Fast Trigonometric functions for Arbitrary Precision numbers.

By Henrik Vestermark (hve@hvks.com)

Abstract:

This is a follow-up to a previous paper that describes the math behind arbitrary precision numbers, see [7]. First of all the original paper was written back in 2013 and quite a few things had happens since then, secondly, I came across some other interesting methods to do the calculation of the trigonometric function. The paper describes in more detail how to do sin(x), cos(x), tan(x), and the inverse arcsin(x), arccos(x), and arctan(x) calculation with arbitrary precision and outline some traditional methods but also introduce an improved version that makes the calculation 5-20 times faster than the original method used in the author own arbitrary precision math packages.

Introduction:

When implementing an arbitrary precision math packages you would use the standard Taylor series calculation for calculating sin(x), cos(x), arcsin(x), and arctan(x) for arbitrary precisions, while 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 improved the performance of the method. We will discuss the various method for calculating trigonometric functions and elaborate on the techniques like clever argument reductions and coefficient scaling to improve the performance of the method. Furthermore, we will analyze some Newton methods for calculating some of the trigonometric functions.

As usual, we will show the actual C++ source for the computation using the author's own arbitrary precision Math library, see [1].

This paper is part of a series of arbitrary precision papers describing methods, implementation details, and optimization techniques. These papers can be found on my website at www.hvks.com/Numerical/papers.html and are listed below:

1. Fast Computation of Math Constants in arbitrary precision. HVE Fast Gamma, Beta, Error, and Zeta functions for arbitrary precision.
2. Fast Gamma, Beta, Error, and Zeta functions for arbitrary precision. HVE Fast Gamma, Beta, Error, and Zeta functions for arbitrary precision.
3. Fast Square Root & Inverse calculation for arbitrary precision math. HVE Fast Square Root & inverse calculation for arbitrary precision
4. Fast Exponential calculation for arbitrary precision math. HVE Fast Exp() calculation for arbitrary precision
5. Fast logarithm calculation for arbitrary precision math. HVE Fast Log() calculation for arbitrary precision
6. Practical implementation of Spigot Algorithms for Transcendental Constants. Practical implementation of Spigot Algorithms for transcendental constants

7. Practical implementation of π algorithms. HVE Practical implementation of PI Algorithms
8. Fast Trigonometric function for arbitrary precision. HVE Fast Trigonometric calculation for arbitrary precision
9. Fast Hyperbolic functions for arbitrary precision. HVE Fast Hyperbolic calculation for arbitrary precision
10. Fast conversion from arbitrary precision number to a string. HVE Fast conversion from arbitrary precision to string
11. Fast conversion from a decimal string to an arbitrary precision number. HVE Fast conversion from string to arbitrary precision

Change log

15-January 2023. Updated some inconsistency in the "Cos(x) using sin(x)" section and corrected the recommendation in the same section. This paper is part of a series of documents within the field of arbitrary precision.
26-January 2023. Cleaning up the grammar.

Contents

The Arbitrary precision library

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Internal format for float_precision variables

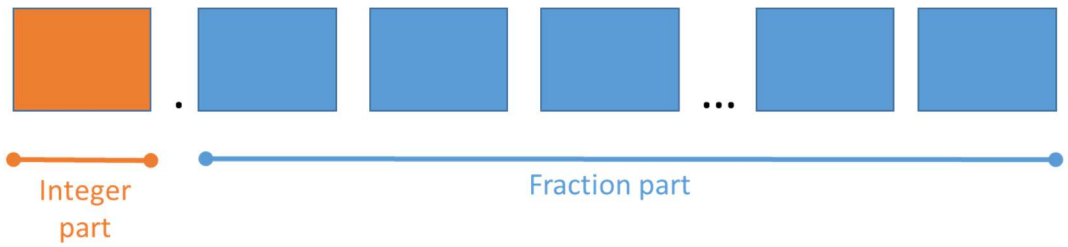
For the internal layout of the arbitrary precision number, we are using the STL vector library declared as:

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

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

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

The Binary format mBinary



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

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

Normalized numbers

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

For more details see [1].

Trigonometric functions

There are quite a few ways you can calculate trigonometric functions with arbitrary precision. Traditional the Taylor series expansion has been used, however in 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 most common one for arbitrary precision libraries is the standard Taylor series expansion method.

Sin(x) using Taylor Series

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

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

Where the similar to the sine hyperbolic functions which the Taylor series is:

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

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 easily reduce any argument > 2π 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..π using the identity:

sin(x)=-sin(x-π) for x≥π.

