

Fast Exponential function for Arbitrary Precision number.

By Henrik Vestermark (hve@hvks.com)

Abstract:

This paper is a follow-up to a previous paper that describes the math behind arbitrary precision numbers. First, the original paper was written in 2013, and quite a few things have happened since then. Secondly, I have encountered other exciting methods for calculating the exponential function. The paper describes how to do e^x -calculation with arbitrary precision and outlines some traditional techniques. It 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. For the function e^x , the Taylor series is not particularly fast in its raw form. However, you can apply techniques that significantly improve the method's performance. We will discuss the various methods for calculating e^x and elaborate on techniques like clever argument reduction and coefficient scaling to enhance the method's performance.

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

This paper is part of a series of arbitrary precision papers describing methods, implementation details, and optimization techniques. 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](#)

Fast Exponential function for Arbitrary Precision number

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

17-June 2024. A section for partial and full coefficient scaling was added.

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. A section on the binary splitting method for e was added.

Fast Exponential function for Arbitrary Precision number

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 the function e^x	7
Argument Reduction.....	8
Example 2: Taylor series, for e^x , using argument reduction	8
The issue with arbitrary precision.....	9
How to Find a reasonable reduction factor?	11
Brent enhancement.....	11
Guard Digits.....	12
Source <code>exp_taylor()</code>	13
e^x using Linear Reduction and Taylor series	14
source <code>exp_linearTaylor()</code>	14
e^x using the coefficient scaling of the Taylor series	14
Further Improvement of the Taylor series methods?	14
No coefficient scaling	15
Partial coefficient scaling.....	15
Full coefficient scaling.....	15
Partial coefficient scaling.....	15
Full coefficient scaling.....	17
e^x using the Sine Hyperbolic function	19
Source for <code>exp(x)</code> using <code>sinh(x)</code>	20
e^x using the inverse and Newton method.....	20
e^x using the binary splitting method.....	21
Argument reduction, for e^x , for the binary splitting method.....	21
Finding a reasonable reduction factor for e^x	23
The precision needed to avoid loss of accuracy.....	23
Source for e^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 <code>computeE</code>	31
Source <code>binarysplittingE</code> (final version).....	31
Recommendation for calculating e	32
Reference	33

Fast Exponential function for Arbitrary Precision number

The Arbitrary precision library

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

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

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

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

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

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

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

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

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

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

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

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

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

Fast Exponential function for Arbitrary Precision number

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

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

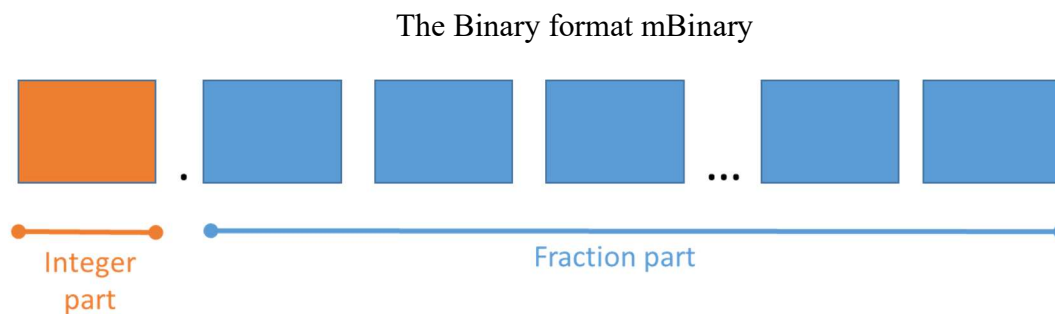
Internal format for float_precision variables

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

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

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

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



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

Other internal class variables like the sign, exponent, precision, and rounding mode are not crucial for understanding the code segments.

Normalized numbers

Fast Exponential function for Arbitrary Precision number

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].

Fast Exponential function for Arbitrary Precision number

e^x

You can calculate e^x in arbitrary precision in several ways. Traditional Taylor series expansion has been used, but some have suggested using the $\sinh()$ function to calculate the $\exp()$. This chapter will examine:

- 1) e^x using the Taylor series & argument reduction.
- 2) e^x using linear reduction + Taylor series & argument reduction.
- 3) e^x using coefficient scaling of the Taylor series & argument reduction.
- 4) e^x using the Sine Hyperbolic function.
- 5) e^x using the inverse and Newton methods.
- 6) e^x using the binary splitting method.

