Fast Exponential function for Arbitrary Precision number.

By Henrik Vestermark (hve@hvks.com)

Abstract:

This is a follow-up to a previous paper that describes the math behind arbitrary precision numbers. First of all the original paper was written back in 2013 and quite a few things had happens since then, secondly, I have come across some other interesting methods to do the exponential function calculation. The paper describes in more detail how to do e^x-calculation with arbitrary precision and outlines some traditional methods but also introduces an improved version that triples the speed of each calculation using automated argument reduction and coefficient scaling.

Introduction:

Usually, when implementing an arbitrary precision math package you would use the standard Taylor series calculation for calculating e^x for arbitrary precisions. The Taylor series for e^x is not particularly fast in its raw form. However, you can apply techniques that significantly improved the performance of the method. We will discuss the various method for calculating e^x and elaborate on the techniques like clever argument reduction and coefficient scaling to improve the performance of the method.

As usual, we will show the actual C++ source for the computation 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 Computation of Math Constants in arbitrary precision. HVE Fast Gamma, Beta, Error, and Zeta functions for arbitrary precision.
2. Fast Gamma, Beta, Error, and Zeta functions for arbitrary precision. HVE Fast Gamma, Beta, Error, and Zeta functions for arbitrary precision.
3. Fast Square Root & Inverse calculation for arbitrary precision math. HVE Fast Square Root & inverse calculation for arbitrary precision
4. Fast Exponential calculation for arbitrary precision math. HVE Fast Exp() calculation for arbitrary precision
5. Fast logarithm calculation for arbitrary precision math. HVE Fast Log() calculation for arbitrary precision
6. Practical implementation of Spigot Algorithms for Transcendental Constants. Practical implementation of Spigot Algorithms for transcendental constants
7. Practical implementation of π algorithms. HVE Practical implementation of PI Algorithms
8. Fast Trigonometric function for arbitrary precision. HVE Fast Trigonometric calculation for arbitrary precision

9. Fast Hyperbolic functions for arbitrary precision. HVE Fast Hyperbolic calculation for arbitrary precision
10. Fast conversion from arbitrary precision number to a string. HVE Fast conversion from arbitrary precision to string
11. Fast conversion from a decimal string to an arbitrary precision number. HVE Fast conversion from string to arbitrary precision

Change log

28-February-2023. Minor corrections.
26-January 2023. Cleaning up the document grammatically.
27-October 2022. Added a section for e^x using the binary splitting method
16-July 2022. Added a section of the binary splitting method for e.

Contents

Abstract: 1
Introduction: 1
Change log 2
The Arbitrary precision library 4
Internal format for float_precision variables 5
Normalized numbers 5
e^x 7
e^x using the Taylor series 7
Example 1. Taylor series for e^x 7
Argument Reduction 8
Example 2: Taylor series for e^x using argument reduction 8
The issue with arbitrary precision 9
Finding a reasonable reduction factor. 11
Brent enhancement 11
Guard Digits 12
Source exp_taylor() 13
e^x using Linear Reduction and Taylor series 14
source exp_linearTaylor() 14
Further Improvement of the methods? 14
e^x using Sine Hyperbolic function 16
Example: without argument reduction 16
Argument Reduction 17
Example: with argument reduction 17
Further improvements of the method? 18
Source for sinh() using coefficient scaling 18
Source for exp(x) using sinh(x) 20
e^x using the inverse and Newton method 21
e^x using the binary splitting method 21
Argument reduction for e^x for the binary splitting method 22
Finding a reasonable reductions factor for e^x 23
The Precision needed to avoid loss of accuracy. 23
Source for exp(x) using binary splitting 24
Which method to use for e^x? 26
Recommendation for calculating e^x 27
The constant e 28
AHJ Sale algorithm for e 28
Binary splitting method 29
Source for Stirling approximation using Newton 30
Source computeE 31
Source binarysplittingE (final version) 31
Recommendation for calculating e 32
Reference 33

