Practical implementation of Interval Arithmetic.

By Henrik Vestermark (hve@hvks.com)

Abstract

Interval arithmetic is a mathematical technique that handles ranges of values rather than precise numbers. Instead of working with a single value, calculations are performed with intervals representing the range of possible values. Each interval has a lower and upper bound, representing the minimum and maximum possible values respectively.
You can find examples of interval arithmetic template classes for the C++ programming language but none that are easy to understand and implement. That is why this paper was written "Practical Implementation of Interval Arithmetic". The paper highlights implementing a general interval template class supporting the float and double type of the C++ programming language.

Introduction

The primary advantage of interval arithmetic is its ability to provide guaranteed bounds on results, which is especially useful for dealing with rounding errors, measurement uncertainties, or imprecise inputs. When calculations are performed, the results are also intervals that encompass all possible outcomes. This is particularly valuable in fields like numerical analysis, computer science, and engineering, where understanding the precision and reliability of computations is crucial. It allows for more robust error handling and uncertainty management in complex calculations.

Building an interval template software class that can handle all interval arithmetic for IEEE754 floating point is simple math. This paper describes the practical implementation aspect of building an interval template class and discusses the tricks and math behind the implementation. The implementation spans from the basic operators like **+,-,*,/** to elementary functions like sqrt(), log(), pow(), exp() and trigonometric functions of sin(), cos(), tan(), asin(), acos() and atan() and hyperebolic functions of sinh(), cosh(), tanh(), asinh(), acosh(), atanh(). Of course, it also supports interval constants like π, ln2, and ln10. The interval arithmetic supports the basic C++ types of *float* and *double*.
.
As usual, we look at the practical implementation aspect of the template class and not so much the theoretical rationale behind it. See [4] Interval Arithmetic: From Principles to Implementation and [5] for some deeper insight into interval arithmetic and the use of IEEE754 floating point format.

Change Log

26-March 2024. This is a completely rewritten second version of the original form 2015.

Contents

History of Interval Arithmetic

The history of interval arithmetic dates back to the mid-20th century and is intertwined with the development of computer science and numerical analysis. Here's a brief overview of its evolution:

Early Concepts around the 1950s. The roots of interval arithmetic can be traced to the work of mathematicians like Archimedes. However, the formal development began in the 1950s with the advent of digital computers.
The development of interval arithmetic as a formalized field is largely credited to Ramon E. Moore. Moore, while working at the Applied Physics Laboratory at Johns Hopkins University and later at Stanford University, developed the basic rules and operations of interval arithmetic. His work was aimed at addressing the problem of error bounds in calculations, which was crucial for reliable scientific computing.
Expansion and Formalization of interval arithmetic happened in the 1970s-1980s. The field expanded significantly during this period. Researchers began to formalize the arithmetic and explore its theoretical properties. They also started to apply it to various problems in scientific computing, engineering, and error analysis.
With the rise of computer science, interval arithmetic started to be integrated into computer algorithms and software in the 1980s-1990s time frame. This was facilitated by the increasing need for reliable numerical computations in engineering and the sciences.
After the 2000s Interval arithmetic has continued to grow and find new applications. It's used in fields ranging from numerical analysis to robotics and control systems. The development of specialized software and hardware to efficiently perform interval arithmetic has also been a focus.

Throughout its history, interval arithmetic has been driven by the need for accurate and reliable numerical computations, especially in the face of uncertain data or rounding errors. Its development reflects a broader trend in mathematics and computer science towards more robust and error-resistant methods. As a curious example
Archimedes is one of the earliest known figures to use a form of interval arithmetic in estimating the value of π. He did this by inscribing and circumscribing polygons around a circle and calculating their perimeters. By increasing the number of sides of the polygons, he was able to narrow the bounds and approximate π more closely. At that time Archimedes couldn't calculate the exact value of π. He determined that π was between $3\,\frac{10}{17}$ (approximately 3.1429) and $3\,\frac{10}{71}$ (approximately 3.1408). This method of approximation was quite sophisticated for its time and is an early historical example of the principles behind interval arithmetic, where values are confined within an upper and lower bound to provide a range of possible values.

Application for Interval Arithmetic

Interval arithmetic has several practical applications across various fields due to its ability to handle uncertainties and provide guaranteed bounds on calculations. Here are some of its primary applications starting with Numerical Analysis which ensures accuracy and

reliability in computations by accounting for rounding errors and other numerical uncertainties.
In rendering computer graphics and geometric computations, interval arithmetic helps in handling inexact data and calculations, leading to more robust algorithms. For systems that require stability and reliability (control systems), interval arithmetic can manage uncertainties in measurements and model parameters. In planning and control of robotics, it’s used to account for the imprecision in sensor data and to ensure safe movements and decisions. In engineering design Engineers use interval arithmetic for designing systems with tolerances and variations in material properties. For simulations and modeling in scientific computing where input data might be uncertain or vary over a range, interval arithmetic provides a way to ensure results account for all possible input values. This also extends to optimization problems, error analysis, and propagation which helps in understanding how errors in input data can affect the outcome of calculations, which is crucial in sensitive applications like aerospace and medical equipment.

By providing a framework to handle and propagate uncertainties, interval arithmetic enhances the robustness and reliability of computational tasks in these and other areas.

Interval Arithmetic

To bound rounding errors when performing floating point arithmetic, we can use interval arithmetic to keep track of rounding errors. After a series of operations using basic operators like **+,-,*,** and / we end with an interval instead of an approximation of the result. The interval width represents the uncertainty of the result but we would know for sure that the correct result will be within this interval.

In interval arithmetic, an interval is typically represented as [a, b], where 'a' is the lower bound and 'b' is the upper bound. Conventionally, it is assumed that a ≤ b, defining what is known as a proper interval. Conversely, if a > b, the interval is termed an improper interval. According to the IEEE 1788 standard for interval arithmetic, both proper and improper intervals are valid and must be correctly handled in computations, as we cannot always assume that a ≤ b. This necessitates determining the lower bound (infimum) of an interval by taking min(a, b), and the upper bound (supremum) by taking max(a, b). This requirement introduces additional steps in the implementation: rather than assuming a ≤ b, code must incorporate functions to find the min and max values for any given interval.

Additionally, an interval where a = b is known as a singleton interval.

For simplicity in formulas and explanations, we often present intervals as if they are proper (a ≤ b). However, in practice, especially in coding implementations, it's crucial to handle intervals correctly according to their proper or improper nature (a>b).

Moreover, it's important to define closed intervals [a, b], semi-closed (or semi-open) intervals (e.g., [a, b) or (a, b]), and open intervals (a, b). Each of these types has different implications for the inclusion or exclusion of the endpoint values in the interval.

The five different interval types are:

Interval type	Definition	Symbol	Comments
Close	a≤x≤b	[a,b]	
Left open	a<x≤b	(a,b]	same as Right close
Right open	a≤x<b	[a,b)	same as Left close
Open	a<x<b	(a,b)	
Empty		∅	Empty interval

Besides the ∅ (empty interval), the 4 other different types are not strictly needed by the standard. However, it is quite convenient to have them otherwise defined an open or semi-open interval will be challenged. For E.g. creating an open interval (0,1) will be something like this in C++:

```
interval<double> i(nextafter(0,+INFINITY), nextafter(1,-INFINITY));
```

A rather inconvenient and cumbersome way of creating this interval while in my implementation can be done like this:

```
interval<double> i(0,1,OPEN);
```

This approach ensures a comprehensive understanding and correct implementation of interval arithmetic, adhering to the IEEE 1788 standard.

An interval is presented by two numbers representing the lower and upper range of the interval:

[a,b] Where $a \leq b$ (for proper intervals)

The 4 basic operators +,-,*, and / for a proper interval are then defined by:

[a,b] + [c,d] = [a+c,b+d]
[a,b] – [c,d] = [a-d, b-c]
[a,b] * [c,d] = [min(ac,ad,bc,bd),max(ac,ad,bc,bd)]
[a,b] / [c,d] = [min($\frac{a}{c},\frac{a}{d},\frac{b}{c},\frac{b}{d}$), max ($\frac{a}{c},\frac{a}{d},\frac{b}{c},\frac{b}{d}$)] and [c,d] does not contain 0.

The division can also be rewritten to:

[a,b] / [c,d] = [a,b] * (1/[c,d]) = [a,b] * [$\frac{1}{d},\frac{1}{c}$]

Now what we don't like is the extra work we would need to do for multiplication of intervals. According to the definition, it will require 8 multiplications to do an interval multiplication. We can however get by with a lot less.

Let's first define an interval class as follows:

Class of [a,b]	Conditions $a \leq b$
Zero	$a=0 \wedge b=0$
Mixed	$a<0 \wedge b \geq 0$
Positive	$a>0$
Negative	$b<0$

Instead of the 8 multiplications, we can get by with 2 in most cases and 4 multiplications in worst case based on the following table:

Class of [a,b]	Class of [c,d]	*
Positive	Positive	[a*c,b*d]
Positive	Mixed	[b*c,b*d]
Positive	Negative	[b*c,a*d]
Mixed	Positive	[a*d,b*d]
Mixed	Mixed	[min(a*d,b*c),max(a*c,b*d)]
Mixed	Negative	[b*c,a*c]
Negative	Positive	[a*d,b*c]
Negative	Mixed	[a*d,a*c]
Negative	Negative	[b*d,a*c]
Zero	-	[0,0]
-	Zero	[0,0]

From a practical standpoint, we can't just use the above definition of "as is". When performing any basic operations, we will always introduce some rounding errors when using a finite representation like the IEEE754 standard for 32-bit binary floating-point arithmetic and 64-bit binary floating-point arithmetic which is common in most programming languages and CPUs from most vendors like Intel, AMS, etc.

General speaking when performing any of the basic operations like +,-,*,/ we have an error that is bound by the $\frac{1}{2}\beta^{1-t}$ using the default rounding mode in IEEE754 or β^{1-t} when rounding towards positive infinity or negative infinity. In IEEE754 binary arithmetic β=2 and t=24 (23+1) for 32-bit float and t=53 (52+1) for 64-bit float, this gives a relative error when performing any of the basic operations

IEEE754	Rounding	Towards $+\infty$ or $-\infty$
32bits float	$\sim 5.96^{-8}$	$\sim 1.19^{-7}$
64bits float	$\sim 2.22^{-16}$	$\sim 1.11^{-16}$

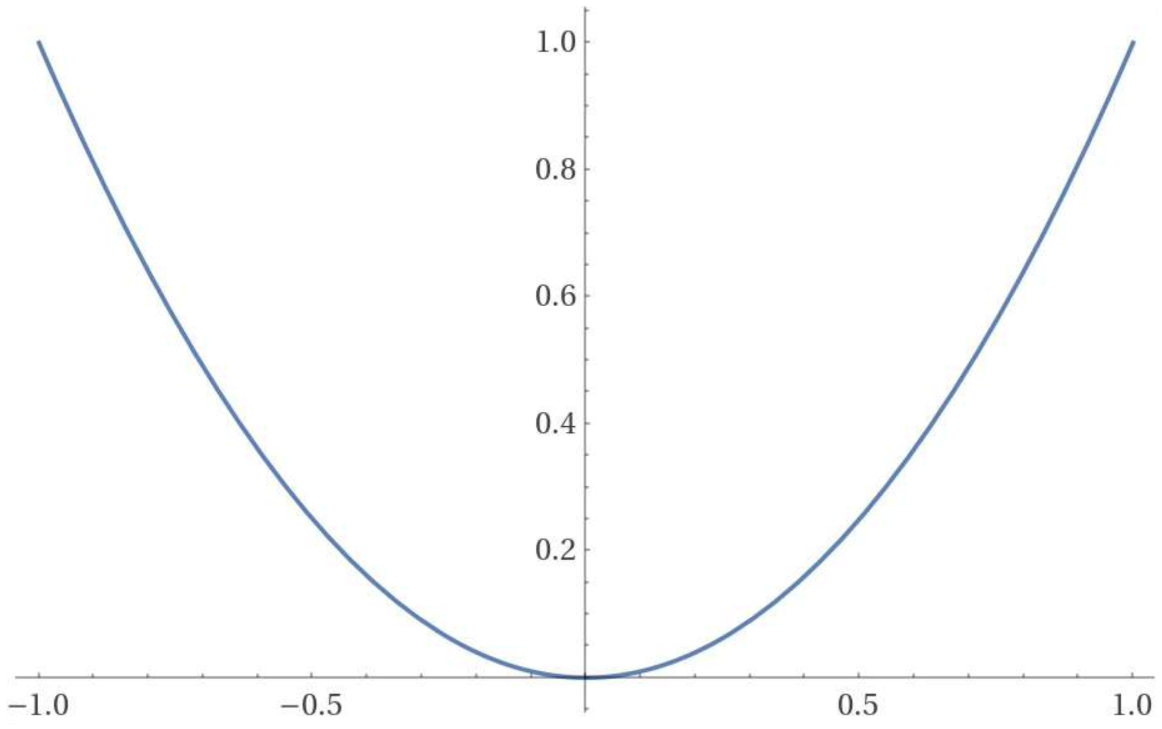
When performing the interval operations, we would need to also control the rounding mode to ensure that the result is really within the interval. $\underline{a}$ means rounding towards -∞ and $\overline{b}$ means rounding towards +∞

$$[\underline{a}, \overline{b}] + [\underline{c}, \overline{d}] = [\underline{a+c}, \overline{b+d}]$$
$$[\underline{a}, \overline{b}] - [\underline{c}, \overline{d}] = [\underline{a-d}, \overline{b-c}]$$
$$[\underline{a}, \overline{b}] * [\underline{c}, \overline{d}] = [\min\underline{(ac, ad, bc, bd)}, \max(\overline{ac, ad, bc, bd}])$$
$$\frac{[\underline{a}, \overline{b}]}{[\underline{c}, \overline{d}]} = [\underline{a}, \overline{b}] * [\underline{\frac{1}{d}}, \overline{\frac{1}{c}}]$$