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

e^x using the Taylor series

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

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

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 the function 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

Fast Exponential function for Arbitrary Precision number

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, we need 45 Taylor's terms to get the result, and if we use $x=0.1$, then we only need 10 Taylor's terms. This leads to the observation that the number of Taylor's terms needed depends heavily on the argument.

Argument Reduction

We prefer to have our $x < 1$ to ensure the Taylor series converges more quickly. We can accomplish that using an argument reduction technique to work with a smaller number to converge faster 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, we can square the result after the Taylor iterations to find the correct value for e^x .

Or, more generally, we can reduce the argument x for some k where:

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

Iterate through the Taylor terms of the reduced argument. $\frac{x}{2^k}$ Then, square the result k times after the Taylor iterations. This makes sense since, for each Taylor term, you need to divide it with the factorial, which is much more time-consuming than squaring the result k times after the Taylor iterations.

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

If we use the previous example 1 and reduce 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, we have 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	

Fast Exponential function for Arbitrary Precision number

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

We get the same results after six Taylor terms if we use an eight-times reduction.

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 first glance. However, when dealing with higher precisions, e.g., 1,000, 10,000, or even 100,000 digits, we suddenly have to perform many 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 $|x| < 1$:

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

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, $n=166,666$. With that amount of Taylor terms, evaluating e^x for high numbers of digits will take a long time. See the table below.

Digits	10^1	10^2	10^3	10^4	10^5	10^6	10^7	10^8	10^9
--------	--------	--------	--------	--------	--------	--------	--------	--------	--------

Fast Exponential function for Arbitrary Precision number

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, reasonable reductions vary between 32 and 64.

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:

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

It 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.

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). Still, it is handled by subtracting one from the exponent (the same as dividing by two or multiplying 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. For example, we can calculate the total workload for a 10,000-digit number using one reduction versus two reductions.

1-reduction workload = $3,861 \cdot (2 \cdot \text{multiplication} + 1 \cdot \text{division}) + 1 \cdot \text{multiplication} = 7,723 \cdot \text{multiplication}$ and $3,861 \cdot \text{division}$.

16-reduction workload: $1,497 \cdot (2 \cdot \text{multiplication} + 1 \cdot \text{division}) + 2 \cdot \text{multiplication} = 2,996 \cdot \text{multiplication}$ and $1,497 \cdot \text{division}$

Fast Exponential function for Arbitrary Precision number

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

How to Find a reasonable reduction factor?

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

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

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 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 high magnitude $|x|$ by 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, you will benefit even more from higher precision by increasing the reduction factor.

Brent enhancement

Fast Exponential function for Arbitrary Precision number

We do not do a repeated number of squares at the back end to avoid loss of precision, 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]:

$$\begin{aligned} e^x - 1 &= (e^{\frac{x}{2}} - 1)(e^{\frac{x}{2}} + 1) \Rightarrow \\ e^x - 1 &= 2(e^{\frac{x}{2}} - 1) + (e^{\frac{x}{2}} - 1)^2 \end{aligned} \quad (6)$$

Guard Digits

When summarizing a Taylor series as e^x , you need quite a lot of summarizing, which will produce round-off errors. In Yacas [5], they estimate the round-off error involving one multiplication, one division, and one addition to be:

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

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.

We use a simple guard digits calculation that we add for our e^x function.

$$2 + \text{ceil}(\log_2(\text{precision})) \text{ as extra guard digits.}$$

Fast Exponential function for Arbitrary Precision number

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 $x > 1$, 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 $n = \text{floor}(x)$ from x , yielding a number between $[0, 1[$ then take the $\exp(x-n)$.

Using the identity:

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

e is a constant and can easily be computed fast, even with arbitrary precision, and then it is just a matter of raising it to the integer power of $\text{floor}(x)$.