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 the second optional parameter is the 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 exponents (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 and that is the class method. .adjustExponent(). This method just adds the parameter to the internal variable that holds the exponent of the float_precision variable. E.g.

```
f.adjustExponent(+1); // Add 1 to the exponent, the same as multiplying the number 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 is an additional method() 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

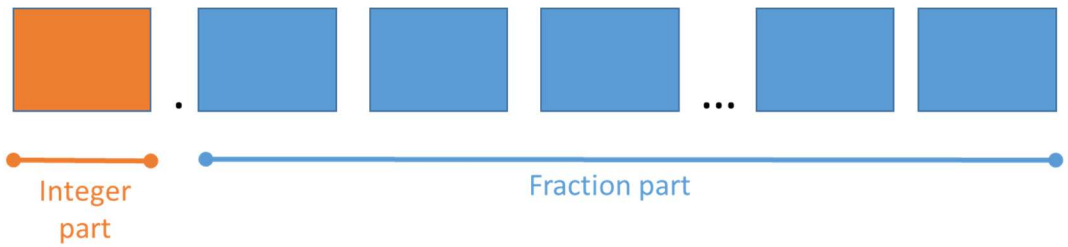
For the internal layout of the arbitrary precision number, we are using the STL vector library declared as:

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

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

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

The Binary format 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].

e^x

There are a couple of ways you can calculate exp(x) in arbitrary precision. Traditional a Taylor series expansion has been used but some have suggested the use of the sinh() function to calculate the exp(): This chapter will examine:

1) Exp(x) using Taylor series & argument reduction.
2) Exp(x) using linear reduction + Taylor series & argument reduction.
3) Exp(x) using linear reduction, Taylor series, argument reduction, and coefficient scaling.
4) Exp(x) using Sine Hyperbolic function with argument reduction & coefficients scaling.
5) Exp(x) using the inverse and Newton methods.
6) Exp(x) using the binary splitting method.

The most common one for arbitrary precision libraries is the standard Taylor series expansion method.

e^x using the Taylor series

For the function, exp(x) we can use the corresponding Taylor series for exp(x) as defined by:

$$\exp(x) = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \cdots \quad (1)$$

We eliminate x< 0 by using the identity: $e^{-x} = \frac{1}{e^x}$ meaning we first calculate e^x and then do the inverse of $\frac{1}{e^x}$.

Unfortunately, this series does not converge very fast and will require many terms to complete.

Example 1. Taylor series for e^x

Using x=1 we get after 17 Taylor series the result of exp(1)= 2.718281828459

Exp(x)		**Original**	**X Reduced**	
x=		**1**	**1**	
Argument reductions=		**0**		
Terms	**Term value**	**Term Sum**	**Exp(x)**	**Error**
1	1.00E+00	1.000000000000	1.000000000000	1.72E+00
2	1.00E+00	2.000000000000	2.000000000000	7.18E-01
3	5.00E-01	2.500000000000	2.500000000000	2.18E-01
4	1.67E-01	2.666666666667	2.666666666667	5.16E-02

5	4.17E-02	2.708333333333	2.708333333333	9.95E-03
6	8.33E-03	2.716666666667	2.716666666667	1.62E-03
7	1.39E-03	2.718055555556	2.718055555556	2.26E-04
8	1.98E-04	2.718253968254	2.718253968254	2.79E-05
9	2.48E-05	2.718278769841	2.718278769841	3.06E-06
10	2.76E-06	2.718281525573	2.718281525573	3.03E-07
11	2.76E-07	2.718281801146	2.718281801146	2.73E-08
12	2.51E-08	2.718281826198	2.718281826198	2.26E-09
13	2.09E-09	2.718281828286	2.718281828286	1.73E-10
14	1.61E-10	2.718281828447	2.718281828447	1.23E-11
15	1.15E-11	2.718281828458	2.718281828458	8.15E-13
16	7.65E-13	2.718281828459	2.718281828459	5.02E-14
17	4.78E-14	2.718281828459	2.718281828459	0.00E+00

That is not too bad, however, if we change the argument to 10 then we need 45 Taylor's terms to get the result and if we use x=0.1 then we only need 10 Taylor terms.
This lead to the observation that the number of Taylor's terms needed depends heavily on the argument to exp(x).

Argument Reduction

We prefer to have our x < 1 to ensure that the Taylor series converges more quickly. We can accomplish that using a technique called *argument reduction* to work with a smaller number to get a faster converging to e^x using fewer *terms* of the Taylor series.

We can use the identity:$e^x = (e^{\frac{x}{2}})^2$ to reduce the argument with a factor of two and then after the Taylor iterations we can square the result to find the correct value of e^x.
Or more generally we can reduce the argument x for some k where:

$$e^x = (e^{\frac{x}{2^k}})^{2^k} \tag{2}$$

Iterate through the Taylor terms of the reduced argument $\frac{x}{2^k}$ and then Square the result k times after the Taylor iterations. This makes sense since for each Taylor term you need to divide with the factorial and that is many times more time-consuming than squaring the result k times after the Taylor iterations.

Example 2: Taylor series for e^x using argument reduction

If using the previous example 1 and reducing the argument twice from one to 0.25 we only need 12 Taylor terms to get the same result as before, saving five Taylor terms but gaining two squaring at the end. However, overall huge savings since we have avoided five time-consuming divisions in Taylor's terms.

Exp(x)	**Original**	**X Reduced**

x=		1	0.25	
Argument reductions=		2		
Terms	Term value	Term Sum	Exp(x)	Error
1	1.00E+00	1.000000000000	1.000000000000	2.84E-01
2	2.50E-01	1.250000000000	2.441406250000	3.40E-02
3	3.13E-02	1.281250000000	2.694855690002	2.78E-03
4	2.60E-03	1.283854166667	2.716831973351	1.71E-04
5	1.63E-04	1.284016927083	2.718209939201	8.49E-06
6	8.14E-06	1.284025065104	2.718278851251	3.52E-07
7	3.39E-07	1.284025404188	2.718281722614	1.25E-08
8	1.21E-08	1.284025416299	2.718281825163	3.89E-10
9	3.78E-10	1.284025416677	2.718281828368	1.08E-11
10	1.05E-11	1.284025416687	2.718281828457	2.69E-13
11	2.63E-13	1.284025416688	2.718281828459	5.77E-15
12	5.97E-15	1.284025416688	2.718281828459	0.00E+00

If we use an eight-times reduction we get the same results after just six Taylors terms.

Exp(x)		Original	X Reduced	
x=		1	0.00390625	
Argument reductions=		8		
Terms	Term value	Term Sum	Exp(x)	Error
1	1.00E+00	1.000000000000	1.000000000000	3.91E-03
2	3.91E-03	1.003906250000	2.712991624253	7.64E-06
3	7.63E-06	1.003913879395	2.718274935741	9.94E-09
4	9.93E-09	1.003913889329	2.718281821729	9.71E-12
5	9.70E-12	1.003913889338	2.718281828454	7.33E-15
6	7.58E-15	1.003913889338	2.718281828459	0.00E+00

The issue with arbitrary precision

17 Taylor's terms to reach a result do not seem so bad at a first glance. However, when we are dealing with higher precisions e.g. 1,000 digits, 10,000, or even 100,000 digits we suddenly have to perform a lot more Taylor terms to find our result. In Yacas's book, of algorithms [5] they found a bound for the number of Taylor terms n needed as a function of the number of precision in digits P assuming |x|< 1:

$$n = \frac{P \cdot \ln(10)}{\ln(P)} - 1 \qquad (3)$$

For P = 1,000 digits you get n=332 Taylor terms are needed. For 10,000 digits, n=2,499, and 100,000 digits you get a whopping n=19,999 Taylor terms and 1M digits, n=166,666 terms. With that amount of Taylor terms, it will take a long time to evaluate exp(x) for high numbers of digits, see table below.

Digits	10^1	10^2	10^3	10^4	10^5	10^6	10^7	10^8	10^9
Taylor terms	9	49	332	2,499	19,999	166,666	1.43M	12.5M	111M

Now to see the effect of argument reduction on improving the Taylor series we have recorded the amount of Taylor terms needed for various argument reductions from 1 to 128 on a random floating-point number between 1.xxx and 9.xxx. From the table, we see that the reduction in the number of Taylor terms varies more than 10-fold between 1 as the reduction factors to a reduction factor of 2^{128}
The Auto reduction is the number of Taylor terms when we automatically find a reasonable reduction factor. Most of the time it varies between 32 to 64 reductions.

Digits	10	100	1,000	10,000	1,000,000
Auto Red.	5	12	75	516	4,393
1 Pred.	17	96	435	3,861	25,197
2 Red.	15	81	393	3,510	23,580
4 Red.	11	60	327	2,962	20,877
8 Red.	8	40	243	2,244	16,941
16 Red.	5	24	159	1,497	12,241
32 Red.	4	13	94	889	7,820
64 Red.	3	8	52	487	4,510
128 Red.	3	5	28	255	2,430

The total number of operations going from one Taylor term to the next is:

$$\frac{x^n}{n!} \rightarrow \frac{x \cdot x^n}{(n+1) \cdot n!}$$

Is two multiplication and one division. The n+1 can be handled using the native C++ types and does not count for the workload for arbitrary precision.

Now doing k reduction will require k multiplication before Taylor iterations and k multiplication at the back-end or 2k multiplication. The front operation multiplication for a normalized arbitrary precision number is not performed as a real multiplication (of 0.5) but handle by just subtracting one from the exponent (which is the same as dividing by two or multiply by 0.5). This does not amount to anything that counts towards the workload and can be ignored. On the back end, it will still require k multiplication. As an example, we can calculate the total workload for a 10,000 digits number using one reduction versus two reductions.

1-eductionworkload = 3,861*(2*multiplication+1 division)+1*multiplication=7,723*multiplication and 3,86*1division.

16-reduction workload: 1,497*(2*multiplication+1*division)+2*multiplication= 2996*multiplication and 1,497 division

Assuming division is 10 times slower than multiplication, you get a total workload of multiplication equivalence of 7,723+10*3,861=46,333 for 1 reduction and 17,966 or 40% reduction in workload.

Finding a reasonable reduction factor.

As can be seen in the above table a higher reduction factor greatly improved the performance. However, how many times reduction is adequate? Yacas book [5] states that at least if x should be lower to $|x|<10^{-M}$ and M should be:

$$|x| < 10^{-M} \; and \; M > \frac{\ln(P)}{\ln(10)} \qquad (4)$$

Measuring the performance indicates that this is not the most optimal selection. Therefore, we multiply the M found above with a constant eight to get a more reasonable reduction factor and then adjust for the magnitude of |x| itself.

The adjustment for the magnitude of |x| is simply the number exponent (power of 2 exponents) to ensure that the number will be well below. This works well for small magnitude |x| and for high magnitude |x|. By just adding the exponent (positive or negative) to the reduction factor.

The performance table below shows the effect of using increasingly higher reduction factors.

All measures are in milliseconds

Digits	10	100	1,000	10,000	1,000,000
Auto Red.	0.11	0.53	17	5,596	291,871
1 Pred.	0.24	2.50	59	39,812	1,810,970
2 Red.	0.20	2.00	67	38,736	1,286,680
4 Red.	0.13	1.57	50	32,372	1,104,910
8 Red.	0.09	1.11	57	24,334	898,426
16 Red.	0.08	0.71	34	16,026	652,547
32 Red.	0.10	0.53	22	9,425	413,501
64 Red.	0.24	0.71	15	5,309	241,661
128 Red.	0.59	0.59	17	3,330	131,452

As you can see for higher precision, you will benefit even more from increasing the reduction factor.

Brent enhancement

To avoid loss of precision we do not do a repeated number of squares at the back end. Instead of just squaring for every number of reductions performed.

$$e^x = (e^{\frac{x}{2}})^2 \quad (5)$$

We use the identity as suggested by Brent [6]:

$$e^x - 1 = \left(e^{\frac{x}{2}} - 1\right)\left(e^{\frac{x}{2}} + 1\right) =>$$
$$e^x - 1 = 2\left(e^{\frac{x}{2}} - 1\right) + \left(e^{\frac{x}{2}} - 1\right)^2 \quad (6)$$

Guard Digits

When summarizing a Taylor series as exp(x) you need quite a lot of summarizing and that will produce round-off errors. In Yacas [5] they estimate the round-off to be approx. per term involving one multiplication, one division, and one addition to be:

$$digits\ lost = \frac{3\ln(n)}{\ln(10)}\ where\ n\ is\ the\ number\ of\ Taylor\ terms \quad (7)$$

.

Lost digits as a function of Taylor terms

Taylor Terms	10	100	1,000	10,000	1,000,000
Lost digits.	3	6	9	12	15

Lost digits adjusted for actual Taylor's terms versus reduction factor

Digits	10	100	1,000	10,000	1,000,000
Auto Red.	2.1	3.2	5.6	8.1	10.9
1 Pred.	3.7	5.9	7.9	10.8	13.2
2 Red.	3.5	5.7	7.8	10.6	13.1
4 Red.	3.1	5.3	7.5	10.4	13.0
8 Red.	2.7	4.8	7.2	10.1	12.7
16 Red.	2.1	4.1	6.6	9.5	12.3
32 Red.	1.8	3.3	5.9	8.8	11.7
64 Red.	1.4	2.7	5.1	8.1	11.0
128 Red.	1.4	2.1	4.3	7.2	10.2

As can be seen, the maximum difference only accounts for 3-4 digits between no reduction and a high reduction factor where a higher reduction factor means less loss of digits.

For our e^x function, we use a simple guard digits calculation that we add

2+ceil(log2(precision)) as extra guard digits.

Source exp_taylor()

