

Fast Trigonometric functions for Arbitrary Precision numbers.

By Henrik Vestermark (hve@hvks.com)

Abstract

This paper presents efficient algorithms for computing trigonometric and inverse trigonometric functions with arbitrary precision. While Taylor series expansions are traditionally used in arbitrary-precision libraries, their direct application becomes increasingly inefficient as precision grows. The methods described here combine structured argument reduction, adaptive working-precision control, and coefficient-scaling techniques to significantly reduce computational cost while maintaining rigorous numerical accuracy.

A unified four-stage computational framework is introduced, consisting of range reduction, argument reduction, series evaluation, and reverse reconstruction. A two-stage precision strategy separates magnitude-dependent precision required for range reduction from convergence-dependent precision required for function evaluation. Rigorous error analyses are provided showing how cancellation during range reduction and error amplification during reverse reduction determine the required working precision. Several implementation variants are analyzed, including partial and full coefficient scaling for Taylor series evaluation, trisection and double-angle reduction strategies, and alternative formulations that compute cosine and tangent via sine. The resulting algorithms reduce the number of expensive division operations and improve performance by approximately 5–20× compared with earlier implementations in the author's arbitrary-precision libraries.

The paper provides practical design guidelines, detailed C++ implementations, and validated precision formulas suitable for high-precision numerical software and arbitrary-precision mathematical packages.

Introduction:

When implementing an arbitrary precision math package, you would use the standard Taylor series calculation for calculating $\sin(x)$, $\cos(x)$, $\arcsin(x)$, and $\arctan(x)$ for arbitrary precision. In contrast, $\tan(x)$ & $\arccos(x)$ can be derived from $\sin(x)$ or $\cos(x)$. The Taylor series for trigonometric functions is not particularly fast in its raw form. However, you can apply techniques that significantly improve the method's performance. We will discuss various methods for calculating trigonometric functions and elaborate on techniques such as clever argument reductions and coefficient scaling to improve performance. Furthermore, we will analyze Newton's methods for calculating trigonometric functions.

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

Fast Trigonometric functions for Arbitrary Precision numbers

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 and 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 an 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](#)
12. Fast Computation of Stirling's numbers in arbitrary precision. [HVE Fast Computation of Stirling numbers in arbitrary precision](#)
13. Fast Prime computation in arbitrary precision. [HVE Fast Prime Computation in arbitrary precision](#)
14. Fast PRNG in arbitrary precision. [HVE Fast PRNG in arbitrary precision](#)
15. Fast Fibonacci sequence in arbitrary precision. [HVE Fast Fibonacci in arbitrary precision](#)
16. Fast arbitrary precision Multiplication. [HVE Fast arbitrary precision integer multiplication](#)
17. Fast Factorial and Binomial computation with arbitrary precision. [HVE Fast factorial and binomial for arbitrary precision.docx](#)

Change log

20-February-2026. Revised $\sin(x)$, $\cos(x)$, and $\tan(x)$ computing and added a rigorous error analysis in the Appendix of this paper.

15-July 2024. Added more methods to the calculation of trigonometric functions.

15-January 2023. I updated some inconsistencies in the “Cos(x) using sin(x)” section and corrected the recommendation there. This paper is part of a series of documents on arbitrary precision.

26-January 2023. Cleaning up the grammar.

Fast Trigonometric functions for Arbitrary Precision numbers

Contents

Abstract	1
Introduction:.....	1
Change log	2
The Arbitrary precision library	5
Internal format for float_precision variables	6
Normalized numbers	7
Trigonometric functions.....	8
Computational Framework for Trigonometric Functions.....	8
The Four-Stage Pipeline	8
Inverse Trigonometric Functions.....	9
Applicable Steps by Function	9
Two-Stage Precision Strategy.....	9
Cross-References	10
Evaluation of trigonometric functions	10
Sin(x) using Taylor Series	10
The issue with arbitrary precision.....	12
How to find a reasonable reduction factor.....	13
Guard Digits.....	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
Source for sin(x) with argument reduction and partial coefficient scaling.....	16
Full coefficient scaling.....	18
Source code for sin(x) using argument reduction and full coefficient scaling.	20
Performance for sin(x)	22
Recommendation for calculating sin(x).....	22
Cos(x):.....	23
Cos(x) using double-angle reduction	25
Source for cos(x) with argument reduction and partial coefficient scaling	25
Cos(x) using full coefficient scaling.	28
Source for cos(x) with argument reduction and full coefficient scaling.....	28
Cos(x) using sin(x).....	30
Source for cos(x) using sin(x).....	30
Source code for cos via sine identity	31
Performance for Cos(x).....	32
Recommendation for calculating cos(x)	32
Tan():.....	33
Source for tan(x)	33
Arcsin(x):	34
Arcsin using Newton's method	34
Arcsin(x) using Taylor series and argument reduction.....	37
Arcsin coefficient scaling	38

Fast Trigonometric functions for Arbitrary Precision numbers

Source for Arcsin(x) with coefficient scaling and argument reduction.....	39
Recommendation for calculating Arcsin(x).....	42
Arccos(x):	42
Source for Arccos(x).....	42
Arctan(x):.....	43
Arctan(x) using the Taylor series.....	43
The issue with arbitrary precision.....	45
Arctan(x) using coefficient scaling.....	46
Source for Arctan(x) with argument reduction & coefficient scaling	46
Arctan(x) using the Euler method.....	48
Arctan(x) using Arcsin().....	49
Source for Arctan(x) using Arcsin()	50
Recommendation for calculating Arctan(x).....	50
Reference	51
Appendix A: Rigorous Error Analysis for Range Reduction	52
Source code for rangeReduction2PI	52
Source code for rangeReductionPI	52
A.1 Purpose and the Catastrophic Cancellation Problem	53
A.2 Notation.....	53
A.3 Error Analysis for rangeReduction2PI().....	54
Step 1: Approximation of π	54
Step 2: Computation of 2π	54
Step 3: Division $x / (2\pi)$	54
Step 4: The floor() operation.....	54
Step 5: Multiplication $r \cdot 2\pi$	54
Step 6: Subtraction $x - r \cdot 2\pi$	54
A.4 Required Precision for rangeReduction2PI().....	55
A.5 Guard Digits Justification	55
A.6 Error Analysis for rangeReductionPI().....	55
Appendix B: Rigorous Error Analysis for sin(x).....	56
B.1 Notation for Stage 2	56
B.2 Choosing the Reduction Factor k.....	56
B.3 Sub-stage 2A: Argument Reduction Error	57
B.4 Sub-stage 2B: Taylor Series Error	57
B.5 Sub-stage 2C: Reverse Reduction Error.....	57
B.6 Total Error for Stage 2	58
B.7 Required Working Precision prec2	58
B.8 Precision Reduction Between Stages	58
Appendix C: Rigorous Error Analysis for cos(x)	59
C.1 Overview of the cos(x) Implementation.....	59
C.2 Error Analysis for the Normal Case: $\cos(x) = \sqrt{1 - \sin^2(x)}$	59
Operation 1: squaring $\sin(v)$	59
Operation 2: subtraction $1 - \sin^2(v)$	59
Operation 3: square root.....	59
C.3 Critical Case: v Near $\pi/2$	60
C.4 Alternative Formula: $\cos(x) = \sin(\pi/2 - x)$	60

Fast Trigonometric functions for Arbitrary Precision numbers

C.5 Sign Determination	60
C.6 Required Working Precision for cos(x)	61
C.7 Comparison with Direct cos(x) via Taylor Series	61