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

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

If π is 'expensive' to calculate (which is usually the case with arbitrary precision) we can omit step 3 since we have a different way to obtain the same thing by just 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 is the fewer terms we would need.

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

After the Taylor series has converged, we use the trisection identity reverse *k* number of times to find our result for sin(x).

To see how this algorithm works let us find the sin(0.7). After the 8th Taylor term, the error is zero 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
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. E.g. for two reductions you get the same result 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	0.07777777778	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 using argument reduction can seriously cut down the number of Taylor terms needed and thereby increase the performance of calculating sin(x).

The issue with arbitrary precision

The Number of Taylor terms to reach a result does not seem so bad at a first glance. In the previous examples, we were only using approx. 15 decimal digits. However, when we are dealing with higher precisions e.g. 1,000 digits, 10,000, or even 100,000 digits we suddenly have to perform a lot more Taylor terms to find our result. In Yacas [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 $x=10^M$:

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

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

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								
10^1	(11)	88	319	1,948	14,022	109,512	898,358	7,615,327
10^0	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. E.g., 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 the argument of 10^{-9} in magnitude. For a precision of 100,000 digits, the factor is only around three and for 100M digits, it is around 2.2. The lesson here is that argument reduction is more efficient for smaller 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 not just the magnitude. It usually gives a little bit less amount of needed Taylor terms. This formula can be quite useful:

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

Finding a reasonable reduction factor.

As can be seen in the above table a higher reduction factor greatly improved the performance. However, how many times reduction is adequate? 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

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

At a starting point of x=1, you get for P=1,00digits that the needed Taylor term is 24. Doing three reductions you get $x=1/3^3 = 0.037$. Using the above formula we expect we would only need 14 Taylor terms. Each Taylor term requires one addition/subtraction, 1 division, and one multiplication which yields a total saving of 10 subtraction, 10 division, and 10 multiplication. Compared to three reductions on the front end are three divisions and on the back end 3 subtraction and nine multiplication a total saving of seven subtraction/addition, one multiplication, and seven division. Since division is a magnitude slower than multiplication and addition/subtraction, we can give a rough saving equivalent with seven divisions. For higher precisions, the saving becomes larger. We automatically calculate the reduction factor as:

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

for higher precisions, and then we adjusted the magnitude of x. After Step 2, we know that x is in the range of $[0..\pi]$ this is equivalent that the exponent of our number (in base 2) being in the range $[-\infty..1]$. We add the exponent to the reduction factor. This has the effect that our reduction factor gets smaller if x is very small preventing us from doing unnecessary reductions. If x is very small, the reduction factor is negative and we simply

do not perform any argument reductions at all. E.g. for P=100 you get 24 and for P=10,000 you get 40. To compensate for the inaccuracy when adding the front and back end calculation, we increase the precision by a quarter of the k factor. The increased precision only generates a small performance penalty compared to the extra saving in Taylor's terms of the overall calculation.

Guard Digits

When summarizing a Taylor series as sin(x) you need quite a lot of summarizing and that will produce round-off errors.

For our sin (x) function, we use a simple guard digits calculation that we add

2+ceil(log10(precision)) as extra guard digits as the working precision.

Further Improvement of the methods?

There is not a lot of things you can do to improve the sin(x) algorithm. However, 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 referred to as 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:

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

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

If the n'th term is not the one starting with the minus sign you can simply 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 have replaced one division for two multiplication. The (n+1)(n+2) can be done using a 32-bit or 64-bit integer since you never get to do so many Taylor terms in real life. There is no need to stop at just grouping two terms together you can do that for three terms or more:

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

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

It is very easy to determine when we need to start with a negative sign by just testing if n'th term divided by 2 is an odd number (start with a minus sign) or an even number starting with plus sign and then alternative the sign thereafter.