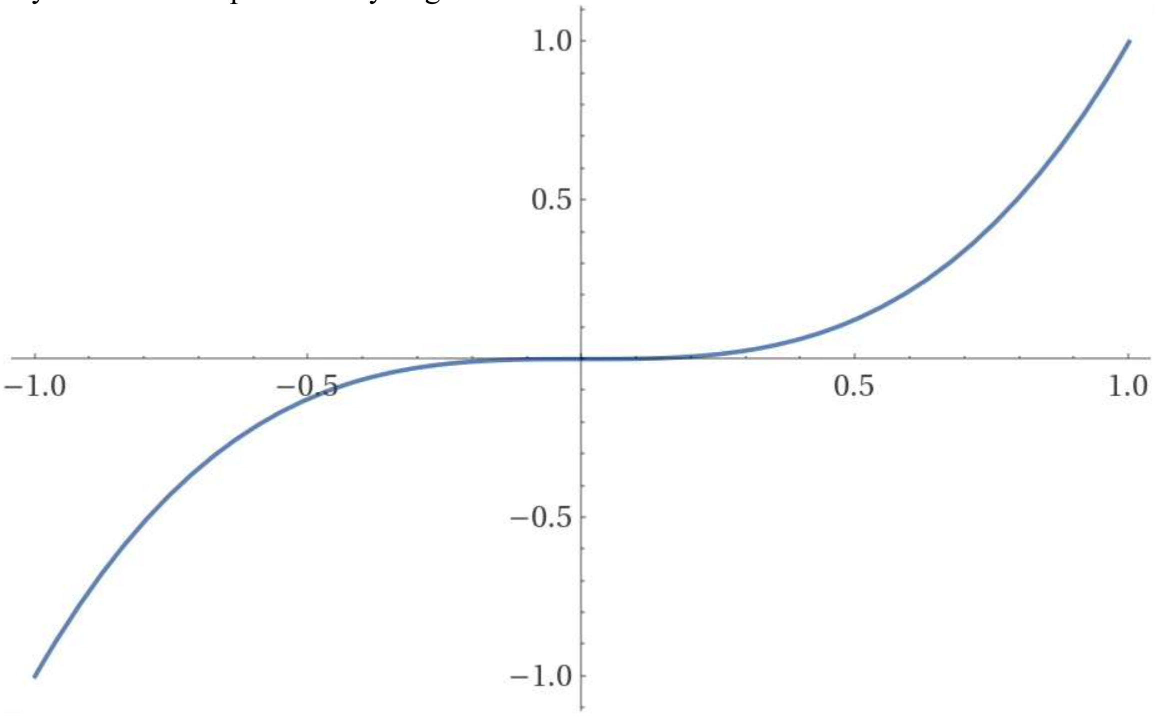
From an IEEE1788 standard, it is also required that an interval can be empty. It is designated with the symbol ∅. There are a few rules for the empty interval.

[a,b]+∅:=[a,b], same goes for ∅+[a,b]:=[a,b]
[a,b]-∅:=[a,b], ∅-[a,b]:=[-b,-a]
[a,b]*∅:=∅, ∅*[a,b]:=∅
[a,b]/∅:=∅, ∅/[a,b]:=∅

Lastly, we also need to be aware of what happens when you square an interval e.g. $[a,b]^2$. If you take the interval [-1,1] and just multiply it with itself using the multiplication formula you get a new interval of [-1,1] which is unnecessarily large. If you look at the plot of x^2 in the range [-1,1] you get:



As you can see the result goes from +1 down to 0 and then rises again to +1. The right result for such an operation is [0,1]. The IEEE 1788 standard kind of foreseen this issue and required a compliant implementation to have a function called sqr() for the squaring of intervals. This issue persists for even power of an interval which needs to be handled properly in an implementation.
If you look at the plot for x^3 you get:



As you can see the value of the operation runs through the interval of [-1,1].

Controlling the rounding

The default rounding mode in the IEEE754 floating-point standard is round to nearest (RN for short). However, as shown in the previous section we need access to round down towards -∞ (RD) and round up towards +∞ (RU).

In interval arithmetic, rounding is a crucial aspect, and it can be implemented either using hardware-based methods or software-based emulation. Hardware-based rounding offers speed and accuracy but lacks flexibility and portability. In contrast, software emulation is more portable and flexible but can be slower and potentially less accurate.

Hardware-Based Rounding

Here are some of the pros of using hardware-based rounding.
Hardware-based rounding is typically faster as it utilizes the processor’s built-in capabilities. It often offers more precise and consistent rounding due to the direct

implementation in the CPU. Reduces the overhead of software emulation, making computations more efficient.

However, the advantages in speed come at a cost, since It relies on specific hardware features, which might not be available or consistent across different platforms or processors.
Hardware implementations are harder to customize or update compared to software. Software relying on specific hardware features may face portability issues across different systems.

As required in IEEE754 round to -∞ (RD) and round to +∞ (RU) are available and it is just a simple matter to switch between these modes from the default round to the nearest.

Controlling Hardware Rounding

To control the rounding process, the IEEE754 offers at least 4 ways of rounding.

1. Rounding to the nearest which is the default rounding method,
2. Rounding down towards -∞
3. Rounding up towards +∞
4. Rounding towards zero (truncating)

Depending on the CPU architecture different instructions enabled these different modes. However, it is unfortunately not portable from architecture to architecture. We will just rely on that somehow the following 3 functions are available that enable these modes.

```
Fp_near();      // Set Floating-point mode to round to the nearest

Fp_Down();      // Set floating-point mode to round down towards -∞

Fp_up();        // Set floating-point mode to round towards +∞
```

As an example of how it can be used. We can implement the interval addition of two intervals [a,b] and [c,d] as follows.

```
      Fp_up();
      f=b+d;
      Fp_down();
      e=a+c;
      Fp_near();
      return [e,f];
```

It seems advantageous to use hardware-controlled rounding, however, in [7] they noted that switching back and forth between the different hardware rounding modes is a relatively expensive operation and surprisingly it turns out that using the software rounding method provides a higher performance in many cases. Although it depends highly on the actual CPU architecture.

Software-Based Rounding (Emulation)

Software-based rounding for sure has its advantages. Software-based rounding is more portable as it does not depend on specific hardware features. It is easier to customize or adapt to specific needs, such as implementing non-standard rounding modes and it ensures consistent behavior across different platforms and hardware.

The drawback of software-based rounding is that it is typically slower than hardware implementations due to the overhead of emulation and it can be more complex to implement and maintain its accuracy and consistency compared to hardware implementations, depending on the quality of the software emulation.

The software-based rounding takes advantage of round to nearest is the default rounding mode in IEEE754. and two algorithms are useful. The *two-sum* and *two-product* algorithms are fundamental in interval arithmetic for maintaining precise bounds. These operations are essential for addressing the problem of rounding errors in floating-point arithmetic. see [7]

1. The *Two-Sum* Algorithm is used to add two floating-point numbers while capturing the error in the result due to rounding. The two-sum operation returns two numbers: the exact sum (as close as the floating-point representation allows) and the error.
2. The *Two-Product* Algorithm is similar to the two-sum, the two-product algorithm multiplies two floating-point numbers, providing the product and the error.

To see how we can take advantage of the two algorithms Let's consider the sum of the arithmetic operation a+b. The result will be rounded to the nearest result (RN). The round to the nearest is either the same as the round down (RD) or the same as rounded up (RU). To figure out which we use the error from the *two-sum* algorithm. If the error is negative then:

RN(a+b)==RU(a+b) and RD(a+b) is predecessor(RN(a+b))

If e is positive then:

RN(a+b)==RD(a+b) and RU(a+b) is successor(RN(a+b))

The successor returns the next representable floating-point number towards $+\infty$ and the predecessor returns the next representable floating-point number towards $-\infty$.

The same analogy can be used for multiplication. If the error is negative then:

RN(a*b)==RU(a*b) and RD(a*b) is predecessor(RN(a*b))

If e is positive then:

RN(a*b)==RD(a*b) and RU(a*b) is successor(RN(a*b))

You can extend that to division, reciprocal, and square root, see the table below.

Operation	**Error<0**		**Error>0**	
RN(a+b)	RU==RN	RD=predecessor	RD=RN	RU=successor
RN(a-b)	RU==RN	RD=predecessor	RD=RN	RU=successor
RN(a*b)	RU==RN	RD=predecessor	RD=RN	RU=successor
RN(a/b)	RU==RN	RD=predecessor	RD=RN	RU=successor
RN(1/b)	RU==RN	RD=predecessor	RD=RN	RU=successor
RN(sqrt(a))	RU==RN	RD=predecessor	RD=RN	RU=successor

Two-sum algorithm was invented in 1965 by O. Møller and comes in two variations. The *two-sum* and the *fast two-sum* algorithm and implementation of this are below. The argument could be either by value or by reference. For standard types like float and double it does not matter too much. However, in anticipation of phase four of the implementation where we will expand the types to include an arbitrary precision floating point class (authors own Arbitrary precision library). Due to the arbitrary precision and large size, it would be much more efficient to use call-by-reference for our arguments.

The original two-sum algorithm in C++.

```cpp
// Implement the twosum algorithm. Assuming round to nearest mode (default
IEEE754)
// sum=(a+b)
// a1=sum-b
// b1=sum-a
// da=a-a1
// db=b-b1
// err=da+db
// return sum, err
// if err is negative then sum=Round_up(a+b), and
// round_down(a+b)=previous(sum)
// if err is positive then sum=Round_down(a+b), and Round_up(a+b)=successor(sum)
//     The twosum algorithm requires 6 floating point operations
std::pair<IT, IT> two_sum(const IT& a, const IT& b)
{
       const IT sum(a + b);
       const IT a1(sum - b);
       const IT b1(sum - a);
       const IT da(a - a1);
       const IT db(b - b1);
       const IT err(da + db);
       return std::make_pair(sum, err);
}
```

The *two-sum* algorithm requires approximately 6 floating-point operations.

A faster version, surprisingly called the fast two-sum algorithm is shown below in C++.

```cpp
// Implement the fast two-sum algorithm
// Assuming round to nearest mode (default IEEE754)
// sum=(a+b)
// a1=sum-a
// err=b-a1
// return sum, err
// if err is negative then sum=Round_up(a+b), and
// round_down(a+b)=previous(sum)
// if err is positive then sum=Round_down(a+b), and
// Round_up(a+b)=successor(sum)
// The fast two-sum algorithm requires only 3 floating point instruction
// and a comparison
static std::pair<IT, IT> fasttwo_sum(const IT& a, const IT& b)
{
       const IT sum(a + b);
       if (abs(a) > abs(b))
       {
              IT tmp(sum - a);
              IT err(b - tmp);
              return std::make_pair(sum, err);
              }
       else
       {
              IT tmp(sum - b);
              IT err(a - tmp);
              return std::make_pair(sum, err);
       }
}
```

which only need 3 floating point operations and comparison operations. However, it requires that a≥b for the sum of a+b.

The Two products were invented by Rump, see C++ implementation below.

```
// Split argument into a right and left. (Dekker's method)
// double has 53bits in mantissa and shifting is therefore (53+1)/2=27
// float has 24bits in mantissa and shifting is therefore (24+1)/2=12
IT split(const IT& a)
{
       const IT C(a * double((1 << 27) + 1));
       IT xright(C - (C - a));
       IT xleft(a - xright);
       return std::make_pair(xright, xleft);
}

// The standard two-product algorithm  (by RUMP's method)
// (xh,xl)=split(a)
// (yh,yl)=split(b)
// p=a*b
// t1=-p+xh*yh
// t2=t1+xh*yl
// t3=t2+xl*yh
// err=t3+xl*yl
// return (p,err)
// if err is negative then sum=Round_up(a*b), and round_down(a*b)=previous(sum)
// if err is positive then sum=Round_down(a*b), and
// Round_up(a*b)=succesor(sum)
// The fast two-product algorithm requires 17 floating-point instruction
// and a comparison
std::pair<IT, IT> two_prod(const IT& a, const IT& b)
{
       const std::pair<IT, IT> x(split(a));
       const std::pair<IT, IT> y(split(b));
       const IT p(a * b);
       const IT t1(-p + x.first * y.first);
```

```
	const IT t2(t1 + x.first * y.second);
	const IT t3(t2 + x.second * y.first);
	const IT err(t3 + x.second * y.second);
	if (abs(p) > ldexp(1, -1021))  // avoiding overflow
			err = x.second * y.second - ((((p * 0.5 - (x.first * 0.5) * 
y.first) * 2) - x.second * y.first) - x.first * y.second);
	else
		err = x.second * y.second - (((p - x.first * y.first) - x.second * 
y.first) - x.first * y.second);
		return std::make_pair(p, err);
	}
```

However, as pointed out by [7] if you have access to a fma operation (Fused-Multiply-Add) you can simplify the *two-product* codes to simple:

The two-product using fma

```
// The fast two product algorithm
// p=a*b
// err=fma(a,b,-p)
// fma is required in IEEE754 and either implemented in hardware or software
// return (p,err)
// if err is negative then sum=Round_up(a*b), and round_down(a*b)=previous(sum)
// if err is positive then sum=Round_down(a*b), and Round_up(a*b)=succesor(sum)
// The fast two-product algorithm requires only 2 floating point instruction
// and a comparison
std::pair<IT, IT> fasttwo_prod(const IT& a, const IT& b)
{
	const IT p(a*b);
	IT err(fma(a, b, -p));

	return std::make_pair(p, err);
	}
```

The fma() has been available since the c99 standard came out.
With that, we are nearly there. We still have to deal with overflow in those two crucial functions. For the *fast_two-sum* the addition a+b can overflow and can give a misleading error direction either positive or negative. The correct response would be in case of overflow to return ±infinity and the error to zero.
For the *two_product* we need to deal with both an overflow situation or underflow in the a*b. For overflow, we will do as for the two-sum algorithm and return the error to zero. In case of underflow, we would need to repeat the product using a scaled product that doesn't underflow and then base the error directions of that computation.