The Arbitrary precision library

If you are already 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.
```

Fast Trigonometric functions for Arbitrary Precision numbers

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

This allows very fast multiplication or 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 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.

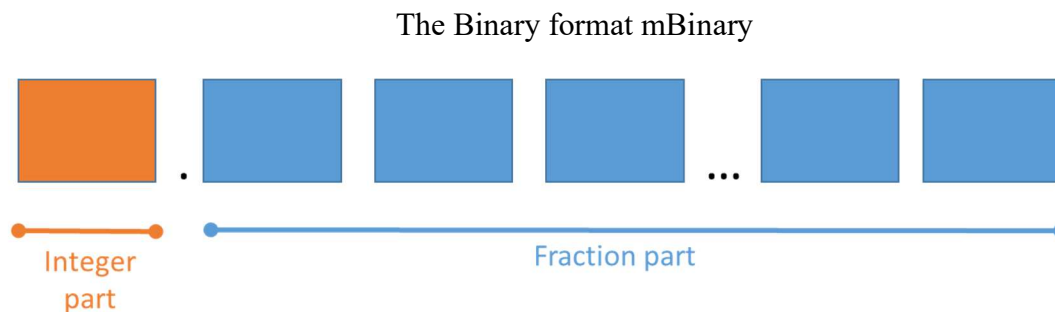
Internal format for `float_precision` variables

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.

Fast Trigonometric functions for Arbitrary Precision numbers

Internal class variables such as the sign, exponent, precision, and rounding mode are not crucial to understanding the code segments.

Normalized numbers

A `float_precision` variable is always stored as a normalized number with a one in the integer portion of the number. The only exception is zero, which is stored as zero. Furthermore, a normalized number has no trailing zeros.

For more details see [1].

Trigonometric functions

Computational Framework for Trigonometric Functions

Before presenting the detailed implementation of each trigonometric function, it is useful to establish a common conceptual framework that applies across all functions. Every function in this library is computed using a structured pipeline of up to four stages. Understanding this pipeline makes it easier to follow the source code and to appreciate why each step is necessary.

The Four-Stage Pipeline

The computation of $\sin(x)$, $\cos(x)$, and $\tan(x)$ follows up to four distinct stages:

a) *Range Reduction [optional]*

For large-magnitude arguments ($|x| \gg 2\pi$), the periodic nature of trigonometric functions is exploited to map x to a mathematically equivalent argument in $[0, 2\pi)$. Because the input can be arbitrarily large (e.g., $x = 10^{50}$), this step must be performed at a working precision that is higher than the target precision. The required precision scales as $p_{\text{target}} + \log_{10}(|x|)$ digits to avoid catastrophic cancellation. This step is skipped when $|x|$ is already small. See Appendix A for a rigorous error analysis. This procedure corresponds to a high-precision argument reduction similar in spirit to the Payne–Hanek method used in correctly rounded floating-point libraries. The required working precision grows with $\log_{10}(|x|)$ because the subtraction of large nearly equal quantities during modulo reduction otherwise destroys significant digits.

b) *Argument Reduction*

The argument is further divided by 3^k (using the trisection formula) to produce a small reduced argument $v_{\text{red}} = v / 3^k$. A small argument makes the Taylor series converge in far fewer terms. The reduction factor k is chosen based on the target precision and the current magnitude of the argument. This step is applied by all six functions.

c) *Taylor Series Evaluation*

The Taylor series for \sin or \arctan is evaluated at the reduced argument. Three variants are described in this paper: no coefficient scaling, partial coefficient scaling, which groups consecutive terms to reduce the number of divisions, and full coefficient scaling, which pre-computes all coefficients and eliminates divisions from the inner loop. The convergence criterion is adaptive: The iteration terminates when adding the next Taylor term no longer changes the accumulated sum at the current working precision. Numerically, this corresponds to the term being smaller than approximately one-half unit in the last place (ULP) under the active rounding mode.

d) *Reverse Argument Reduction*

The result is mapped back to the original argument range by applying the trisection identity k times. For sine, this identity is $\sin(3y) = 3\sin(y) - 4\sin^3(y)$. Each reverse step amplifies any error in the current result by a factor of

Fast Trigonometric functions for Arbitrary Precision numbers

approximately 3, which is why the working precision in Stage 2 must be chosen to absorb $k \cdot \log_2(3)$ extra bits. This step is required only for the forward functions (sin, cos, tan) and is not needed for the inverse functions.

Inverse Trigonometric Functions

The inverse functions $\text{asin}(x)$, $\text{acos}(x)$, and $\text{atan}(x)$ use only steps b) and c) from the pipeline above:

Range reduction (step a) is not required because the domains of the inverse functions are inherently bounded: asin and acos accept arguments in $[-1, 1]$, and atan accepts all real arguments but produces results bounded by $(-\pi/2, \pi/2)$. No large-magnitude problem arises.

Reverse argument reduction (step d) is not required because the inverse trisection identity is applied differently: the argument reduction for asin and atan uses an algebraic substitution to reduce the input to a small value, and the corresponding reverse step is an arithmetic correction rather than a recursive identity. The Taylor series for atan converges directly for small arguments.

Applicable Steps by Function

The table below summarises the pipeline steps applied to each function. Range reduction is marked Optional because it is only invoked when the magnitude of the argument exceeds 2π ; for typical arguments, it is skipped entirely.

Function	a) Range Red.	b) Arg. Red.	c) Taylor Series	d) Reverse Red.
sin(x)	Optional	Yes	Yes	Yes
cos(x)	Optional	Yes	Yes	Yes
tan(x)	Optional	Yes	Yes	Yes
asin(x)	--	Yes	Yes	--
acos(x)	--	Yes	Yes	--
atan(x)	--	Yes	Yes	--

Two-Stage Precision Strategy

An important implementation detail shared by $\text{sin}(x)$, $\text{cos}(x)$, and $\text{tan}(x)$ is the two-stage precision strategy. The precision formulas used here are not heuristic. A complete derivation of the required working precision for range reduction and reverse reconstruction is provided in Appendices A and B.

Stage 1 precision (prec1): Used for range reduction only. Must be high enough to compensate for the input's magnitude. Formula: $\text{prec1} = p_target + \text{ceil}(\log_2(|x|) / \log_2(10)) + \text{guard_digits}$. For $x = 10^{50}$ and $p_target = 100$ this gives $\text{prec1} \approx 160$ digits.

Stage 2 precision (prec2): Used for argument reduction, Taylor series, and reverse reduction. Depends on the trisection count k and the target precision, but not on the original magnitude of x . Formula: $\text{prec2_bits} = p_target_bits + k * 1.585 + \log_2(p_target) + \text{guard_bits}$. For $p_target = 100$, this typically gives $\text{prec2} \approx 115$ digits, substantially less than prec1 .

The key insight is that after range reduction, the argument is bounded by 2π regardless of how large the original x was. Stage 2, therefore, operates on a problem of fixed size, and

Fast Trigonometric functions for Arbitrary Precision numbers

its precision requirement depends only on the convergence properties of the Taylor series, not on the original input magnitude. This separation makes arbitrary-precision trigonometric computation practical even for very large arguments.

The precision is explicitly reduced between the two stages by calling `v.precision(prec2)` before entering Stage 2. This is not merely a rounding step; it is a deliberate performance optimization. Since the Taylor series cost scales as $O(\text{prec}^2)$ per iteration and there are $O(p_target)$ iterations, operating at `prec1` throughout Stage 2 would increase the computation time by approximately $(\text{prec1} / \text{prec2})^2$, which can easily be a factor of 2 or more.

Cross-References

The rigorous error bounds that justify the precision formulas above are derived in the appendices:

Appendix A presents a complete error analysis of the range-reduction functions `rangeReduction2PI()` and `rangeReductionPI()`, including a proof that precision must scale with $\log_{10}(|x|)$ and a mathematical justification for the guard digit counts.

Appendix B derives the working precision formula for `sin(x)`, showing how the 3^k error amplification in the reverse reduction drives the `prec2` requirement.

Appendix C derives the error analysis for `cos(x)`, including the threshold condition $|\cos(v)| > 0.1$ that determines when the `sqrt` formula is safe versus when the alternative `sin(pi/2 - v)` formula must be used.

Evaluation of trigonometric functions

There are many ways you can calculate trigonometric functions with arbitrary precision. Traditionally, the Taylor series expansion has been used; however, this chapter will examine:

- 1) `Sin(x)` using Taylor series, argument reduction, and coefficient scaling.
- 2) `Cos(x)` using Taylor series, argument reduction, and coefficient scaling.
- 3) `Tan(x)` using various methods.
- 4) `Arcsin(x)` using Taylor series, argument reduction, and coefficient scaling
- 5) `Arccos(x)` using `arcsin(x)`
- 6) `Arctan(x)` using Taylor series, argument reduction, and coefficient scaling.
- 7) `Arctan(x)` using other methods.

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

Sin(x) using Taylor Series

The standard way of calculating `sin(x)` is using the Taylor Series. `Sin(x)` can be found with the Taylor series:

Fast Trigonometric functions for Arbitrary Precision numbers

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} \dots \quad (1)$$

Similar to the sine hyperbolic functions, the Taylor series is:

$$\sinh(x) = x + \frac{x^3}{3!} + \frac{x^5}{5!} + \frac{x^7}{7!} + \frac{x^9}{9!} \dots \quad (2)$$

Where the only difference is the alternating sign between the Taylor Terms, $\sin(x)$ is defined for any real number.

However, before we start the Taylor series, we first reduce the argument x . We will do that in four steps.

Step 1: We notice that $\sin(x)$ is cyclic with a period of 2π , so we can quickly reduce any argument $> 2\pi$ so it falls between zero and 2π by simply taking x modulo 2π .

Step 2: We can further reduce x so it is between $0.. \pi$ using the identity:

$$\sin(x) = -\sin(x-\pi) \text{ for } x \geq \pi.$$

Step 3: We reduce it further by using the symmetry around $\frac{\pi}{2}$ to the range $0.. \frac{\pi}{2}$:

$$\sin(x) = \sin\left(x - \frac{\pi}{2}\right) \text{ for } x \geq \frac{\pi}{2}$$

If π is 'expensive' to calculate (usually the case with arbitrary precision), we can omit step 3 since we have a different way to obtain the same thing by increasing the argument reduction factor. See the section on finding a reasonable reduction factor.

Step 4: Finally, we reduced the argument k number of times using the trisection identity:

$$\sin(3x) = 3\sin(x) - 4\sin^3(x)$$

Until x is below a certain threshold, it is obvious from the $\sin(x)$ Taylor series that the smaller x , the fewer terms we would need.

Argument reduction reduces the number of Taylor iterations, minimizes round-off errors, and reduces calculation time.

After the Taylor series has converged, we use the trisection identity to reverse- k times to find the result for $\sin(x)$.

Let us find the $\sin(0.7)$ to see how this algorithm works. The error is zero after the 8th Taylor term, and the result is ~ 0.6442176872 .

sin(x)		Original	X Reduced	
x=		0.7	0.7	
Taylor reductions=		0		
Terms	Term value	Term Sum	sin(x)	Error
1	7.00E-01	0.70000000000	0.7000000000	-5.58E-02
2	5.72E-02	0.642833333	0.6428333333	1.38E-03
3	1.40E-03	0.64423391667	0.6442339167	-1.62E-05

Fast Trigonometric functions for Arbitrary Precision numbers

4	1.63E-05	0.64421757653	0.6442175765	1.11E-07
5	1.11E-07	0.64421768773	0.6442176877	-4.94E-10
6	4.95E-10	0.64421768724	0.6442176872	1.55E-12
7	1.56E-12	0.64421768724	0.6442176872	-3.66E-15
8	3.63E-15	0.64421768724	0.6442176872	0.00E+00

We can see the effect of Step 4 by increasing the number of argument reductions. For example, you get the same result for two reductions after only five iterations. The argument is reduced twice from 0.7 to ~ 0.077

sin(x)		Original	X Reduced	
x=		0.7	0.077777778	
Taylor reductions=		2		
Terms	Term value	Term Sum	sin(x)	Error
1	7.78E-02	0.077777777778	0.6447587967	-5.41E-04
2	7.84E-05	0.07769936	0.6442175235	1.64E-07
3	2.37E-08	0.07769938357	0.6442176873	-2.36E-11
4	3.42E-12	0.07769938357	0.6442176872	2.00E-15
5	2.87E-16	0.07769938357	0.6442176872	0.00E+00

If we do four argument reductions in step 4, we get the result after only three iterations.

sin(x)		Original	X Reduced	
x=		0.7	0.008641975	
Taylor reductions=		4		
Terms	Term value	Term Sum	sin(x)	Error
1	8.64E-03	0.00864197531	0.6442243516	-6.66E-06
2	1.08E-07	0.008641868	0.6442176872	2.49E-11
3	4.02E-13	0.00864186774	0.6442176872	0.00E+00

Again, we notice that argument reduction can significantly reduce the number of Taylor terms needed, thereby improving the performance of calculating $\sin(x)$.

The issue with arbitrary precision

The number of Taylor terms used to reach a result does not seem so bad at first glance. In the previous examples, we only used approx. 15 decimal digits. However, when dealing with higher precision, e.g., 1,000, 10,000, or even 100,000 digits, we suddenly need many more Taylor terms to obtain the result. In Yacas [5], they found a bound for the number of Taylor terms, n , needed for the $\sin(x)$ as a function of the number of precision in digits P and the magnitude, M , of the argument $x=10^M$:

Fast Trigonometric functions for Arbitrary Precision numbers

$$2(n + 1) \approx \frac{(P - M) \cdot \ln(10)}{\ln(P - M) - 1 - M \cdot \ln(10)} \Rightarrow$$

$$n \approx \frac{1}{2} \frac{(P - M) \cdot \ln(10)}{\ln(P - M) - 1 - M \cdot \ln(10)} - 1 \quad (3)$$

The number of Taylor terms needed for sin(x) as a function of precision and argument magnitude.

Digits	10 ¹	10 ²	10 ³	10 ⁴	10 ⁵	10 ⁶	10 ⁷	10 ⁸
x								
10¹	(11)	88	319	1,948	14,022	109,512	898,358	7,615,327
10⁰	8	31	194	1,402	10,951	89,835	761,532	6,608,768
10⁻¹	3	19	140	1,095	8,983	76,153	660,876	5,837,230
10⁻²	2	14	109	898	7,615	66,087	583,723	5,227,006
10⁻³	1	11	90	761	6,608	58,372	522,700	4,732,291
10⁻⁴	1	9	76	661	5,837	52,270	473,229	4,323,125
10⁻⁵	1	7	66	584	5,227	47,323	432,312	3,979,084
10⁻⁶	1	6	58	522	4,732	43,231	397,908	3,685,765
10⁻⁷	1	6	52	473	4,323	39,791	368,576	3,432,721
10⁻⁸	1	5	47	432	3,979	36,857	343,272	3,212,190
10⁻⁹	(1)	5	43	398	3,686	34,327	321,219	3,018,284

The table above is quite interesting. For this reason, the effect of argument reduction for a precision of 100 digits reduces the number of Taylor terms by a factor of six between arguments of 1 and 10⁻⁹ in magnitude. For a precision of 100,000 digits, the factor is only about 3; for 100M digits, it is about 2.2. The lesson is that argument reduction is more efficient for lower precision than for higher precision. However, overall argument reduction is beneficial at any precision. There is another approximation for n based on the actual value of x rather than just its magnitude. It usually gives a bit fewer of the required Taylor terms. This formula can be pretty helpful:

$$n \approx \frac{P \cdot \ln(10)}{2(\ln(P) - \ln(x))} - 1 \quad (4)$$

How to find a reasonable reduction factor.

As shown in the table above, a higher reduction factor greatly improved performance. However, how many times is the reduction adequate? The argument reduction on the front end is a division per reduction. In the back end, you do this as many times as you did the reduction on the front end. $\sin(3x) = 3\sin(x) - 4(\sin^3(x))$ taking $\sin(x)$ out as a factor you get this: $\sin(3x) = \sin(x)(3 - 4(\sin^2(x)))$ or one subtraction and three multiplication. Using

Fast Trigonometric functions for Arbitrary Precision numbers

$$n \approx \frac{P \cdot \ln(10)}{2(\ln(P)-1) \cdot (x)} - 1 \quad (5)$$

Starting at $x=1$, for $P=1,00$ digits, the needed Taylor term is 24. Doing three reductions, you get $x = 1/33 = 0.037$. We expect to use the above formula to only need 14 Taylor terms. Each Taylor term requires one addition/subtraction, one division, and one multiplication, yielding a total saving of 10 subtractions, 10 divisions, and 10 multiplications. Compared to three reductions on the front end, there are three divisions, and on the back end, three subtractions and nine multiplications, a total saving of seven subtractions/additions, one multiplication, and seven divisions. Since division is a magnitude slower than multiplication and addition/Using subtraction, we can give a rough savings estimate with seven divisions. For higher precision, the savings become larger.

We automatically calculate the reduction factor as follows:

$$k = 8 \left\lceil \frac{2}{3} \ln(2) * \ln(P) \right\rceil \quad (6)$$

For higher precision, we adjusted the magnitude of x . After Step 2, we know that x is in the range of $[0..\pi]$. This is equivalent to the exponent of our number (in base 2) being in the range $[-\infty..1]$. We add the exponent to the reduction factor. This means our reduction factor decreases as x approaches zero, preventing unnecessary reductions. If x is very small, the reduction factor is negative, and we do not perform any argument reductions. For example, for $P=100$, you get 24; for $P=10,000$, you get 40. To compensate for inaccuracies in the front-end and back-end calculations, we increase the precision by approximately half the k factor. The increased precision incurs only a minor performance penalty compared with the extra savings in Taylor's overall calculation.

Guard Digits

During Taylor accumulation, many additions of decreasing magnitude are performed. To prevent loss of significant digits due to rounding during intermediate steps, additional working precision (“guard digits”) is used. Empirically the choice

$$\text{guard} = 2 + \text{ceil}(\log_{10}(P))$$

provides a sufficient safety margin while keeping the performance impact negligible.

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
- Full coefficient scaling

Fast Trigonometric functions for Arbitrary Precision numbers

No coefficient scaling

No-coefficient scaling is simply the regular use of the Taylor series, as presented above. Each Taylor term is computed, including 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 it is added to the final sum. It can be computationally expensive due to repeated division operations, especially when used 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 is a slight decrease in numerical precision, as errors can accumulate in the grouped terms before 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 first summing all scaled Taylor series terms and then dividing the final sum by the factorial. This method may be the fastest for reducing the number of division operations, but it may also introduce the largest 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 $\sin(x)$:

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

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 called coefficient scaling) and reduce the number of divisions. Consider the n 'th and the $n+1$ term, assuming the n 'th term is the negative part (for the moment):

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

Moreover, group them:

Fast Trigonometric functions for Arbitrary Precision numbers

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

If the n'th term is not the one starting with the minus sign, you can just flip the sign in the above equation, yielding:

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

Then, you replace one division with two multiplications. The (n+1)(n+2) can be done using a 32-bit or 64-bit integer, since you never get to use that many Taylor terms in real life. There is no need to stop at just grouping two terms. You can do that for three terms or more:

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

Saving two divisions, however, gained a few more additions and multiplications. It is straightforward to determine when to start with a negative sign by testing if n'th term divided by 2 is an odd number (begin with a minus sign) or an even number starting with a plus sign, and then alternate the sign after that.

Source for `sin(x)` with argument reduction and partial coefficient scaling.

```
float_precision sin(const float_precision& x) {
    if (isnan(x) || isinf(x))
        return FP_QUIET_NAN;
    if (x.iszero())
        return x;

    const int group = 5;
    const size_t p_target = x.precision();
    const size_t p_target_bits = (size_t)(p_target * log2(10));
    int sign = x.sign();
    float_precision v = abs(x);
    uintmax_t loopcnt = 1;
    uintmax_t i;

    // =====
    // STAGE 1: Range Reduction mod 2π
    // =====
    // Stage 1: Range reduction
    rangeReduction2PI(v); // High precision, handles large x
    //v.precision(p_target + 5); // Reduce precision for next stage, but keep some
guard bits

    bool sign_flip;
    rangeReductionPI(v, sign_flip); // Low precision, x is small now
    if (sign_flip)
        sign *= -1;

    // =====
    // STAGE 2: Core Computation (Trisection + Taylor)
```

Fast Trigonometric functions for Arbitrary Precision numbers

```
// =====  
// Now v is in [0, π], compute reduction factor k  
intmax_t k = 8 * (intmax_t)ceil(log(2) * log(p_target));  
k = (intmax_t)ceil(2.0 * k / 3);  
// Adjust k based on actual argument size  
k += v.exponent();  
k = std::max((intmax_t)0, k); // ensure that we dont do negative reduction  
  
// Precision for core computation  
// Must account for error amplification by 3^k  
size_t extra_bits = 0;  
extra_bits += (size_t)(k * 1.585); // Error amplification: k * log2(3)  
extra_bits += (size_t)ceil(log2(p_target / 3.0 * 5.0)); // Roundoff  
extra_bits += 20; // Guard bits  
  
size_t prec2_bits = p_target_bits + extra_bits;  
size_t prec2 = (size_t)(prec2_bits / log2(10)) + 1;  
  
// IMPORTANT: Reduce precision to p2 (might be less than p1!)  
v.precision(prec2);  
  
// Now perform core computation at precision p2  
float_precision sinx, r, vsq, terms;  
sinx.precision(prec2);  
r.precision(prec2);  
vsq.precision(prec2);  
terms.precision(prec2);  
  
// Argument reduction by 3^k  
float_precision c3(3, prec2);  
r = pow(c3, float_precision(k, prec2));  
v /= r;  
  
// Taylor series evaluation  
vsq = v.square();  
r = v;  
sinx = v;  
  
if (group == 1)  
{  
    // Now iterate using Taylor expansion, one term at a time  
    for (i = 3; ; i += 2, ++loopcnt)  
    {  
        r *= vsq / float_precision(i * (i - 1));  
        r.change_sign();  
        if (sinx + r == sinx)  
            break;  
        sinx += r;  
    }  
}  
else  
{  
    // Do group Taylor terms at a time  
    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(prec2); vn[i].precision(prec2);  
        if (i == 1) vn[1] = vsq;  
        if (i > 1) vn[i] = vn[i - 1] * vsq;  
    }  
    // Now iterate  
    for (i = 3; ; )  
    {  
        intmax_t j;  
        for (j = group - 1; j >= 0; --j)  
        {  
            if (j == group - 1)  
            {
```

Fast Trigonometric functions for Arbitrary Precision numbers

```
    j), prec2);
    cn[j] = float_precision((i + 2 * j - 1) * (i + 2 *
    if ((i / 2 + j - 1) & 0x1) // Odd
        cn[j].change_sign();
    }
    else
    {
        cn[j] = -cn[j + 1] * float_precision((i + 2 * j - 1)
    * (i + 2 * j), prec2);
    }
}

cn[0] = abs(cn[0]).inverse();
// Adding from smallest to largest number
terms = vn[group - 1];
if ((i / 2 + group - 1) & 0x1)
    terms.change_sign();
for (j = group - 1; j >= 2; --j)
    terms += cn[j] * vn[j - 1];
terms += cn[1];
r *= vsq * cn[0];
terms *= r;
i += 2 * group; // Update term count
loopcnt += group;
if (sinx + terms == sinx) // Reach precision?
    break; // yes terminate loop
sinx += terms; // Add Taylor terms to result
if (group > 1)
    r *= vn[group - 1]; // adjust r to last Taylor
term
}
}

// Reverse reduction
float_precision c4(4, prec2);
for (intmax_t i = k; i > 0; --i) {
    sinx *= (c3 - c4 * sinx.square());
}

// =====
// STAGE 3: Round to target precision
// =====
sinx.mode(x.mode());
sinx.precision(p_target);
if (sign < 0)
    sinx.change_sign();
return sinx;
}
```

Full coefficient scaling

Now, why stop at grouping only a few Taylor terms? In [8], 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:

$$n!e^x = n! - \dots + ((+(-n(n-1)(n-2)) + (-n(n-1) + (n+x)x)x)x \dots \quad (8)$$

Here, you summed up the series and postponed the division to a single final division to calculate e^x . This approach is worth considering since division is an expensive operator with arbitrary precision. The +- sign indicates that you alternate between + and - depending on the power of the Taylor term.

Fast Trigonometric functions for Arbitrary Precision numbers

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. Because intermediate partial sums can grow much larger than the final result, rounding errors accumulate before normalization occurs. The method, therefore, trades numerical stability for fewer division operations and performs best only when the reduced argument is sufficiently small.

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

$$(n + 0.5) \ln(n) - n(\ln(|x|) + 1) \approx P \cdot \ln(10) + \ln\left(\sqrt{\frac{2}{\pi}}\right) \quad (9)$$

Where P is the needed decimal precision.

We then have our $f(y)$:

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

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

And $f'(y)$:

$$f'(y) = \frac{n+0.5}{n} + \ln(n) - \ln(|x|) - 1 \quad (11)$$

Applying Newton's method yields:

$$y_{i+1} = y_i - \frac{f(y_i)}{f'(y_i)} \Rightarrow \quad (12)$$

$$y_{i+1} = y_i - \frac{(n_i+0.5) \ln(n_i) - n_i(\ln(|x|)+1) - P \cdot \ln(10) + 0.22579}{\frac{n_i+0.5}{n_i} + \ln(n_i) - \ln(|x|) - 1} \quad (13)$$

You get:

$$n_{i+1} = n_i - \frac{(n_i+0.5) \ln(n_i) - n_i(\ln(|x|)+1) - P \cdot \ln(10) + 0.22579}{\left(\frac{n_i+0.5}{n_i}\right) + \ln(n_i) - \ln(|x|) - 1} \quad (14)$$

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

Fast Trigonometric functions for Arbitrary Precision numbers

$$n_0 = \frac{d \cdot \ln(10)}{\ln(d \cdot \ln(10) - \ln(|x|) - \ln(|-\ln(|x|)|))} \quad (15)$$

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 need only a few iterations to determine the number of Taylor terms required.

Therefore, we can replace the previous code for partial coefficient scaling for five Taylor terms with this one for full coefficient scaling.

Source code for $\sin(x)$ using argument reduction and full coefficient scaling.