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;
}
```

e^x using the coefficient scaling of the Taylor series

Further Improvement of the Taylor series methods?

Coefficient scaling can be considered an improvement over the standard method.

Generally speaking, you can have three options here.

- No coefficient scaling
- Partial coefficient scaling

Fast Exponential function for Arbitrary Precision number

- Full coefficient scaling

No coefficient scaling

No coefficient scaling is just the regular use of the Taylor series, as presented above. Each Taylor term is computed, including the division, which performs poorly compared to the other operators, particularly in arbitrary precision libraries. Although the method is straightforward, it ensures that each term is as accurate as possible before contributing to the final sum. It can be computationally expensive due to repeated division operations, mainly when used in connection with arbitrary precision libraries.

Partial coefficient scaling

You can group more Taylor terms before dividing. For example, you reduce the number of division operations by calculating five Taylor terms at a time and then dividing the sum of these terms by the appropriate factorial. Partial coefficient scaling will enhance performance since division is generally more computationally intensive than multiplication or addition. The trade-off here is a potential slight decrease in numerical precision, as errors could accumulate in the grouped terms before the division normalizes them.

Full coefficient scaling

When doing a full coefficient scaling, you postpone the division until all Taylor terms have been computed. This method involves summing all scaled Taylor series terms first and dividing the final sum by the factorial. This method could be the fastest in reducing the number of division operations, but it may also lead to the most significant error. When terms with large magnitudes are summed without immediate normalization, floating point errors can accumulate, particularly for higher-order terms or larger values of x .

The discussion of partial and full coefficient scaling is as follows.

Partial coefficient scaling

Consider the Taylor series expansion of $\exp(x)$:

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

The issue is the division for each term. Since division is often slower than calculation and addition, you could group two or more Taylor terms (sometimes called coefficient rescaling) and reduce the number of divisions. Consider the n 'th and the $n+1$ terms:

Fast Exponential function for Arbitrary Precision number

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

Moreover, group them:

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

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. You can do that for three terms:

$$\dots \frac{(n+1)(n+2)x^n + (n+2)x^{n+1} + x^{n+2}}{(n+2)!} \dots \Rightarrow$$
$$\dots \frac{x^n(x^2 + (n+2)x + n^2 + 3n + 2)}{(n+2)!} \dots$$

Saving two divisions, however, gained a few more additions 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, implementing this grouping of Taylor terms will be highly advantageous. With four to five terms grouped, you get a speedup of 2-3 times compared to not grouping terms.

Iterations Source for five 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)
{
```

Fast Exponential function for Arbitrary Precision number

```
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;
}
```

Full coefficient scaling

Now, why stop with the grouping of only a few Taylor terms? In [9], they devised a new computation of the Taylor series, postponing the division until all Taylor terms had been calculated.

He noticed that by evaluating the Taylor series backward, you can set this recursion:

$$n!e^x = n! + \dots + ((n(n-1)(n-2) + (n(n-1) + (n+x)x)x)x) \dots \quad (11)$$

Here, you summed up the series and postponed the division to one final division to calculate e^x . This approach is worth considering since division is an expensive operator with arbitrary precision.

The only perceived drawback is that we need to know how many Taylor terms we need before computation. This is contrary to all the other methods presented here, where you could summarize the Taylor terms until no more change was detected for a given precision of the result in the sum, thereby having an automated way of stopping the summation of the Taylor terms.

Luckily, figuring out the needed number of Taylor terms is not difficult. We are using Sterling approximation for the factorial. We can write the error terms required for a given decimal precision P .

$$(n + 0.5) \ln(n) - n(\ln(|x|) + 1) \approx P \cdot \ln(10) + \ln\left(\sqrt{\frac{2}{\pi}}\right) \quad (12)$$

Where P is the needed decimal precision.

We then have our $f(y)$:

$$f(y) = (n + 0.5) \ln(n) - n - P \cdot \ln(10) + 0.22579 \quad (13)$$

Where $\ln\left(\sqrt{\frac{2}{\pi}}\right) = 0.22579$

And $f'(y)$:

Fast Exponential function for Arbitrary Precision number

$$f'(y) = \frac{n+0.5}{n} + \ln(n) - \ln(|x|) - 1 \quad (14)$$

Applying the Newton's method yields:

$$y_{i+1} = y_i - \frac{f(y_i)}{f'(y_i)} \Rightarrow \quad (15)$$

$$y_{i+1} = y_i - \frac{(n_i+0.5)\ln(n_i) - n_i(\ln(|x|)+1) - P \cdot \ln(10) + 0.22579}{\frac{n_i+0.5}{n_i} + \ln(n_i) - \ln(|x|) - 1} \quad (16)$$

You get:

$$n_{i+1} = n_i - \frac{(n_i+0.5)\ln(n_i) - n_i(\ln(|x|)+1) - P \cdot \ln(10) + 0.22579}{\left(\frac{n_i+0.5}{n_i}\right) + \ln(n_i) - \ln(|x|) - 1} \quad (17)$$

Since we need an integral number of Taylor terms, we don't need to carry that much precision. As a starting point, [9] suggested.

$$n_0 = \frac{d \cdot \ln(10)}{\ln(d \cdot \ln(10) - \ln(|x|) - \ln(|-\ln(|x|)|))} \quad (18)$$

The starting formula works well for the Newton iteration since $0 < x < 1$ due to argument reduction. Since we are only interested in the ceiling n_{i+1} , we will only need a few iterations to find the number of Taylor terms required.