The final *two_sum* and *two_product* functions are below.

```
// Implement the fast two-sum algorithm
// Assuming round to nearest mode (default IEEE754)
// sum=(a+b)
// a1=sum-a
// err=b-a1
// return sum, err
// if err is negative then sum=Round_up(a+b), and
// round_down(a+b)=previous(sum)
// if err is positive then sum=Round_down(a+b), and Round_up(a+b)=successor(sum)
// The fast two-sum algorithm requires only 3 floating point instructions and
// a comparison
// Note if an overflow occurs the sum will be +infinity and the error is set
// to 0.
static std::pair<IT, IT> fasttwo_sum(const IT& a, const IT& b)
{
	const IT sum(a + b);
	if (abs(a) > abs(b))
	{
		IT tmp(sum - a);
		IT err(b - tmp);
		if (sum == infinity_interval<IT>()) // If overflow occurs then set 
err=0
			err = IT(0);
		return std::make_pair(sum, err);
	}
	else
	{
		IT tmp(sum - b);
		IT err(a - tmp);
		if (sum == infinity_interval<IT>()) // If overflow occurs then set 
err=0
			err = IT(0);
		return std::make_pair(sum, err);
	}
}

// The fast two product algorithm
// p=a*b
// err=fma(a,b,-p)
// fma is required in IEEE754 and either implemented in hardware or software
// return (p,err)
// if err is negative then sum=Round_up(a*b), and round_down(a*b)=previous(sum)
// if err is positive then sum=Round_down(a*b), and Round_up(a*b)=succesor(sum)
// The fast two-product algorithm requires only 2 floating point instruction
// and a comparison
// Note if an overflow occurs the product will be +infinity and the error is 
// set to 0.
std::pair<IT, IT> fasttwo_prod(const IT& a, const IT& b)
{
	const IT p(a*b);
	IT err(fma(a, b, -p));
	if (p == infinity_interval<IT>()) // If overflow occurs then set err=0
		err = IT(0);
	if (p != IT(0) && abs(p) < underflow_interval<IT>()) // Test for underflow 
conditions
	{// Recalculate the err using the scale products
		int aexp, bexp;
		IT ascale(frexp(a, &aexp)); // ascale [1.0,2.0)
		IT bscale(frexp(b, &bexp)); // bscale [1.0,2.0)
		IT p2(ascale * bscale );
		err = fma(ascale, bscale, - p2);
	}
	return std::make_pair(p, err);
}
```

Software libraries and systems that support interval arithmetic often implement these algorithms to ensure that all possible values within the intervals are accurately accounted for. Some well-known software and libraries include INTLAB: A MATLAB toolbox for interval arithmetic that provides reliable computations with rounding error bounds. The Boost Interval Arithmetic Library: Part of the Boost C++ Libraries, it supports interval arithmetic operations with various policies for controlling the rounding and handling of interval bounds.

These tools and libraries are used in various fields, from scientific computing to engineering and finance, wherever accurate error tracking and robust interval calculations are needed. They implement not just *two-sum* and *two-products* but a whole range of arithmetic and mathematical operations adapted for interval arithmetic.

Interval class requirements

With the definition of interval arithmetic done and the two crucial algorithms *two-sum* and *two-product,* we can now turn to the requirement of an interval class.
It is always important to have your requirements ready before starting the C++ coding process. I usually divide it up into phases so I start with a base implementation and then expand it further through the defined phases. For each phase, you add testing of the phase functionalities.

Phase One: establishing the base functionality
All the essential operators: +,-,*,/,=,+=,-=,*=,/=.
Constructors for a float, double, or long double to interval type.
Conversion operators from interval to regular float, double, long double, short, int, long.
Support of closed, semi-closed, semi-open, and open intervals.
Support the empty interval set ∅.
cout and cin operator.
Comparison operators ==,!=,<,<=,>,>=.
Essentials methods:

negate: Negate the interval (same as monadic -).
toString: return the string representation of an interval number.
width: return the width of the interval. (standard names this wid).
center: return the center of the interval (standard names this mid).
radius: return the radius of the interval (standard names this rad).
infimum: return the infimum of the interval (standard name it inf).
supremum: return the supremum of the interval. (standard name it sup).
leftinterval: return or set the lower range of the interval as is.
rightinterval: return or set the upper range of the interval as is.
isEmpty: return the Boolean value of emptiness of the interval.
isProper: return true if the interval is a proper interval otherwise false.
isImproper: return true if the interval is an improper interval otherwise false.
isPoint: return true if the interval is a singleton, otherwise false.
isClass: return the classification of the interval.
abs: return the absolute value of the interval.
sqrt: return the square root of the interval.
sqr: return the square of the interval (x^2).

Phase Two: expanding the base functionality

mixed data type for arithmetic operations.

manifest interval constants: LN2, LN10, e, PI,
exponential (exp) and logarithm functions. (log,log10)
power x^y
Union: Union of two intervals
Intersection: intersections of two intervals
Interior: is the interval an interior to another interval
Precedes: Does the interval precede another interval
Subset: is the interval a subset of another interval
Inclusion: is the interval included in another interval
In: is the float or double number within the interval

Phase Three: Completed all the auxiliary functions
Trigonometric interval functions
Hyperbolic interval functions:

Phase Four:
Expand support for arbitrary precision interval type
other useful functions e.g. gamma, beta, etc.

Phase One Implementation

Is all about establishing the base functionality of the interval class.

Defining the interval template class

With the implementation split into phases, we are now ready to begin our initial design. We need to create a template class for our interval functionality. In essence similar to the standard complex template library in C++. Also, the class needs to hold the left and right sides of the interval plus the interval type to be able to cope with Closed, Semi-open, or fully open intervals as previously discussed. Not surprisingly we end up with this template class definition and the private parts and variables.

```
// Interval class
// Realistically the class Type can be float, or double.
// Any other type is not supported
//
template<class IT> class interval {
       IT left;                   // Left interval
       IT right;                  // Right interval
       enum interval_type type;   // Interval type CLOSE, OPEN, LEFT_OPEN,
                                     RIGHT_OPEN, (RIGHT_CLOSE is synonym for
                                     LEFT_OPEN and LEFT_CLOSE same as RIGHT_OPEN
       public:

   };
```

We have also established our two functions fasttwo_sum() and fasttwo_prod() as internal private functions to the class and we get the following:

```
// Interval class
// Realistically the class Type can be float, or double.
// Any other type is not supported
//
template<class IT> class interval {
	IT left;			// Left interval
	IT right;			// Right interval
	enum interval_type type;	// Interval type CLOSE, OPEN, LEFT_OPEN,
					// RIGHT_OPEN, (RIGHT_CLOSE is synonym for
					// LEFT_OPEN and LEFT_CLOSE same as RIGHT_OPEN
					// and EMPTY

	// Implement the fast two-sum algorithm
	// Assuming round to nearest mode (default IEEE754)
	// sum=(a+b)
	// a1=sum-a
	// err=b-a1
	// return sum, err
	// if err is negative then sum=Round_up(a+b), and
	// round_down(a+b)=previous(sum)
	// if err is positive then sum=Round_down(a+b), and
	// Round_up(a+b)=successor(sum)
	// The fast two-sum algorithm requires only 3 floating point
	// instruction and a comparison
	static std::pair<IT, IT> fasttwo_sum(const IT& a, const IT& b)
	{
		const IT sum(a + b);
		if (abs(a) > abs(b))
		{
			IT tmp(sum - a);
			IT err(b - tmp);
			if (sum == infinity_interval<IT>()) // If overflow occurs
then set err=0
				err = IT(0);
			return std::make_pair(sum, err);
		}
		else
		{
			IT tmp(sum - b);
			IT err(a - tmp);
			if (sum == infinity_interval<IT>()) // If overflow occurs
then set err=0
				err = IT(0);
			return std::make_pair(sum, err);
		}
	}
	// The fast two product algorithm
	// p=a*b
	// err=fma(a,b,-p)   // fma is required in IEEE754 and either
	// implemented in hardware or software
	// return (p,err)
	// if err is negative then sum=Round_up(a*b), and
	// round_down(a*b)=previous(sum)
	// if err is positive then sum=Round_down(a*b), and
	// Round_up(a*b)=succesor(sum)
	// The fast two-product algorithm requires only 2 floating-point
	// instruction and a comparison
	std::pair<IT, IT> fasttwo_prod(const IT& a, const IT& b)
	{
		const IT p(a*b);
		IT err(fma(a, b, -p));
```

```
              if (p == infinity_interval<IT>()) // If overflow occurs then set
err=0
                     err = IT(0);
              if (p != IT(0) && abs(p) < underflow_interval<IT>()) // Test for
underflow conditions
              {// Recalculate the err using the scale products
                     int aexp, bexp;
                     IT ascale(frexp(a, &aexp)); // ascale [1.0,2.0)
                     IT bscale(frexp(b, &bexp)); // bscale [1.0,2.0)
                     IT p2(ascale * bscale );
                     err = fma(ascale, bscale, - p2);
              }
              return std::make_pair(p, err);
       }

       public:

   };
```

That was the private part of our template class. Now to the constructors for the class.

Interval Constructors

As our requirement goes our interval class can be declared with zero, one, two, or 3 parameters. Like in.

```
// Declare an interval variable with an empty class ∅
Interval<double> i;

// Declare an interval variable initialized with a point
Interval<float> fi(2.0)

// Declare an interval variable as the interval [1,2]
Interval<double> di(1.0,2.0);

// Declare an interval variable with the open interval (3,4)
Interval<double> dit(3.0,4.0,OPEN);

// Declare an improper closed interval variable [3,2]
Interval<float> dit2(3,2);
```

```
// constructor. zero, one or two arguments for type IT
interval();                // Empty interval
interval(const IT& d);     // Singleton interval
// Regular interval with interval_type (default CLOSE)
interval(const IT& l, const IT& h, const enum interval_type t=CLOSE);
```

The interval type is defined below:

```
// The five different interval types
// # Close   a<=x<=b       [a,b]
// # Left open a<x<=b      (a,b]  same as Right close
// # Right open a<=x<b     [a,b)  same as Left close
// # Open a<x<b            (a,b)
// # EMPTY interval        Ø
enum interval_type { EMPTY, CLOSE, LEFT_OPEN, RIGHT_OPEN, OPEN,
                     RIGHT_CLOSE=LEFT_OPEN, LEFT_CLOSE=RIGHT_OPEN };
```

Interval Coordinate functions

We would also need some coordinating functions. E.g. get the left or right value of the interval or change them for that matter. Likely we need to find the width (wid) of the interval, the radius (rad) of the interval, and the midpoint (mid) for converting an interval to a scalar element and we also need to define the required infimum (inf) and supremum (sup) plus the "is" method functions as required by the IEEE 1788 standard.

We also need some direct method to get and set the left and right interval plus getting or setting the interval type ([], (], [), []) and I use **leftinterval()** and **rightinterval()** for the interval ends

```
// Coordinate functions.
IT rightinterval() const;                // Return rightinterval bound
IT leftinterval() const;                 // Return leftinterval bound
IT rightinterval(const IT&);             // Set and return rightinterval bound
IT leftinterval(const IT&);       // Set and return leftinterval bound
enum interval_type intervaltype() const; // Return interval type
enum interval_type intervaltype(enum interval_type); // Set and return interval
type

// IEEE1788 standard functions
IT inf() const;     // Return infimum of interval
IT sup() const;     // Return supremum of interval
IT mid() const;     // Return midpoint of interval
IT rad() const;     // Return radius of interval
IT wid() const;     // Return width of interval
IT mig() const;     // Return Mignitude. inf(|x|)
IT mag() const;     // Return Magnitude. sup(|x|)

// Is methods as required per IEEE 1788 standard
bool isEmpty() const;
bool isPoint() const;
bool isImproper() const;
bool isProper() const;
```

Previously I have introduced the term interval class as a way to classify a given interval type. E.g. zero, positive, negative, mixed, etc. This can also be a helpful function to support. And the toString() function that generates a string representation of the interval.

```
// Miscellaneous but useful coordinate functions
enum int_class isClass() const;
std::string toString() const;    // Convert interval to a string
```

the enumerated type int_class is defined as follows:

```
// The eight different interval classification
// # ZERO            a=0 && b=0
// # POSITIVE0       a==0 && b>0
// # POSITIVE1       a>0 && b>0
// # POSITIVE        a>=0 && b>0
// # NEGATIVE0       a<0 && b==0
// # NEGATIVE1       a<0 && b<0
// # NEGATIVE        a<0 && b<=0
// # MIXED           a<0 && b>0
enum int_class { NO_CLASS, ZERO, POSITIVE0, POSITIVE1, POSITIVE, NEGATIVE0,
                 NEGATIVE1, NEGATIVE, MIXED };
```

Interval Conversion operators

Is all about defining conversion operators to convert an interval to any of the built-in types in C++.

```
// Conversion Operators
operator short() const                  // Conversion to short
operator int() const                    // Conversion to int
operator long() const                   // Conversion to long
operator long long() const              // Conversion to long
operator unsigned short() const         // Conversion to unsigned short
operator unsigned int() const           // Conversion to unsigned int
operator unsigned long() const          // Conversion to unsigned long
operator unsigned long long() const     // Conversion to unsigned long
operator double() const;                // Conversion to double
operator float() const;                 // Conversion to float
```

Interval Essential operators

This is all the C++ assignments operators, see below.

```
// Essential operators
interval<IT>& operator= ( const interval<IT>& );
interval<IT>& operator+=( const interval<IT>& );
interval<IT>& operator-=( const interval<IT>& );
interval<IT>& operator*=( const interval<IT>& );
interval<IT>& operator/=( const interval<IT>& );
```

Interval non-essential operators

These are the +,-,*,/ and the monadic version of +,- plus the Boolean ==,!=,>,<,>=,<= operators.

```
// Arithmetic +,-,*,/ Binary and Unary
template<class IT> interval<IT> operator+( const interval<IT>&, const
interval<IT>& );
template<class IT> interval<IT> operator+( const interval<IT>& );    // Unary
template<class IT> interval<IT> operator-( const interval<IT>&, const
interval<IT>& );
template<class IT> interval<IT> operator-( const interval<IT>& );    // Unary
template<class IT> interval<IT> operator*(const interval<IT>&, const
interval<IT>&);
template<class IT> interval<IT> operator/( const interval<IT>&, const
interval<IT>& );
```

with the class declaration defined and the outer declaration in place, we can now move to the implementation of it.