```
// sin(x) using argument reduction and full coefficient scaling
//
static float_precision sinM(const float_precision & x, const int klimit = 16) {
    if (isnan(x) || isinf(x))
        return FP_QUIET_NAN;
    if (x.iszero())
        return x;

    const int group = 5;
    const size_t p_target = x.precision();
    const size_t p_target_bits = (size_t)(p_target * log2(10));
    int sign = x.sign();
    float_precision v = abs(x);
    uintmax_t loopcnt = 1;
    uintmax_t i;
    const float_precision c3(3), c4(4);

    // =====
    // STAGE 1: Range Reduction mod 2π
    // =====
    // Stage 1: Range reduction
    rangeReduction2PI(v); // High precision, handles large x
    //v.precision(p_target + 5); // Reduce precision for next stage, but keep some
guard bits

    bool sign_flip;
    rangeReductionPI(v, sign_flip); // Low precision, x is small now
    if (sign_flip)
        sign *= -1;

    // =====
    // STAGE 2: Core Computation (Trisection + Taylor)
    // =====
    // Now v is in [0, π], compute reduction factor k
    intmax_t k = 8 * (intmax_t)ceil(log(2) * log(p_target));
    k = (intmax_t)ceil(2.0 * k / 3);
    // Adjust k based on actual argument size
    k += v.exponent();
    k = std::max((intmax_t)0, k); // ensure that we dont do negative reduction

    // Precision for core computation
    // Must account for error amplification by 3^k
    size_t extra_bits = 0;
    extra_bits += (size_t)(k * 1.585); // Error amplification: k * log2(3)
    extra_bits += (size_t)ceil(log2(p_target / 3.0 * 5.0)); // Roundoff
    extra_bits += 20; // Guard bits

    size_t prec2_bits = p_target_bits + extra_bits;
    size_t prec2 = (size_t)(prec2_bits / log2(10)) + 1;

    // IMPORTANT: Reduce precision to p2 (might be less than p1!)
    float_precision sinx, vsq, terms, r;
    v.precision(prec2);
    r.precision(prec2);
```

Fast Trigonometric functions for Arbitrary Precision numbers

```
sinx.precision(prec2);
vsq.precision(prec2);
terms.precision(prec2);

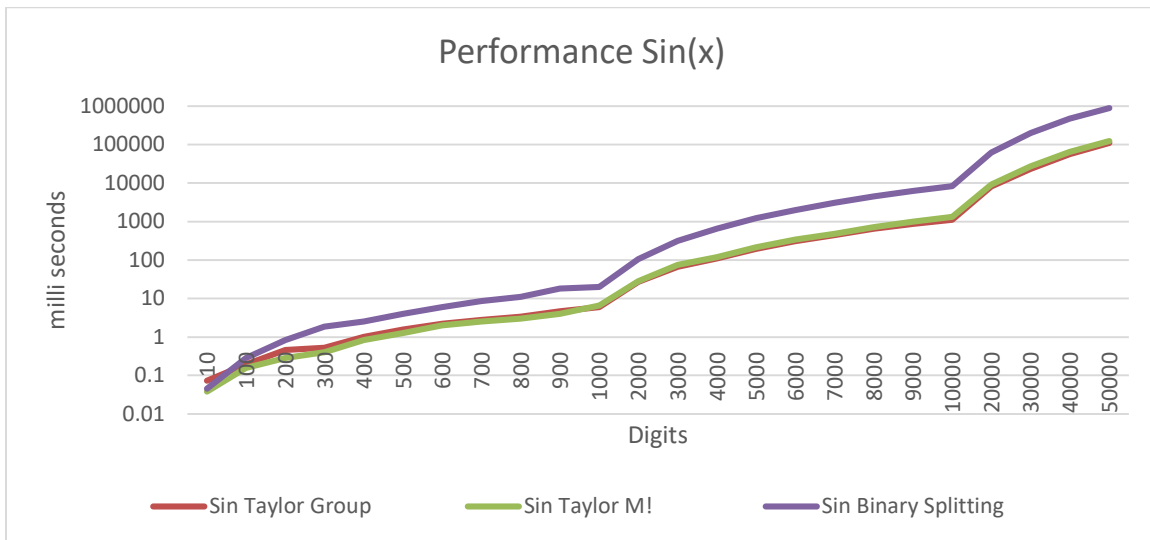
// Now use the trisection identity sin(3x)=3*sin(x)-4*sin(x)^3
// k times k. Where k is the number of reduction factor based on the needed
precision of the argument.
r = c3;
r = pow(r, float_precision(k)); // Since r and k is an integer this wil be faster
v /= r;
vsq = v.square();
r = v;
sinx = v;

// How many Taylor terms
double M, y, dy, xdouble;
xdouble = double(v);
M = prec2 * log(10) / (log(prec2 * 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) - prec2 * 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;
}

int m = int(M + 1);
//m += m & 0x1 ? 0 : 1; // Ensure the odd terms
m += 1 - m % 2; // Ensure m is odd
// Do (m+1)/2 Taylor terms in reverse order and deferred the division to after the
Taylor terms looping
int_precision mFak(1);
sinx = vsq;
bool addTerm = ((m + 1) / 2) % 2 != 0; // Initialize based on the first term's
parity
for (i = m; i >= 3; i -= 2, ++loopcnt)
{
    mFak *= int_precision(i * (i - 1)); // Build the next m*(m-1)
    if(addTerm) // Odd?
        sinx += float_precision(mFak, prec2);
    else
        sinx -= float_precision(mFak, prec2);
    sinx *= (i == 3) ? v : vsq;
    addTerm = !addTerm;
}
sinx /= float_precision(mFak, prec2);
for (; k > 0; --k)
    sinx *= (c3 - c4 * sinx.square());

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

Performance for $\sin(x)$



Performance chart for $\sin(x)$ calculation.

The table above shows that binary splitting for computing $\sin(x)$ is not preferred. The two other methods, using the Taylor series with partial coefficient scaling, are neck-and-neck with those using the Taylor series with full coefficient scaling. We notice that the full coefficient scaling is slightly faster, up to around 1,000 digits, whereas the partial coefficient scaling takes over and is approximately 10% faster than the full coefficient scaling method. This is a pattern we have seen before for other elementary functions.

Recommendation for calculating $\sin(x)$

Based on the performance measure of the various $\sin(x)$ methods, we recommend:

- Always use the cyclic and symmetry rules to reduce the x to the range $[0.. \pi]$
- It is unnecessary to reduce it to the range $[0.. \frac{\pi}{2}]$ Using symmetry, avoiding another calculation of π .
- Use Taylor for $\sin(x)$ using an aggressive reduction factor to speed up the Taylor term calculation.
- Use Coefficient scaling to increase performance.
 - a. Full coefficient scaling is faster, up to around 1,000 digits of precision.
 - b. Partial coefficient scaling is approx. 10% faster than full coefficient scaling above 1,000 digits of precision.
- Maintain separate precision levels for range reduction and evaluation. The dependence of working precision on the trisection count k follows directly from the error amplification analysis derived in Appendix B.

Fast Trigonometric functions for Arbitrary Precision numbers

Cos(x):

For $\cos(x)$, we again use a Taylor series until any further terms do not change the result to the given precision.

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} \dots \text{for any real value } x$$

We can map the equivalent four-step $\cos(x)$ procedure into the interval. $[0 \dots \frac{\pi}{2}]$.

Step 1: We notice that $\cos(x)$ is cyclic with a period of 2π so we can quickly reduce any argument $> 2\pi$ so it falls between 0 and 2π by simply taking x modulo 2π .

Step 2: We can further reduce x so it is between $0 \dots \pi$ using the identity:
 $\cos(2\pi-x) = \cos(x)$ for $x \geq \pi$.

Step 3: We reduce it further by using the symmetry around $\frac{\pi}{2}$ to the range $0 \dots \frac{\pi}{2}$:
 $\cos(x) = -\cos\left(x - \frac{\pi}{2}\right)$ for $x \geq \frac{\pi}{2}$.

Suppose π is 'expensive' to calculate (usually the case with arbitrary precision). In that case, we can omit step 3, since we can achieve the same result by increasing the argument reduction factor. See the section on finding a reasonable reduction factor.

Step 4: Finally, we reduced the argument k number of times using the trisection identity:
 $\cos(3x) = -3\cos(x) + 4\cos^3(x)$

Until x is below a certain threshold, it is obvious from the $\cos(x)$ Taylor series that the smaller the x , the fewer terms we would need. We could also use the double-angle identity:

$$\cos(2x) = 2\cos^2(x) - 1$$

Although the trisection identity serves us well for calculating $\sin(x)$, it turns out that there is a much higher loss of precision when using the trisection identity over the double-angle formula. See later.

This argument reduction is used to reduce the number of Taylor iterations, minimize round-off errors, and minimize calculation time.

After the Taylor series has converged, we use the trisection or double angle identity reverse k number of times to find our $\cos(x)$ result.

Let us find the $\cos(0.7)$ to see how this algorithm works. The error is zero after the 8th Taylor term, and the result is ~ 0.7648421873 .

cos(x)		Original	X Reduced	
x=		0.7	0.7	
Taylor reductions=		0		
Terms	Term value		cos(x)	Error
1	1.00E+00	1.0000000000	1.0000000000	-2.35E-01
2	2.45E-01	0.7550000000	0.7550000000	9.84E-03

Fast Trigonometric functions for Arbitrary Precision numbers

3	1.00E-02	0.7650041667	0.7650041667	-1.62E-04
4	1.63E-04	0.7648407653	0.7648407653	1.42E-06
5	1.43E-06	0.7648421950	0.7648421950	-7.76E-09
6	7.78E-09	0.7648421873	0.7648421873	2.88E-11
7	2.89E-11	0.7648421873	0.7648421873	-7.76E-14
8	7.78E-14	0.7648421873	0.7648421873	0.00E+00

We can see the effect in Step 4 by increasing the number of argument reductions. For example, you get the same result for two reductions after only five iterations. The argument is reduced twice from 0.7 to ~ 0.077

cos(x)					
		Original	X Reduced		
x=		0.7	0.077777778		
Taylor reductions=		2			
Terms	Term value		cos(x)	Error	
1	1.00E+00	1.0000000000	1.0000000000	-2.35E-01	
2	3.02E-03	0.9969753086	0.7647284320	1.14E-04	
3	1.52E-06	0.9969768334	0.7648422102	-2.29E-08	
4	3.07E-10	0.9969768331	0.7648421873	2.48E-12	
5	3.32E-14	0.9969768331	0.7648421873	2.78E-15	

If we do four argument reductions in step 4, we get the result after only four iterations.

cos(x)					
		Original	X Reduced		
x=		0.7	0.008641975		
Taylor reductions=		4			
Terms	Term value		cos(x)	Error	
1	1.00E+00	1.0000000000	1.0000000000	-2.35E-01	
2	3.73E-05	0.9999626581	0.7648407840	1.40E-06	
3	2.32E-10	0.9999626584	0.7648421873	-3.63E-12	
4	5.79E-16	0.9999626584	0.7648421873	-2.81E-13	

Again, we notice that argument reduction can significantly reduce the number of Taylor terms needed, thereby improving performance in calculating $\cos(x)$.

We notice that the error has increased, and we cannot find a better answer than an absolute error of $\sim 1E-13$. The higher the reduction factor, the worse it gets. It should be noted that this issue arises only from the use of a reduction factor, not from the Taylor series.

Many of the same arguments used in calculating $\sin(x)$ also apply to $\cos(x)$, including aggressive argument reduction, coefficient scaling, etc. We have to be careful not to reduce our argument too aggressively. The stability difference between cosine and sine

Fast Trigonometric functions for Arbitrary Precision numbers

reconstruction is quantified in Appendix C, where the error amplification factor is shown to increase from approximately 3 to 9 for the cosine trisection identity.

Cos(x) using double-angle reduction

Unlike $\sin(x)$, cosine is locally flat near zero since $\cos(0)=0$. During reverse argument reconstruction, relative errors therefore amplify more rapidly when using the trisection identity. This conditioning effect explains why aggressive trisection reduces accuracy for $\cos(x)$ but not for $\sin(x)$. It is, therefore, better to use the double-angle formula:

$$\cos(2x) = 2\cos^2(x) - 1 \quad (16)$$

Alternatively, it is even better written as:

$$\cos(2x) = 2(1 - \cos(x))^2 - 4(1 - \cos(x)) + 1 \quad (17)$$

Although it does not prevent round-off errors, it is less sensitive than the trisection formula. We calculate the reduction factor for $\cos(x)$ as:

$$k = 2[\ln(2) * \ln(P)] \quad (18)$$

For higher precision, we made adjustments for the magnitude of x . After Step 2, we know that x lies in the range of $[0 \dots \pi]$. This is equivalent to the exponent of our number (in base 2) being in the range $[-\infty \dots 1]$. We add the exponent to the reduction factor. This means that our reduction factor decreases as x becomes very small, preventing unnecessary reductions. If x is very small, the reduction factor is negative, and we do not perform any argument reductions.

Source for $\cos(x)$ with argument reduction and partial coefficient scaling

```
static float_precision cos(const float_precision& x, const int klimit = 16)
{
    float_precision r, cosx, vsq, terms;
    const float_precision c1(1), c2(2), c3(3), c4(4);
    const int group = 5;
    const size_t p_target = x.precision();
    const size_t p_target_bits = (size_t)(p_target * log2(10));
    int sign = x.sign();
    float_precision v = abs(x);
    uintmax_t loopcnt = 1;
    uintmax_t i;

    // =====
    // STAGE 1: Range Reduction mod 2π
    // =====
    rangeReduction2PI(v); // v now in [0, 2π) at high precision

    // Determine sign based on quadrant in [0, 2π)
    float_precision pi = _float_table(_PI, v.precision());
    float_precision pi_2 = pi;
    pi_2.adjustExponent(-1); // π/2
    float_precision three_pi_2 = pi + pi_2; // 3π/2

    // Determine if cos is negative
    // cos > 0 for v ∈ [0, π/2) ∪ (3π/2, 2π)
```

Fast Trigonometric functions for Arbitrary Precision numbers

```
// cos < 0 for v ∈ (π/2, 3π/2)
bool cos_negative = (v > pi_2 && v < three_pi_2);

// Now reduce v to [0, π/2] for computation
float_precision two_pi = pi * c2;
if (v > pi)
    v = two_pi - v; // Reflect from [π, 2π] to (0, π]
if (v > pi_2)
    v = pi - v; // Reflect from (π/2, π] to [0, π/2)

// v is now in [0, π/2]
//
// =====
// STAGE 2: Core Computation (Trisection + Taylor)
// =====
// Now v is in [0, π], compute reduction factor k
intmax_t k = 2 * (intmax_t)ceil(log(2) * log(p_target));
k = (intmax_t)ceil(2.0 * k / 3);
// Adjust k based on actual argument size
k += v.exponent();
k = std::max((intmax_t)0, k); // ensure that we dont do negative reduction

// Precision for core computation
// Must account for error amplification by 3^k
size_t extra_bits = 0;
extra_bits += (size_t)(k * 1.585); // Error amplification: k * log2(3)
extra_bits += (size_t)ceil(log2(p_target / 3.0 * 5.0)); // Roundoff
extra_bits += 20; // Guard bits

size_t prec2_bits = p_target_bits + extra_bits;
size_t prec2 = (size_t)(prec2_bits / log2(10)) + 1;

// IMPORTANT: Reduce precision to p2 (might be less than p!)
v.precision(prec2);
cosx.precision(prec2);
r.precision(prec2);
vsq.precision(prec2);
terms.precision(prec2);

// Now use the trisection identity cos(3x)=-3*cos(x)+4*cos(x)^3
// k times k. Where k is the number of reduction factor based on the needed
precision of the argument.
r = c3;
r = pow(r, float_precision(k)); // Since r and k is an integer this will be faster
v /= r;
vsq = v.square();
r = c1;
cosx = r;

if (group == 1)
{
    // Now iterate using taylor expansion
    for (i = 2;; i += 2, ++loopcnt)
    {
        r *= vsq / float_precision(i*(i - 1)); // de is only 20 digits
        standard precision but since v2 is precision is will be extended to v2 precision //(
        m<USHRT_MAX? float_precision( m * (m-1) ) : float_precision(m) * float_precision(m-1) );
        r.change_sign();
        if (cosx + r == cosx)
            break;
        cosx += r;
        // terms = restore_cosx(cosx, k); // DEBUG
        // cout << "\tCosx=" << terms << endl; // DEBUG
    }
}
else
{ // no overflow possible until terms exceed 7,100 ~ A little more than
145,000 digits
    std::vector<float_precision> vn(group); // vn[0] is not used
```