```
float_precision expTaylor(const float_precision x)
	{
	size_t precision=x.precision()+2+(size_t)ceil(log(x.precision()));
	unsigned int i;
	intmax_t k = 0;
	float_precision r, expx, v(x);
	const float_precision c1(1), c2(2);

	if (v.iszero())
		return v = c1;
	// Automatically calculate optimal reduction factor
	k = 8*(intmax_t)ceil(log(2)*log(precision));
	k +=  v.exponent() + 2;
	k = std::max((intmax_t)0, k);
	precision +=  k / 2;
	// Do iteration using higher precision plus compensate for reduction factor
	v.precision(precision);
	r.precision(precision);
	expx.precision(precision);

	if (v.sign() < 0)
		v.change_sign();
	v.adjustExponent(-k);

	// Do the first two iterations
	r = v.square();
	r.adjustExponent(-1);  // multiply with 0.5
	expx = c1+v+r;
	// Now iterate
	for (i = 3; ; ++i)
		{
		r *= v / float_precision(i,precision);
		if (expx + r == expx)
			break;
		expx += r;
		}

	// Brent enhancement avoids loss of significant digits when x is small.
	if (k>0)
		{
		expx -= c1;
		for (; k > 0; k--)
			expx = (c2 + expx)*expx;
		expx += c1;
		}
	if (x.sign() < 0)
		expx = _float_precision_inverse(expx);

	// Round to the same precision as argument and rounding mode
	expx.mode(x.mode());
	expx.precision(x.precision());
	loopcnt_taylor = i;
	return expx;
	}
```

e^x using Linear Reduction and Taylor series

If x > 1 then instead of adding reductions to get |x| < 1. Then we use a linear reduction until |x| < 1 and then use the standard Taylor series as above with argument reduction. The linear reduction is to subtract the n=floor(x) from x yielding a number between [0,1[then take the exp(x-n).

Using the identity:

$$e^x = e^{floor(x)}e^{x-floor(x)} \Rightarrow e^x = (e)^{floor(x)}e^{x-floor(x)} \quad (8)$$

e is a constant and can easily be computed fast even in arbitrary precision and then it is just a matter of raising it to the integer power of floor(x).

source exp_linearTaylor()