Implementing the interval class

Most of the implementation is simple and follows the class definition.

1. Constructor's
2. Coordinate methods
3. Conversion operators
4. Essential operators

Implementing the constructor's

The constructors are simple and just populate the interval class with initial values. If no parameters are given the interval will be initialized as an EMPTY (∅) interval. Notice that for the 3 arguments constructor, the third argument is defaulted to a closed interval, if not present.

```
// Construct an empty interval
template<class IT> inline  interval<IT>::interval()
       :left(IT(0)), right(IT(0)), type(EMPTY) {}      // Set EMPTY interval type

// Construct a singleton interval
// Since IT is either float or double and the argument is of the same type
// we can't catch any conversion error for up or down-conversion of the argument
template<class IT> inline  interval<IT>::interval(const IT& d)
       : left(d), right(d), type(CLOSE) {}      // Default is closed type

// Construct a regular interval
// Since IT is either float or double and the argument is of the same type
// we can't catch any conversion error for up or down-conversion of the argument
template<class IT> inline  interval<IT>::interval(const IT& l, const IT& h, const
enum interval_type t)
       : left(l), right(h), type(t) {}
```

Implementing the coordinate functions (methods)

These are all the methods as defined in the interval class. Most of these methods are relatively easy to implement. For the methods we have

 rightinterval(), leftinterval(), intervaltype(). These are simple functions that just return the left, right, or type of interval. There is no computation needed and it just returns the current values. There is also an overloaded version that allows you to set these private variables. There is also the isClass() and toString() methods.

 The next group is the required IEEE1788 standard functions

- inf()
- sup()
- mid()
- rad()
- wid()
- mig()
- mag()
- isEmpty()
- isPoint()
- isImproper()
- isProper()

Two methods inf() and sup() require a little explanation.

The inf() function stands for "infimum" or "greatest lower bound (GLB)." It returns the lowest possible value in an interval, effectively representing the lower bound of the interval. This function is crucial for ensuring that any operations or comparisons involving intervals take into account the lowest possible value that an element of the interval can assume, thus maintaining accuracy and reliability in mathematical computations. This means that regardless of whether the interval is proper (left≤right) or improper (left>right) it still returns the infimum of the interval. This is also valid regardless of whether the interval is closed, open, or semi-open.

The sup() function stands for "supremum" or "least upper bound (LUB)." It returns the highest possible value in an interval, effectively representing the upper bound of the interval. Similarly, the sup() function ensures that the highest possible value within an interval is considered during operations or comparisons. This is vital for encompassing the full range of possible values within an interval and for accurate interval arithmetic computations. Again regardless of whether the interval is proper (left≤right) or improper (left>right) it still returns the infimum of the interval. This is also valid regardless of whether the interval is closed, open, or semi-open

In interval arithmetic, these functions are used to define intervals (e.g., [inf(x), sup(x)] for an interval x) and to perform arithmetic operations between intervals. When operations such as addition, subtraction, multiplication, or division are performed between intervals, the inf() and sup() functions help determine the resulting interval by calculating the new bounds based on the operation performed and in consideration of the interval type and properness of the interval.

```
// Coordinate functions.

// Return the right interval "as is"
template<class IT> inline IT interval<IT>::rightinterval() const
{ return right; }

// Return the left interval "as is"
template<class IT> inline IT interval<IT>::leftinterval() const
{ return left; }
```

```
// Set the right interval "as is".
// If an interval is empty set the interval type to CLOSE
template<class IT> inline IT interval<IT>::rightinterval(const IT& r)
{
	if (type == EMPTY)
		type = CLOSE;
	right = r;
	return right;
}

// Set the left interval "as is".
// If an interval is empty set the interval type to CLOSE
template<class IT> inline IT interval<IT>::leftinterval(const IT& l)
{
	if (type == EMPTY)
		type = CLOSE;
	left = l;
	return left;
}

// Return the current intervaltype
template<class IT> inline enum interval_type interval<IT>::intervaltype() const
{ return type; }

// Set the interval type
// The below table is for a proper interval
//                    CLOSE []      OPEN ()       LEFT_OPEN (]  RIGHT_OPEN [)
//      CLOSE  []     #             -,+           -,#           #,+
//      OPEN   ()     +,-           #             #,-           +,#
//      LEFT_OPEN (]  +,#           #,+           #             +,+
//      RIGHT_OPEN [) #,-           -,#           -,-           #
//
//      For improper intervals we preserve the improperness in the result.
//
template<class IT> inline enum interval_type interval<IT>::intervaltype(const
enum interval_type to)
{
	interval<IT> x = *this;
	const IT infi(INFINITY);

	if (x.type != to)
	{
		if (this->isImproper())
		{	// If improper swap the interval and switch the half-open
intervals
			std::swap(x.left, x.right);
			if (x.type == LEFT_OPEN)
				x.type = RIGHT_OPEN;
			else
				if (x.type == RIGHT_OPEN)
					x.type = LEFT_OPEN;
		}

		switch (x.type)
		{
		case CLOSE:
			// if the interval is already CLOSE then do nothing
			if (to == LEFT_OPEN || to == OPEN)
				x.left = nextafter(x.left, -infi);
			if (to == RIGHT_OPEN || to == OPEN)
```

```
                                x.right = nextafter(x.right, +infi);
                        break;
                case OPEN:
                        // if the interval is already Open then do nothing
                        if (to == RIGHT_OPEN || to == CLOSE)
                                x.left = nextafter(x.left, +infi);
                        if (to == LEFT_OPEN || to == CLOSE)
                                x.right = nextafter(x.right, -infi);
                        break;
                case LEFT_OPEN:
                        // If the interval is already RIGHT_CLOSE same as LEFT_OPEN
then do nothing
                        if (to == RIGHT_OPEN || to == CLOSE)
                                x.left = nextafter(x.left, +infi);
                        if (to == RIGHT_OPEN || to == OPEN)
                                x.right = nextafter(x.right, +infi);
                        break;
                case RIGHT_OPEN:
                        // If the interval is already LEFT_CLOSE then do nothing
                        if (to == LEFT_OPEN || to == OPEN)
                                x.left = nextafter(x.left, -infi);
                        if (to == LEFT_OPEN || to == CLOSE)
                                x.right = nextafter(x.right, -infi);
                        break;
                }
                x.type = to;
                if (this->isImproper())
                {
                        std::swap(x.left, x.right);
                }
                *this = x;
        }
        return x.type;
}

// compute the infimum(greatest lower bound) of an interval, taking into account
the type of interval
// (closed, open, left-open, or right-open) and whether the interval is proper
// (left endpoint is less than or equal to right endpoint) or improper(left
endpoint is greater than the right endpoint).
template<class IT> inline IT interval<IT>::inf() const
{
        // For a closed interval, directly return the minimum of left and right.
        if (type == CLOSE)
                return min(left, right);

        IT adjustedLeft = left;
        IT adjustedRight = right;
        const IT infi(INFINITY);

        // Adjust left boundary for open intervals
        if (type == LEFT_OPEN || type == OPEN)
                adjustedLeft = nextafter(left, (left <= right) ? +infi : -infi);

        // Adjust right boundary for open intervals, taking into account improper
intervals
        if (type == RIGHT_OPEN || type == OPEN)
                adjustedRight = nextafter(right, (left <= right) ? -infi : +infi);

        // Return the minimum of the adjusted boundaries
        return std::min(adjustedLeft, adjustedRight);
```

```
}

// Optimizing the function for calculating the supremum(least upper bound) of an
interval follows
// a similar approach to optimizing the infimum calculation
template<class IT> inline IT interval<IT>::sup() const
{
       // For a closed interval, directly return the maximum of left and right.
       if (type == CLOSE)
              return max(left, right);

       IT adjustedLeft = left;
       IT adjustedRight = right;
       const IT infi(INFINITY);

       // Adjust left boundary for open intervals
       if (type == LEFT_OPEN || type == OPEN)
              adjustedLeft = nextafter(left, (left <= right) ? +infi : -infi);

       // Adjust right boundary for open intervals, considering proper and
improper intervals
       if (type == RIGHT_OPEN || type == OPEN)
              adjustedRight = nextafter(right, (left <= right) ? -infi : +infi);

       // Return the maximum of the adjusted boundaries
       return std::max(adjustedLeft, adjustedRight);
}

// Return interval midpoint
template<class IT> inline IT interval<IT>::mid() const
{ if (right == left) return left; else  return (right + left) / IT(2); }

// Return interval radius
template<class IT> inline IT interval<IT>::rad() const
{ IT r; r = (right - left) / IT(2); return r; }

// Return interval width
template<class IT> inline IT interval<IT>::wid() const
{ IT r; r = right - left; if (r < IT(0)) r = -r; return r; }

// Return mignitude of class
template<class IT> inline IT interval<IT>::mig() const
{
       IT l = inf();
       IT r = sup();
       l = abs(l);
       r = abs(r);
       return std::min(l, r);
}

// Return magnitude of interval
template<class IT> inline IT interval<IT>::mag() const
{
       IT l = inf();
       IT r = sup();
       l = abs(l);
       r = abs(r);
       return std::max(l, r);
}

// Required is... methods
```

```
template<class IT> inline bool interval<IT>::isProper() const
{ return left<=right; }
template<class IT> inline bool interval<IT>::isImproper() const
{ return right<left; }
template<class IT> inline bool interval<IT>::isPoint() const
{ return left==right; }
template<class IT> inline bool interval<IT>::isEmpty() const
{ return type==EMPTY; }

// Return interval classification
template<class IT> inline enum int_class interval<IT>::isClass() const
{
	if (left == IT(0) && right == IT(0)) return ZERO;
	if (left == IT(0) && right >  IT(0)) return POSITIVE0;
	if (left >  IT(0) && right >  IT(0)) return POSITIVE1;
	if (left >= IT(0) && right >  IT(0)) return POSITIVE;
	if (left <  IT(0) && right == IT(0)) return NEGATIVE0;
	if (left <  IT(0) && right <  IT(0)) return NEGATIVE1;
	if (left <  IT(0) && right <= IT(0)) return NEGATIVE;
	if (left <  IT(0) && right >  IT(0)) return MIXED;
	return NO_CLASS;
}

// Return the interval as a String representation
template<class IT> inline std::string interval<IT>::toString() const
{
	std::string s;
	enum interval_type t = intervaltype();
	std::ostringstream strs;

	strs.precision(25);
	strs << (t == LEFT_OPEN || t == OPEN ? "(" : "[");
	strs << left;
	strs << ",";
	strs << right;
	strs << (t == RIGHT_OPEN || t == OPEN ? ")" : "]");

	return strs.str();
}
```

Implementing the conversion operators

It is trivial to implement the conversion operators since it just has to return the midpoint of the interval when conversion to any integers type or float and double types. If the interval is empty the conversion is undefined.

```
// Conversion Operators
template<class IT> inline interval<IT>::operator short() const
{	// Conversion to short
	return static_cast<short>(mid());
}
template<class IT> inline interval<IT>::operator int() const
{	// Conversion to int
	return static_cast<int>(mid());
}

template<class IT> inline interval<IT>::operator long() const
```

```
{       // Conversion to long
        return static_cast<long>(mid());
}
template<class IT> inline interval<IT>::operator long long() const
{       // Conversion to long long
        return static_cast<long long>(mid());
}

template<class IT> inline interval<IT>::operator unsigned short() const
{       // Conversion to unsigned short
        return static_cast<unsigned short>(mid());
}
template<class IT> inline interval<IT>::operator unsigned int() const
{       // Conversion to unsigned int
        return static_cast<unsigned int>(mid());
}

template<class IT> inline interval<IT>::operator unsigned long() const
{       // Conversion to unsigned long
        return static_cast<unsigned long>(mid());
}

template<class IT> inline interval<IT>::operator unsigned long long() const
{       // Conversion to long long
        return static_cast<unsigned long long>(mid());
}

template<class IT> inline interval<IT>::operator double() const
{       // Conversion to double
        return static_cast<double>(mid());
}
template<class IT> inline interval<IT>::operator float() const
{       // Conversion to float
        return static_cast<float>(mid());
}
```

Implementing the basic operators

Now that we can control rounding via software emulation it is pretty straightforward to implement interval arithmetic in a portable way. An assignment is just a copy from the right to the left side of the = operator.

```
// Assignment operator. Works for all class types
//
template<class IT> inline interval<IT>& interval<IT>::operator=( const
interval<IT>& a )
{
        left = a.left;
        right = a.right;
        type = a.type;
        return *this;

}
```

Interval Addition

The Interval addition is:

$$[\underline{a},\overline{b}]+[\underline{c},\overline{d}]=[\underline{a+c},\overline{b+d}]$$

