07 \cdot P \tag{7}$$

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

Source Euler Brent-McMillan

```
static float_precision EulerConstant(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+log(kmax)/log(10));
	uintmax_t k;
	float_precision res(0,workprec);
	float_precision vn(0,workprec), sn(0,workprec), vsq(0,workprec);
	float_precision vsum(1, workprec), ssum(0,workprec), hsum(0,workprec);
	int_precision np(1), kfac(1);

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

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

Brent enhancement

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

$$U(n) = S(n) - V(n) \cdot \ln(n) \tag{8}$$

Then

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

And

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

Where A_k is:

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

Furthermore, we use B_k as the nth term of the V(n) series.

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

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

Algorithm Brent summation trick

$$A_0 = -\ln(n), B_0 = 1$$

$$B_k = B_{k-1} \cdot \frac{n^2}{k^2}$$

$$A_k = \frac{1}{k}\left(A_{k-1} \cdot \frac{n^2}{k} + B_k\right) = A_{k-1} \cdot \frac{n^2}{k^2} + \frac{B_k}{k}$$

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.

```
static float_precision EulerConstant2(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 + log(kmax) / log(10));
	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 = _float_precision_inverse(fpk);  // 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;
}
```

Binary splitting method for γ

Lastly, you can use the Binary splitting technic as outlined in [4]

Algorithm: Binary splitting method for γ (7 variables)

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

And P(b-1,b)=1;  Q(b-1,b)=b; R(b-1,b)=$n^2$; S(b-1,b)=$b^2$; T(b-1,b)=$n^2$;
U(b-1,b)=$n^2$; V(b-1,b)=$b^3$;
```

Algorithm 2

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

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

and let the formula reverse bottom up.

In the end, you find γ by:

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

In [4] they found that i=3.5911214766686221366n as 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 quite easily 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.

Algorithm: Binary splitting method for γ (5 variables)

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

And P(b-1,b)=1;  Q(b-1,b)=b; R(b-1,b)=$n^2$; T(b-1,b)=$n^2$; U(b-1,b)=$n^2$;
```

Algorithm 3

At the end you find γ 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 5 variables version does perform better but not impressively better.

Source of Binary splitting method for the Euler-Mascheroni constant

The main call is the call to computeEulerDigits() which initialize and call the recursive binary splitting function binarysplittingEuler().

```
static void binarysplittingEuler(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)
	{
		uintmax_t bsq = b * 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;
	binarysplittingEuler(a, mid, nsq, p, q, r, t, u);           // interval [a..mid]
	binarysplittingEuler(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;
	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;
	return;
}

static float_precision computeEulerdigits(const uintmax_t precision)
{
	const uintmax_t n = (uintmax_t)ceil((precision * log(10) + log(3.14159265)) / 4);
	uintmax_t k = uintmax_t(ceil(n * 3.5911214766686221366));
	int_precision p, q, r, t, u;
	float_precision fp, fq;

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

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

	q *= r + q * q;
	fp = float_precision(u, precision + 1); fq = float_precision(q, precision + 1);
	fp /= fq;
	fq = float_precision(n);
	fp -= log(fq);
	fp.precision(precision);
	return fp;
}
```

It is relatively easy to create a threaded version of the binary splitting algorithm and it has been proven advantageous to increase the performance by just threading the computational task.

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

Source of the threaded Binary splitting method for the Euler-Mascheroni constant

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

```
static float_precision computeEulerdigitsThread(const uintmax_t precision)
{
	const uintmax_t n = (uintmax_t)ceil((precision * log(10) + log(3.14159265)) / 4);
	uintmax_t k = uintmax_t(ceil(n * 3.5911214766686221366));
	int_precision p, q, r, t, u;
	int_precision pp, qq, rr, tt, uu, tmp;
	float_precision fp, fq;