Fast Trigonometric functions for Arbitrary Precision numbers

```
std::vector<float_precision> cn(group); //

if (group > 3 && klimit == 0)
    i = 0;

for (i = 0; i < group; ++i)
{
    cn[i].precision(prec2); vn[i].precision(prec2);
    if (i == 1) vn[1] = vsq;
    if (i > 1) vn[i] = vn[i - 1] * vsq;
}
// Now iterate
for (i = 2; ; )
{
    intmax_t j;
    for (j = group - 1; j >= 0; --j)
    {
        if (j == group - 1)
        {
            cn[j] = float_precision((i + 2 * j - 1)*(i + 2 * j),
prec2);

            if ((i / 2 + j - 1) & 0x1) // Odd
                cn[j].change_sign();
        }
        else
        {
            cn[j] = -cn[j + 1] * float_precision((i + 2 * j -
1)*(i + 2 * j), prec2);
        }
    }

    cn[0] = abs(cn[0]).inverse();
    // Summing adding from smallest to largest number
    terms = vn[group - 1];
    if ((i / 2 + group - 1) & 0x1)
        terms.change_sign();
    for (j = group - 1; j >= 2; --j)
        terms += cn[j] * vn[j - 1];
    terms += cn[1];

    r *= vsq*cn[0];
    terms *= r;
    i += 2 * group; // Update term count
    loopcnt += group;
    if (cosx + terms == cosx) // Reach precision
        break; // yes terminate loop
    cosx += terms; // Add taylor terms to result
    if (group > 1)
        r *= vn[group - 1]; // ajust r to last Taylor term
}

}

//cosx.precision(4 * precision);
for (; k > 0; --k)
    cosx *= (c4 * cosx.square() - c3);

if (cos_negative)
    cosx = -cosx;

// Round to same precision as argument and rounding mode
cosx.mode(x.mode());
cosx.precision(x.precision());
loopcnt_cos = loopcnt;
return cosx;
}
```

Fast Trigonometric functions for Arbitrary Precision numbers

Cos(x) using full coefficient scaling.

As with $\sin(x)$, we can improve performance by using partial or full coefficient scaling. The version with full coefficient scaling follows the same layout as for $\sin(x)$, but uses the Taylor series for $\cos(x)$. It is shown below using the trisection identity for argument reduction. (You could just as well have used the double-angle formula instead.)

Source for $\cos(x)$ with argument reduction and full coefficient scaling

```
// cos(x) using argument reduction and full coefficient scaling
//
static float_precision cosM(const float_precision& x, const int klimit = 16)
{
    float_precision r, cosx, vsq, terms;
    const float_precision c1(1), c2(2), c3(3), c4(4);
    const int group = 5;
    const size_t p_target = x.precision();
    const size_t p_target_bits = (size_t)(p_target * log2(10));
    int sign = x.sign();
    float_precision v = abs(x);
    uintmax_t loopcnt = 1;
    uintmax_t i;

    // =====
    // STAGE 1: Range Reduction mod 2π
    // =====
    rangeReduction2PI(v); // v now in [0, 2π) at high precision

    // Determine sign based on quadrant in [0, 2π)
    float_precision pi = _float_table(_PI, v.precision());
    float_precision pi_2 = pi;
    pi_2.adjustExponent(-1); // π/2
    float_precision three_pi_2 = pi + pi_2; // 3π/2

    // Determine if cos is negative
    // cos > 0 for v ∈ [0, π/2) ∪ (3π/2, 2π)
    // cos < 0 for v ∈ (π/2, 3π/2)
    bool cos_negative = (v > pi_2 && v < three_pi_2);

    // Now reduce v to [0, π/2] for computation
    float_precision two_pi = pi * c2;
    if (v > pi)
        v = two_pi - v; // Reflect from [π, 2π) to (0, π]
    if (v > pi_2)
        v = pi - v; // Reflect from (π/2, π] to [0, π/2)

    // v is now in [0, π/2]
    //
    // =====
    // STAGE 2: Core Computation (Trisection + Taylor)
    // =====
    // Now v is in [0, π], compute reduction factor k
    intmax_t k = 2 * (intmax_t)ceil(log(2) * log(p_target));
    k = (intmax_t)ceil(2.0 * k / 3);
    // Adjust k based on actual argument size
    k += v.exponent();
    k = std::max((intmax_t)0, k); // ensure that we dont do negative reduction

    // Precision for core computation
    // Must account for error amplification by 3^k
    size_t extra_bits = 0;
    extra_bits += (size_t)(k * 1.585); // Error amplification: k * log2(3)
    extra_bits += (size_t)ceil(log2(p_target / 3.0 * 5.0)); // Roundoff
    extra_bits += 20; // Guard bits

    size_t prec2_bits = p_target_bits + extra_bits;
    size_t prec2 = (size_t)(prec2_bits / log2(10)) + 1;
```

Fast Trigonometric functions for Arbitrary Precision numbers

```
// IMPORTANT: Reduce precision to p, (might be less than p.!)
v.precision(prec2);
cosx.precision(prec2);
r.precision(prec2);
vsq.precision(prec2);
terms.precision(prec2);

// Check that argument is larger than 2*PI and reduce it if needed.
// No need for high precision. we just need to figure out if we need to Calculate
PI with a higher precision
if (abs(v) > float_precision(2 * 3.14159265))
{ // Reduce argument to between 0..2P
  cosx = _float_table(_PI, prec2);
  cosx.adjustExponent(+1); // Multiply with 2
  if (abs(v) > cosx)
  {
    r = v / cosx;
    (void)modf(r, &r);
    v -= r * cosx;
  }
  if (v < float_precision(0))
    v += cosx;
}

// Reduced it further to between 0..PI.
// However avoid calculating PI is not needed.
// No need for high precision. we just need to figure out if we need to Calculate
PI with a higher precision
if (abs(v) > float_precision(3.14159265))
{
  r = _float_table(_PI, prec2);
  if (v > r)
    v = r * c2 - v; // cos(x)=cos(2PI - x) for x >= PI
}

// Now use the trisection identity cos(3x)=-3*cos(x)+4*cos(x)^3
// k times k. Where k is the number of reduction factor based on the needed
precision of the argument.
r = c3;
r = pow(r, float_precision(k)); // Since r and k is an integer this wil be faster
v /= r;
vsq = v.square();

// How many Taylor terms
double M, y, dy, xdouble;
xdouble = double(v);
M = prec2 * log(10) / (log(prec2 * 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) - prec2 * 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;
}

int m = int(M + 1);
//m += m & 0x1 ? 1 : 0; // Ensure the even terms
m += m % 2; // Ensure m is odd
// Do (m+1)/2 Taylor terms in reverse order and deferred the division to after the
Taylor terms looping
int_precision mFak(1);
cosx = vsq;
bool addTerm = ((m + 1) / 2) % 2 != 0; // Initialize based on the first term's
parity
for (i = m; i >= 4; i -= 2, ++loopcnt)
```

Fast Trigonometric functions for Arbitrary Precision numbers

```
{
    mFak *= int_precision(i * (i - 1)); // Build the next m*(m-1)
    if (addTerm) // Odd?
        cosx += float_precision(mFak, prec2);
    else
        cosx -= float_precision(mFak, prec2);
    cosx *= vsq;
    addTerm = !addTerm;
}
mFak *= int_precision(2);
cosx /= float_precision(mFak, prec2);
cosx += c1; // add the first Taylor term last!

for (; k > 0; --k)
    cosx *= (c4 * cosx.square() - c3);

// Round to same precision as argument and rounding mode
cosx.mode(x.mode());
cosx.precision(x.precision());
loopcnt_cos = loopcnt;
return cosx;
}
```

Cos(x) using sin(x)

Since we have a speedy and robust implementation of $\sin(x)$ that does not suffer from the same issue of using a high reduction factor compared to $\cos(x)$, it could be interesting to calculate $\cos(x)$ using $\sin(x)$:

$$\cos(x) = \sqrt{1 - \sin^2(x)} \quad (19)$$

This approach assumes that the square root routine operates at a precision equal to or higher than that of $\sin(x)$. Otherwise, cancellation in $1 - \sin^2(x)$ may introduce additional rounding error.

This increases performance by a factor of 2 compared to the traditional method of calculating $\cos(x)$ directly, and it is recommended.

Source for $\cos(x)$ using $\sin(x)$

```
float_precision cos(const float_precision& x) {
    if (isnan(x) || isinf(x))
        return FP_QUIET_NAN;

    const size_t p_target = x.precision();
    float_precision v = abs(x);

    // =====
    // STAGE 1: Range Reduction mod 2π
    // =====
    rangeReduction2PI(v); // v now in [0, 2π)

    // For cosine, DON'T use rangeReductionPI here!
    // Instead, do the sign determination based on [0, 2π) quadrant
    // =====
    // STAGE 2: Core Computation
    // =====
    size_t p_work = p_target + 2;
    v.precision(p_work);

    float_precision pi = _float_table(_PI, p_work);
    float_precision pi_2 = pi;
```

Fast Trigonometric functions for Arbitrary Precision numbers

```
pi_2.adjustExponent(-1); // π/2
float_precision three_pi_2 = pi + pi_2; // 3π/2

// Determine sign from quadrant in [0, 2π)
// cos > 0 for v ∈ [0, π/2) ∪ (3π/2, 2π)
// cos < 0 for v ∈ (π/2, 3π/2)
bool cos_negative = (v > pi_2 && v < three_pi_2);

// Now reduce v to [0, π/2] for computation
// using: cos(π - v) = -cos(v) and cos(π + v) = -cos(v)
if (v > pi)
    v = pi * float_precision(2) - v; // Reflect: 2π - v
if (v > pi_2)
    v = pi - v; // Reflect: π - v

// Now v is in [0, π/2]

// Check if v is near π/2
double v_dbl = double(v);
bool near_pi_2 = (v_dbl > 3.14159265358979323846 / 2 - 0.1);

float_precision cosx(0, p_work);
const float_precision c1(1, p_work);

if (near_pi_2) {
    // cos(v) = sin(π/2 - v) for v near π/2
    float_precision arg = pi_2 - v;
    cosx = sin(arg);
}
else {
    // cos(x) = √(1 - sin²(x)) [always positive since v ∈ [0, π/2]]
    float_precision sinx = sin(v);
    cosx = sqrt(c1 - sinx.square());
}

// Apply sign from quadrant determination
if (cos_negative)
    cosx = -cosx;

cosx.precision(p_target);
cosx.mode(x.mode());
return cosx;
}
```

There is another alternative to using the identity: $\cos(x) = \sin\left(\frac{\pi}{2} - x\right)$. If you have a fast generation of π , you will experience a similar performance to the $\cos(x) = \sqrt{1 - \sin^2(x)}$ But I think relying on the faster $\text{sqrt}(x)$ function will be safer.

Source code for cos via sine identity

```
// cos(x)=sin(pi/2-x)
float_precision testcosviasinidentity(const float_precision& x) {
    size_t p_target = x.precision();

    // Stage 1: Range reduce x to [0, 2π)
    float_precision v = abs(x); // cos is even: cos(-x) = cos(x)
    rangeReduction2PI(v); // v now in [0, 2π) at high precision

    // Stage 2: Compute π/2 - v with modest extra precision
    size_t prec = v.precision() + 5; // Just a few guard digits
    float_precision pi_2 = _float_table(_PI, prec);
    pi_2.adjustExponent(-1); // π/2

    float_precision arg = pi_2 - v;

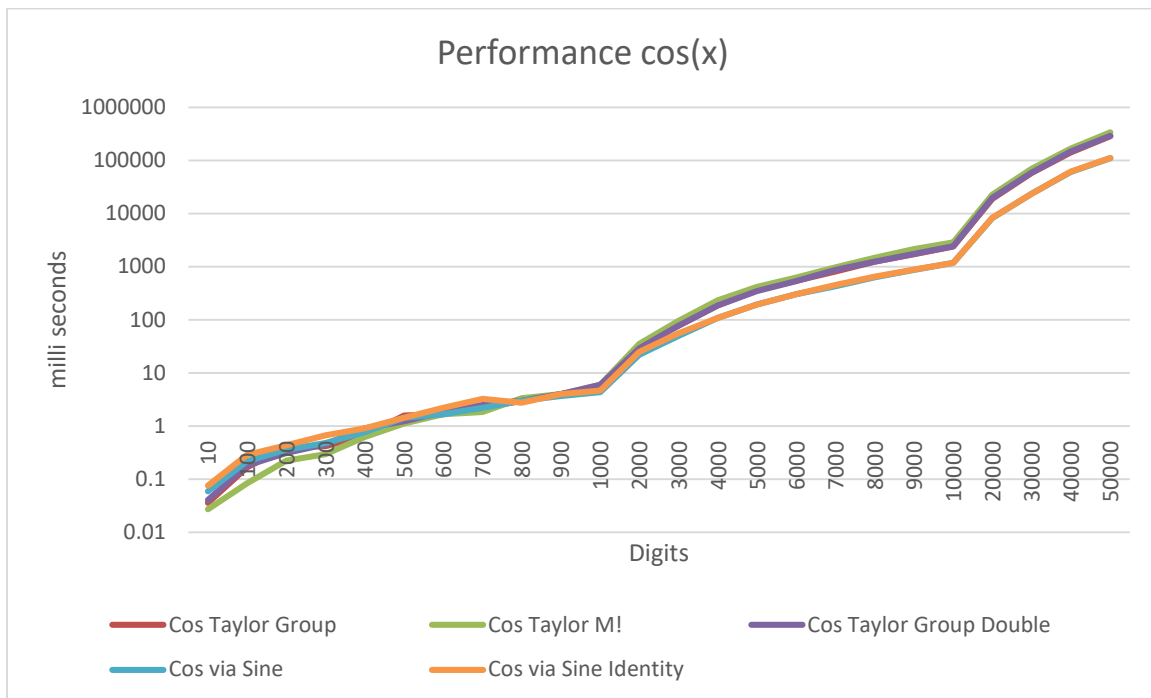
    // Stage 3: Call sin() which will do its own precision management
```

Fast Trigonometric functions for Arbitrary Precision numbers

```
arg.precision(p_target); // Reduce to target before calling sin
float_precision result = sin(arg);

result.precision(p_target);
result.mode(x.mode());
loopcnt_cos = loopcnt_sin;
return result;
}
```

Performance for Cos(x)



Performance graph for cos(x) with various methods.

As the graph above shows, computing cos(x) via the sine identity is by far the fastest method of the two methods that show similar performance. There is an exception: for small precision in cos(x), using full coefficient scaling is slightly faster below 800-1000 digits.

Recommendation for calculating cos(x)

Based on the performance measure of the various cos(x) methods, we recommend:

- Always use the cyclic and symmetry rules to reduce the x to the range $[0, \pi]$
- It is unnecessary to reduce it to the range $[0, \dots, \frac{\pi}{2}]$ using symmetry, avoiding another calculation of π .
- Use the double angle formula instead of the trisection formula for argument reduction.

Fast Trigonometric functions for Arbitrary Precision numbers

- Do not use the Taylor series for $\cos(x)$ with an aggressive reduction factor to speed up the Taylor term calculation. If you do it anyway, use it with a coefficient scaling to increase performance.
- Use $\cos(x) = \sqrt{1 - \sin^2(x)}$ or $\cos(x) = \sin\left(\frac{\pi}{2} - x\right)$
- It is the preferred method for calculating $\cos(x)$, two times faster than the other $\cos(x)$ methods.
- Maintain separate precision levels for range reduction and evaluation.

Tan():

We could use a Taylor series for $\tan(x)$; however, since we have an efficient implementation of $\sin(x)$, it is better to use the identity:

$$\tan(x) = \frac{\sin(x)}{\sqrt{1 - \sin^2(x)}} \quad (20)$$

However, before we start the calculation, we first reduce the argument x to a value between 0 and 2π , then call $\sin(x)$ (see above).