Code for addition using the fastwo_sum. Where .inf() returns the lowest number of the interval. And .sup() returns the highest number in the interval.

```
// += operator. Works for all classes.
// Always return a "proper" and closed [] interval
// a:=a+[EMPTY] or b:=[EMPTY]+b or [EMPTY]:=[EMPTY]+[EMPTY]
template<class IT> inline interval<IT>& interval<IT>::operator+=(const
interval<IT>& a)
{
	// Handle EMPTY interval first
	if (a.type == EMPTY)
		return *this;
	if (type == EMPTY)
		return (*this = a);

	const IT infi(INFINITY);
	// Neither a or b is [EMPTY]
	std::pair<IT, IT> xlow = fasttwo_sum(inf(), a.inf());
	std::pair<IT, IT> xright = fasttwo_sum(sup(), a.sup());
	left = xlow.first;
	right = xright.first;
	// Any adjustment?
	if (xlow.second < IT(0))
		left = nextafter(left, -infi);
	if (xright.second > IT(0))
		right = nextafter(right, +infi);
	type = CLOSE;
	return *this;
}
```

Interval Subtraction

The Interval subtraction is:

$$[\underline{a},\overline{b}]-[\underline{c},\overline{d}]=[\underline{a-d},\overline{b-c}]$$

Code for subtraction using the *fasttwo_sum*.

```
// -= operator. Works for all classes.
// Always return a "proper" and closed [] interval
// a=a-[EMPTY] or -b=[EMPTY]-b or [EMPTY]=[EMPTY]-[EMPTY]
//
template<class IT> inline interval<IT>& interval<IT>::operator-=(const
interval<IT>& a)
{
	// Handle EMPTY interval first
	if (a.type == EMPTY)
		return *this;
	if (type == EMPTY)
		return (*this = -a);
	const IT infi(INFINITY);
	// Neither a or b is [EMPTY]
	std::pair<IT, IT> xlow = fasttwo_sum(inf(), -a.sup());
	std::pair<IT, IT> xright = fasttwo_sum(sup(), -a.inf());
	left = xlow.first;
	right = xright.first;
	if (xlow.second < IT(0))
		left = nextafter(xlow.first, -infi);
	if (xright.second > IT(0))
		right = nextafter(xright.first, +infi);
	type = CLOSE;
	return *this;
}
```

Interval Multiplication

The interval multiplication is:

$$[\underline{a},\overline{b}] * [\underline{c},\overline{d}] = [\min\underline{(ac,ad,bc,bd)}, \max\overline{(ac,ad,bc,bd}])$$

Code for multiplication using the *fasttwo_product* algorithm and with some optimization.

```
// Works for all classes.
// [EMPTY]:=a*[EMPTY] or [EMPTY]:=[EMPTY]*b or [EMPTY]:=[EMPTY]*[EMPTY]
// Please note that this is for all integer classes. interval<int>, 
interval<long>,
// were there is no loss of precision
// Instead of doing the mindless low = MIN(low*a.right, 
low*a.low,right*a.low,right*a.right) and
// right = MAX(low*a.right, low*a.low,right*a.low,right*a.right) requiring a 
total of 8 multiplication
//
//   low, right, a.low, a.right    result
//    +     +     -     +        -  +  [ right*a.low, right*a.right ] 2205
//    +     +     -     -        -  -  [ right*a.low, low*a.right ]
//    +     +     +     +        +  +  [ low*a.low, right*a.right ]
//    -     +     +     +        -  +  [ low*a.right, right*a.right ]
//    -     +     -     +        -  +  [ MIN(low*a.right,right*a.low), 
MAX(low*a.low,right*a.right) ]
//    -     +     -     -        -  -  [ right*a.low, low*a.low ]
//    -     -     +     +        -  -  [ low*a.right, right,a.low ]
//    -     -     -     +        -  -  [ low*a.right, low*a.low ]
//    -     -     -     -        +  +  [ right*a.right, low * a.low ]
//
template<class IT> inline interval<IT>& interval<IT>::operator*=(const 
interval<IT>& a)
{
	// Handle EMPTY interval first  Ø
	if (type == EMPTY)
		return *this;
	if (a.type == EMPTY)
		return (*this = a);

	// Neither a or b is Ø
	// Extract intervals through inf() and sup()
	IT al = inf();
	IT ah = sup();
	IT bl = a.inf();
	IT bh = a.sup();

	auto multiply = [&](const IT& x, const IT& y)
	{
		std::pair<IT, IT> tmp = interval<IT>::fasttwo_prod(x, y);
		interval<IT> res;
		const IT infi(INFINITY);
		res.left = res.right = tmp.first;
		if (tmp.second < IT(0))
			res.left = nextafter(tmp.first, -infi);
		if (tmp.second > IT(0))
			res.right = nextafter(tmp.first, +infi);
		return res;
	};

	interval<IT> itmp;
	type = CLOSE;
	// Shortcuts
	if (al >= IT(0) && bl >= IT(0))
	{// Both intervals positive
		itmp = multiply(al, bl);
		left = itmp.left;
		itmp = multiply(ah, bh);
		right = itmp.right;
		return *this;
	}
	if (ah < IT(0) && bh < IT(0))
	{// Both intervals negative
		itmp = multiply(al, bl);
		right = itmp.right;
		itmp = multiply(ah, bh);
		left = itmp.left;
		return *this;
	}
	if (al >= IT(0) && bh < IT(0))
	{// [A] interval positive, [B] interval negative
		itmp = multiply(ah, bl);
		left = itmp.left;
		itmp = multiply(al, bh);
		right = itmp.right;
		return *this;
	}
	if (ah < IT(0) && bl >= IT(0))
	{// [A] interval negative, [B] interval positive
		itmp = multiply(al, bh);
		left = itmp.left;
		itmp = multiply(ah, bl);
		right = itmp.right;
		return *this;
	}
	// Otherwise, we have a mixed interval. Make all 4 combinations
	itmp = multiply(al, bl);
	left = itmp.left;
	right = itmp.right;
	itmp = multiply(al, bh);
	left = min(left, itmp.left);
	right = max(right, itmp.right);
	itmp = multiply(ah, bl);
	left = min(left, itmp.left);
	right = max(right, itmp.right);
```

```
        itmp = multiply(ah, bh);
        left = min(left, itmp.left);
        right = max(right, itmp.right);

        return *this;
}
```

Interval Division

The interval division is:

$$\frac{[\underline{a},\overline{b}]}{[\underline{c},\overline{d}]} = [\underline{a},\overline{b}] * [\frac{1}{\underline{d}},\frac{\overline{1}}{c}]$$

```
// Works for all classes
// [EMPTY]:=a/[EMPTY] or [EMPTY]:=[EMPTY]/b or [EMPTY]:=[EMPTY]/[EMPTY]
// Please note that this is for all integer classes. interval<int>,interval<long>
// where there is no loss of precision
template<class IT> inline interval<IT>& interval<IT>::operator/=(const
interval<IT>& b)
 {
        const IT infi(INFINITY);
        // Handle EMPTY interval first
        if (type == EMPTY)
               return *this;
        if (b.type == EMPTY)
               return (*this = b);

        interval<IT> tmp = *this;  // Save a copy
        // Compute the reverse of y e.g. 1/y
        auto inverse = [&](const IT& y, const bool up)
        {
               IT res = IT(1) / y, r;

               r = -fma(res, y, IT(-1));
               if (up == false)
               {
                      if (r < IT(0))
                             res = nextafter(res, -infi);
               }
               else
               {
                      if (r > IT(0))
                             res = nextafter(res, +infi);
               }

               return res;
        };

        IT bl = b.inf();
        IT bh = b.sup();
        if (bl == IT(0) && bh == IT(0))
        {
               left = -infi;
               right = +infi;
```

```
                *this *= tmp;
                return *this;
        }
        if (bh == IT(0))
        {     // b.low is !=0
                right = inverse(bl, true);
                left = -infi;
                *this *= tmp;
                return *this;
        }
        if (bl == IT(0))
        {     // b.right is !=0
                left = inverse(bh, false);
                right = +infi;
                *this *= tmp;
                return *this;
        }
        // neither b.low or b.right is zero
        left = inverse(bh, false);
        right = inverse(bl, true);
        *this *= tmp;
        return *this;
 }
```

Implementation of monadic and dyadic arithmetic operators

We simply compute the dyadic operators with the use of the essential operators to simplify the code.

```
// Binary + operator specialization for only interval<IT> arguments
// Works for all classes
//
template<class IT> inline interval<IT> operator+(const interval<IT>& a, const
interval<IT>& b)
{
       interval<IT> result(a);
       result += b;
       return result;
}

// Unary + operator
// Works for all classes
//
template<class IT> inline interval<IT> operator+(const interval<IT>& a)
{
       return a;
}

// Binary - operator
// Works for all classes
//
template<class IT> inline interval<IT> operator-(const interval<IT>& a, const
interval<IT>& b)
{
       interval<IT> result(a);
       result -= b;
       return result;
}
```

```
// Unary - operator
// Works for all classes
//
template<class IT> inline interval<IT> operator-(const interval<IT>& a)
{
	interval<IT> result(0);
	result -= a;
	return result;
}

// Binary * operator
// Works for all classes
//
template<class IT> inline interval<IT> operator*(const interval<IT>& a, const
interval<IT>& b)
{
	interval<IT> result(a);
	result *= b;
	return result;
}

//Binary / operator
// Works for all classes
//
template<class IT> inline interval<IT> operator/(const interval<IT>& a, const
interval<IT>& b)
{
	interval<IT> result(a);

	if (result == b)
	{
		enum int_class intclass = b.isClass();
		if (intclass != ZERO && intclass != POSITIVE0 && intclass !=
NEGATIVE0)
		{
			result = interval<IT>(1, 1);
			return result;
		}
	}
	result /= b;
	return result;

}
```

Implementation of the comparison operators

In this section, we handle all the 6 comparison operators. ==,!=, >=, <=, >, <. Notice that we also need to be able to handle the empty interval ∅. If both intervals are ∅, then we return false. If one of the intervals is ∅ then we return true. Notice that some implementations return undefined when both intervals are ∅.

```
// == operator
// Works for all classes
//
template<class IT> inline bool operator==(const interval<IT>& a, const
interval<IT>& b)
{
	if (a.intervaltype() == EMPTY && b.intervaltype() == EMPTY)
		return false; // Both EMPTY=> return false
	if (a.intervaltype() == EMPTY || b.intervaltype() == EMPTY)
		return true;  // One but not both are EMPTY => return true
	// Check for equality. By using inf() and sup() we do not have to worry
	about Improper interval
	// or interval type != CLOSE
	return a.inf() == b.inf() && a.sup() == b.sup();
}

// != operator
// Works for all classes
//
template<class IT> inline bool operator!=(const interval<IT>& a, const
interval<IT>& b)
{
	return !(a == b);
}
// >= operator
// Works for all classes
//
template<class IT> inline bool operator>=(const interval<IT>& a, const
interval<IT>& b)
{
	if (a.intervaltype() == EMPTY && b.intervaltype() == EMPTY)
		return false;
	if (a.intervaltype() == EMPTY || b.intervaltype() == EMPTY)
		return true;
	if (a.inf() >= b.sup())
		return true;
	return false;
}
// <= operator
// Works for all classes
//
template<class IT> inline bool operator<=(const interval<IT>& a, const
interval<IT>& b)
{
	if (a.intervaltype() == EMPTY && b.intervaltype() == EMPTY)
		return false;
	if (a.intervaltype() == EMPTY || b.intervaltype() == EMPTY)
		return true;
	if (a.sup() <= b.inf())
		return true;
	return false;
}

// > operator
// Works for all classes
//
template<class IT> inline bool operator>(const interval<IT>& a, const
interval<IT>& b)
{
	if (a.intervaltype() == EMPTY && b.intervaltype() == EMPTY)
		return false;
	if (a.intervaltype() == EMPTY || b.intervaltype() == EMPTY)
		return true;
	if (a.inf() > b.sup() )
		return true;
	return false;
}
```

```
// < operator
// Works for all classes
//
template<class IT> inline bool operator<(const interval<IT>& a, const
interval<IT>& b)
{
	if (a.intervaltype() == EMPTY && b.intervaltype() == EMPTY)
		return false;
	if (a.intervaltype() == EMPTY || b.intervaltype() == EMPTY)
		return true;
	if (a.sup() < b.inf())
		return true;
	return false;

}
```

Implementing the sqr(x) and sqrt(x) functions

We need to add a special code for handling x^2. See previous sections. For the sqrt(x) we again can use the fma() library function to quickly bound the interval.

```
// sqr(x)=x^2
template<class IT> inline interval<IT> sqr(const interval<IT>& x)
{
	IT left = x.inf();
	IT right = x.sup();
	IT tmpl = left;
	IT tmpr = right;
	interval<IT> r(0);

	left *= left;
	right *= right;
	r.rightinterval(max(left, right));
	// Contained zero?
	if ( tmpl > 0 && tmpr > 0)
		r.leftinterval(min(left, right));
	return r;
}

// sqrt(x)
template<class IT> inline interval<IT> sqrt(const interval<IT>& x)
{
	const IT infi(INFINITY);
	IT sq, r;
	IT left, right;

	// Find leftinterval bound
	sq = sqrt(x.inf());
	left = sq;
	r = -fma(sq, sq, -x);
	if (r < IT(0))
		left = nextafter(left, -infi);

	// Find rightinterval bound
	sq = sqrt(x.sup());
	right = sq;
	r = -fma(sq, sq, -x);
```

```
        if (r > IT(0))
               right = nextafter(right, +infi);
        return interval<IT>(left, right);
}
```

Implementing the abs(x) functions

In interval arithmetic, the absolute value (abs) of an interval is defined in a way that captures all possible absolute values of any number within the interval. The result is always a non-negative interval. The definition depends on the position of the interval concerning zero:

If the interval is entirely non-negative (i.e., both its lower and upper bounds are greater than or equal to zero), then the absolute value of the interval is the interval itself, since all values within it are already non-negative.

If the interval is entirely negative (i.e., both its lower and upper bounds are less than zero), then the absolute value of the interval is obtained by taking the absolute values of its bounds and swapping them (since the lower bound will be the most negative and, thus, have the highest absolute value).