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

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

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

	// wait for both threads to terminate
	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 = float_precision(u, precision + 1); fq = float_precision(q, precision + 1);
	fp /= fq;
	fq = float_precision(n);
	fp -= log(fq);
	fp.precision(precision);
	return fp;
}
```

Performance of Euler-Mascheroni constant

It is quite clear by looking at the performance chart that the binary splitting method is superior for the computation of the Euler-Mascheroni constant. The traditional Brent-McMillan (Brent 1) and Brent Summation trick (Brent 2) methods can’t be recommended for fast computation of the Euler-Mascheroni constant. However Brent Summarization trick is a significant improvement over the standard Brent-McMillan formula.

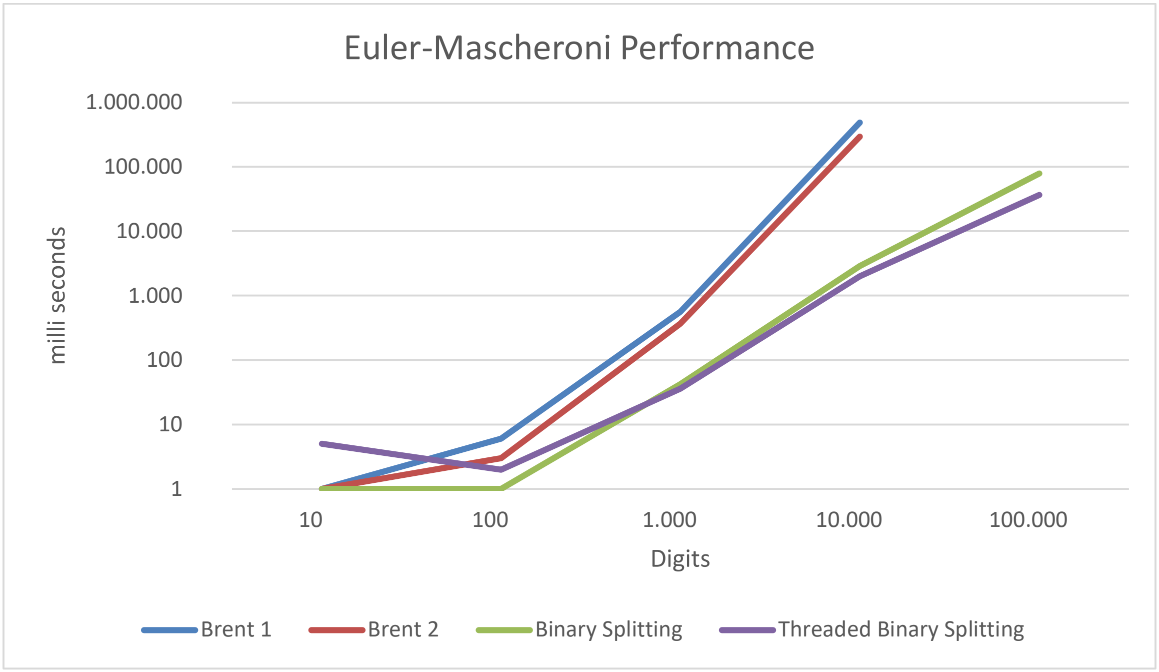


Figure 1 Euler-Mascheroni Performance

Recommendation for the Euler-Mascheroni constant

Based on the performance chart below, I recommend the Binary splitting method and 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:

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

The Catalan constant is ~0.9159655941772…

This series also convert 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)

Ramanujan's method I

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

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

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 efficiently summarize the series. One of the drawbacks of this method is that we need to calculate π, ln(2+$\sqrt{3}$), and $\sqrt{3}$ to arbitrary precision. Horner's schema looks like this:

$$1 + \frac{1}{2\cdot3}\left(\frac{1}{3} + \frac{2}{2\cdot5}\left(\frac{1}{5} + \frac{3}{2\cdot7}\left(\frac{1}{7} + \cdots\right)\right)\right) \tag{17}$$

And the algorithm for the Horner schema sum.

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

Algorithm 4

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:

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

In [5] Free use 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<=n;++k)
        tmp=k/(2k+1)
        $B_k=B_{k-1}$*tmp
        $C_k=C_{k-1}$*tmp+$B_k$/(2k+1)
        $G_k=G_{k-1}+C_k$
```

Algorithm 5

We need a little bit more to achieve P, decimal precision compare to the first method. In [5] 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.