We can, therefore, replace the previous code for coefficient scaling for five Taylor terms with this one.

Source code for `exp(x)` using full coefficient scaling.

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

    if (v.iszero())
        return v = c1;
    if (klimit < 0)
        // 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);
    }
    else k = klimit;
    precision += k / 2;
    // Do iteration using higher precision plus compensate for reduction factor
    v.precision(precision);
    r.precision(precision);
```

Fast Exponential function for Arbitrary Precision number

```
expx.precision(precision);

if (v.sign() < 0)
    v.change_sign();
v.adjustExponent(-k);
// How many Taylor terms
double M,y,dy, xdouble;
xdouble = double(v);
M= precision * log(10) / (log(precision * log(10) - log(abs(xdouble)) -
log(abs(-log(abs(xdouble))))));
for (i = 1;; ++i)
{
    y = (M + 0.5) * log(M) - M * (log(abs(xdouble)) + 1.0) - precision *
log(10) + 0.22579;
    dy = (M + 0.5) / M + log(M) - log(abs(xdouble)) - 1;
    if(int(M) == int(M -y / dy))
        break;
    M+= -y/dy;
}

loopcnt_taylor = size_t(M+1);
int_precision mFak(size_t(M+1));
expx = float_precision(mFak,precision) + v;
for (i = int(M); i > 0; --i)
{
    expx *= v;
    mFak *= int_precision(i);
    expx += float_precision(mFak,precision);
}
expx /= float_precision(mFak, precision);

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

// Round to same precision as argument and rounding mode
expx.mode(x.mode());
expx.precision(x.precision());

return expx;
}
```

e^x using the Sine Hyperbolic function

It is less used, but the fastest way to calculate e^x is using the Sine Hyperbolic function using the identity:

$$e^x = \sinh(x) + \sqrt{1 + \sinh^2(x)} \quad (19)$$

Fast Exponential function for Arbitrary Precision number

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

$$\sinh(x) = x + \frac{x^3}{3!} + \frac{x^5}{5!} + \frac{x^7}{7!} + \frac{x^9}{9!} \dots \quad (20)$$

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

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \dots \quad (21)$$

Except for each term, we go faster toward zero with the $\sinh(x)$. We should expect that we will need fewer Taylor terms for a given precision than the $\exp(x)$ Taylor series, which is why using $\sinh(x)$ to calculate e^x makes sense. See [10] for a detailed description of how to implement $\sinh(x)$.

Source for $\exp(x)$ using $\sinh(x)$

```
float_precision expSinh(const float_precision& x)
{
    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);
            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 can quickly compute $\ln(x)$. Which you usually do not have when using arbitrary precision. The method solves the equation $x=e^y$ by taking the

Fast Exponential function for Arbitrary Precision number

$\ln()$ of both sides $\ln(x) = y$ and then solving it using the Newton method, which yields the iteration:

$$x_{n+1} = x_n(1 + y - \ln(x_n)) \quad (22)$$

The Newton method has a quadratic convergence rate doubling the 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 $\exp(x)$:

$$\exp(x) = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \dots \quad (23)$$

However, instead of calculating the series 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 . Then, it is just a matter of dividing p with q to get the approximation for e .

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

$Q(0,k)$ 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 $Q(a,b)$ using the binary splitting method. (a and b are integers and $a < b$) following the recursion:

Algorithm: Binary splitting method for e

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

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

To make the algorithm efficient, we must ensure that $|x| < 1$. That can be done quickly by using argument reduction as described under e^x using the Taylor series. We expect that the Taylor series will converge faster if $|x| \ll 1$.

Fast Exponential function for Arbitrary Precision number

Calculate how many Taylor terms we need as a function of the required decimal digits of e . We resort to the Stirling approximation formula for $k!$. 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:

$$\frac{x^k}{k!} < 10^{-P} \tag{25}$$

Where we use the Stirling approximation for $k!$:

$$k! \approx \left(\frac{k}{e}\right)^k \sqrt{2\pi k}, \text{ Stirling approximation} \tag{26}$$

This yield:

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

Taking $\log()$ on both sides, you get:

$$-k \cdot \log(x) + k \cdot (\log(k) - 1) + \frac{1}{2}\log(2\pi k) > P \cdot \log(10) \tag{27}$$

To solve this for k , we can use Newton's methods to find a solution within a few iterations. Notice that 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 table above clearly shows the effect of using the argument reduction technique in binary splitting. We can apply the same argument reduction formula already established at the start of the explanation of e^x .