```
float_precision expLinearTaylor(const float_precision x)
	{
	size_t precision = x.precision() + 2 + (size_t)ceil(log(x.precision()));
	float_precision expx, v(x);
	const float_precision c1(1);

	if (v.iszero())
		return v = c1;
	expx.precision(precision);
	v = floor(v);
	if (v >= c1)
		{
		expx = _float_table(_EXP1, precision);
		expx = pow(expx, abs(v));
		v = expTaylor(x - v);
		expx *= v;
		}
	else
		expx = expTaylor(x);
	expx.mode(x.mode());
	expx.precision(x.precision());
	return expx;
	}
```

Further Improvement of the methods?

There is not a lot of things you can do to improve the exp(x) algorithm. However, consider the Taylor series expansion of exp(x):

$$\exp(x) = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \cdots \quad (9)$$

The issue is the division for each term. Since division is many times slower than calculation and addition. You could group two or more Taylor terms (sometimes referred to as coefficient rescaling) and reduce the number of divisions. Consider the n'th and the n+1 term:

$$\dots\frac{x^n}{n!}+\frac{x^{n+1}}{(n+1)!}\dots$$

Moreover, group them:

$$\dots\frac{(n+1)x^n}{(n+1)n!}+\frac{x^{n+1}}{(n+1)!}\dots =>$$

$$\dots\frac{(n+1)x^n+x^{n+1}}{(n+1)!}\dots$$

Then you have replaced one division with an extra multiplication. The (n+1) can be done using a 32-bit or 64-bit integer since you never get to do many Taylor terms in real life. There is no need to stop at just grouping two terms together you can do that for three terms:

$$\dots\frac{(n+1)(n+2)x^n+(n+2)x^{n+1}+x^{n+2}}{(n+2)!}\dots =>$$

$$\dots\frac{x^n(x^2+(n+2)x+n^2+3n+2)}{(n+2)!}\dots$$

Saving two divisions, however, gaining a few more addition and multiplications.

In general, you can add a g group together:

$$\frac{\sum_n^{n+g}(\prod_{i=1}^{g}(n+i))x^{n+i-1}}{(n+g)!} \quad (10)$$

Because arbitrary precision division is, much more time-consuming to calculate it will be highly advantageous to implement this grouping of Taylor terms. With four to five terms grouped, you get a speedup of 2-3 times compared to not grouping terms together.

Iterations Source for 5 terms rescaling of coefficients replacing:

```
// Now iterate
for (i = 3; ; ++i)
	{
	r *= v / float_precision(i, precision);
	if (expx + r == expx)
		break;
	expx += r;
	}
```

With this:

```
const float_precision v2(v.square()), v3(v2*v), v4(v2*v2);
float_precision terms(0,precision);
// Now iterate
for (i = 3; ; i += 5)
	{
	uintmax_t j = (i + 2)*(i + 3)*(i + 4);
	r *= v / float_precision(i*(i + 1)*j, precision);
	terms = r*(float_precision((i + 1)*j) + float_precision(j)*v +
float_precision((i + 3)*(i + 4))*v2 + float_precision(i + 4)*v3 + v4);
	if (expx + terms == expx)
		break;
	expx += terms;
	r *= v4;
	}
```

e^x using Sine Hyperbolic function

Less use but the fastest way to calculate exp(x) is using the Sine Hyperbolic function using the identity:

$$\exp(x) = \sinh(x) + \sqrt{1 + \sinh(x)^2} \quad (11)$$

Where the sinh(x) can be found with the Taylor series:

$$\sinh(x) = x + \frac{x^3}{3!} + \frac{x^5}{5!} + \frac{x^7}{7!} + \frac{x^9}{9!} \ldots \quad (12)$$

The sinh(x) Taylor series looks familiar to the Taylor series for exp(x) (every second term is removed):

$$\exp(x) = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \cdots \quad (13)$$

Except that, for each term, we go faster towards zero with the sinh(x) and we should expect that we would need fewer Taylor terms for a given precision compare to the exp(x) Taylor series.

Example: without argument reduction

Using no argument reduction. We need 9 Taylor terms to get the result compared to 17 for exp(x) using the Taylor series.

Exp(x)		**Original**	**X Reduced**	
x=		**1**	**1**	
Argument reductions=		**0**		
Terms	**Term value**	**Term Sum**	**Exp(x)**	**Error**
1	1.00E+00	1.00000000000	2.4142135624	3.04E-01

2	1.67E-01	1.16666666667	2.7032574095	1.50E-02
3	8.33E-03	1.17500000000	2.7179274124	3.54E-04
4	1.98E-04	1.17519841270	2.7182769296	4.90E-06
5	2.76E-06	1.17520116843	2.7182817840	4.44E-08
6	2.51E-08	1.17520119348	2.7182818282	2.84E-10
7	1.61E-10	1.17520119364	2.7182818285	1.35E-12
8	7.65E-13	1.17520119364	2.7182818285	4.88E-15
9	2.81E-15	1.17520119364	2.7182818285	0.00E+00

Argument Reduction

As for the regular Taylor, series for exp(x), it is clear that we prefer to have our |x| < 1 to ensure that the Taylor series converge more quickly. We again use *argument reduction* to work with a smaller number to get a faster converging to e^x using fewer *terms* of the Taylor series.

We can use the trisection identity: $\sinh(3x) = \mathrm{Sinh}(x)(3 + 4\mathrm{Sinh}(x)^2)$ to reduce the argument with a factor of three and then after the Taylor iterations we restore and find the correct value for sinh(x) by applying this formula the same number of times we did when reducing the argument.

Example: with argument reduction

With two reductions, you get the result after only five Taylor terms compare to 12

Exp(x)		**Original**	**X Reduced**	
x=		**1**	**0.111111111**	
Taylor reductions=		**2**		
Terms	**Term value**	**Term Sum**	**Exp(x)**	**Error**
1	1.11E-01	0.11111111111	2.7127251898	5.56E-03
2	2.29E-04	0.11133973480	2.7182783961	3.43E-06
3	1.41E-07	0.11133987592	2.7182818275	1.01E-09
4	4.15E-11	0.11133987596	2.7182818285	1.73E-13
5	7.11E-15	0.11133987596	2.7182818285	0.00E+00

With 8 times reduction you get the result after two 2 Taylor terms compare to 6 using standard exp(x) Taylor series.

Exp(x)		**Original**	**X Reduced**	
x=		**1**	**0.000152416**	
Taylor reductions=		**8**		
Terms	**Term value**	**Term Sum**	**Exp(x)**	**Error**
1	1.52E-04	0.00015241579	2.7182818179	1.05E-08
2	5.90E-13	0.00015241579	2.7182818285	0.00E+00

Granted it is not fair to compare it this way since the standard exp(x) argument reduction is only a factor of two per reduction compared to a factor of three using the sinh(x) trisection identity.

Further improvements of the method?

The same technique for coefficient rescaling (grouping of Taylor terms) can be applied here as well. Consider the Taylor series for sine hyperbolic:

$$\sinh(x) = x + \frac{x^3}{3!} + \frac{x^5}{5!} + \frac{x^7}{7!} + \frac{x^9}{9!} \ldots \quad (14)$$

The issue again clearly is the division for each term. Since division is many times slower than calculation and addition. You could group two or more Taylor terms (sometimes referred to as coefficient rescaling) and reduce the number of divisions. Consider the n'th and the n+1 term:

$$\ldots \frac{x^n}{n!} + \frac{x^{n+2}}{(n+2)!} \ldots$$

Moreover, group them:

$$\ldots \frac{(n+1)(n+2)x^n}{(n+1)(n+2)n!} + \frac{x^{n+2}}{(n+2)!} \ldots =>$$

$$\ldots \frac{(n+1)(n+2)x^n + x^{n+2}}{(n+2)!} \ldots$$

Then you have replaced one division with two extra multiplication. The (n+1)(n+2) can be done using 64-bit integer arithmetic since you never get to do some many Taylor terms in real life that it will overflow. There is no need to stop at just grouping two terms together you can do that for three terms:

For grouping three Taylor terms, you get:

$$\ldots \frac{(n+1)(n+2)(n+3)(n+4)x^n + (n+3)(n+4)x^{n+2} + x^{n+4}}{(n+4)!} \ldots =>$$

$$\ldots \frac{(n+3)(n+4)(\,(n+1)(n+2)x^n + x^{n+2}) + x^{n+4}}{(n+4)!} \ldots$$

Source for sinh() using coefficient scaling