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

Where:

a_i=(0,1/2,-1/2,1/4,0,-1/8,1/8,-1/16) 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);
	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;
}
```

Lupas Binary Splitting method

Lupas series for the Catalan constant is:

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

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:
    $P(b-1,b)=(32(2b-1)b^3)(40b^2+56b+19)(-1)^b$
    $Q(b-1,b)=(4b+1)^2(4b+3)^2$
    $R(b-1,b)=32(2b-1)b^3$
```

Algorithm 6

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

In the end, you find G by:

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

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

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

In [4] they found the linearly convergent cost to be ~11.5, which makes 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);

	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 publish two methods back in 2008 and 2019. The first method used:

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

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

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

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

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

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

And:
$P(b-1,b)= b^3(3b+2)$
$Q(b-1,b)=-(2b+1)^3(2b-1)^3$
$R(b-1,b)=b^3(2b+1)^3$
```

Algorithm 7

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

In the end, you find G by:

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

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

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

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

$$G = -\frac{1}{1024}\sum_{k=1}^{\infty}\frac{(-40\quad)^k(45136\quad{}^4-57184k^3+2124\quad{}^2-3160k+165)}{k^3(2k-1)^3}\left(\frac{(2k)!^6(3k)!^3}{k!^3(6k)!^3}\right) \qquad (27)$$

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

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

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

And:
P(b-1,b)= $45136b^4-57184b^3+21240b^2-3160b+165$
Q(b-1,b)=$-27(6b-1)^3(6b-5)^3$
R(b-1,b)=$512b^3(2b-1)^3$
```

Algorithm 8

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

In the end, you find G by:

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

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

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

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)
		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
                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 publish two formulas in 2010. The short and long formula.

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

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)= $580b^2-184b+15$
    Q(b-1,b)= $9(6b-1)^2(6b-5)^2$
    R(b-1,b)= $32b^3(2b-1)$
```

Algorithm 9

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

In the end, you find G by:

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

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

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

Pilehrood also has a long version formula:

$$G = -\frac{1}{64}\sum_{k=1}^{\infty} \frac{(-256)^k(419840k^6-915456k^5+782848k^4-332800k^3+73256k^2-7800k+31\ )}{k^3(2k-1)(4k-1)^2(4k-3)^2\binom{8k}{4k}^2\binom{2k}{k}} \tag{33}$$

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

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

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

    And:
    P(b-1,b)=$(-1)^b(419840b^6 - 915456b^5 + 782848b^4 - 332800b^3 + 73256b^2 - 7800b + 315)$
    Q(b-1,b)=$(8b-1)^2(8b-3)^2(8b-5)^2(8b-7)^2$
    R(b-1,b)=$32b^3(2b-1)(4b-1)^2(4b-3)^2$
```

Algorithm 10

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

In the end, you find G by:

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

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

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

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;
        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(4bm3)^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;
        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;
}
```

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 you can look at the below table that outlines the name, implementation type, error, and precision requirement.

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

Not surprisingly the performance depends heavily on the convergence speed and implementation type e.g. 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 Lucas, 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 and among the classical series the Broadhurst method is by far the fastest.

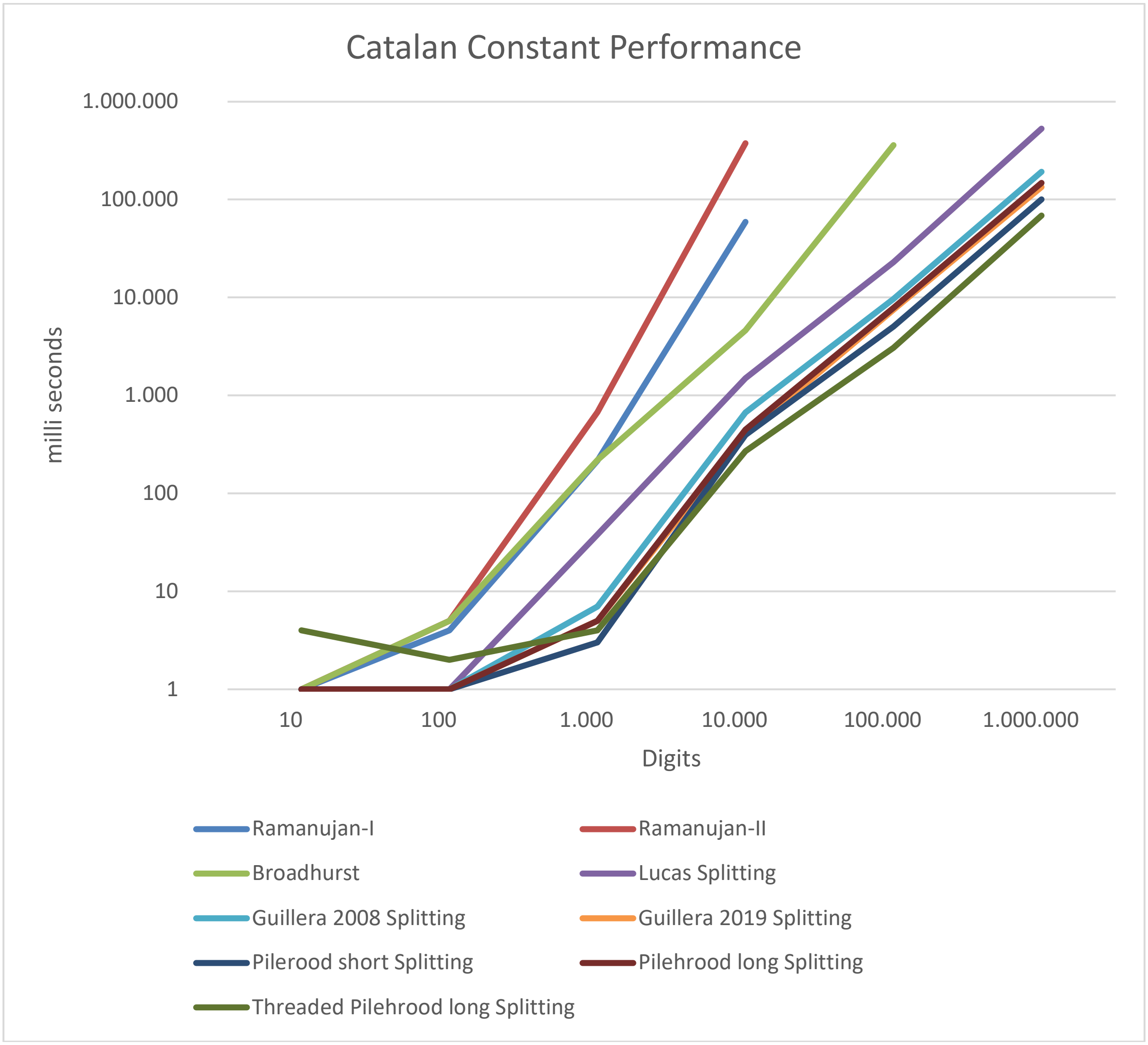


Figure 2 Catalan Constant Performance chart

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. Also if performance is required then use or implement a threaded version of the Pilehrood method. It is very easy to create a 2, 3, 4, or more core-threaded version of the binary splitting method. If we only want a classical method then I recommend the Broadhurst method.

Apéry's constant ζ(3)

Is the common short name for the ζ(3) value. This is a specialized formula for the ζ(3) instead of using the more general computation of ζ(s) in [6]. As I mentioned in [6] there has been researched into finding a formula, series, etc. for the odd integer's values of the zeta function. One of them is the value of ζ(3). There is two formula that comes to mind and these are [4]:

- Amdeberhan-Zeilberger series
- Wedeniwski series

Amdeberhan-Zeilberger series

This series is given by Amdeberhan-Zeilberger back in 1997.

$$\zeta(3) = \frac{1}{64}\sum_{k=0}^{\infty}\frac{(-1)^k(205k^2+250k+77)(k!)^{10}}{(2k+1)^5} \qquad (36)$$

Now by now, we should have learned that the most efficient computation is by using the binary splitting method.

Algorithm: Binary splitting method for ζ(3) (1997)

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

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

Algorithm 11

And then

$$\zeta(3) = \frac{P(0,n)+77Q(0,n)}{64Q(0,n)} + O(1024^{-n}) \qquad \textbf{(37)}$$

Which have a linearly convergent cost of ~2.89 which is slightly higher than the next Wedeniwski method.

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

$$n = \left\lceil P \frac{\ln(10)}{\ln(1024)} \right\rceil \tag{38}$$

Source Amdeberhan-Zeilberger Binary splitting method

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