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

```
float_precision sin(const float_precision& x, const int klimit = 16, int group =
1)
	{
	const int group = 5;
	size_t precision = x.precision() + 2 + (size_t)ceil(log10(x.precision()));
	intmax_t k;
	uintmax_t i;
	int sign;
	uintmax_t loopcnt = 1;
	float_precision r, sinx, v(x), vsq, terms;
	const float_precision c3(3), c4(4);

	// Check for argument reduction and increase precision if necessary
	// Automatically calculate optimal reduction factor as a power of two
	k = 8 * (intmax_t)ceil(log(2)*log(precision));

	// Now use the trisection identity sin(3x)=sin(x)(3+4Sin^2(x))
	// until the argument has been reduced 2/3*k times.
	// Converting power of 2 to power of 3.
	k = (intmax_t)ceil(2.0*k / 3);
	precision += k / 4;
	r.precision(precision);
	sinx.precision(precision);
	v.precision(precision);
	vsq.precision(precision);
	terms.precision(precision);

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

	// Check that argument is larger than 2*PI and reduce it if needed
	// to the range [0..2*PI].
	// No need for high precision. We just need to figure out if we need
	// to Calculate PI with a higher precision
	if (v > float_precision(2 * 3.14159265))
```

```
            {
            // Reduce argument to between 0..2PI
            sinx = _float_table(_PI, precision);
            sinx.adjustExponent(+1);  // same as sinx*= c2;
            if (abs(v) > sinx)
                  {
                  r = v / sinx;
                  (void)modf(r, &r);
                  v -= r * sinx;
                  }
            if (v < float_precision(0))
                  v += sinx;
            }

      // 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 (v > float_precision(3.14159265))
            {
            if(sinx.iszero())    // PI not call before. Then increase reduction
                                 // factor with one reducing sinx from interval
                                 // [3.14..6.28] to max [1.05..2.10].
                  {
                  ++k;
                  }
            else
                  {
                  sinx = _float_table(_PI, precision); // We don't need to worry
that we called it a second time since it will be cached from the first calculation
                  if (v > sinx)
                        {
                        v -= sinx;
                        sign *= -1;  // Change sign
                        }
                  }
            }

      // Adjust k for the final value of v when v is small (less than 1).
      // We know it is in the interval between [0...PI]
      // This indicates that the exponent is in the range [-inf..1]
      // Avoid unnecessary argument reduction if v is small
      k += v.exponent();
      k = std::max((intmax_t)0, k);

      // Now use the trisection identity sin(3x)=3*sin(x)-4*sin(x)^3
      // k times k.  Where k is the number of reduction factors 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 = v;
      sinx = v;

      if (group == 1)
            {
```

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

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

		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;
			}
		// Now iterate
		for (i = 3; ; )
			{
			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), precision);
					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), precision);
					}
				}

			cn[0] = abs(cn[0]).inverse();
			// Summing adding from smallest to the 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];     // ajust r to last Taylor term
                }
        }

    // reverse argument reduction
    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();
    return sinx;
    }
```

Recommendation for calculating sin(x)

Based on the performance measure of the various sin(x) methods recommend:

- Always use the cyclic and symmetry rules to reduce the x to the range $[0.\pi]$
- It is unnecessary to reduce it down to the range $[0\ldots\frac{\pi}{2}]$ 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

Cos(x):

For cos(x) we again use a Taylor series until any additional addition does not change the result for the given precision of the number.

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

We can use the equivalent four steps procedure for cos(x), mapping it into the interval $[0\ldots\frac{\pi}{2}]$.

Step 1: We notice that cos(x) is cyclic with a period of 2π so we can easily reduce any argument > 2π 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..π using the identity:
cos(2π-x)=cos(x) for x≥π.

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

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

If π is 'expensive' to calculate (which is usually the case with arbitrary precision) we can omit step 3 since we have a different way to obtain the same thing by just 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 x is the fewer terms we would need. We could also use the double-angle identity:

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

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

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

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

To see how this algorithm works let us find the cos(0.7). After the 8th Taylor term, the error is zero 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**
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. E.g. for two reductions you get the same result 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 using argument reduction can seriously cut down the number of Taylor terms needed and thereby increase the performance in calculating cos(x).

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

Although many of the same arguments used in the calculation of sin(x) also apply for cos(x), including aggressive use of argument reduction, coefficients scaling, etc. We have to be careful how aggressive our argument reduction can be.

Cos(x) using double angle reduction

Argument reduction reduces x to a much smaller value that is much more sensitive to round-off errors for cos(x) than its counterpart for sin(x). It is, therefore, better to use the double-angle formula:

$$\cos(2x) = 2cos^2(x) - 1 \qquad (8)$$

Alternatively, even better written as:

$$\cos(2x) = 2(1 - \cos(x))^2 - 4(1 - \cos(x)) + 1 \qquad (9)$$

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

$$k = 2\lceil \ln(2) * \ln(P) \rceil \quad (10)$$

for higher precisions, and then we made adjustments for the magnitude of x. After Step 2, we know that x is in the range of [0...π] this is equivalent that the exponent of our number (in base 2) being in the range [-∞...1]. We add the exponent to the reduction factor. This has the effect that our reduction factor gets smaller if x is very small preventing us from doing unnecessary reductions. If x is very small, the reduction factor is negative and we simply do not perform any argument reductions at all.

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