```
float_precision sinh(const float_precision& x)
	{
	const int group=3;
	size_t precision = x.precision() + 2+(size_t)ceil(log10(x.precision()));
```

```
	size_t loopcnt = 2;
	intmax_t k;
	uintmax_t i;
	float_precision r, sinhx, v(x), vsq, terms;
	const float_precision c1(1), c3(3), c4(4);

	if (x.sign() < 0)
		v.change_sign();

	// Automatically calculate optimal reduction factor as a power of two
	k = 8 * (intmax_t)ceil(log(2)*log(precision));
	k += v.exponent() + 2;

	// Now use the trisection identity sinh(3x)=sinh(x)(3+4Sinh^2(x))
	// until the argument has been reduced 2/3*k times.
	// Converting power of 2 to power of 3.
	k = 2*(intmax_t)ceil(2.0*k / 3);

	// Adjust the precision
	precision += k;
	v.precision(precision);
	r.precision(precision);
	sinhx.precision(precision);
	vsq.precision(precision);
	terms.precision(precision);
	r = c3;
	r = pow(r, float_precision(k)); // Since r and k is an integer this is fast
	v /= r;
	vsq = v.square();
	r = v;
	sinhx = v;

	if (group == 1)
		{// No Coefficients rescaling
		// Now iterate using Taylor expansion
		for (i = 3;; i += 2, ++loopcnt)
			{
			r *= vsq / float_precision(i*(i - 1));
			if (sinhx + r == sinhx)
				break;
			sinhx += r;
			}
		}
	else
		{
		std::vector<float_precision> vn(group);  // vn[0] is not used
		std::vector<float_precision> cn(group);

		for (i = 0; i < group; ++i)
			{
			cn[i].precision(precision); vn[i].precision(precision);
			if (i == 1) vn[i] = vsq;
			if (i > 1) vn[i] = vn[i-1] * vsq;
			}
		// Now iterate
		for (i = 3; ; )
			{
			int j;
```

```
                    for (j = group - 1; j >= 0; --j)
                        {
                        if (j == group - 1)
                            {
                            cn[j] = float_precision((i + 2*j-1)*(i+2*j));
                            }
                        else
                            {
                            cn[j] = cn[j + 1] * float_precision((i + 2*j-
1)*(i+2*j));
                            }
                        }
                    for (j = 2, terms = cn[1]; j < group; ++j)
                        terms += cn[j] * vn[j - 1];
                    terms += vn[group - 1];
                    r *= vsq / cn[0];
                    terms *= r;
                    i += 2*group;               // Update term count
                    loopcnt += group;
                    if (sinhx + terms == sinhx) // Reach precision
                        break;                  // yes terminate loop
                    sinhx += terms;             // Add Taylor terms to result
                    if (group > 1)
                        r *= vn[group - 1];  // ajust r to last Taylor term
                    }
                }

        for (; k > 0; k--)
                {
                sinhx *= (c3 + c4*sinhx.square());
                }

        // Round to same precision as argument and rounding mode
        sinhx.mode(x.mode());
        sinhx.precision(x.precision());
        if (x.sign() < 0)
                sinhx.change_sign();
        return sinhx;
        }
```

Source for exp(x) using sinh(x)

```
float_precision expSinh(const float_precision& x, unsigned int klimit, int group =
1)
        {
        size_t precision=x.precision()+2+(size_t)ceil(log10(x.precision()));
        float_precision v(x);
        const float_precision c1(1);

        v.precision(precision);
        if (v.sign() < 0)
                v.change_sign();

        if (floor(v) == v)  // v is an Integer
                {// use the 100 times faster shortcut exp(v)=exp(1)^v
                v = _float_table(_EXP1, precision);
                v = pow(v, abs(x));
```

```
        }
    else
        {
        v = sinh(v,klimit,group);
        v += sqrt(c1 + v.square());
        v.precision(precision);
        }