Alternatively, we could use the Taylor series for $\tan(x)$:

$$\tan(x) = x + \frac{x^3}{3} + \frac{2x^5}{15} + \frac{17x^5}{315} + \frac{62x^9}{2835} + \dots + \frac{2^{2n}(2^{2n}-1)B_n x^{2n-1}}{(2n)!} + \dots \quad (21)$$

Where B_n is the Bernoulli number; however, since we don't know how many Bernoulli numbers we need, this will require it to be calculated on the fly and, therefore, will be way more complicated to implement than the identity for $\tan(x)$ using $\sin(x)$.

Source for tan(x)

```
float_precision tan(const float_precision& x) {
    if (isnan(x) || isinf(x))
        return FP_QUIET_NAN;

    const size_t p_target = x.precision();
    const size_t p_target_bits = (size_t)(p_target * log2(10));
    float_precision v = abs(x);

    // =====
    // STAGE 1: Range Reduction mod 2π (same as sin)
    // =====
    rangeReduction2PI(v); // v now in [0, 2π)

    // =====
    // STAGE 2: Core Computation
    // =====
    // Adaptive precision for tan(x)
    // tan has worse error amplification (1/cos³) than cos (1/cos²)
    size_t extra;
    if (p_target < 40)
        extra = p_target / 4 + 10; // ~35% extra for low precision
    else if (p_target < 100)
```

Fast Trigonometric functions for Arbitrary Precision numbers

```
        extra = p_target / 10 + 5; // ~15% extra for medium precision
    else
        extra = 5; // Small fixed extra for high precision
    size_t prec2 = p_target + extra;
    // Reduce precision from prec1 to prec2 for core computation
    v.precision(prec2);

    // Get  $\pi$  at working precision prec2
    float_precision pi = _float_table(_PI, prec2);
    float_precision pi_2 = pi / 2;
    float_precision pi_3_2 = pi_2 * 3;
    // Check for singularities
    if (v == pi_2 || v == pi_3_2)
        throw float_precision::domain_error();

    // Check if v is near  $\pi/2$  or  $3\pi/2$ 
    double v_dbl = double(v);
    double pi_dbl = 3.14159265358979323846;
    double v_mod_pi = fmod(v_dbl, pi_dbl);
    bool near_pi_2 = (fabs(v_mod_pi - pi_dbl / 2) < 0.1);
    const float_precision c1(1, prec2);
    float_precision tanx(0, prec2);

    if (near_pi_2) {
        // Near  $\pi/2$  or  $3\pi/2$ : use complementary angle
        //  $\tan(\pi/2 + \delta) = -\cot(\delta) = -\cos(\delta)/\sin(\delta)$ 
        float_precision delta = v - pi_2;
        if (abs(delta) > pi_2) // Actually near  $3\pi/2$ 
            delta = v - pi_3_2;
        // For small  $\delta$ :  $\tan(\delta) = \sin(\delta)/\sqrt{1 - \sin^2(\delta)}$ 
        float_precision sin_delta = sin(delta);
        float_precision tan_delta = sin_delta / sqrt(c1 - sin_delta.square());
        //  $\tan(v) = -1/\tan(\delta)$ 
        tanx = -c1 / tan_delta;
    }
    else {
        // Away from  $\pi/2$ : use  $\tan(x) = \sin(x) / \sqrt{1 - \sin^2(x)}$ 
        float_precision sinx = sin(v);
        float_precision cosx = sqrt(c1 - sinx.square());
        // Determine sign of cos from quadrant
        if (v < pi_2 || v > pi_3_2)
            tanx = sinx / cosx;
        else
            tanx = sinx / (-cosx);
    }

    // Handle sign (tan is odd function:  $\tan(-x) = -\tan(x)$ )
    if (x.sign() < 0)
        tanx.change_sign();
    // Round to target precision
    tanx.mode(x.mode());
    tanx.precision(p_target);
    return tanx;
}
```

Arcsin(x):

We have a few options. Either we can find $\arcsin(x)$ using the Newton method, or we can use a Taylor series for $\arcsin(x)$.

Arcsin using Newton's method

To find the value of $\arcsin(x)$, it is trendy to resort to a Newton iteration when solving the equation $\arcsin(a)=x \Rightarrow a=\sin(x)$.

Fast Trigonometric functions for Arbitrary Precision numbers

Restating the problem as $f(x)=\sin(x)-a=0$ and applying the Newton method we get:
Where $f(x)=\sin(x)-a$ and $f'(x)=\cos(x)$.

$$x_{n+1} = x_n - \frac{\sin(x_n)-a}{\cos(x_n)} \quad (22)$$

We stop when $x_n=x_{n-1}$ for any given precision of the number. We do not want to calculate both $\sin(x)$ and $\cos(x)$, so we replace $\cos(x)$ with the identity:

$$\cos(x) = \sqrt{1 - \sin^2(x)} \quad (23)$$

Yields:

$$x_{n+1} = x_n - \frac{\sin(x_n)-a}{\sqrt{1-\sin^2(x_n)}} \quad (24)$$

To speed up the iteration and ensure convergence, we repeatedly reduced the argument x to a small value using the identity:

$$\text{Arcsin}(x) = 2 \cdot \text{ArcSin}\left(\frac{x}{\sqrt{2}\sqrt{1+\sqrt{1-x^2}}}\right) \quad (25)$$

The x argument will always be between $-1 \leq x \leq 1$ per definition, so we only need a maximum of two argument reductions to get below 0.5.

You can obtain k , the number of reductions, by repeatedly applying the recurrence below k times. Set $x_0=x$ and k is the number of reductions:

$$x_k = \frac{x_{k-1}}{\sqrt{2}\sqrt{1+\sqrt{1-x_{k-1}^2}}} \quad (26)$$

until x_m is sufficiently low. We can start with an initial guess of $\arcsin(x)$ using standard IEEE754. This gives us a starting guess for the Newton iteration with at least 15 significant digits. The Newton iteration will converge quickly with a convergence rate of 2, meaning the number of correct digits doubles per iteration. After we find the new x_n , we will need to multiply the result with $x = x_n \cdot 2^k$ To reverse the argument reduction we did before the Newton iteration.

Let us find the $\arcsin(0.3)$ to see how this algorithm works. After only three iterations, the error is zero, and the result is ~ 0.304693 .

ArcSin(x)	Newton	Original	X Reduced
x=		0.3	0.3
No Reduction		0	
Iteration	x	ArcSin(x)	Error
1	0.304689231	0.304689230851802	3.42E-06
2	0.304692654	0.304692654013555	1.84E-12

Fast Trigonometric functions for Arbitrary Precision numbers

3	0.304692654	0.304692654015398	0.00E+00
----------	-------------	-------------------	----------

Now, assuming we did not do any argument reduction for a moment, we will see a much slower convergence when x gets near 1. The slow convergence observed when x approaches ± 1 is expected because the condition number of $\arcsin(x)$ grows as $1/\sqrt{1-x^2}$. Small changes in x therefore produce large changes in the result, reducing Newton's effective quadratic convergence unless argument reduction is applied first. See below.

ArcSin(x)	Newton	Original	X Reduced
x=		1	1
No Reduction		0	
Iteration	x	ArcSin(x)	Error
1	1.293407993	1.293407993026020	2.77E-01
2	1.432998367	1.432998366665080	1.38E-01
3	1.502006577	1.502006576891840	6.88E-02
4	1.536415021	1.536415021395350	3.44E-02
5	1.553607368	1.553607367680850	1.72E-02
6	1.562202059	1.562202058854760	8.59E-03
7	1.566499219	1.566499219274400	4.30E-03
8	1.568647776	1.568647776340770	2.15E-03
9	1.569722052	1.569722051981120	1.07E-03
10	1.570259189	1.570259189439680	5.37E-04
11	1.570527758	1.570527758123650	2.69E-04
12	1.570662042	1.570662042459940	1.34E-04
13	1.570729185	1.570729184627400	6.71E-05
14	1.570762756	1.570762755710630	3.36E-05
15	1.570779541	1.570779541251150	1.68E-05
16	1.570787934	1.570787934020610	8.39E-06
17	1.57079213	1.570792130414050	4.20E-06
18	1.570794229	1.570794228613920	2.10E-06
19	1.570795278	1.570795277678890	1.05E-06
20	1.570795802	1.570795802251530	5.25E-07
21	1.570796064	1.570796064492250	2.62E-07
22	1.570796196	1.570796195702950	1.31E-07
23	1.570796261	1.570796260914560	6.59E-08
24	1.570796295	1.570796294618790	3.22E-08
25	1.570796312	1.570796311871080	1.49E-08
26	1.570796319	1.570796319310360	7.48E-09

Even after 26 iterations, we only get a decent result with an error margin of 7.48E-9, while with two argument reductions, we have the result with only three iterations.

ArcSin(x)	Newton	Original	X Reduced
-----------	--------	----------	-----------

Fast Trigonometric functions for Arbitrary Precision numbers

x=		1	0.382683432
No Reduction		2	
Iteration	x	ArcSin(x)	Error
1	0.392678725	1.570714899985370	2.04E-05
2	0.392699082	1.570796326451610	8.58E-11
3	0.392699082	1.570796326794900	0.00E+00

This example demonstrates the benefit of reducing arguments before applying Newton's iterations.

Using Newton's iteration results in relatively few iterations; however, it is still not as fast as the direct approach using the Taylor series; see the next section.

Arcsin(x) using Taylor series and argument reduction

Instead of the Newton method, we can use the Taylor Series for arcsin(x) given by:

$$\text{Arcsin}(x) = x + \frac{x^3}{2 \cdot 3} + \frac{3x^5}{2 \cdot 4 \cdot 5} + \frac{3 \cdot 5x^7}{2 \cdot 4 \cdot 6 \cdot 7} + \frac{3 \cdot 5 \cdot 7x^9}{2 \cdot 4 \cdot 6 \cdot 8 \cdot 9} \dots \quad (27)$$

This gives us a more direct approach to arcsin(x), and when applied together with the reductions, we see a speed-up in the calculation in the range of two. This method will become increasingly faster as precision rises compared to the Newton version.

The Taylor series seems a little hard to digest. If we denote the n'th Taylor term, r, we can go from one Taylor term to the next using the following recurrence:

$$r_1 = x$$

$$r_n = r_{n-1} \frac{(2n-3)^2 \cdot x^2}{(2n-1)(2n-2)} \text{ for } n = 2, 3, \dots, m$$

We calculate the reducing factor, k, as:

$$2 \cdot [\ln(2) * \ln(\text{precision})] \quad (28)$$

Adjust the reduction factor downwards if x is small to avoid unnecessary reductions. We should be careful not to be too aggressive because of the reduction formula:

$$\text{Arcsin}(x) = 2 \cdot \text{ArcSin}\left(\frac{x}{\sqrt{2}\sqrt{1+\sqrt{1-x^2}}}\right) \quad (29)$$

Require one division and two square roots (the $\sqrt{2}$ is a constant that can be calculated before the reduction), two multiplications, and two additions/subtractions. The benefit of

Fast Trigonometric functions for Arbitrary Precision numbers

using reduction slows down and becomes counterproductive when the reduction factor exceeds 30-40.

Below is an example of using the Taylor Series to calculate $\text{arcSin}(x)$ with $x = 0.3$.

ArcSin(x)	Taylor	Original	X Reduced	
x=		0.3	0.3	
No Reduction		0		
Terms	Term Value	Taylor sum	Arcsin(x)	Error
1	3.00E-01	0.3000000000000000	0.3000000000000000	4.69E-03
2	4.50E-03	0.3045000000000000	0.3045000000000000	1.93E-04
3	1.82E-04	0.3046822500000000	0.3046822500000000	1.04E-05
4	9.76E-06	0.304692013392857	0.304692013392857	6.41E-07
5	5.98E-07	0.304692611400670	0.304692611400670	4.26E-08
6	3.96E-08	0.304692651032278	0.304692651032278	2.98E-09
7	2.77E-09	0.304692653798869	0.304692653798869	2.17E-10
8	2.00E-10	0.304692653999250	0.304692653999250	1.61E-11
9	1.49E-11	0.304692654014168	0.304692654014168	1.23E-12
10	1.13E-12	0.304692654015302	0.304692654015302	9.53E-14
11	8.78E-14	0.304692654015390	0.304692654015390	7.55E-15
12	6.88E-15	0.304692654015397	0.304692654015397	6.66E-16

After 12 Taylor terms, we have 15-16 correct decimal digits. If we run it with a reduction factor of two, we get:

ArcSin(x)	Taylor	Original	X Reduced	
x=		0.3	0.076099521	
No Reduction		2		
Terms	Term Value	Taylor sum	Arcsin(x)	Error
1	7.61E-02	0.076099520968904	0.304398083875615	2.95E-04
2	7.35E-05	0.076172971428661	0.304691885714644	7.68E-07
3	1.91E-07	0.076173162841418	0.304692651365671	2.65E-09
4	6.60E-10	0.076173163501238	0.304692654004951	1.04E-11
5	2.60E-12	0.076173163503838	0.304692654015353	4.47E-14
6	1.11E-14	0.076173163503849	0.304692654015397	3.89E-16

The same result is achieved after only six iterations. This again demonstrates that argument reduction can significantly reduce the workload.

Arcsin coefficient scaling

We have seen that we can typically achieve 2-3 times better performance by implementing coefficient scaling. If we try to group two Taylor terms to avoid a division, we get from the Taylor terms listed above:

Fast Trigonometric functions for Arbitrary Precision numbers

$$r_1 = x, \quad r_n = r_{n-1} \frac{(2n-3)^2 \cdot x^2}{(2n-1)(2n-2)} \text{ for } n = 2, 3, \dots, m$$

If we denote for simplicity $u_1 = (2n-3)^2, l_1 = (2n-1)(2n-2)$ and the following term u_2 and l_2 , we get from the above recurrence when grouping two terms:

$$\text{Two Taylor terms} = r_{n-1} \frac{u_1 \cdot x^2}{l_1} + r_{n-1} \frac{u_1 \cdot x^2}{l_1} \cdot \frac{u_2 \cdot x^2}{l_2} \Rightarrow$$

$$r_{n-1} x^2 \left(\frac{u_1}{l_1} + \frac{u_1}{l_1} \cdot \frac{u_2 \cdot x^2}{l_2} \right) \Rightarrow$$

$$r_{n-1} x^2 \left(\frac{u_1 l_2}{l_1 l_2} + \frac{u_1 u_2 \cdot x^2}{l_1 l_2} \right) \Rightarrow$$

$$r_{n-1} x^2 \left(\frac{u_1 l_2 + u_1 u_2 \cdot x^2}{l_1 l_2} \right)$$

The new recurrence for r , grouping two Taylor terms, is given by:

$$r_1 = x, \quad r_{n+1} = r_{n-1} x^4 \frac{u_1 u_2}{l_1 l_2}$$

Continue one by grouping three Taylor terms you get.

$$r_{n-1} x^2 \left(\frac{u_1 l_2 l_3 + u_1 u_2 l_3 \cdot x^2 + u_1 u_2 u_3 \cdot x^4}{l_1 l_2 l_3} \right)$$

The new r_{n+2} is given by:

$$r_1 = x, \quad r_{n+2} = r_{n-1} x^6 \frac{u_1 u_2 u_3}{l_1 l_2 l_3}$$

You can continue on this path. In the current implementation, we group five Taylor terms and scale the coefficients accordingly.