```
// Zeta(3) Amdeberhan - Zeilberger(1997)
static void binarysplittingApery1(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 = bsq * bsq * b;		// No overflow if b<=7'131
		else
			if (b <= 2'642'245)
			{	// Max b^3
				r = bsq * b;
				r *= bsq;
			}
			else
			{	// Max b^2
				r = bsq; r *= r; r *= b;
			}

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

			}
			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;
	binarysplittingApery1(a, mid, p, q, r);	// interval [a..mid]
	binarysplittingApery1(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 computeAperydigits1(const uintmax_t precision)
{
	intmax_t kmax = intmax_t(ceil(precision * log(10) / log(1024)));
	int_precision p, q, r;
	float_precision fp, fq;

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

	binarysplittingApery1(0, kmax, p, q, r);
	p += int_precision(77) * q;
	q *= int_precision(64);
	fp = float_precision(p, precision + 1);
	fq = float_precision(q, precision + 1);

	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. It is easy to continue this and create a three-way, four-way, or even higher-threaded version.

```
static float_precision computeAperydigits1Thread(const uintmax_t precision)
{
	intmax_t kmax = intmax_t(ceil(precision * log(10) / log(1024)));
	int_precision p, q, r, pp, qq, rr;
	float_precision fp, fq;

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


	std::thread first([=, &p, &q, &r]()
		{binarysplittingApery1(0, kmax/2, p, q, r); });       // interval [a..k/2]

	std::thread second([=, &pp, &qq, &rr]()
		{binarysplittingApery1(kmax/2, kmax, pp, qq, rr); }); // interval [k/2..k]

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

	// 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 = float_precision(p, precision + 1);
	fq = float_precision(q, precision + 1);

	fp /= fq;
	fp.precision(precision);
	return fp;
}
```

Wedeniwski series

This series is given by Amdeberhan-Zeilberger back in 1997.

$$\zeta(3) = \frac{1}{24}\sum_{k=0}^{\infty}\frac{(-1)^k(126392k^5+412708k^4+531578k^3+336367k^2+104000k+1264\ \ )((2k+1)!(2k)!k!)^3}{(3k+2)!(4k+3)!^3} \quad (39)$$

Algorithm: Wedeniwski Binary splitting method for ζ(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 12

And then

$$\zeta(3) = \frac{P(0,n)+12463Q(0,n)}{10368Q(0,n)} + O(110592^{-n}) \quad (40)$$

Which have a linearly convergent cost of ~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:

$$n = \left\lceil P\frac{\ln(10)}{\ln(110592)}\right\rceil \quad (41)$$

Source Wedeniwski binary splitting method

```
static void binarysplittingApery2(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 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
		//24(3b+1)(3b+2)(4b+1)^3(4b+3)^3
		q = b4p1 * (b4p1+2); q *= q * q; q *= 24 * b3p1*(b3p1+1);
		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;
        binarysplittingApery2(a, mid, p, q, r);        // interval [a..mid]
        binarysplittingApery2(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 computeAperydigits2(const uintmax_t precision)
{
        intmax_t kmax = intmax_t(ceil(precision * log(10) / log(110592)));
        int_precision p, q, r;
        float_precision fp, fq;

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

        binarysplittingApery2(0, kmax, p, q, r);
        p += int_precision(12463) * q;
        q *= int_precision(10368);
        fp = float_precision(p, precision + 1);
        fq = float_precision(q, precision + 1);

        fp /= fq;
        fp.precision(precision);
        return fp;
}
```

Apéry Constant ζ(3) performance

Both Binary splitting methods outperform the general zeta function implementation. 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 ~0.198*Precision terms versus ~0.332*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.