If the interval spans zero (i.e., its lower bound is negative and its upper bound is positive), then the absolute value of the interval is from zero to the maximum of the absolute values of the lower and upper bounds.

Source code for abs(x)

```
// abs([a,b])
// if a>=0 in [a,b] then |[a,b]|==[a,b]
// if b<0 in [a,b] then |[a,b]|=[-b,-a]
// if a<0 & b>0 in [a,b] then |[a,b]|=[0,max(-a,b)]
template<class IT> inline interval<IT> abs( const interval<IT>& a )
        {
        if (a.inf() >= IT(0) ) // entirely positive
               return a;
        else
               if (a.sup() < IT(0) ) // Entirely negative
                      return -a;

        return interval<IT>(IT(0), max(-a.inf(),a.sup()));
        }
```

Implementation of the cin & cout operators

The standard output **cout** and input **cin** are straightforward. **cout** output the interval in the format of:

[left, right]

When the interval is closed:

(left, right] or [left, right) for semi-open intervals and

And

(left, right) for fully open intervals.

If the interval is empty an ∅ is outputted.

While **cin** can handle a little variation to the above. Any syntax for the input is legal

[left, right]
[interval] which is equivalent to [interval, interval]

you can mix in '(' or ')' instead of square bracket '[' or ']' to indicate semi or full-open intervals.

Phase One conclusion

This was the goal for the first phase which defined and implemented all the base base-level functionality that is needed for the next two phases.

However, we do have a shortcoming. Where the result is not completely accurate. This has to do with the constructor. The constructor parameter is of the template type IT (either float, double, or long double). As per C++ standard rules when an actual parameter is of a lesser type than IT then an automatic conversion is applied to the desired IT type. However, the conversion happens before executing the constructor and therefore any inaccuracies in the conversion are not caught by the interval class. An example will clarify the issue.

```
Interval<float> x(16777216);
Interval<float> y(16777217);
```

Will create the interval [16777216.0,16777216.0] for both x and y. The issue is that the integer 16777217 is larger than what accurately can be converted to a float 23-bit mantissa while 16777216 just fits in.

The same is true for converting a 64-bit double to a 32-bit float type. Even for an interval<double> variable a 64-bit integer larger than 9,007,199,254,740,992 will not convert correctly to a double (a double mantissa is 52-bit.

In phase 2 we will introduce mixed data type which magically allows us to address and fix the above issue.

This concludes Phase One of the practical implementation of an interval template library. We now have the basic functionality established and we can continue implementing Phase Two and Phase Three.

Phase Two implementation

In phase 2 we expand on phase one functionality. One of the most important ones is the handling of mixed data type in all the interval arithmetic operations plus handling mixed data type in constructor's

mixed data type for arithmetic operations

This is important and it solved an issue with the Phase One implementation. E.g. If you try this:

```
	interval<float> f(16777216);
```

The float interval f is correct [16777216.0,16777216.0]. However, if you try this:

```
	interval<float> f(16777217);
```

The float interval f is incorrect [16777216.0,16777216.0]. If you try the next integer:

```
	interval<float> f(16777218);
```

The float interval f is again correct [16777218.0,16777218.0].

We simply do not catch that an integer larger than 16777216 cannot always be represented correctly in the 32-bit float format. (mantissa is 23-bits).
The same goes for 64-bit integers above 9,007,199,254,740,992 converting to an interval<double> (mantissa is 52-bits).
The reason why phase one didn't catch this issue is that C++ always converts up or down to the requested constructor type before calling the constructor and therefore the constructor does not know that the argument is inaccurate.
By adding constructors that can handle different types we can catch any up or down converting so it can be handled correctly within the constructor. By letting the constructor handle the conversion we get the correct result for

```
	interval<float> f(16777217);
```

The float interval f is incorrect [16777216.0,16777218.0].

Lastly, we were also missing that we could use another interval as the constructor element for a new interval.
We therefore add the following constructors to the interval class.

```
// Constructor for mixed type IT != _X (base types).
// Allows auto construction of e.g. interval<double> x(float)
// Notice that the phase one constructor above is still valid when both the
// interval type IT and the argument is also of the same type
template<typename _X> interval(const _X&); // Singleton interval

// Regular interval with interval_type (default CLOSE)
```

```
template<typename _X, typename _Y> interval(const _X& , const _Y&, const enum
interval_type t=CLOSE);

// constructor for any other type to IT. Both up and down conversions
// are possible
// constructor for an interval<_X> argument
template<typename _X> interval(const interval<_X>& a);
```

We can now correct catch the original constructor type and ensure proper conversion to the interval.

```
// Constructor for creating mixed-type intervals when
// IT != _X (base types), enabling automatic
// construction of intervals from different types
// (e.g., interval<double> x = float).
// For initializations with integral types, check the value against the
// max without loss of precision,
// adjust the interval to ensure it fits within the bounds of float or
// double values.
// If float or double limits are exceeded, set the left interval to the
// correct lower bound,
// while the right interval is adjusted accordingly.
// Note: This template constructor is preferred over the simpler constructor
template<class IT> template<typename _X> inline interval<IT>::interval(const _X&
x)
{
	const bool integral = std::is_integral<_X>::value;
	const bool isIntegral = std::is_integral<_X>::value;
	const bool isTargetFloat = std::is_same<IT, float>::value;
	const bool isSourceDoubleOrLongDouble = std::is_same<_X, double>::value ||
std::is_same<_X, long double>::value;
	const IT infi(infinity_interval<IT>());// infi(INFINITY);

	// up promoting is accurate
	left = IT(x);
	right = IT(x);
	if(isTargetFloat&& isSourceDoubleOrLongDouble)
	{	// Downpromoting from double to float.
		// Uppromotion from float to double is always accurate
		auto adjustBoundaries = [&](const _X& val)
		{
			_X e = val - _X(left);
			if (e < _X(0)) left = nextafter(left, -infi);
			e = val - _X(right);
			if (e > _X(0)) right = nextafter(right, +infi);
		};
		adjustBoundaries(x);
	}
	if (isIntegral)
	{	// Handle integer promotion to IT
		const intmax_t absX = intmax_t(abs(x));
		auto maxFloat = 16'777'216; // 2^24
		auto maxDouble = 9'007'199'254'740'992; // 2^53
		bool exceedsFloat = isTargetFloat && absX > maxFloat;
		bool exceedsDouble = !isTargetFloat && absX > maxDouble;

		if (exceedsFloat || exceedsDouble)
		{
			_X e = x - _X(left);
			if (e > _X(0)) right = nextafter(right, +infi);
```

```
                        if (e < _X(0)) left = nextafter(left, -infi);
                }
        }
        type = CLOSE;
}

// Constructor for creating mixed type intervals (IT != _X, base types)
// with two or three arguments, facilitating the automatic construction of
// intervals from different types (e.g., interval<double> x = {float, float}).
// For initializations with integral types, it verifies the value against
// the maximum that can be handled
// without loss of precision, adjusting the interval to fit within the bounds of
float or double values.
// Should the float or double limits be exceeded, the left interval is set to the
correct lower bound,
// while the right interval is adjusted accordingly.
// This template function is preferred over the simpler constructor from
// phase one, except when the argument type matches the interval class type (IT).
// To simplify implementation, the single argument constructor is called twice.
// Then, the minimum of the two left intervals and the maximum of the two right
// intervals are determined to establish the final interval bounds.
template<class IT> template<typename _X, typename _Y> inline
interval<IT>::interval(const _X& x, const _Y& y, const enum interval_type t)
{
        const interval<IT> ll(x);   // Call mixed singleton constructor
        const interval<IT> rr(y);   // Call mixed singleton constructor

        left = min(ll.inf(), rr.inf());
        right = max(ll.sup(), rr.sup());
        type = t;
        //if x>y originally was improper then return it as an improper interval
        if (IT(x) > IT(y))
                std::swap(left, right);
        return;
}

// constructor for any other interval<_X> to interval<IT>.
// e.g. interval<float> i1(2,3);
// interval<float> i2(i1);
template<class IT> template<typename _X> inline interval<IT>::interval(const
interval<_X>& a) //
{
        // Call two argument mixed constructor
        const interval<IT> x(a.leftinterval(), a.rightinterval(),
a.intervaltype());
        left=x.left;
        right= x.right;
        type = x.type;
}

// Constructor for creating mixed type intervals (IT != _X, base types)
// with two or three arguments, facilitating the automatic construction of
// intervals from different types (e.g., interval<double> x = {float, float}).
// For initializations with integral types, it verifies the value against the
// maximum that can be handled without loss of precision, adjusting the
// interval to fit within the bounds of float or double values.
// Should the float or double limits be exceeded, the left interval is set
// to the correct lower bound, while the right interval is adjusted accordingly.
// This template function is preferred over the simpler constructor from
// phase one, except when the argument type matches the interval class
// type (IT).
// To simplify implementation, the single argument constructor is called
// twice. Then, the minimum of the two left intervals and the maximum of the
// two right intervals are determined to establish the final interval bounds.
template<class IT> template<typename _X, typename _Y> inline
interval<IT>::interval(const _X& x, const _Y& y, const enum interval_type t)
{
       const interval<IT> ll(x);  // Call mixed singleton constructor
       const interval<IT> rr(y);  // Call mixed singleton constructor

       left = std::min(ll.inf(), rr.inf());
       right = std::max(ll.sup(), rr.sup());
       type = t;
       //if x>y originally then return it as an improper interval
       if (x > y)
              std::swap(left, right);
       return;
}

// constructor for any other interval<_X> to interval<IT>.
// e.g. interval<float> i1(2,3);
// interval<float> i2(i1);
template<class IT> template<typename _X> inline interval<IT>::interval(const
interval<_X>& a) //
{
       // Call two argument mixed constructor
       const interval<IT> x(a.leftinterval(), a.rightinterval(),
a.intervaltype());
       left=x.left;
       right= x.right;
       type = x.type;
}
```

Interval functions

Here we have all the interval functions:

a. Union: Union of two intervals (join)
b. Intersection: Intersections of two intervals
c. Interior: Is the interval an interior to another interval
d. Precedes: Does the interval precede another interval
e. Subset: Is the interval a subset of another interval
f. Inclusion: Is the interval included in another interval
g. In: Is the float or double number within the interval

We have an interesting issue with the union function. The name 'union' is a keyword in C (C++) and can therefore not be used. Instead, we used the word join for interval unions. A special note on the union (join) function is that the standard prescribes that if the union of two intervals is disjoint then the function returns two intervals instead of one. I used the pair template to return two intervals. In case the two intervals are not disjoint then the second interval returned is the empty interval ∅.

The intersection as the name applied returns the intersection of the two intervals or the ∅ for the empty interval. Notice that intersection is also implemented using the & and &= operator.

```
// Return union
// The name join is used since union is a reserved word in C++
// It follows the IEEE 1788 standard by returning two intervals if the interval
// a and b are not connected
// otherwise, if return the joined interval and the second interval of the
// pair returned is the empty
// interval
// Notice the |= operates or the binary operator | returns the union of the
// two intervals by combining them into one interval
//
template<class IT> inline std::pair<interval<IT>, interval<IT> > join(const
interval<IT>& a, const interval<IT>& b)
{
	if (a.sup() < b.inf())  // interval do not connect
	{	// return two interval
		return std::make_pair <interval<IT>, interval<IT> >(a, b);
	}
	// Return the union of the two intervals.
	interval<IT> c(min(a.inf(),b.inf()),max(a.sup(),b.sup()));
	interval<IT> d; // Empty set
	return std::make_pair <interval<IT>, interval<IT> >(c, d);
}

// Return the interval intersection of the two intervals.
template<class IT> inline interval<IT> intersection(const interval<IT>& a, const
interval<IT>& b)
{
	interval<IT> c(a);
	c &= b;
	return c;
}

// if a is a subset of b then return true otherwise false
template<class IT> inline bool subset(const interval<IT>& a, const interval<IT>&
b)
{
	if (b.inf() <= a.inf() && a.sup() <= b.sup())
		return true;
	return false;
}

// if a is an interior of b then return true otherwise false
template<class IT> inline bool interior(const interval<IT>& a, const
interval<IT>& b)
{
	if (b.inf() < a.inf() && a.sup() < b.sup())
		return true;
	return false;
}

// if a precedes b then return true otherwise false
template<class IT> inline bool precedes(const interval<IT>& a, const
interval<IT>& b)
{
	if (a.sup() < b.inf() )
		return true;
	return false;
}

// inclusion between two intervals.
// If a is a subset of b then return +1,
// if b is a subset of a then return +1
// otherwise return 0
template<class IT> inline int inclusion(const interval<IT>& a, const
interval<IT>& b)
{
	if( subset(a, b))
		return -1;
	if (subset(b, a))
		return +1;
	return 0;
}
```

Manifest interval constants: LN2, LN10, e, PI

In computational mathematics and applications requiring interval arithmetic, the precision and accuracy of manifest constants such as π, e, LN2, and LN10 are paramount. Traditional approaches, which define these constants with fixed precision, often fall short in tasks demanding arbitrary precision levels. Furthermore, there is the issue that decimal representations cannot always be accurately defined in the IEEE754 standard. E.g. the value for π is 3.1459265…however, that number is not an exact representation in the float type in the IEEE754 standard. The closest number that can be done in the 32-bit float type is 3.14159250 or 3.14159274. Since the last number is the closest (Round to nearest) to the real π it will return the number 3.14159274 for the C++ float type. The lower part can then be found using the nextafter() function in the direction of -infinity to find the lower part 3.14159250. You can manually find all the other constants similarly for both the float and double type.