    if (x.sign() < 0)
        v = _float_precision_inverse(v);
    // Round to the same precision as argument and rounding mode
    v.mode(x.mode()); v.mode(x.mode());
    v.precision(x.precision());
    return v;
    }
```

e^x using the inverse and Newton method

This method is only relevant if you have a very fast way to compute ln(x). Which you usually do not have when using arbitrary precision. The method solves the equation x=exp(y) by taking the ln() of both sides ln(x) = y and then solving it using the Newton method, which yields the iteration:

$$x_{n+1} = x_n(1 + y - \ln(x_n)) \quad (15)$$

The Newton method has a quadratic convergence rate doubling the number of correct digits for each iteration. However, it is many times slower than any of the previous methods.

e^x using the binary splitting method

This method expands on the same method for calculating e, see the section on constants.

It used the Taylor series for exp(x):

$$\exp(x) = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \cdots \quad (16)$$

However, instead of calculating the series as above we implement it using the binary splitting method.
The binary splitting methods (see [8]) equate the Taylor series terms with two variables p and q and then it is just a matter of dividing p with q to get the approximation for e.

$$e = \sum_{k=0}^{\infty} \frac{1}{k!} = 1 + \frac{x}{1} + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \cdots = \frac{P(0,k)}{Q(0,k)} \quad (17)$$

Here Q(0,k) is an integer, but P(0,k) is a *float_precision* variable. (Since x can be any real value). The notation P(0,k)/Q(0,k) represents the first k terms of the above series. For any

given value of a & b, we can compute P(a,b) and Q(a,b) as follows using the binary splitting method. (a and b are integers and a<b) following the recursion:

Algorithm: Binary splitting method for e

$m = \frac{a+b}{2}$ *integer division*
P(a,b)=P(a,m)Q(m,b)+P(m,b)
Q(a,b)=Q(a,m)Q(m,b)
And P(b-1,b)=x^b; Q(b-1,b)=b;

Algorithm 1

You continue this recursive breakdown until a+1=b and you set P(a,b)=x^b and Q(a,b)=b and let the formula reverse bottom up.

Argument reduction for e^x for the binary splitting method

Now to make the algorithm efficient we need to ensure that |x| <1. That can be done easily by just using argument reduction as previously describe under exp(x) using the Taylor series. We expect that if |x| << 1 then the Taylor series will converge faster.

To calculate how many Taylor terms we need as a function of required decimal digits of e. We resort to the Stirling approximation formula for ! We notice that to get P decimal precision of e^x and the number of Taylor terms is k we need it to satisfy the equation that:

$$\frac{x^k}{k!} < 10^{-P} \quad (18)$$

Where we use the Stirling approximation for k!:

$$k! \approx \left(\frac{k}{e}\right)^k \sqrt{2\pi k}\,, Stirling\ approximation \quad (19)$$

This yield:

$$\frac{x^k}{\left(\frac{k}{e}\right)^k \sqrt{2\pi k}} < 10^{-P} =>$$

Taking log() on both sides you get:

$$-k \cdot \log(x) + k \cdot (\log(k) - 1) + \frac{1}{2}\log(2\pi k) > P \cdot log(10) \quad (20)$$

To solve this for k, we can use Newton's methods that find a solution within a few iterations. Notice we only need to find the next higher integral number for k.

Taylor terms needed as a function of x

Digits	10	100	1,000	10,000	100,000	1,000,000

x						
1	14	70	450	3,249	25,206	205,022
10^{-1}	7	45	325	2,521	20,502	172,350
10^{-2}	5	33	252	2,050	17,235	148,429
10^{-3}	4	25	205	1,724	14,843	130,202
10^{-4}	3	21	173	1,484	13,020	115.878
10^{-5}	2	18	149	1,302	11,588	104,339
10^{-6}	2	15	130	1,159	10,434	94,852
10^{-7}	2	13	116	1,044	9,485	86,920
10^{-8}	2	12	105	949	8,692	80,194
10^{-9}	2	11	95	869	8,020	74,419

The above table clearly shows the effect of using the argument reduction technic in the binary splitting method. We can apply the same argument reduction formula already established at the start of the explanation of e^x.

Finding a reasonable reductions factor for e^x

As can be seen in the above table a higher reduction factor greatly improved the performance. However, how many times reduction is adequate? Yacas book [6] states that at least x should be lower to $|x|<10^{-M}$ and M should be:

$$|x| < 10^{-M} \; and \; M > \frac{\ln(P)}{\ln(10)} \qquad (21)$$

Where P is the precision in decimal digits.

Measuring the performance indicates that this is not the most optimal selection. Therefore, we multiply the M found above with a constant eight to get a more reasonable reduction factor and then adjust for the magnitude of |x| itself.

The adjustment for the magnitude of |x| is simply the number exponent (power of 2 exponents) to ensure that the number will be well below one. This works well for small magnitude |x| and for high magnitude |x|. By just adding the exponent (positive or negative) to the reduction factor.

The Precision needed to avoid loss of accuracy.

Looking at the algorithm we can see for P(a,b):

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

We multiply each P(a,m) with Q(m,b) where Q is the factorial. This will create pretty big numbers as we increase the number of terms we need. To see how big we can again use the Stirling approximation for !

$$k! \approx \left(\frac{k}{e}\right)^k \sqrt{2\pi k}\,, Stirling\ approximation \qquad (22)$$

Using log10(k!) We find the number of decimal digits as the size of k!

$$\log 10(k!) \approx log10(\left(\frac{k}{e}\right)^k \sqrt{2\pi k}\,) =>$$

$$\mathrm{k} \cdot log10(k) - k + \frac{1}{2} log10(2\pi k) \approx \mathrm{k} \cdot log10(k) - k, for\ large\ k \qquad (23)$$

Digits	10	100	1,000	10,000	100,000	1,000,000
Size of k! in decimal digits	9	100	2,000	30,000	400,000	5,000,000

Table of the decimal size of various values for !

As expected,! Is a powerful factor where we need to adjust upward the needed accuracy or precision when we calculate ex at some precision. The adjustment amount is much larger than we are used to dealing with using regular methods for e^x. However, if we use argument reduction it counteracts the need to handle calculation with a significantly higher number of digits.

Source for exp(x) using binary splitting

The source consists of 3 functions. ComputeExp() that drives the recursive function binarysplittingExp() and an outer function xsterling_approx that calculates the needed number of terms to evaluate

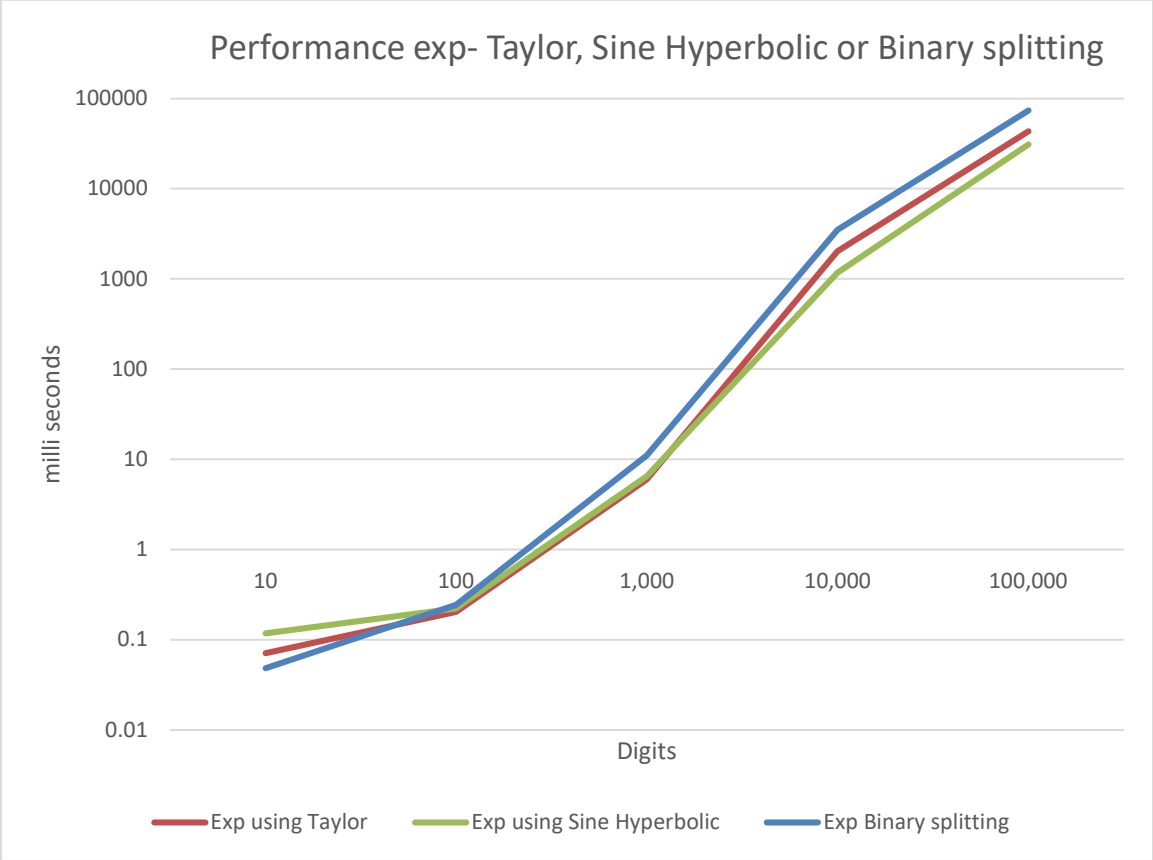
```
// Stirling approximation for calculating e^x
static uintmax_t xstirling_approx(uintmax_t digits, eptype xexpo)
{
	double xnew, xold;
	const double test = (digits + 1) * log(10);
	// x^n/k!<10^-p, where p is the precision of the number
	// x^n~2^x'exponent
	// Stirling approximation of k!~Sqrt(2*pi*k)(k/e)^k.
	// Taken ln on both sides you get:
	//     -k*log(2^xexpo) + k*(log((k)-1)+0.5*log(2*pi*m)=test=>
	//     -k*xexpo*log(2) + k*(log((k)-1)+0.5*log(2*pi*m)=test
	// Use the Newton method to find in less than 4-5 iteration
	for (xold = 5, xnew = 0; ; xold = xnew)
	{
	double  f = -xold*xexpo*log(2)+xold*(log(xold)-1)+0.5*log(2*
3.141592653589793 * xold);
		double f1 = 0.5 / xold + log(xold) - xexpo * log(2);
		xnew = xold - (f - test) / f1;
		if ((uintmax_t)ceil(xnew) == (uintmax_t)ceil(xold))
			break;
	}
	return (uintmax_t)ceil(xnew);
}
```