Fast Exponential function for Arbitrary Precision number

Finding a reasonable reduction factor for e^x

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

$$|x| < 10^{-M} \text{ and } M > \frac{\ln(P)}{\ln(10)} \quad (28)$$

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 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 high magnitude $|x|$ by 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 big numbers as we increase the terms we need. To see how big we can again use the Stirling approximation for !

$$k! \approx \left(\frac{k}{e}\right)^k \sqrt{2\pi k}, \text{ Stirling approximation} \quad (29)$$

Using $\log_{10}(k!)$ We find the number of decimal digits as the size of $k!$

$$\log_{10}(k!) \approx \log_{10}\left(\left(\frac{k}{e}\right)^k \sqrt{2\pi k}\right) \Rightarrow$$

$$k \cdot \log_{10}(k) - k + \frac{1}{2} \log_{10}(2\pi k) \approx k \cdot \log_{10}(k) - k, \text{ for large } k \quad (30)$$

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 !

Fast Exponential function for Arbitrary Precision number

As expected! It is a decisive factor where we must adjust the required accuracy or precision upward when calculating e^x at some precision. The adjustment amount is much more considerable than we are used to using regular methods for e^x . However, argument reduction counteracts the need to handle calculations with significantly higher digits.

Source for e^x using binary splitting

The Source consists of 3 functions. ComputeExp() that drives the recursive function binarysplittingExp() and an outer function xstirling_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;
```

Fast Exponential function for Arbitrary Precision number

```
    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 two 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;

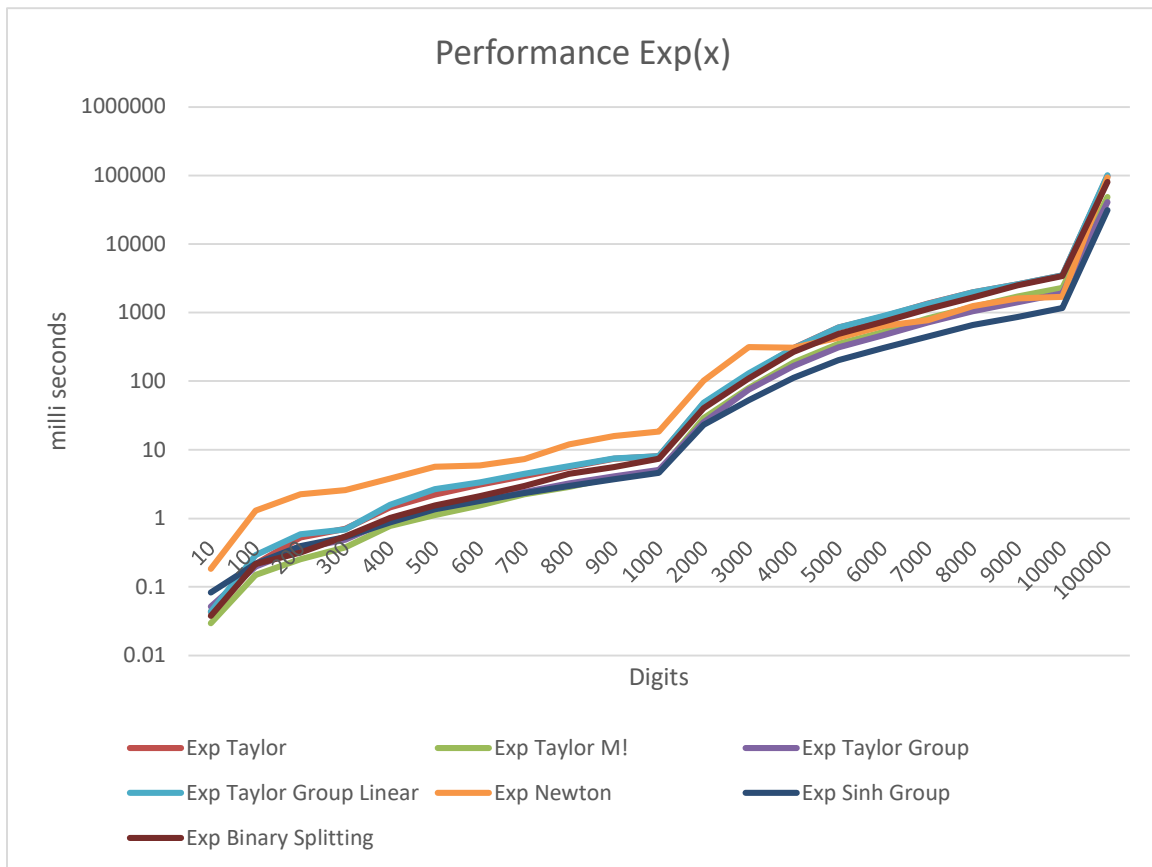
    // Reverse argument reduction
    // Brent enhancement avoids loss of significant digits when x is small.
    if (r > 0)
    {
        p -= c1;
        for (; r > 0; --r)
            p *= (c2 + p);
        p += c1;
    }
    if (x.sign() < 0)
        p = _float_precision_inverse(p);
}
```

Fast Exponential function for Arbitrary Precision number