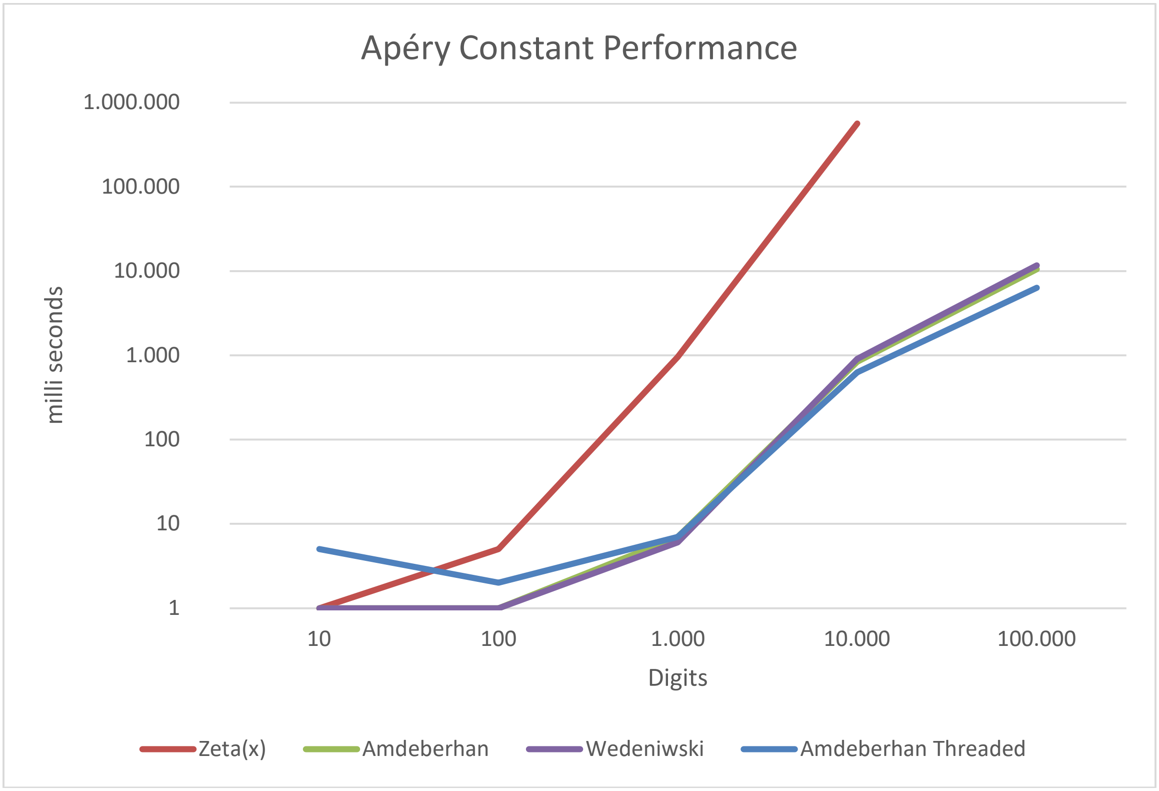


Figure 3 Apéry Constant Performance

Recommendation for the constant ζ(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 simpler to implement.
4) Implement the threaded version for the binary splitting method if speed is of the essence.

Reference

1) Arbitrary precision library package. Arbitrary Precision C++ Packages (hvks.com)
2) Numerical recipes in C++, 3rd 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
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.