```
float_precision cos(const float_precision& x)
	{
	const int group = 5;
	size_t precision = x.precision() + 2 + (size_t)ceil(log10(x.precision()));
	intmax_t k, i;
	uintmax_t loopcnt = 1;
	float_precision r, cosx, v(x), vsq, terms;
	const float_precision c1(1), c2(2), c4(4);

	// Check for argument reduction and increase precision if necessary
	// Automatically calculate optimal reduction factor as a power of two
	k = 2 * (intmax_t)ceil(log(2)*log(precision));

	precision += k;
	r.precision(precision);
	cosx.precision(precision);
	v.precision(precision);
	vsq.precision(precision);
	terms.precision(precision);

	// 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, precision);
		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, precision);
		if (v > r)
			v = r * c2 - v;		// cos(x)=cos(2PI - x) for x >= PI
		}

	// Adjust k for the final value of v when v is small (less than 1).
	// We know it is in the interval between [0...PI]
	// This indicates that the exponent is in the range [-inf..1]
	// Avoid unnecessary argument reduction if v is small
	k += v.exponent();
	k = std::max((intmax_t)0, k);

	// Now use the double identity cos(2x)=2cos(x)^2-1
	// k times k.  Where k is the number of reduction factor based
	// on the needed precision of the argument.
	v.adjustExponent(-k);  // Divide with 2^k
	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));;
			r.change_sign();
			if (cosx + r == cosx)
				break;
			cosx += r;
			}
		}
	else
		{
		std::vector<float_precision> vn(group);  // vn[0] is not used
		std::vector<float_precision> cn(group);  //

		for (i = 0; i < group; ++i)
			{
			cn[i].precision(precision); vn[i].precision(precision);
			if (i == 1) vn[1] = vsq;
			if (i > 1) vn[i] = vn[i - 1] * vsq;
			}
		// Now iterate
		for (i = 2; ; )
			{
			// Re-calculate the coefficients
			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), precision);
                                    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), precision);
                                    }
                              }

                        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 (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
                        }
                  }

      // Double formula cos(2x)=cos^2(x)-1=-4(1-cos(x)+2*(1-cosx)^2+1
      for (; k > 0; --k)
            {
            v = c1 - cosx;
            cosx  = -c4*v + c2*v.square() + c1;
            }

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

Cos(x) using sin(x)

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

$$\cos(x) = \sqrt{1 - sin^2(x)} \qquad (11)$$

It turns out that this increases the performance by a factor of 2 times the traditional way of calculating cos(x) directly and is therefore recommended.
There is another alternative to using the identity: $\cos(x) = \sin(\frac{\pi}{2} - x)$. If you have a fast generation π you will experience a similar performance as the $\cos(x) = \sqrt{1 - sin^2(x)}$ but in my opinion, it will be safer to rely on the faster sqrt(x) function.

Source for cos(x) using sin(x)

```
float_precision cos(const float_precision& x)
	{
	size_t precision = x.precision() + 2 + (size_t)ceil(log10(x.precision()));
	float_precision cosx(x), sinx(x);
	const float_precision c1(1);
	double d;

	sinx = sin(x);
	cosx = sqrt(c1 - sinx.square());
	d = x; d = cos(d);
	if( d < 0 )
		cosx = -cosx;
	cosx.mode(x.mode());
	cosx.precision(x.precision());
	return cosx;
	}
```

Recommendation for calculating cos(x)

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

- Always use the cyclic and symmetry rules to reduce the x to the range $[0.\pi]$
- It is unnecessary to reduce it down to the range $[0\ldots\frac{\pi}{2}]$ using symmetry avoiding another calculation of π.
- Use the double angle formula for argument reduction instead of the trisection formula.
- Do not use the Taylor series for cos(x) with an aggressive reductions factor to speed up the Taylor term calculation. If you do it anyway then use it with coefficient scaling to increase performance.
- Use $\cos(x) = \sqrt{1 - sin^2(x)}$ as the preferred method for calculating cos(x) which is two times faster than the other cos(x) methods.