```
static void binarysplittingExp(const float_precision& x, float_precision& xp,
const uintmax_t a, const uintmax_t b, float_precision& p, float_precision& q)
{
	float_precision pp(0, x.precision() + 1);
	float_precision qq(0,pp.precision());
	uintmax_t mid;

	if (b - a == 1)
	{// No overflow using 64bit arithmetic
		xp *= x;
		p = xp;
		q = float_precision(b);
		return;
	}
	if (b - a == 2)
	{// No overflow using 64bit arithmetic if b<=4'294'967'296
		xp *= x;
		p = x + float_precision(b);
		p *= xp;
		xp *= x;
		q = float_precision(b * (b - 1));
		return;
	}

	mid = (a + b) / 2;
	binarysplittingExp(x, xp, a, mid, p, q);  // interval [a..mid]
	binarysplittingExp(x, xp, mid, b, pp, qq);// interval [mid..b]
	// Reconstruct interval [a..b] and return p & q
	p *= qq;
	p += pp;
	q *= qq;
}


// Binary splitting with recursion and argument reduction.
static float_precision computeExp(const float_precision& x, int kk = 1)
{
	size_t precision = x.precision()+2+(size_t)ceil(log10(x.precision()));
	const float_precision c1(1), c2(2);
	float_precision p, pp, v(x), xp(1), q, qq;
	uintmax_t k;
	intmax_t r;

	// Automatically calculate optimal reduction factor as a power of two
	r = 8 * (intmax_t)ceil(log(2) * log(precision));
	r += v.exponent() + 1;
	r = std::max((intmax_t)0, r);

	// Adjust the precision
	precision += (intmax_t)floor(log10(precision)) * r;
	v.precision(precision);
	p.precision(precision);
	pp.precision(precision);
	xp.precision(precision);
	q.precision(precision);
	qq.precision(precision);

	// e^-x==1/e^x
	if (v.sign() < 0)
		v.change_sign();
	v.adjustExponent(-r);
```

```
        // Calculate needed Taylor terms
        k = xstirling_approx(v.precision(), v.exponent());
        if (k < 2)
                k = 2;  // Minimum 2 terms otherwise it cant split

        //Need to calculate [0..k]
        binarysplittingExp2(v, xp, 0, k, p, q);
        // Adjust and calculate exp(x)
        pp = q;
        p += pp;
        p /= pp;

        // Reverse argument reduction
        // Brent enhancement avoids loss of significant digits when x is small.
        if (r > 0)
        {
                p -= c1;
                for (; r > 0; --r)
                        p *= (c2 + p);
                p += c1;
        }
        if (x.sign() < 0)
                p = _float_precision_inverse(p);

        // Round to the same precision as argument and rounding mode
        p.mode(x.mode());
        p.precision(x.precision());
        return p;
}
```

Which method to use for e^x?

By measuring the performance, we get a clear advantage of using the sine hyperbolic function to calculate e^x, particularly with an increasing number of digits. The use of the Binary splitting method is interesting but lacks the performance of the two other methods.



Time in milliseconds between the three methods for evaluating e^x

Recommendation for calculating e^x

Based on the performance measure recommend:

1) Exp() using sinh() is the fastest of the two methods and is therefore recommended.
2) Always use argument reduction to increase performance
3) For large e^x, the initial linear reduction seems to work faster than just using argument reduction to lower the number.
4) Coefficient rescaling (or grouping of terms) can speed up calculation by a factor of two-three and is therefore recommended.

The constant e

The transcendental constant e (same as exp(1)) can be more beneficial calculated by other methods than the ones presented in the previous section. There is a Spigot-like algorithm from the computer Journal 1968 (A H J Sale) that I have modified to serve the purpose of use in the arbitrary precision library. The algorithm is a magnitude faster than using the Taylor series for calculating exp(1) even with the enhancement presented in this paper. Please ref to [3] for further details. However, that is not the only fast algorithm. We will also present calculating e using the binary splitting method.

AHJ Sale algorithm for e

The original algorithm was presented in [3] back in the sixties and accompanied by an Algol 60 version. The original code has been ported to the C++ environment with a few additional improvements. The result of the calculation is delivered as a decimal string see the source code below. The function is called with the wanted number of digits for e. Based on this the needed number of Taylor Terms is calculated and then the main loop delivers one decimal number per loop.

Source AHJ Sale algorithm for e

```
// From The Computer Journal 1968 (A H J Sale) written in Algol 60 and ported with
some modification
// to c++
static std::string spigot_e(const size_t digits)
	{
	unsigned int m;
	unsigned int tmp, carry;
	double test = (digits + 1) * log(10);
	bool first_time = true;
	unsigned int *coef;
	std::string ss("2.");
	ss.reserve(digits + 16);
	double xnew, xold;