Source for `Arcsin(x)` with coefficient scaling and argument reduction

```
float_precision asin(const float_precision& x)
{
    const int group = 5;
    size_t precision = x.precision() + 2 + (size_t)ceil(log10(x.precision()));
    intmax_t k, i;
    size_t loopcnt = 1;
    int sign;
    float_precision r, asinx, v(x), vsq, lc, uc, terms;
    const float_precision c1(1), c2(2);
```

Fast Trigonometric functions for Arbitrary Precision numbers

```
if (x > c1 || x < -c1)
    throw float_precision::domain_error();

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

// Automatically calculate optimal reduction factor as a power of two
k = 2 * (intmax_t)ceil(log(2)*log(precision));
// Adjust k for the final value of v when v is small (less than 1).
// We know it is in the interval between [0..1]
// This indicates that the exponent is in the range [-inf..0]
// Avoid unnecessary argument reduction if v is small
k += v.exponent();
k = std::max((intmax_t)0, k);

// Adjust the precision
precision += k / 4;
r.precision(precision);
asinx.precision(precision);
v.precision(precision);
vsq.precision(precision);
lc.precision(precision);
uc.precision(precision);
terms.precision(precision);

// Now use the identity arcsin(x)=2arcsin(x/(sqrt(2)*sqrt(1+sqrt(1-x*x))))
// k number of times
r = _float_table(_SQRT2, precision);
for (i = 0; i < k; ++i)
    v /= r * sqrt(c1 + sqrt(c1 - v.square()));

vsq = v.square();
r = v;
asinx = v;
if (group == 1)
{
    // Now iterate using Taylor expansion
    for (i = 3;; i += 2, ++loopcnt)
    {
        // Multiplication fits into 64-bit
        uc = float_precision((i - 2) * (i - 2));
        lc = float_precision(i * i - i);
        r *= uc * vsq / lc;
        if (asinx + r == asinx)
            break;
        asinx += r;
    }
}
else
{
    std::vector<float_precision> vn(group); // vn[0] is not used
    std::vector<float_precision> un(group);
    std::vector<float_precision> ln(group);

    for (i = 0; i < group; ++i)
        { // Adjust to working precision
```

Fast Trigonometric functions for Arbitrary Precision numbers

```
un[i].precision(precision);
ln[i].precision(precision);
vn[i].precision(precision);
if (i == 1) vn[1] = vsq;
if (i > 1) vn[i] = vn[i - 1] * vsq;
}
// Now iterate
for (i = 3;; )
{
    // Recalculate the coefficients
    intmax_t j; uintmax_t tmp;
    for (j = group - 1; j >= 0; --j)
    {
        if (j == group - 1)
        {
            tmp = (i - 2); tmp *= tmp;
            uc = float_precision(tmp); un[j] = uc;
            tmp = (i - 1 + j * 2); tmp *= tmp + 1;
            lc = float_precision(tmp); ln[j]=lc;
        }
        else
        {
            tmp = i - 4 + (group - j) * 2; tmp *= tmp;
            uc = float_precision(tmp);
            un[j] = uc;
            tmp = i - 1 + j * 2; tmp *= tmp + 1;
            lc = float_precision(tmp);
            ln[j] = lc;
            un[j] *= un[j + 1]; ln[j] *= ln[j + 1];
        }
    }

    ln[0] = ln[0].inverse();
    // Adding from smallest to largest number
    uc = terms = vn[group - 1]* un[0];
    for (j = group - 1; j >= 2; --j)
        terms += (un[group-j]*ln[j]) * vn[j - 1];
    terms += un[group - 1] * ln[1];
    i += 2*group;
    loopcnt += group;
    r *= vsq * ln[0];
    terms *= r;
    if (asinx + terms == asinx)
        break;
    asinx += terms;
    if (group > 1)
        r *= uc;      // ajust r to last Taylor term
}

// Reverse argument reduction
if (k > 0)
    asinx.adjustExponent(+k); // asinx*=2^k

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

Fast Trigonometric functions for Arbitrary Precision numbers

```
if (sign < 0)
    asinx.change_sign();
return asinx;
}
```

Recommendation for calculating Arcsin(x)

Based on the performance measure of the various arcsin(x) methods, we recommend:

- The preferred method uses the Taylor series for arcsin(), argument reduction, and coefficient scaling.
- Arcsin() using the Newton method does not perform as well as the Taylor series method. The performance issue gets worse with increasing precision.
- Only use a moderate number of argument reductions since it is very time-consuming to calculate. (This involves a division and calculation of two square roots.)

Arccos(x):

To find Arccos(x), we used the identity:

$$\text{Arccos}(x) = \frac{\pi}{2} - \text{Arcsin}(x) \quad (30)$$

There is not much else you can do.

Source for Arccos(x)

```
float_precision acos( const float_precision& x )
{
    size_t precision;
    float_precision y;
    const float_precision c1(1);

    if( x > c1 || x < -c1 )
        throw float_precision::domain_error(); }

    precision = x.precision();
    y = _float_table( _PI, precision );
    y.adjustExponent(-1);
    y -= asin( x );

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

Fast Trigonometric functions for Arbitrary Precision numbers

Arctan(x):

There are two interesting methods to use. One is the standard Taylor series, and the other is contributed by Euler and is considered faster (at least fewer terms are needed).

Arctan(x) using the Taylor series

For $\arctan(x)$, we can use a Taylor series until any additional addition does not change the result for the given precision of the number:

$$\text{Arctan}(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \frac{x^9}{9} - \dots, \text{ where } |x| \leq 1 \quad (31)$$

However, before we start the Taylor series, we first need to reduce the argument x to a smaller value, which will make the Taylor series run faster by using fewer Taylor terms. We use the identity:

$$\text{Arctan}(x) = 2 \cdot \arctan\left(\frac{x}{1+\sqrt{1+x^2}}\right) \quad (32)$$

k number of times until x is sufficiently low.

This argument reduction is done to reduce the number of Taylor steps, minimize the round-off errors and calculation time, and ensure that our Taylor series is stable.

We calculate the reducing factor, k , as:

$$2 \cdot \lceil \ln(2) * \ln(\text{precision}) \rceil \quad (33)$$

Adjust the reduction factor downwards if x is small to avoid unnecessary reductions. We should be careful not to be too aggressive because of the reduction formula:

$$\text{Arctan}(x) = 2 \cdot \arctan\left(\frac{x}{1+\sqrt{1+x^2}}\right) \quad (34)$$

It requires one division, one square root, and two additions. The benefit of using reduction slows down and becomes counterproductive when the reduction factor exceeds 30-40.

After the Taylor series converges, we multiply the result by 2^k to obtain our $\arctan(x)$ result. Looking closer at the argument reduction, you will notice that we never need more than one argument reduction to reduce $x > 1$ to $x < 1$. The first reduction will give us a max of ± 1 since:

$$\lim_{x \rightarrow \infty} \left(\frac{x}{1 + \sqrt{1 + x^2}} \right) = 1$$

or

Fast Trigonometric functions for Arbitrary Precision numbers

$$\lim_{x \rightarrow -\infty} \left(\frac{x}{1 + \sqrt{1 + x^2}} \right) = -1$$

Let us find the arctan(0.3) to see how this algorithm works. After the 13th Taylor term, the errors do not decrease, and the result is ~0.291456794477867.

ArcTan(x)		Taylor	Original	X Reduced	
x=			0.3	0.3	
No Reduction			0		
Terms	Term Value	Taylor sum	Arctan(x)	Error	
1	3.00E-01	0.3000000000000000	0.3000000000000000	8.54E-03	
2	9.00E-03	0.2910000000000000	0.2910000000000000	4.57E-04	
3	4.86E-04	0.2914860000000000	0.2914860000000000	2.92E-05	
4	3.12E-05	0.291454757142857	0.291454757142857	2.04E-06	
5	2.19E-06	0.291456944142857	0.291456944142857	1.50E-07	
6	1.61E-07	0.291456783100130	0.291456783100130	1.14E-08	
7	1.23E-08	0.291456795364153	0.291456795364153	8.86E-10	
8	9.57E-10	0.291456794407559	0.291456794407559	7.03E-11	
9	7.60E-11	0.291456794483524	0.291456794483524	5.66E-12	
10	6.12E-12	0.291456794477407	0.291456794477407	4.60E-13	
11	4.98E-13	0.291456794477905	0.291456794477905	3.77E-14	
12	4.09E-14	0.291456794477864	0.291456794477864	3.16E-15	
13	3.39E-15	0.291456794477867	0.291456794477867	2.22E-16	

If we take a two-argument reduction, we reduce the number of Taylor terms taken. E.g., arctan(0.3) gives the result after only six Taylor terms.

ArcTan(x)		Taylor	Original	X Reduced	
x=			0.3	0.072993423	
No Reduction			2		
Terms	Term Value	Taylor sum	Arctan(x)	Error	
1	7.30E-02	0.072993423050513	0.291973692202050	5.17E-04	
2	1.30E-04	0.072863785762585	0.291455143050342	1.65E-06	
3	4.14E-07	0.072864200190164	0.291456800760656	6.28E-09	
4	1.58E-09	0.072864198612959	0.291456794451837	2.60E-11	
5	6.54E-12	0.072864198619495	0.291456794477981	1.13E-13	
6	2.85E-14	0.072864198619467	0.291456794477867	5.55E-16	

If we do four argument reductions, we only need four Taylor terms to get the result. As we have seen before, argument reduction is crucial for reducing the number of Taylor terms required as precision increases.

Fast Trigonometric functions for Arbitrary Precision numbers

The issue with arbitrary precision

The number of Taylor terms used to reach a result does not seem so bad at first glance. In the previous examples, we only used approx. 15 decimal digits. However, when dealing with higher precisions, e.g., 1,000, 10,000, or even 100,000 digits, we suddenly need to evaluate many more Taylor terms to obtain our result. You can find the approximate value for the number of Taylor Terms n by:

$$\frac{x^{2n-1}}{2n-1} < 10^{-P} \quad (35)$$

P is the precision in decimal digits, and $|x| < 1$. The terms we dropped are the 2^{n+1} terms. Given

$$\frac{x^{2n+1}}{2n+1} = 10^{-P} \Rightarrow$$

$$(2n + 1) \ln(x) - \ln(2n + 1) = -P \cdot \ln(10) \quad (36)$$

$-\ln(2n+1)$ is small compare to $(2n+1)\ln(x)$ so we drop it and get:

$$(2n + 1) \ln(x) \approx -P \cdot \ln(10) \Rightarrow$$

$$(2n + 1) \approx \frac{-P \cdot \ln(10)}{\ln(x)} \Rightarrow$$

$$n \approx \frac{-P \cdot \ln(10) - \ln(x)}{2 \cdot \ln(x)} \quad (37)$$

Now, if we use $x=10^M$ where M is the magnitude of the number, we can further simplify it:

$$n \approx \frac{-P-M}{2 \cdot M} \quad (38)$$

The number of Taylor terms needed for arctan(x) as a function of precision and argument magnitude.

Digits	10^1	10^2	10^3	10^4	10^5	10^6	10^7	10^8
x								
10^{-1}	5	50	500	5,000	50,000	500,000	5,000,000	50,000,000
10^{-2}	2	25	250	2,500	25,000	250,000	2,500,000	25,000,000
10^{-3}	1	16	166	1,666	16,666	166,666	1,666,666	16,666,666
10^{-4}	1	12	125	1,250	12,500	125,000	1,250,000	12,500,000
10^{-5}	1	10	100	1,000	10,000	100,000	1,000,000	10,000,000
10^{-6}	0	8	83	833	8,333	83,333	833,333	8,333,333
10^{-7}	0	7	71	714	7,142	71,428	714,285	7,142,857
10^{-8}	0	6	62	625	6,250	62,500	625,000	6,250,000
10^{-9}	0	5	55	555	5,555	55,555	555,555	5,555,555

Fast Trigonometric functions for Arbitrary Precision numbers

This table indicates the usefulness of argument reduction.

The table above is quite interesting. For example, the effect of argument reduction for a precision of 100 digits reduces the number of Taylor terms by a factor of six between arguments of -1 in magnitude down to an argument of 10^{-9} in magnitude, which is around a factor of 10 times fewer Taylor Terms. However, overall argument reduction is beneficial at any precision.

Arctan(x) using coefficient scaling

We have seen that we can typically achieve 2-3 times better performance by implementing coefficient scaling. If we try to group two Taylor terms to avoid a division, we get from the Taylor series for arctan, where n denotes the n 'th Taylor term for arctan. If term n is even, we start with a minus sign; otherwise, +, and then we alternate the sign for each Taylor term as we advance:

$$\begin{aligned} \text{Two Taylor terms: } & -\frac{x^{n-1}}{2n-1} + \frac{x^{n+1}}{2n+1} \Rightarrow \\ & -\frac{(2n+1)x^{n-1} + (2n-1)x^{n+1}}{(2n-1)(2n+1)} \Rightarrow \\ & x^{n-1} \cdot \frac{-(2n+1) + (2n-1)x^2}{(2n-1)(2n+1)} \end{aligned}$$

If we group three Taylor terms, we get the following:

$$x^{n-1} \cdot \frac{-(2n+1)(2n+3) + (2n-1)(2n+3)x^2 - (2n-1)(2n+1)x^4}{(2n-1)(2n+1)(2n+3)}$$

We can continue grouping Taylor terms. From a practical point of view, grouping five Taylor terms is reasonable, as it doubles performance compared to not doing so.

Source for Arctan(x) with argument reduction & coefficient scaling

```
float_precision atan(const float_precision& x)
{
    const int group = 5;
    size_t precision = x.precision() + 2 + (size_t)ceil(log10(x.precision()));
    intmax_t k, i;
    size_t loopcnt = 1;
    float_precision r, atanx, v(x), vsq, terms;
    const float_precision c1(1);

    Automatically calculate the optimal reduction factor as a power of two
    k = 2 * (intmax_t)ceil(log(2)*log(precision));
    if (v.exponent() >= 0)
        ++k; // We only need one reduction to get x below 1
    Else // Avoid unnecessary argument reduction if v is small
        k += v.exponent();
}
```

Fast Trigonometric functions for Arbitrary Precision numbers

```
k = std::max((intmax_t)0, k);

// Adjust the precision
if (k > 0)
    precision += k / 4; ;
r.precision(precision);
atanx.precision(precision);
v.precision(precision);
vsq.precision(precision);
terms.precision(precision);

// Transform the solution to ArcTan(x)=2*ArcTan(x/(1+sqrt(1+x^2)))
for (i = k; i>0; --i )
    v = v / (c1 + sqrt(c1 + v.square()));

vsq = v.square();
r = v;
atanx = v;
if (group == 1)
{
    // Now iterate using Taylor expansion
    for (i = 3;; i += 2, ++loopcnt)
    {
        v *= vsq;
        v.change_sign();
        r = v / float_precision(i);
        if (atanx + r == atanx)
            break;
        atanx += r;
    }
}
else
{
    std::vector<float_precision> vn(group); // vn[0] is not used
    std::vector<float_precision> cn(group+1);

    for (i = 0; i < group; ++i)
    {
        cn[i].precision(precision); vn[i].precision(precision);
        if (i == 1) vn[1] = vsq;
        if (i > 1) vn[i] = vn[i - 1] * vsq;
    }
    cn[group].precision(precision);
    // Now iterate
    for (i = 3;; )
    {
        // Recalculate the coefficients
        intmax_t j, m;
        for (j = 0, cn[group]=c1; j < group; ++j)
        {
            cn[j] = c1;
            cn[group] *= float_precision(i + 2 * j);
            for (m = 0; m < group; ++m)
            {
                if (m == j) continue;
                cn[j] *= float_precision(i + 2 * m);
            }
            if ((i + 2 * j )/2 & 0x1)
```

Fast Trigonometric functions for Arbitrary Precision numbers

```

        cn[j].change_sign();
    }

    cn[group] = cn[group].inverse();
    // Summing adding from smallest to the largest number
    terms = vn[group - 1] * cn[group-1];
    for (j = group - 1; j >= 2; --j)
        terms += cn[j-1] * vn[j - 1];
    terms += cn[0];
    i += 2 * group;
    loopcnt += group;
    r *= vsq;
    terms *= r * cn[group];
    if (atanx + terms == atanx)
        break;
    atanx += terms;
    if (group > 1)
        r *= vn[group - 1]; // ajust r to last Taylor term
}

atanx.adjustExponent(k); // multiply with 2^k

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

```

Arctan(x) using the Euler method

Euler devised another series for arctan that supposedly converges more quickly than the Taylor series. The series can be expressed (alternatively) as:

$$\text{Arctan}(x) = \sum_{n=0}^{\infty} \frac{2^{2n}(n!)^2}{(2n+1)!} \frac{z^{2n+1}}{(1+x^2)^{n+1}} \quad (39)$$

For $x > 0.4$, fewer terms were required than the equivalent Taylor series, e.g., $\text{arctan}(0.6)$, which requires 25 terms to get the result. Using the Taylor series requires 30 terms. As x increased, it got worse. However, for $x < 0.4$, the Taylor and Euler series require approximately the same number of terms.