Tan(x):

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:

$$tan(x) = \frac{\sin(x)}{\sqrt{1-sin^2(x)}} \qquad (12)$$

However, before we start the calculation we first reduce the argument x so it falls between 0 and 2π and then call Sin(x) (see above).

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

$$\tan(x) = x + \frac{x^3}{3} + \frac{2x^5}{15} + \frac{17x^5}{315} + \frac{62x^9}{2835} + \cdots \frac{2^{2n}(2^{2n}-1)B_n x^{2n-1}}{(2n)!} + \cdots \qquad (13)$$

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 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 )
   {
   const size_t precision=x.precision() + 2;
   float_precision tanx, r, v(x), pi;
   const float_precision c1(1), c3(3);

   // Increase working precision
   tanx.precision( precision );
   v.precision( precision );
   pi.precision( precision );

   // Check that argument is larger than 2PI and reduce it if needed.
   pi = _float_table( _PI, precision );
   tanx = pi;
   tanx.adjustExponent(+1);        // 2*PI
   if( abs( v ) > tanx )
       {
       r = v / tanx;
       (void)modf( r, &r );
       v -= r * tanx;
       }
   if( v < float_precision( 0 ) )
      v += tanx;

   pi.adjustExponent(-1);   // pi *= 0.5;
   if( v == pi || v ==  pi * c3 )
      { throw float_precision::domain_error(); }

   tanx = sin( v );
   if( v < pi || v > pi * c3 )
      tanx /= sqrt( c1 - tanx.square() );
   else
      tanx /= -sqrt( c1 - tanx.square() );
```

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

Arcsin(x):

We have a few options. Either we can find arcsin (x) using the Newton method or we can do it using a Taylor series for arcsin (x).

Arcsin using Newton's method

To find arcSin(x) it is very popular to resort to a Newton iteration when solving the equation arcSin(a)=x => a=sin(x).

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

$$x_{n+1} = x_n - \frac{\sin(x_n)-a}{\cos(x_n)} \quad (14)$$

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:

$$\cos(x) = \sqrt{1 - sin^2(x)} \quad (15)$$

Yields:

$$x_{n+1} = x_n - \frac{\sin(x_n)-a}{\sqrt{1-sin^2(x_n)}} \quad (16)$$

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

$$Arcsin(x) = 2 \cdot ArcSin(\frac{x}{\sqrt{2}\sqrt{1+\sqrt{1-x^2}}}) \quad (17)$$

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

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

$$x_k = \frac{x_{k-1}}{\sqrt{2}\sqrt{1+\sqrt{1-x_{k-1}^2}}} \quad (18)$$

until x_m is sufficiently low. Now we can start with an initial guess of arcsin(x) using standard IEEE754. This gives us a starting guess for the Newton iteration with a least 15 significant digits and 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.

To see how this algorithm works let us find the arcSin(0.3). 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
3	0.304692654	0.304692654015398	0.00E+00

Now assuming for a moment we did not do any argument reduction we will see a much slower convergence when x get near 1. 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**
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 using argument reduction before applying the Newton iterations.

Using Newton's iteration gives the result in relatively few iterations however still not very fast compared to the direct approach using the Taylor series, see 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:

$$Arcsin(x) = x + \frac{x^3}{2\cdot 3} + \frac{3x^5}{2\cdot 4\cdot 5} + \frac{3\cdot 5x^7}{2\cdot 4\cdot 6\cdot 7} + \frac{3\cdot 5\cdot 7x^9}{2\cdot 4\cdot 6\cdot 8\cdot 9} \ldots \qquad (19)$$

This gives us a more direct approach to arcsin(x) and applied together with the dynamic argument reductions we see a speed up in the calculation in the range of two. As the precision rise, this method will become increasingly faster than 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:

$$r_1 = x$$

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

We calculate the reducing factor, k as:

$$2 \cdot \lceil \ln(2) * \ln(precision) \rceil \quad (20)$$

and 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:

$$Arcsin(x) = 2 \cdot ArcSin\left(\frac{x}{\sqrt{2}\sqrt{1+\sqrt{1-x^2}}}\right) \quad (21)$$

Require one division and two square roots (the $\sqrt{2}$ is a constant that can be calculated before the reduction), two multiplication, and two addition/subtracting. The benefit of using reduction slows down and becomes counterproductive when the reduction factor exceeds 30-40.

Below is an example of using the Taylor Series for calculating 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	0.300000000000000	0.300000000000000	4.69E-03
2	4.50E-03	0.304500000000000	0.304500000000000	1.93E-04
3	1.82E-04	0.304682250000000	0.304682250000000	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 the result with 15-16 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 reduce the workload significantly.

Arcsin coefficient scaling

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

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

If we denoted 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 together:

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

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

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

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

The new recurrence for r, grouping two Taylor terms together 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 together 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 use a grouping of 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);

	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
                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
                    }
             }

      // Revere 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());

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

Recommendation for calculating Arcsin(x)

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

- The preferred method is to use the Taylor series for arcsin(), together with 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. (Involving a division and two square roots calculation).

Arccos(x):

To find Arccos(x) we used the identity:

$$Arccos(x) = \frac{\pi}{2} - Arcsin(x) \qquad (22)$$

It 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;
}
```