	// Stirling approximation of m!~Sqrt(2*pi*digits)(digits/e)^digits.
	// Taken ln on both sides you get:
	//     m*(Math.log((m)-1)+0.5*Math.log(2*Math.pi*m);
	// Use the Newton method to find in less than 4-5 iterations
	for (xold = 5, xnew = 0; ; xold = xnew)
		{
		double  f=xold*(log(xold)-1)+0.5*log(2*3.141592653589793*xold);
		double f1 = 0.5 / xold + log(xold);
		xnew = xold - (f - test) / f1;
		if ((int)ceil(xnew) == (int)ceil(xold))
			break;
		}
	m = (unsigned int)ceil(xnew);
	if (m < 5)
		m = 5;
	coef = new unsigned int[m + 1];

	for (size_t i = 1; i < digits; ++i, first_time = false)
		{
```

```
            carry = 0;
            for (unsigned int j = m; j >= 2; j--)
                  {
                  if (first_time == true)
                        tmp = 10;
                  else
                        tmp = coef[j] * 10;
                  tmp += carry;
                  carry = tmp / (j);
                  coef[j] = tmp % (j);
                  }
            ss.append(1, (char)(carry + '0'));

      delete[] coef;
      return ss;
      }
```

Binary splitting method

The binary splitting methods (see [8]) equate the Taylor series terms with two integers p and q and then it is just a matter of dividing p with q to get the approximation for e.

$$e = \sum_{k=0}^{\infty} \frac{1}{k!} = 1 + \frac{1}{1} + \frac{1}{2} + \frac{1}{6} + \frac{1}{24} + \cdots = \frac{P(0,k)}{Q(0,k)} \qquad (24)$$

The notation P(0,k)/Q(0,k) represents the first k terms of the above series. For any given value of a & b, we can compute P(a,b) and Q(a,b) as follows using the binary splitting method. (a and b are integers and a<b) following the recursion:

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

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

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

And P(b-1,b)=1; Q(b-1,b)=b;

Algorithm 2

You continue this recursive breakdown until a+1=b and you set P(a,b)=1 and Q(a,b)=b and let the formula reverse bottom up.

Source Binary splitting for e

```
void binarysplittingE(const uintmax_t a, const uintmax_t b, int_precision& p,
int_precision& q)
      {
      int_precision pp, qq;
      uintmax_t mid;

      if (b - a == 1)
            {// No overflow using 64bit arithmetic
            p = int_precision(1);
            q = int_precision(b);
```

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

Notice if you need more than the first 19 Taylor Terms you will need more than 64-bit variables to hold p and q. You would need to switch to arbitrary integer precision. This is done using the type int_precision (instead of e.g. uintmax_t for 64-bit environment) from the author's arbitrary precision packages.

To calculate how many Taylor terms we need as a function of required decimal digits of e. We resort to the Stirling approximation formula for! We notice that to get the P decimal precision of e and the number of Taylor terms is k we need it to satisfy the equation that $k!>10^P$.

$$k! \approx \left(\frac{k}{e}\right)^k \sqrt{2\pi k}\,, Stirling\ approximation \qquad (25)$$

$$\left(\frac{k}{e}\right)^k \sqrt{2\pi k} > 10^P \Rightarrow$$

Taking log() on both sides you get:

$$k \cdot (\log(k) - 1) + \frac{1}{2}\log(2\pi k) > P \cdot \log(10) \qquad (26)$$

To solve this for k, we can use Newton's methods that find a solution within a few iterations. Notice we only need to find the next higher integral number for k.

Source for Stirling approximation using Newton

```
uintmax_t stirling_approx(uintmax_t digits)
      {
      double xnew, xold;
      const double test = (digits + 1) * log(10);
      // Stirling approximation of k!~Sqrt(2*pi*k)(k/e)^k.
      // Taken ln on both side you get: k*(log((k)-1)+0.5*log(2*pi*m);
      // Use the Newton method to find in less than 4-5 iteration
      for (xold = 5, xnew = 0; ; xold = xnew)
            {
            double f=xold*(log(xold)-1)+0.5*log(2*3.141592653589793*xold);
            double f1=0.5/xold+log(xold);
            xnew = xold - (f - test) / f1;
            if ((uintmax_t)ceil(xnew) == (uintmax_t)ceil(xold))
                  break;
            }
      return (uintmax_t)ceil(xnew);
      }
```

The binary splitting is called from the below source. Which set up the call to binarysplittingE() and call the stirling_approx() for calculating the needed number of Taylor terms.

Source computeE

```
// Binary splitting requested digits decimal numbers of e
 float_precision computeEdigit(const uintmax_t digits)
       {
       uintmax_t k;
       int_precision p, q;
       float_precision fp, fq;

       fp.precision(digits); // Set the precision to digits decimal digits
       fq.precision(digits); // Set the precision to digits decimal digits
       // Calculate the required number of Taylor terms from digits
       k = stirling_approx(digits);
       binarysplittingE(0, k, p, q);
       p += q;
       fp = float_precision(p,digits); fq = float_precision(q,digits);
       fp /= fq;
       return fp;
       }
```

To reduce the number of recursive calls and increase the performance you would not have to wait until a+1=b before setting p, q for the first time. We can use a pre-calculated formula that calculates p and q directly when a+2=b, a+3=b, and a+4=b to reduce the number of recursive calls we make and speed up the performance.

We can now present the final version of binarysplittingE().

Source binarysplittingE (final version)

```
void binarysplittingE(const uintmax_t a, const uintmax_t b, int_precision& p,
int_precision& q)
       {
       int_precision pp, qq;
       uintmax_t mid;

       if (b - a == 1)
              {// No overflow using 64bit arithmetic
              p = int_precision(1);
              q = int_precision(b);
              return;
              }
       if (b - a == 2 /* && b <= 4'294'967'296ull*/)
              {// No overflow using 64bit arithmetic if b<=4'294'967'296
              p = int_precision(b + 1);
              q = int_precision(b*(b - 1));
              return;
              }
       if (b - a == 3 && b <= 2'642'245ull)
              {// No overflow using 64bit arithmetic if b<=2'642'245
              uintmax_t b1 = b*(b - 1);
              p = int_precision(b1 + b + 1);
              q = int_precision(b1*(b - 2));
```

```
            return;
            }
      if (b - a == 4 && b <= 65'536ull)
            {// No overflow using 64bit arithmetic if b<=65'536
            uintmax_t b1 = b*(b - 1), b2 = b1*(b - 2);
            p = int_precision(b2 + b1 + b + 1);
            q = int_precision(b2*(b - 3));
            return;
            }
      mid = (a + b) / 2;
      binarysplittingE(a, mid, p, q);  // interval [a..mid]
      binarysplittingE(mid, b, pp, qq);// interval [mid..b]
      // Reconstruct interval [a..b] and return p & q
      p = p*qq + pp;
      q *= qq;
      }
```

Recommendation for calculating e

Use the binary splitting method, which is approx. 20 times faster than the AHJ Sale method when calculating e with 100,000 digits. The binary splitting method is approx. 40 times faster than using the Taylor series for exp(1) describe in the previous chapter.

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) Practical implementation of Spigot Algorithms for Transcendental Constants. Practical implementation of Spigot Algorithms for transcendental constants (hvks.com)
4) Methods of Computing square roots; May 17-2013; http://en.wikipedia.org/wiki/Methods_of_computing_square_roots
5) The Yacas book of algorithms, Version 1.3.3, April 1 2013 by the Yacas team
6) Richard Brent & Paul Zimmermann, Modern Computer Arithmetic, Version 0.5.9 17 October 2010; http://maths-people.anu.edu.au/~brent/pd/mca-cup-0.5.9.pdf
7) The Math behind arbitrary precision for integer and floating point arithmetic. The Math behind arbitrary precision (hvks.com)
8) Methods of computing binary splitting. Mathematical Constants and computation (free.fr) (direct link) Binary splitting method (free.fr)