ArcTan(x)	Euler	Original	X Reduced	
x=		0.6	0.6	
No Reduction		0		
Terms	Term Value	Euler sum	Arctan(x)	Error
1	4.41E-01	0.441176470588235	0.441176470588235	9.92E-02
2	7.79E-02	0.519031141868512	0.519031141868512	2.14E-02
3	1.65E-02	0.535518013433747	0.535518013433747	4.90E-03
4	3.74E-03	0.539258732192246	0.539258732192246	1.16E-03

Fast Trigonometric functions for Arbitrary Precision numbers

5	8.80E-04	0.540138901311893	0.540138901311893	2.81E-04
6	2.12E-04	0.540350706715016	0.540350706715016	6.88E-05
7	5.18E-05	0.540402460071436	0.540402460071436	1.70E-05
8	1.28E-05	0.540415246194786	0.540415246194786	4.25E-06
9	3.19E-06	0.540418431664964	0.540418431664964	1.07E-06
10	7.99E-07	0.540419230498042	0.540419230498042	2.70E-07
11	2.01E-07	0.540419431884533	0.540419431884533	6.84E-08
12	5.10E-08	0.540419482874974	0.540419482874974	1.74E-08
13	1.30E-08	0.540419495832545	0.540419495832545	4.44E-09
14	3.30E-09	0.540419499135455	0.540419499135455	1.14E-09
15	8.44E-10	0.540419499979607	0.540419499979607	2.91E-10
16	2.16E-10	0.540419500195850	0.540419500195850	7.47E-11
17	5.55E-11	0.540419500251357	0.540419500251357	1.92E-11
18	1.43E-11	0.540419500265630	0.540419500265630	4.95E-12
19	3.68E-12	0.540419500269306	0.540419500269306	1.28E-12
20	9.48E-13	0.540419500270254	0.540419500270254	3.30E-13
21	2.45E-13	0.540419500270499	0.540419500270499	8.54E-14
22	6.33E-14	0.540419500270562	0.540419500270562	2.21E-14
23	1.64E-14	0.540419500270579	0.540419500270579	5.66E-15
24	4.24E-15	0.540419500270583	0.540419500270583	1.44E-15
25	1.10E-15	0.540419500270584	0.540419500270584	3.33E-16

As with the Taylor series, argument reduction greatly reduced the number of terms needed. For example, $\arctan(0.6)$ uses a reduction factor of four and requires only five terms (the same as the Taylor series).

ArcTan(x)	Euler	Original	X Reduced	
x=		0.6	0.033789069	
No Reduction		4		
Terms	Term Value	Euler sum	Arctan(x)	Error
1	3.38E-02	0.033750535945282	0.540008575124517	4.11E-04
2	2.57E-05	0.033776195334449	0.540419125351177	3.75E-07
3	2.34E-08	0.033776218744006	0.540419499904093	3.66E-10
4	2.29E-11	0.033776218766888	0.540419500270213	3.71E-13
5	2.32E-14	0.033776218766912	0.540419500270584	3.33E-16

Another drawback is that each Euler term requires more computational power than its corresponding Taylor series term. Overall, using the Euler version of $\arctan(x)$ over the Taylor series version is not worth it.

Arctan(x) using Arcsin()

It could be interesting to use the identity:

Fast Trigonometric functions for Arbitrary Precision numbers

$$\text{Arctan}(x) = \text{Arcsin}\left(\frac{x}{\sqrt{1+x^2}}\right) \quad (40)$$

Particularly if you want to reduce code size and reuse the existing `arcsin()` implementation. However, the performance is slightly slower (20%-30%) than using the Taylor series for `arctan()`.

Source for `Arctan(x)` using `Arcsin()`

```
float_precision atan_asin(const float_precision& x)
{
    size_t precision = x.precision()+2+(size_t)ceil(log10(x.precision()));
    float_precision atanx(x);
    const float_precision c1(1);

    atanx.precision(precision);
    atanx = testasin(atanx / sqrt(c1 + atanx.square()));
    // Round to same precision as argument and rounding mode
    atanx.mode(x.mode());
    atanx.precision(x.precision());
    return atanx;
}
```

Recommendation for calculating Arctan(x)

Based on the performance measure of the various `arctan(x)` methods, we recommend:

- The preferred method uses the Taylor series for `arctan(x)`, argument reduction, and coefficient scaling.
- `Arctan(x)` using the Euler series has no advantages over the Taylor series for argument < 0.4 . For argument $x > 0.4$, sticking with the Taylor series and using the recommended argument reduction and coefficient scaling for increased performance is more beneficial.
- `Arctan(x)` using `arcsin(x)` is a slower alternative that can be used to simplify and reduce code size.
- Only use a moderate number of argument reductions since it is very time-consuming to calculate. (Involving a division and a square root calculation).

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) HVE Fast Log() calculation for arbitrary precision; [Fast Log\(\) calculation for arbitrary precision \(hvks.com\)](#)
- 4) HVE Fast Exp() calculation for arbitrary precision; [Fast Exp\(\) calculation for arbitrary precision \(hvks.com\)](#)
- 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) Ronald W Potter. Arbitrary Precision Calculation of selected higher functions. ISBN #:978-1-312-59943-7

Appendix A: Rigorous Error Analysis for Range Reduction

This appendix presents the formal error bounds that justify the working-precision rules used throughout the main text.

All error bounds assume correctly rounded floating-point arithmetic with relative error bounded by one unit roundoff $\varepsilon = 2^{-p}$, where p denotes the working precision in bits.

Higher-order terms $O(\varepsilon^2)$ are neglected throughout.

Range reduction maps an arbitrary input argument x to an equivalent bounded interval by exploiting the periodicity of trigonometric functions. This appendix derives rigorous error bounds for the two range-reduction functions, `rangeReduction2PI()` and `rangeReductionPI()`, used by `sin(x)`, `cos(x)`, and `tan(x)`.

Decimal precision values are converted to binary precision using $\log_2(10)$ where required.

Source code for `rangeReduction2PI`

```
static void rangeReduction2PI(float_precision& v) {
    if (v.iszero())
        return;
    const size_t p_target = v.precision();

    // Calculate precision needed for range reduction
    // This handles arbitrarily large x
    size_t prec1 = p_target;
    if (v.exponent() > 1) {
        prec1 += (size_t)ceil(v.exponent() / log2(10));

        if (p_target < 40)
            prec1 += 5;
        prec1 += 5;
    }

    v.precision(prec1);
    // Quick check
    if (v <= float_precision(6.28318530717958647692))
        return;

    // Perform range reduction to [0, 2π)
    float_precision two_pi = _float_table(_PI, prec1);
    two_pi.adjustExponent(+1); // Multiply by 2
    if (v > two_pi) {
        float_precision r = v / two_pi;
        modf(r, &r);
        v -= r * two_pi;
    }

    if (v < float_precision(0))
        v += two_pi;
    // v is now in [0, 2π) at precision prec1
}
```

Source code for `rangeReductionPI`

```
static void rangeReductionPI(float_precision& v, bool& sign_flip) {
    sign_flip = false;

    // Add small margin to current precision
    size_t prec = v.precision();
    v.precision(prec);
    // Quick check - already in [0, π)?
    if (v <= float_precision(3.14159265358979323846))
        return;
    // Compute π
    float_precision pi = _float_table(_PI, prec);
```

Fast Trigonometric functions for Arbitrary Precision numbers

```
// Reduce to [0, π)
if (v > pi) {
    v -= pi;
    sign_flip = true;
}
}
```

A.1 Purpose and the Catastrophic Cancellation Problem

For a large argument such as $x = 10^{50}$, the naive approach to computing $x \bmod 2\pi$ subtracts two numbers of magnitude 10^{50} whose difference lies in $[0, 2\pi)$. When two numbers of magnitude 10^{50} are subtracted to produce a result of magnitude approximately 1, catastrophic cancellation occurs, and approximately 50 decimal digits of precision are lost in the subtraction alone.

As a concrete example, if π is computed to only 100 digits but $x = 10^{50}$, the error in $r \cdot 2\pi$ (where $r = \text{floor}(x / 2\pi)$) is approximately:

$$\text{error} \sim (10^{50} / (2\pi)) \cdot 2^{(-\text{precision_bits})} \sim 10^{50} \cdot 10^{(-100)} = 10^{(-50)}$$

This error of $10^{(-50)}$ is larger than the entire interval $[0, 2\pi) \sim 6.28$. The reduced angle would therefore be completely wrong. The solution is to compute π with precision that scales with the magnitude of x .

A.2 Notation

Throughout this appendix, the following notation is used:

x	: original input argument (can be arbitrarily large)
v	: reduced argument after range reduction
p_target	: target precision in decimal digits
$prec1$: working precision for range reduction in decimal digits
$prec1_bits$: working precision in bits = $prec1 \cdot \log_2(10)$
E	: binary exponent of x , where $x \sim 2^E$
eps	: unit roundoff = $2^{(-prec1_bits)}$
pi_exact	: true mathematical value of π
pi_comp	: computed value of π at precision $prec1$

Throughout this appendix, precision is sometimes expressed in decimal digits and sometimes in binary bits, depending on the derivation. Conversion between the two uses the relation

$$\text{precision_bits} = \text{precision_digits} \cdot \log_2(10).$$

Error bounds are derived in binary form because floating-point roundoff is naturally expressed in powers of two, while final precision requirements are reported in decimal digits for consistency with the arbitrary-precision interface.

Fast Trigonometric functions for Arbitrary Precision numbers

The binary exponent E satisfies $E = \text{floor}(\log_2(|x|))$. For $x = 10^k$ we have $E \sim k \cdot \log_2(10) \sim k \cdot 3.322$.

A.3 Error Analysis for rangeReduction2PI()

The algorithm computes $v = x - \text{floor}(x / (2\pi)) \cdot 2\pi$ in the following steps:

Step 1: Approximation of π

The computed value of π satisfies:

$$|\text{pi_comp} - \text{pi_exact}| \leq 2^{(-\text{prec1_bits})}$$

Step 2: Computation of 2π

Multiplication by 2 is exact in binary floating-point arithmetic because it simply increments the exponent by 1 with no rounding error. However, the approximation error in π doubles:

$$2 \cdot \text{pi_comp} = 2 \cdot \text{pi_comp} \quad (\text{exact multiplication, no rounding error})$$

$$|2 \cdot \text{pi_comp} - 2 \cdot \text{pi_exact}| = 2 \cdot |\text{pi_comp} - \text{pi_exact}| \leq 2^{(1 - \text{prec1_bits})}$$

This is a magnification of error, not a rounding error from the operation itself.

Step 3: Division $x / (2\pi)$

The error in the quotient $r = x / (2 \cdot \text{pi_comp})$ comes from two sources. First, using an approximate value of $2 \cdot \pi$ introduces:

$$|x / 2 \cdot \text{pi_comp} - x / 2 \cdot \text{pi_exact}| \sim (|x| / (2\pi))(\text{eps_2pi} / 2\pi)$$

where $\text{eps_2pi} = |2 \cdot \text{pi_comp} - 2 \cdot \text{pi_exact}| \leq 2^{(1 - \text{prec1_bits})}$. Second, rounding the division result adds at most:

$$|x / 2\pi| \cdot 2^{(-\text{prec1_bits})}$$

For large x , the second term dominates, giving:

$$|r_comp - r_exact| \sim |x| / (2\pi) \cdot 2^{(-\text{prec1_bits})}$$

Step 4: The floor() operation

The floor function returns the largest integer less than or equal to the argument, provided the argument fits within the working precision. For $|x| < 10^{300}$, this is satisfied comfortably.

Step 5: Multiplication $r \cdot 2\pi$

The product $r \cdot 2\pi_comp$ has error from two sources: the approximation error in 2π and the rounding of the product. Using $|r| \sim |x| / (2\pi)$ and $|r \cdot 2 \cdot \text{pi_comp}| \sim |x|$:

$$|r \cdot 2 \cdot \text{pi_comp} - r \cdot 2 \cdot \text{pi_exact}| \leq (|x| / (2\pi)) \cdot 2^{(1 - \text{prec1_bits})} + |x| \cdot 2^{(-\text{prec1_bits})}$$

$$\sim |x| \cdot 2^{(-\text{prec1_bits})} \cdot (1/\pi + 1)$$

$$\sim |x| \cdot 2^{(-\text{prec1_bits})} \cdot 1.318$$

Step 6: Subtraction $x - r \cdot 2\pi$

The final subtraction yields $v = x - r \cdot 2 \cdot \text{pi_comp}$. The result v lies in $[0, 2\pi)$, so the rounding error of this subtraction is at most $2\pi \cdot 2^{(-\text{prec1_bits})} \sim 6.28 \cdot 2^{(-\text{prec1_bits})}$, which is negligible compared to the propagated error from the multiplication. The dominant error term is therefore:

Fast Trigonometric functions for Arbitrary Precision numbers

$$|v_{\text{comp}} - v_{\text{exact}}| \sim 1.32 \cdot |x| \cdot 2^{-(\text{prec1_bits})}$$

A.4 Required Precision for rangeReduction2PI()

To guarantee that the error in the reduced angle is negligible compared to the target precision, we require:

$$1.32 \cdot |x| \cdot 2^{-(\text{prec1_bits})} \leq 2^{-(\text{p_target_bits})}$$

Taking logarithms base-2 and using $\log_2(|x|) \sim E$:

$$\text{prec1_bits} \geq \text{p_target_bits} + E + \log_2(1.32)$$

$$\text{prec1_bits} \geq \text{p_target_bits} + E + 0.4$$

Converting to decimal digits by dividing by $\log_2(10) \sim 3.322$:

$$\text{prec1} \geq \text{p_target} + \text{ceil}(E / \log_2(10)) + 1 \text{ (minimum theoretical requirement)}$$

Range reduction requires working precision that increases linearly with the argument's binary exponent. Consequently, argument reduction for transcendental functions cannot be performed at fixed precision when $|x|$ is unbounded.

A.5 Guard Digits Justification

The analysis above captures only the dominant error terms. In practice, additional guard digits are needed to account for roundoff accumulation and second-order terms. Each of the approximately 5 floating-point operations contributes at most one ulp of rounding error. This adds $\text{ceil}(\log_2(5)) \sim 3$ bits, or about 0.9 decimal digits.

The analysis neglected smaller terms such as the error in computing $1 / (2\pi)$ and cross-product error terms. These add another 0.5 to 1 decimal digit.

Low-precision edge cases, when p_target is small (below 40 digits), the relative impact of each rounding error is larger, and boundary effects near representable numbers become more significant.

Combining these contributions, the recommended precision formula with guard digits is:

if $E > 1$:

$$\text{prec1} = \text{p_target} + \text{ceil}(E / \log_2(10)) + 5 \quad // \text{p_target} \geq 40$$

$$\text{prec1} = \text{p_target} + \text{ceil}(E / \log_2(10)) + 10 \quad // \text{p_target} < 40$$

This has been empirically validated: with $\text{p_target} = 20$ and $x = 10^{50}$ ($E \sim 166$), the formula gives $\text{prec1} = 20 + 50 + 10 = 80$ digits, which produces a result accurate to better than $10^{(-20)}$ as confirmed by comparison with a 250-digit Wolfram Alpha reference value.

A.6 Error Analysis for rangeReductionPI()

After `rangeReduction2PI()`, the value v lies in $[0, 2\pi)$ and therefore has magnitude at most 6.28. Reducing to $[0, \pi)$ using $v = v - \pi$ (when $v > \pi$) involves only numbers of moderate size. The subtraction error satisfies:

$$\text{error} \leq \pi \cdot 2^{-(\text{prec_bits})} + 2^{-(\text{prec_bits})} \sim 4.15 \cdot 2^{-(\text{prec_bits})}$$

Fast Trigonometric functions for Arbitrary Precision numbers

For this to be negligible compared to the target precision:

$$\text{prec_bits} \geq \text{p_target_bits} + \log_2(4.15) \approx \text{p_target_bits} + 2.05$$

In decimal: $\text{prec} \geq \text{p_target} + 1$ digit minimum. A safety margin of 5 digits is used in practice:

$$\text{prec} = \text{v.precision()} + 5 \quad // \text{v.precision()} \text{ is already prec1 from stage 1}$$

No additional precision scaling of x 's original magnitude is required here, because that large magnitude has already been eliminated by `rangeReduction2PI()`. This is why `rangeReductionPI()` is much simpler than `rangeReduction2PI()`.

Appendix B: Rigorous Error Analysis for $\sin(x)$

After range reduction to v in $[0, \pi]$, the computation of $\sin(v)$ uses three sub-stages: argument reduction by trisection, dividing v by 3^k , Taylor series evaluation of \sin at the reduced argument, and reverse reduction applying the trisection identity k times. This appendix derives rigorous error bounds for each sub-stage and the total error, and presents the working-precision formula for Stage 2.