Arctan(x):

There are two interesting methods to use. One is the standard Taylor series and the other one is contributed to Euler which is considered faster than the Taylor series (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:

$$Arctan(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \frac{x^9}{9} - \cdots, where\ |x| \leq 1 \qquad (23)$$

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

$$Arctan(x) = 2 \cdot \arctan\left(\frac{x}{1+\sqrt{1+x^2}}\right) \qquad (24)$$

k number of times until x is sufficiently low.

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

We calculate the reducing factor, k as:

$$2 \cdot \lceil \ln(2) * \ln(precision) \rceil \qquad (25)$$

and 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:

$$Arctan(x) = 2 \cdot \arctan\left(\frac{x}{1+\sqrt{1+x^2}}\right) \quad (26)$$

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

After the Taylor series has converged, we multiply the result with 2^k to find our result for arctan(x). Now 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:

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

or

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

To see how this algorithm works let us find the arctan(0.3). After the 13th Taylor terms the errors do not get lower 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	0.300000000000000	0.300000000000000	8.54E-03
2	9.00E-03	0.291000000000000	0.291000000000000	4.57E-04
3	4.86E-04	0.291486000000000	0.291486000000000	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

Now if we take two-argument reduction we reduced 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 to lowering the number of Taylor terms needed when precision is increased.

The issue with arbitrary precision

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

$$\frac{x^{2n-1}}{2n-1} < 10^{-P} \tag{27}$$

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

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

$$(2n+1)\ln(x) - \ln(2n+1) = -P \cdot \ln(10) \tag{28}$$

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

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

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

$$n \approx \frac{-P \cdot \ln(10) - \ln(x)}{2 \cdot \ln(x)} \tag{29}$$

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

$$n \approx \frac{-P-M}{2 \cdot M} \tag{30}$$

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

This table indicates the usefulness of argument reduction.

The table above is quite interesting. E.g., 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 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 gain typically 2-3 times better performance if we implement coefficient scaling. If we try to group two Taylor terms to avoid a division, we get from the Taylor series for arctan where n denoted 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 going forward:

$$Two\ Taylor\ terms:\ -\frac{x^{n-1}}{2n-1}+\frac{x^{n+1}}{2n+1} =>$$

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

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

If we group three Taylor terms, we get:

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

We can continue grouping Taylor terms. From a practical point of view, grouping five Taylor terms together is a reasonable amount as it will double the performance compared to not doing it.

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();
	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)
                              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:

$$Arctan(x) = \sum_{n=0}^{\infty} \frac{2^{2n}(n!)^2}{(2n+1)!} \frac{z^{2n+1}}{(1+x^2)^{n+1}} \quad (31)$$

For x>0.4 required fewer Terms than the equivalent Taylor series, e.g. arctan(0.6) requires 25 terms to get the result. While using the Taylor series requires 30 Taylor terms. As x increased, it get worse. However, for x<0.4 the Taylor series and the 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
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 using argument reduction greatly reduced the number of terms needed. E.g. arctan(0.6) using a reduction factor of four requires only 5 terms (same as for 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 the corresponding Taylor series. Overall it is not worth using the Euler version of arctan(x) over the Taylor series version.

Arctan(x) using Arcsin()

It could be interesting to use the identity:

$$Arctan(x) = \text{Arcsin}\left(\frac{x}{\sqrt{1+x^2}}\right) \qquad (32)$$

Particularly if you want to reduce the size of your code and reuse existing code for arcsin(). 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 recommend:

- The preferred method is to use the Taylor series for arctan(x), together with 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, it is more beneficial to stick with the Taylor series and use the recommended argument reduction and coefficient scaling for increased performance.
- Arctan(x) using arcsin(x) is an alternative that is slower but 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)