```
// 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 increasing digits. The Binary splitting method is interesting but lacks the performance of the two other methods.



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

Looking at the chart, the Exp Taylor M! (full coefficient scaling) is the fastest method to around 1,000 digits, then both the Exp Taylor Group (partial coefficient scaling) and Exp using sinh (with partial coefficient scaling) take over with a lead of the latter that continues with increasing number of digits. Neither Exp Taylor (no coefficient scaling) Exp Taylor group linear, exp binary splitting, and exp using Newton methods come near the three leading methods.

Recommendation for calculating e^x

Based on the performance measure, recommend:

- 1) e^x using $\sinh()$ is recommended and is the fastest of all methods.
- 2) Always use argument reduction to increase performance.
- 3) Coefficient rescaling (or grouping of terms) can speed up calculation by a factor of two-three and is therefore recommended.
 - a. Use full coefficient scaling below 1,000 digits precision.
 - b. Use partial coefficient scaling above 1,000 digits precision.

The constant e

The transcendental constant e (same as e^1) can be more beneficial when calculated by methods other than those 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. Even with the enhancement presented in this paper, the algorithm is a magnitude faster than using the Taylor series for calculating e^1 . Please refer 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] in the sixties and accompanied by an Algol 60 version. The original code has been ported to the C++ environment with additional improvements. The calculation result 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  $m! \sim \sqrt{2\pi m} (m/e)^m$ .
    // Taken ln on both sides, you get:
    //  $m * (\text{Math.log}(m) - 1) + 0.5 * \text{Math.log}(2 * \text{Math.pi} * m)$ ;
    // 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)
    {
```

Fast Exponential function for Arbitrary Precision number

```
    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 .

$$e = \sum_{k=0}^{\infty} \frac{1}{k!} = 1 + \frac{1}{1} + \frac{1}{2} + \frac{1}{6} + \frac{1}{24} + \dots = \frac{P(0,k)}{Q(0,k)} \quad (31)$$

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 $Q(a,b)$ using the binary splitting method. (a and b are integers and $a < b$) following the recursion:

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

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);
        }
```

Fast Exponential function for Arbitrary Precision number

```
        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 that 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.

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

$$k! \approx \left(\frac{k}{e}\right)^k \sqrt{2\pi k}, \text{ Stirling approximation} \quad (32)$$
$$\left(\frac{k}{e}\right)^k \sqrt{2\pi k} > 10^P \Rightarrow$$

Taking $\log()$ on both sides, you get:

$$k \cdot (\log(k) - 1) + \frac{1}{2} \log(2\pi k) > P \cdot \log(10) \quad (33)$$

To solve this for k , we can use Newton's methods to 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*k);
    // 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);
}
```

Fast Exponential function for Arbitrary Precision number

The binary splitting is called from the below Source. This sets up the call to `binarysplittingE()` and calls the `stirling_approx()` to calculate 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 and 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));
        }
```

Fast Exponential function for Arbitrary Precision number

```
        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. Twenty times faster than the AHJ Sale method when calculating e with 100,000 digits. The binary splitting method is approx. Forty times faster than using the Taylor series for e^1 described 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 \(free.fr\)](#)
- 9) Ronald W Potter. Arbitrary Precision Calculation of selected higher functions. ISBN #:978-1-312-59943-7
- 10) Fast Hyperbolic functions for arbitrary precision. [HVE Fast Hyperbolic calculation for arbitrary precision](#)