Doing so you would end up with the below very static code.

```
// The interval for pi with float accuracy
const interval<float> FPI_INTERVAL(3.141'592'50, 3.141'592'74);
// The interval for PI with double accuracy
const interval<double> DPI_INTERVAL(3.141'592'653'589'793'1,
3.141'592'653'589'793'6);

// The interval for e with float accuracy
const interval<float> FE_INTERVAL(2.718'281'75, 2.718'281'98);
// The interval for e with double accuracy
const interval<double> DE_INTERVAL(2.718'281'828'459'045'1,
2.718'281'828'459'045'5);

// The interval for LN2 with float accuracy
const interval<float> FLN2_INTERVAL(0.693'147'123, 0.693'147'182);
// The interval for e with double accuracy
const interval<double> DLN2_INTERVAL(0.693'147'180'559'945'29,
0.693'147'180'559'945'40);

// The interval for LN10 with float accuracy
const interval<float> FLN10_INTERVAL(2.302'584'89,2.302'585'12);
// The interval for LN10 with double accuracy
```

```
const interval<double>
DLN10_INTERVAL(2.302'585'092'994'045'5,2.302'585'092'994'045'9);
```

We now have access to both the left and right intervals of these constants, highlighted a static approach to defining manifest constants, fixed to float and double precision levels. While effective for standard applications, this method restricts the adaptability and precision enhancement essential for advanced computational tasks, especially in scientific computing where arbitrary precision is a necessity and particularly when considering templates function the static form is not ideal.

Instead, we can introduce a sophisticated methodology employing template factory functions, offering a versatile solution to generate these constants dynamically at any required precision.
The transition to template factory functions represents a paradigm shift in how we manage manifest constants within interval arithmetic frameworks. This approach leverages the power of templates in C++ to dynamically adjust the precision of constants according to the computational context.
Template factory functions are designed to generate constants at runtime with precision tailored to the specific needs of a calculation. This method stands in contrast to the static definition of constants, offering a more flexible and scalable solution.

The primary advantages of using template factory functions include:
Scalability of Precision where precision can be adjusted dynamically, enabling calculations to be as accurate as necessary without being constrained by predetermined limits. Enhanced Code Reusability and Maintainability where a single function can serve multiple data types and precision requirements, simplifying codebase management and enhancing readability finally Enhanced Accuracy, by allowing for higher precision levels, calculations involving manifest constants become more accurate, reducing the risk of errors in critical computations.

The core of this approach lies in the implementation of template factory functions for each manifest constant. Here is an in-depth look at how these functions are structured:

For each constant (π, e, LN2, LN10), a template factory function is defined. These functions check the type at compile time and return the appropriate interval based on the specified or implied precision.
For standard float and double, predefined intervals ensure compatibility with existing codebases. When turning to arbitrary precision types which is "float_precision" (explained in Phase four), the function dynamically calculates the constant to the specified precision, accounting for rounding to ensure the interval is accurate.
Precision control is a standout feature for arbitrary precision types. Users can specify the desired precision, allowing for a high degree of flexibility in computational tasks. The C++ code would be:

```
    pi_interval<float_precision>(50);
```

This call generates an interval for π with 50 decimal digits of precision, showcasing the function's ability to cater to specific precision needs.

For Standard Types like float and double the declaration would look like this:

```
auto pi_float = pi_interval<float>();
auto pi_double = pi_interval<double>();
```

Arbitrary Precision Types with Specified Precision:

```
auto pi_high_precision = pi_interval<float_precision>(100);
```

In comparison with the Fixed-Precision Approach, the flexibility and scalability of the template factory function approach significantly outperform the fixed-precision method, particularly in high-precision and dynamic computing environments. By allowing precision to be adjusted according to the computational requirements, this methodology not only enhances accuracy but also broadens the scope of possible applications.

The mentioned template factory function code is below for the standard types of float and double (Notice the use of **constexpr** allowing it to be handled at compile time:

```
// Get PI for standard float types
template<typename IT> constexpr interval<IT> pi_interval(const size_t precision =
float_precision_ctrl.precision())
{
	if constexpr (std::is_same<IT, float>::value)
		return interval<IT>(IT(3.141'592'50), IT(3.141'592'74));
	else if constexpr (std::is_same<IT, double>::value)
		return interval<IT>(IT(3.141'592'653'589'793'1),
IT(3.141'592'653'589'793'6));
	else
	static_assert(is_floating_point<IT>::value, "Unsupported type for
pi_interval.Type must be float, double.");
}

// Get e at the precision for IT.(float_precision also based on the precision)
template<typename IT> constexpr interval<IT> e_interval(const size_t precision =
float_precision_ctrl.precision())
{
	if constexpr (std::is_same<IT, float>::value)
		return interval<IT>(IT(2.718'281'75), IT(2.718'281'98));
	else if constexpr (std::is_same<IT, double>::value)
		return interval<IT>(IT(2.718'281'828'459'045'1),
IT(2.718'281'828'459'045'5));
	else
		static_assert(is_floating_point<IT>::value, "Unsupported type for
pi_interval.Type must be float, double.");
}

// Get e at the precision for IT.(float_precision also based on the precision)
template<typename IT> constexpr interval<IT> ln2_interval(const size_t precision
= float_precision_ctrl.precision())
{
	if constexpr (std::is_same<IT, float>::value)
		return interval<IT>(IT(0.693'147'123), IT(0.693'147'182));
	else if constexpr (std::is_same<IT, double>::value)
```

```
		return interval<IT>(IT(0.693'147'180'559'945'29), 
IT(0.693'147'180'559'945'40));
	else
		static_assert(is_floating_point<IT>::value, "Unsupported type for 
pi_interval.Type must be float, double.");
}

// Get e at the precision for IT.(float_precision also based on the precision)
template<typename IT> constexpr interval<IT> ln10_interval(const size_t precision 
= float_precision_ctrl.precision())
{
	if constexpr (std::is_same<IT, float>::value)
		return interval<IT>(IT(2.302'584'89), IT(2.302'585'12));
	else if constexpr (std::is_same<IT, double>::value)
		return interval<IT>(IT(2.302'585'092'994'045'5), 
IT(2.302'585'092'994'045'9));

	else
		static_assert(is_floating_point<IT>::, "Unsupported type for 
pi_interval.Type must be float, double.");
}
```

Logarithm functions (Log,log10)

There are two approaches you can take here. The first approach is to use the Taylor series to compute log(x).

The Taylor series for log(x) is:

$$\ln(x) = 2(\frac{x-1}{x+1} + \frac{1}{3}(\frac{x-1}{x+1})^3 + \frac{1}{5}(\frac{x-1}{x+1})^5 + \cdots)$$

If we use $z = \frac{x-1}{x+1}$ we get:

$$\ln(x) = 2(z + \frac{1}{3}z^3 + \frac{1}{5}z^5 + \cdots)$$

And with the below implementation using interval arithmetic. You would need to call the function twice for an interval. E.g.

```
	Interval<double> x(2.5, 3.5);
	Interval<double> resleft, resright;

	Resleft=interval_log(x.inf());
	Resright=interval_log(x.sup());
	Return interval<double>( min(resleft.inf(),resright.inf()),
	                         max(resleft.sup(),resright.sup()));
```

```
static interval<double> interval_log(double x)
```

```
{
	// First handle the shortcuts
	if (x < 0) { return interval<double>(NAN, NAN); }
	if (x == 0) { return interval<double>(-INFINITY, -INFINITY); }
	if (x == 1) { return interval<double>(0); }
	if (x == 2) { return interval<double>(DLN2_INTERVAL); }
	if (x == 10) { return interval<double>(DLN10_INTERVAL); }

	const interval<double> c1(1.0);
	interval<double> zn, zsq, sum, delta;
	int i, exponent;
	// Split the significant and exponent
	x = frexp(x, &exponent);
	if (x == 0.5)
	{// True Power of two. More accurate to just multiply (exponent-1) * LN2
		zn = DLN2_INTERVAL;
		zn *= interval<double>(exponent - 1);
		return zn;
	}
	zn = interval<double>(x);
	// Taylor series of log(x)
	// log(x)=2( z + z^3/3 + z^5/5 ...)
	// where z=(x-1)/(x+1)
	// The fraction part is [0.5,1] (base 2) after removing the exponent
	// Initialize the iteration (zn-1)/(zn+1)
	zn = (zn - c1) / (zn + c1);
	zsq = zn * zn;
	sum = zn;

	// Iterate using Taylor series log(x) == 2( z + z^3/3 + z^5/5 ... )
	for (i = 3;; i += 2)
	{
		zn *= zsq;
		delta = zn / interval<double>(i);
		if (sum.mid() + delta.mid() == sum.mid())
			break;
		sum += delta;
	}
	sum *= interval<double>(2.0);// Finally multiply with 2

	if (exponent != 0)
	{	// restore the exponent
		sum += interval<double>(exponent) * DLN2_INTERVAL;
	}
	return sum;
}
```

This approach functions well but returns a larger interval than is strictly necessary.

The next approach is to recognize that the standard library log(x) function is rounded to the nearest result. We can assume that the result is always within 1 ULP (ULP stands for *Units in the Last Place*. It is a measure used to express the precision and accuracy of floating-point calculations. Specifically, it quantifies the difference between a calculated floating-point number and the nearest representable value in the floating-point system being used for the correct result).

The ULP represents the error bound of the accuracy of mathematical functions in C++, the ULP can help define the maximum error margin. For instance, if an elementary function guarantees an accuracy of 1 ULP, it means the result will be no more than one floating-point representation away from the exact (as per the floating-point system) result. To finalize the interval around these elementary functions we then just add the next available floating-point number up and down and then return the result. This approach turns out to get much tighter bounds compared with using a Taylor expansion for log(x).

```
// log(x)
template<class IT> inline interval<IT> log(const interval<IT>& x)
{
	const IT infi(infinity_interval<IT>());// infi(INFINITY);
	const IT l(x.inf());
	const IT r(x.sup());
	const bool isIEEE754Float = std::is_floating_point<IT>::value;

	// Early return for intervals that start with negative or zero
	if ( r <= IT(0))
		return interval<IT>(NAN, NAN); // Entire interval is undefined for x 
<= 0

	// Initialize lower and upper with direct log calculations or -INFINITY for 
l <= 0
	IT lower((l <= IT(0)) ? -infi : log(l));
	IT upper(log(r));

	// Apply shortcuts for well-known constants, adjusting for precision
	if (l == IT(1)) lower = IT(0);
	else if (isIEEE754Float && l == IT(2)) lower = ln2_interval<IT>().inf();
	else if (isIEEE754Float && l == IT(10)) lower = ln10_interval<IT>().inf();
	else lower = nextafter(lower, -infi); // Adjust for precision if not a 
shortcut value

	if (r == IT(1)) upper = IT(0);
	else if (isIEEE754Float && r == IT(2)) upper = ln2_interval<IT>().sup();
	else if (isIEEE754Float && r == IT(10)) upper = ln10_interval<IT>().sup();
	else upper = nextafter(upper, +infi); // Adjust for precision if not a 
shortcut value

	// Ensure lower is not mistakenly set to a non-NaN value when l <= 0
	if (l <= IT(0) && r > IT(0))
		lower = -infi;

	return interval<IT>(lower, upper);
}
```

The computation of log10(x) is found similarly.

```
// log10(x)
template<class IT> inline interval<IT> log10(const interval<IT>& x)
{
	const IT infi(infinity_interval<IT>());// infi(INFINITY);
	const IT l(x.inf());
	const IT r(x.sup());

	// Early return for intervals that start with negative or zero
	if (r <= IT(0))
		return interval<IT>(NAN, NAN); // Entire interval is undefined for x 
<= 0

	// Initialize lower and upper bounds with direct log10 calculations or -
INFINITY for l <= 0
	IT lower(l <= IT(0) ? -infi : log10(l));
	IT upper(log10(r));

	// Apply shortcuts for well-known constants, adjusting for precision
	if (l == IT(1)) lower = IT(0);
	else if ( l == IT(10)) lower = IT(1);
	else lower = nextafter(lower, -infi); // Adjust for precision if not a 
shortcut value

	if (r == IT(1)) upper = IT(0);
	else if ( r == IT(10)) upper = IT(1);
	else upper = nextafter(upper, +infi); // Adjust for precision if not a 
shortcut value

	// Ensure lower is not mistakenly set to a non-NaN value when l <= 0
	if (l <= IT(0) && r > IT(0))
		lower = -infi;

	return interval<IT>(lower, upper);
}
```

Exponential (exp)

For the exp(x) you do have a choice between using the Taylor expansion for exp(x) as defined by:

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

We eliminate x< 0 by using the identity: $e^{-x} = \frac{1}{e^x}$ meaning we first calculate e^x and then do the inverse of $\frac{1}{e^x}$. The Taylor series expansion does convert slowly towards the result. In real life, a technique called argument reduction needs to be applied to improve the convergence rate. There are other methods using the Hyperbolic functions, or a simple Newton iteration. Regardless of the method the bound of computing exp(x) results in a wider bound that is strictly necessary. As for the log(x) function, we can use that the standard library exp(x) delivered a pretty accurate rounded result. And we simply again widen the result with the next and previously available floating-point number.

```
// exp(x)
template<class IT> inline interval<IT> exp(const interval<IT>& x)
{
	const IT infi(infinity_interval<IT>());// infi(INFINITY);
	const bool isIEEE754Float = std::is_floating_point<IT>::value;
	const IT l(x.inf());
	const IT r(x.sup());
	IT leftexp(exp(l));
	IT rightexp(exp(r));

	// Directly handle the special cases with exact values
	if (l == IT(0)) leftexp = IT(1); // e^0 = 1, exact
	else
		if (isIEEE754Float && l == IT(1))
			leftexp = e_interval<IT>().inf(); // e^1, use predefined 
constant
		else
			leftexp = nextafter(leftexp, -infi); // Adjust unless it's a 
special case

	if (r == IT(0)) rightexp = IT(1); // e^0 = 1, exact
	else
		if (isIEEE754Float && r == IT(1))
			rightexp = e_interval<IT>().sup(); // e^1, use predefined 
constant
		else
			rightexp = nextafter(rightexp, +infi); // Adjust unless it's 
a special case

	// Create and return the interval from the calculated or adjusted values
	return interval<IT>(leftexp, rightexp);
}
```

Power(x,y)

The interval power function pow(x,y) is implemented using two functions.