B.1 Notation for Stage 2

<code>p_target</code>	: target precision in decimal digits
<code>p_target_bits</code>	: target precision in bits = $\text{p_target} \cdot \log_2(10)$
<code>prec2</code>	: working precision for Stage 2 in decimal digits
<code>prec2_bits</code>	: working precision in bits
<code>k</code>	: trisection reduction count
<code>v</code>	: reduced argument in $[0, \pi]$ from Stage 1
<code>v_red</code>	: $v / 3^k$ after argument reduction
<code>eps2</code>	: unit roundoff at $\text{prec2} = 2^{(-\text{prec2_bits})}$

B.2 Choosing the Reduction Factor k

The reduction factor k is chosen so that $|v_red| = |v| / 3^k$ is small enough for the Taylor series to converge rapidly. The formula used is:

$$k_base = 8 \cdot \text{ceil}(\log(2) \cdot \log(\text{p_target}))$$

$$k = \text{ceil}(2 \cdot k_base / 3)$$

$$k = \text{max}(0, k + \text{v.exponent}()) \quad // \text{adjust for actual magnitude of } v$$

The adjustment by `v.exponent()` handles the case where v is already small (for example, $v = 0.001$ requires fewer reductions than $v = 3$). The `max(0, ...)` prevents negative k values.

B.3 Sub-stage 2A: Argument Reduction Error

The argument reduction computes $v_red = v / 3^k$. With exact arithmetic, this is trivial, but in floating-point, the division introduces a relative rounding error. The error in v_red satisfies:

$$\begin{aligned} |v_red_comp - v_red_exact| &\leq |v_red_exact| \cdot eps2 + small_correction \\ &\sim (|v| / 3^k) \cdot eps2 \end{aligned}$$

Since $|v| \leq \pi$ and \sin is Lipschitz with constant 1, this argument error propagates to an error in $\sin(v_red)$ of:

$$eps_2A \leq \pi \cdot 3^{(-k)} \cdot eps2 \sim 3.15 \cdot 3^{(-k)} \cdot eps2$$

B.4 Sub-stage 2B: Taylor Series Error

The Taylor series for \sin is:

$$\sin(x) = x - x^3/3! + x^5/5! - x^7/7! + \dots$$

The adaptive stopping criterion terminates iteration when $\sin x + term == \sin x$ in floating-point arithmetic, which occurs when $|term| < ulp(\sin x)$. At this point, the truncation error satisfies:

$$eps_truncation \leq |\sin x| \cdot eps2 \sim |v_red| \cdot eps2$$

For $|v_red| < 0.1$ (which the choice of k guarantees), this is at most $0.1 \cdot eps2$.

Roundoff accumulation over m Taylor iterations (each contributing approximately 5 floating-point operations) adds:

$$eps_roundoff \leq 5 \cdot m \cdot eps2 \sim (5 \cdot p_target / 3) \cdot eps2$$

Because $m \sim p_target / 3$, iterations are typically required. The total Taylor series error is therefore:

$$eps_2B \sim 2 \cdot p_target \cdot |v_red| \cdot eps2$$

B.5 Sub-stage 2C: Reverse Reduction Error

The reverse reduction applies the trisection identity k times:

$$\sin(3y) = 3\sin(y) - 4\sin^3(y)$$

Let $f(s) = 3 \cdot s - 4 \cdot s^3$. The derivative is:

$$f'(s) = 3 - 12s^2$$

For small s (just after Taylor evaluation, $|s| \sim |v_red| < 0.1$):

$$|f'(s)| \sim 3 - 12(0.1)^2 = 2.88 \sim 3$$

Each reverse reduction step, therefore, amplifies the existing error bounded by 3 for $|s| \leq 0.1$

and adds its own rounding error. After k iterations:

$$eps_2C \sim 3^k \cdot eps_2B + eps_iter \cdot \sum_{i=1}^k 3^i$$

Fast Trigonometric functions for Arbitrary Precision numbers

$$\sim 3^k \cdot \text{eps_2B} + 3^k \cdot 5 \cdot \text{eps2}$$

$$\sim 2 \cdot \text{p_target} \cdot 3^k \cdot \text{eps2}$$

This 3^k amplification is the dominant error term and drives the working precision requirement.

B.6 Total Error for Stage 2

Combining all sub-stages (with `eps_2C` dominating):

$$\begin{aligned} \text{eps_total} &\sim 2 \cdot \text{p_target} \cdot 3^k \cdot \text{eps2} \\ &= 2 \cdot \text{p_target} \cdot 3^k \cdot 2^{(-\text{prec2_bits})} \end{aligned}$$

This shows that reverse reconstruction, not Taylor truncation, dominates the precision requirement. Increasing Taylor accuracy alone cannot compensate for insufficient working precision during reconstruction.

B.7 Required Working Precision prec2

Setting `eps_total` $\leq 2^{(-\text{p_target_bits})}$ and solving for `prec2_bits`:

$$\begin{aligned} 2 \cdot \text{p_target} \cdot 3^k \cdot 2^{(-\text{prec2_bits})} &\leq 2^{(-\text{p_target_bits})} \\ \text{prec2_bits} &\geq \text{p_target_bits} + \log_2(2 \cdot \text{p_target}) + k \cdot \log_2(3) \\ \text{prec2_bits} &\geq \text{p_target_bits} + k \cdot 1.585 + \log_2(\text{p_target}) + 1 \end{aligned}$$

Converting to decimal digits and adding 20 guard bits for safety:

$$\begin{aligned} \text{extra_bits} &= \text{ceil}(k \cdot 1.585) && // 3^k \text{ amplification} \\ &+ \text{ceil}(\log_2(\text{p_target} / 3.0 \cdot 5.0)) && // \text{roundoff accumulation} \\ &+ 20 && // \text{guard bits} \\ \text{prec2_bits} &= \text{p_target_bits} + \text{extra_bits} \\ \text{prec2} &= \text{ceil}(\text{prec2_bits} / \log_2(10)) + 1 \end{aligned}$$

This is exactly the formula implemented in the `sin(x)` source code. The guard bits of 20 have been validated empirically across precision levels from 20 to 200 digits and for arguments ranging from 0.5 to 10^{50} .

B.8 Precision Reduction Between Stages

After Stage 1, `v` has precision `prec1`, which may be much larger than `prec2` (for example `prec1 = 160` and `prec2 = 115` for `p_target = 100` with `x = 1050`). The statement `v.precision(prec2)` before the Taylor series is therefore not just a round-down but an actual reduction that saves substantial computation time. The two-stage strategy, where Stage 1 uses high precision only for range reduction and Stage 2 uses lower precision tailored to Taylor convergence, is critical to performance.

Specifically, since Taylor series computation cost scales roughly as $O(\text{prec2}^2)$ per iteration and there are $O(\text{p_target})$ iterations, operating at `prec2` instead of `prec1` throughout would increase cost by a factor of approximately $(\text{prec1} / \text{prec2})^2$, which for the example above is $(160 / 115)^2 \sim 1.94$, nearly doubling the computation time.

Fast Trigonometric functions for Arbitrary Precision numbers

Because multiplication costs dominate arbitrary-precision arithmetic, reducing working precision from prec_1 to prec_2 roughly reduces total complexity by a factor of $(\text{prec}_1/\text{prec}_2)^2$, explaining the observed performance improvement.

Appendix C: Rigorous Error Analysis for $\cos(x)$

The computation of $\cos(x)$ uses the identity $\cos(x) = \sqrt{1 - \sin^2(x)}$ combined with a threshold check near $\pi/2$, where this formula loses accuracy. This appendix analyses the error in this approach and shows when the alternative formula $\cos(x) = \sin(\pi/2 - x)$ must be used instead.

C.1 Overview of the $\cos(x)$ Implementation

After range reduction to v in $[0, \pi]$ using the same `rangeReduction2PI()` and `rangeReductionPI()` functions described in Appendix A (and with the same error bounds), $\cos(x)$ is computed as follows:

For v NOT near $\pi/2$ (the normal case): compute $\sin(v)$ at working precision $p_{\text{work}} = p_{\text{target}} + 2$, then form $\cos(v) = \sqrt{1 - \sin^2(v)}$ and determine the sign from the quadrant. The sign is positive for v in $[0, \pi/2)$ and negative for v in $(\pi/2, \pi]$.

For v NEAR $\pi/2$ (within 0.1 radians): use $\cos(v) = \sin(\pi/2 - v)$ instead, which avoids the accuracy loss in the square root formula. The threshold check uses double-precision arithmetic because only the coarse position relative to $\pi/2$ matters.

C.2 Error Analysis for the Normal Case: $\cos(x) = \sqrt{1 - \sin^2(x)}$

Let $s = \sin(v)$ be the computed sine with error δ_s satisfying $|\delta_s| \leq \text{ulp}(s)$ at precision p_{work} . The error propagates through three operations.

Operation 1: squaring $\sin(v)$

$$(\sin(v) + \delta_s)^2 = \sin^2(v) + 2 \cdot \sin(v) \cdot \delta_s + \delta_s^2$$

For $|\sin(v)| \leq 1$ the error in $\sin^2(v)$ satisfies:

$$\delta_{\text{sq}} \leq 2 \cdot |\sin(v)| \cdot |\delta_s| + \text{ulp}(\sin^2(v)) \leq 3 \cdot \text{ulp}$$

Operation 2: subtraction $1 - \sin^2(v)$

The result equals $\cos^2(v)$. The subtraction adds at most $\text{ulp}(\cos^2(v))$, giving:

$$\delta_{\text{sub}} \leq 3 \cdot \text{ulp} + \text{ulp}(\cos^2(v)) \approx 4 \cdot \text{ulp}$$

Operation 3: square root

For a value $y = \cos^2(v)$ with error δ_y , the square root satisfies:

$$\begin{aligned} \sqrt{y + \delta_y} &\approx \sqrt{y} + \delta_y / (2 \cdot \sqrt{y}) \\ &= |\cos(v)| + \delta_y / (2 \cdot |\cos(v)|) \end{aligned}$$

The error in $\cos(v)$ is therefore:

$$\begin{aligned} \delta_{\text{cos}} &\approx \delta_{\text{sub}} / (2 \cdot |\cos(v)|) + \text{ulp}(\cos(v)) \\ &\approx 4 \cdot \text{ulp} / (2 \cdot |\cos(v)|) + \text{ulp} \\ &= 2 \cdot \text{ulp} / |\cos(v)| + \text{ulp} \end{aligned}$$

Fast Trigonometric functions for Arbitrary Precision numbers

For $|\cos(v)|$ well away from zero, a small multiple of ulp bounds this, and $p_work = p_target + 2$ provides adequate precision. However, when $|\cos(v)|$ is small (v near $\pi/2$), the factor $1 / |\cos(v)|$ can be large and the error blows up.

This behaviour reflects the condition number of the square-root formulation, which grows proportionally to $1 / |\cos(v)|$ near $\pi/2$. The loss of accuracy, therefore, arises from the problem conditioning rather than from deficiencies of the algorithm itself.

C.3 Critical Case: v Near $\pi/2$

When $v = \pi/2 - \delta$ for small $\delta > 0$:

$$|\cos(v)| = |\cos(\pi/2 - \delta)| = \sin(\delta) \sim \delta$$

The relative error in the computed $\cos(v)$ satisfies:

$$\delta_{\cos} / |\cos(v)| \sim 2 \cdot \text{ulp} / \cos^2(v) = 2 \cdot \text{ulp} / \delta^2$$

For this to be acceptable (less than $2^{-(p_target_bits)}$), we need:

$$2 \cdot \text{ulp} / \delta^2 \leq 2^{-(p_target_bits)}$$

$$\text{ulp} \leq \delta^2 \cdot 2^{(p_target_bits - 1)}$$

$$2^{(p_work_bits)} \leq \delta^2 \cdot 2^{(p_target_bits - 1)}$$

$$p_work_bits \geq p_target_bits + 1 - 2 \cdot \log_2(\delta)$$

For $\delta = 0.1$ (the threshold): $1 - 2 \cdot \log_2(0.1) \sim 1 + 6.64 \sim 8$ extra bits ~ 2.4 extra digits. With $p_work = p_target + 2$ this is comfortably satisfied.

For $\delta < 0.1$ (which triggers the alternative formula): $p_work = p_target + 2$ would be insufficient. For example, $\delta = 0.001$ requires $1 - 2 \cdot \log_2(0.001) \sim 1 + 19.9 \sim 21$ extra bits ~ 6.3 extra digits. This is why the threshold check at 0.1 radians is necessary.

C.4 Alternative Formula: $\cos(x) = \sin(\pi/2 - x)$

When v is within 0.1 radians of $\pi/2$, the implementation uses:

$$\text{arg} = \pi/2 - v$$

$$\cos(v) = \sin(\text{arg})$$

The argument $\text{arg} = \pi/2 - v$ satisfies $|\text{arg}| < 0.1$, which means $\sin(\text{arg}) \sim \text{arg}$ is small, and the Taylor series for \sin converges very rapidly. The error analysis is identical to that in Appendix B, and $p_work = p_target + 2$ is sufficient.

The key accuracy advantage is that this avoids the catastrophic cancellation in $1 - \sin^2(v)$ when $\sin^2(v) \sim 1$, and avoids the subsequent error amplification in the square root. Instead, the small difference $\pi/2 - v$ is computed directly and fed to $\sin()$, which handles it accurately.

C.5 Sign Determination

After range reduction, v lies in $[0, \pi]$. Within this interval $\cos(v)$ is:

$$\cos(v) \geq 0 \text{ for } v \text{ in } [0, \pi/2]$$

$$\cos(v) \leq 0 \text{ for } v \text{ in } [\pi/2, \pi]$$

Fast Trigonometric functions for Arbitrary Precision numbers

The sign is therefore determined by comparing v to $\pi/2$ at working precision. In the near- $\pi/2$ case, the $\sin(\pi/2 - v)$ formula naturally produces the correct sign because $\arg = \pi/2 - v$ is positive for $v < \pi/2$ and negative for $v > \pi/2$.

Additionally, if `rangeReductionPI()` set `sign_flip = true` (meaning the original reduced angle was in $[\pi, 2\cdot\pi)$), the result is negated because $\cos(v + \pi) = -\cos(v)$.

C.6 Required Working Precision for $\cos(x)$

Combining the analysis above:

Normal case ($|\cos(v)| > 0.1$, i.e., v not within 0.1 rad of $\pi/2$): The error in $\sqrt{1 - \sin^2(v)}$ is bounded by approximately $20 \cdot \text{ulp}$, so `p_work = p_target + 2` (giving roughly 6 extra bits) is sufficient.

Near $\pi/2$ case ($|\arg| < 0.1$): The function $\sin(\arg)$ is called with $|\arg| < 0.1$, so the `sin()` internal precision management (Stage 2 at `prec2`) handles the accuracy requirements. No additional precision beyond `p_target + 2` is needed for the outer `cos()` wrapper.

In summary, the formula `p_work = p_target + 2` is adequate for $\cos(x)$ provided the threshold-based switching between the two formulas is in place. Without the threshold check, values close to $\pi/2$ would require `p_work = p_target + 2 \cdot |\log_{10}(|\cos(v)|)| + 5`, which could be arbitrarily large.

C.7 Comparison with Direct $\cos(x)$ via Taylor Series

An alternative implementation would compute $\cos(x)$ directly via its Taylor series $\cos(x) = 1 - x^2/2! + x^4/4! - \dots$, using a cosine-specific trisection identity $\cos(3y) = 4\cos^3(y) - 3\cos(y)$ for the reverse reduction step.

The derivative of $f(c) = 4\cdot c^3 - 3\cdot c$ with respect to c is $f'(c) = 12\cdot c^2 - 3$. For $c = \cos(y)$ near 1 (small y), this gives $f'(1) = 9$, meaning each reverse reduction step amplifies errors by a factor of approximately 9 rather than 3 as for sine. The required working precision would therefore satisfy:

$$\begin{aligned} \text{prec2_bits} &\geq \text{p_target_bits} + k \cdot \log_2(9) + \dots \\ &\geq \text{p_target_bits} + k \cdot 3.17 + \dots \end{aligned}$$

compared to `p_target_bits + k \cdot 1.585` for $\sin(x)$. This makes a direct $\cos(x)$ Taylor implementation approximately twice as expensive in working precision (and therefore roughly 4 times slower for the Taylor series computation) compared to the $\sin(x)$ -based implementation used here.

The $\sin(x)$ -based implementation $\cos(x) = \pm \sqrt{1 - \sin^2(x)}$ is therefore both simpler and more efficient, provided the threshold switching near $\pi/2$ is correctly implemented. Direct cosine reconstruction amplifies rounding errors approximately three times faster than sine reconstruction because the reverse trisection identity has a derivative magnitude of 9 instead of 3. Consequently, computing $\cos(x)$ indirectly via $\sin(x)$ is asymptotically more efficient for arbitrary-precision arithmetic.