1) One that computes the pow(x,y), where x is an interval and y is a floating point number.
2) One that computes the pow(x,y) where both x and y are intervals.

```
// pow(x,y) where x is an interval and y id a double
//
template<class IT> inline interval<IT> pow(const interval<IT>& x, const IT y)
{
	// Check for special cases
	if (y == IT(0)) // Anything to the power of 0 is 1
		return interval<IT>(1);

	//interval<IT> lhs, rhs;
	const IT infi(INFINITY);
	const IT l = x.inf();
	const IT r = x.sup();

	IT lp = pow(l, y);
	IT rp = pow(r, y);

	if (floor(l) != l || floor(r) != r)
	{	// if either is not an integer then we do not have an exact power
		lp = nextafter(lp, (lp > IT(0)) ? -infi : +infi);
		rp = nextafter(rp, (rp > IT(0)) ? +infi : -infi);
	}
	// else Both are integers => trust the result

	// Ensure correct interval ordering for the result
```

```
        return interval<IT>(std::min(lp, rp), std::max(lp, rp));
}


// pow(x) we have to do it manually
// x^y == exp( y * ln( x ) ) );
//                  interval      singleton
// x
//
template<class IT> inline interval<IT> pow(const interval<IT>& x, const
interval<IT>& y)
{
        interval<IT> c;

        if (y.isPoint())     // is y a singleton interval?
                return pow(x, y.inf());
        if (x.isPoint())     // x is a point, y is an interval
        {
                // ??
        }
        // Both x and y are intervals
        IT yi = y.inf();
        IT ys = y.sup();
        // if y is an integer?
        if (floor(yi) == yi && floor(ys) == ys)
        { // raise to the power of an integer interval
                interval<IT> lhs = pow(x, yi);
                interval<IT> rhs = pow(x, ys);
                IT lmin = std::min(lhs.inf(), lhs.inf());
                IT lmax = std::max(lhs.sup(), lhs.sup());
                IT rmin = std::min(rhs.inf(), rhs.inf());
                IT rmax = std::max(rhs.sup(), rhs.sup());
                c = interval<IT>(std::min(lmin, rmin), std::max(lmax, rmax));
                return c;
        }

        // Otherwise, do it the hard way
        c = log(x);
        c *= y;
        c = exp(c);

        return c;
}
```

This concludes the Phase 2 implementation.

Phase Three implementation

In Phase Three we handle all the trigonometric functions and the hyperbolic functions plus any miscellaneous functions not implemented in Phase One and Two. The Trigonometric function is well known to have a periodicity of 2π and within that period fluctuates between -1 and +1. E.g. sin(π/2)=1, sin(3π/2)=-1 and sin(0) or sin(π)=0. From an interval arithmetic standpoint, we need to capture the maximum and minimum of the interval, and contrary to monotonic functions like log() and exp() we cannot just take the value from both endpoints of the interval but need to find the maximum and minimum value the sin() function pass through between these the left and right interval points.

We could use the Taylor series for the trigonometric functions and some simple identities as listed below.

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

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

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

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

$$arccos(x) = \frac{\pi}{2} - arcsin(x)$$

$$arctan(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \frac{x^9}{9} - \cdots, where\ |x| \le 1$$

However, as we saw for the log() and exp() functions it is simply faster and gives a more accurate bound of the interval by just using the built-in function in the C++ standard library (which usually has an accuracy of <= 1 ULP) and just add an interval around the result to ensure the interval is bounded correctly.

```
// sin(x)
// The function for calculating the sine over an interval can be significantly
// optimized and simplified
// to handle the periodic nature of the sine function and ensure it correctly
// covers the range of sine
// values within the specified interval.
// Here's an optimized approach that considers the sine function's properties:
// The sine function is periodic with a period of 2π, and its range is between
// -1 and 1.
// For any input interval, the sine function's output interval might wrap
// around this range.
// If the interval's width is greater than or equal to 2π, the sine function
// covers its entire range of [-1,1].
// For intervals smaller than 2π, calculate the exact sine values at the
// interval's endpoints
// and check for any critical points (multiples of 2π / 2) within the interval
// to determine the maximum
// and minimum sine values.
// This function works for both the build-in types: float, double, or long double
// but also for the float_precision class. (arbitrary precision). This is done
// by ensuring that
// local float_precision declaration is performed at the precision of x.
// Implemented via the use of constexpr (requires c++17)
// lambda functions. the variable l and r inherits the precision from the
// call to fmod()
template<class IT> inline interval<IT> sin(const interval<IT>& x)
{
	const IT infi(infinity_interval<IT>());// infi(INFINITY);
	const IT pi = [&]()->IT {
		if constexpr (std::is_floating_point<IT>::value)
		{ // for floating point type
			return acos(IT(-1)); // More precise pi value
		}
		else
		{ // For float_precision class. Use maximum precision of the left or
right interval
			return _float_table(_PI,
max(x.leftinterval().precision(),x.rightinterval().precision()));
		}
	}(); // The Lambda is immediately invokeed
	const IT twopi(IT(2) * pi);
	IT l(x.inf());
	IT r(x.sup());

	// If the interval width is >= 2pi, the sine function covers the full range
[-1, 1]
	if (x.sup() - x.inf() >= twopi)
		return interval<IT>(IT(-1), IT(1));

	// Calculate sine values at the interval's endpoints
	IT sin_l(sin(l));
	IT sin_r(sin(r));

	// Check for critical points within the interval
	IT sin_min(min(sin_l, sin_r));
	IT sin_max(max(sin_l, sin_r));

	// Check passing critical ponts by normalizing l and r
	l=fmod(x.inf(), twopi); // Normalize l within a single period
	r=l + fmod(x.sup() - x.inf(), twopi); // Calculate r based on l and the
interval width
	// Normalize angles to be within [0, 2*pi)
	if (l < IT(0))		l += twopi;
	if (r >= twopi)		r -= twopi;
	if (l <= pi / IT(2) && pi / IT(2) <= r)
		sin_max = IT(1); // pi/2 is within interval
	if (l <= IT(1.5) * pi && IT(1.5) * pi <= r)
		sin_min = IT(-1); // 3*pi/2 is within interval

	// Established a safety interval around the result to ensure correct bound
for the computation
	if (sin_min != IT(-1)  && sin_min != IT(0) )
		sin_min = nextafter(sin_min, -infi);
	if( sin_max != IT(1) && sin_max != IT(0) )
		sin_max= nextafter(sin_max, +infi);

	// Create and return the interval based on calculated min and max sine
values
	return interval<IT>(IT(sin_min), IT(sin_max));
}
```

```
// Same layout as for the sin(x) with the needed change for cos(x)
// The normalization of the input interval l and r remains the same as for
// the sin(x), as it's based on the periodicity of the trigonometric functions.
// The critical points for maximum and minimum values are adjusted for the
// cos(x) function. Specifically,
// cos(x) reaches its maximum value of 1 at 0 and 2π, and its minimum value
// of − 1 at π.
// The check for these critical points within the given interval is updated
// to reflect the cosine function's behavior.
// The return statement creates and returns an interval of type IT based on
// the calculated minimum and maximum
// values of cos(x) within the specified interval.
// This function works for both the build in types: float, double or long double
// but also for the float_precision class. (arbitrary precision).
// This is done by ensure that
// local float_precision declaration is performed at the precision of x.
// Implemented via the use of constexpr (requires c++17)
// lambda functions. the variable l and r inherits the precision from the
// call to fmod()
template<class IT> inline interval<IT> cos(const interval<IT>& x)
{
	const IT infi(infinity_interval<IT>());// infi(INFINITY);
	const IT pi = [&]()->IT {
		if constexpr (std::is_floating_point<IT>::value)
		{ // for floating point type
			return acos(IT(-1)); // More precise pi value
		}
		else
		{ // For float_precision class. Use maximum precision of the left or 
right interval
			return _float_table(_PI, max(x.leftinterval().precision(), 
x.rightinterval().precision()));
		}
	}(); // The Lambda is immediately invokeed
	const IT twopi(IT(2) * pi);
	IT l(x.inf());
	IT r(x.sup());

	// If the interval width is >= 2pi, the cos function covers the full range 
[-1, 1]
	if (x.sup() - x.inf() >= twopi)
		return interval<IT>(IT(-1), IT(1));

	// Calculate cosine values at the interval's endpoints
	IT cos_l(cos(l));
	IT cos_r(cos(r));

	// Check for critical points within the interval
	IT cos_min(min(cos_l, cos_r));
	IT cos_max(max(cos_l, cos_r));

	// Check passing critical ponts by normalizing l and r
	l = fmod(x.inf(), twopi); // Normalize l within a single period
	r = l + fmod(x.sup() - x.inf(), twopi); // Calculate r based on l and the 
interval width
	// Normalize angles to be within [0, 2*pi)
	if (l < IT(0))		l += twopi;
	if (r >= twopi)		r -= twopi;
	if (r<l)
		cos_max = IT(1.0); // 0 or 2*pi is within interval
	//if (l <= pi && pi <= r)
	if(l<=pi && r >= pi)
		cos_min = IT(-1.0); // pi is within interval

	// Established a safety interval around the result to ensure correct bound 
for the computation
	if (cos_min != IT(-1) && cos_min != IT(0) )
		cos_min = nextafter(cos_min, -infi);
	if (cos_max != IT(1) && cos_max != IT(0) )
		cos_max = nextafter(cos_max, +infi);

	// Create and return the interval based on calculated min and max cosine 
values
	return interval<IT>(IT(cos_min), IT(cos_max));
}

// This code makes several key assumptions and considerations:
// It normalizes the input interval to a single period of 2π to manage
// the periodicity of tan(x).
// It checks if the interval crosses a vertical asymptote by examining the
// range of the interval and the relative
// positions of l and r. If the interval crosses an asymptote, the function
// can potentially take on all real values,
// so the interval is set to (-∞, ∞).
// If the interval does not include an asymptote, the function calculates
// the tangent at the endpoints of the interval
// and uses these to determine the minimum and maximum values of tan(x)
// within the interval.
// It returns an interval representing the range of tan(x) over the
// specified interval, taking into account the
// possibility of infinite values.
// This approach captures the basic behavior of the tangent function over
// an interval, but it simplifies the handling
// of asymptotes and does not account for multiple discontinuities within
// a larger interval. For more complex cases,
// additional logic would be required to segment the interval and handle
// each segment individually.
// This function works for both the build-in types: float, double, or long
// double but also for the float_precision class. (arbitrary precision).
// This is done by ensuring that
// local float_precision declaration is performed at the precision of x.
// Implemented via the use of constexpr (requires c++17)
// lambda functions. the variable l and r inherits the precision from the
// call to fmod()
template<class IT> inline interval<IT> tan(const interval<IT>& x)
{
	const IT infi(infinity_interval<IT>());// infi(INFINITY);
	const IT pi = [&]()->IT {
		if constexpr (std::is_floating_point<IT>::value)
		{ // for floating point type
			return acos(IT(-1)); // More precise pi value
		}
		else
		{ // For float_precision class. Use maximum precision of the left or 
right interval
			return _float_table(_PI, max(x.leftinterval().precision(), 
x.rightinterval().precision()));
		}
	}(); // The Lambda is immediately invokeed
	const IT twopi(IT(2) * pi);
	IT l(fmod(x.inf(), twopi)); // Normalize l within a single period
```

```
       IT r(l + fmod(x.sup() - x.inf(), twopi)); // Calculate r based on l and the
interval width

       // Normalize angles to be within [0, 2*pi)
       if (l < IT(0)) l += twopi;
       if (r >= twopi) r -= twopi;

       // Check if the interval includes a vertical asymptote
       if (x.sup() - x.inf() >= pi || (floor((l + pi / IT(2)) / pi) != floor((r +
pi / IT(2)) / pi))) {
              // The function covers an entire period or crosses an asymptote,
range is all real numbers
              return interval<IT>(IT(-infi), IT(+infi));
       }

       // Calculate tangent values at the interval's endpoints
       IT tan_l(tan(x.inf()));
       IT tan_r(tan(x.sup()));

       // Given the properties of tan(x), if the interval does not include an
asymptote,
       // the minimum and maximum can be directly computed from the interval's
endpoints.
       IT tan_min(min(tan_l, tan_r));
       IT tan_max(max(tan_l, tan_r));

       // Established a safety interval around the result to ensure correct bound
for the computation
       if (tan_min != pi)
              tan_min = nextafter(tan_min, -infi);
       if (tan_max != pi)
              tan_max = nextafter(tan_max, +infi);

       // Create and return the interval based on calculated min and max tangent
values
       return interval<IT>(IT(tan_min), IT(tan_max));
}

// asin(x)
// This function works for both the build in types: float, double or long double
// but also for the float_precision class. (arbitrary precision).
// This is done by ensure that
// local float_precision declaration is performed at the precision of x.
// Implemented via the use of constexpr (requires c++17)
// lambda functions.
//
template<class IT> inline interval<IT> asin(const interval<IT>& x)
{
       // Check if the input interval exceeds the domain of arcsin
       if (x.inf() < IT(-1) || x.sup() > IT(1))
              throw std::domain_error("arcsin is undefined for values outside the
interval [-1, 1].");

       const IT infi(infinity_interval<IT>());// infi(INFINITY);
       // Calculate arcsin values at the interval's endpoints
       const IT asin_l(asin(x.inf()));
       const IT asin_r(asin(x.sup()));

       // Ensure the interval is correctly oriented
       IT asin_min(min(asin_l, asin_r));
       IT asin_max(max(asin_l, asin_r));
       // Established a safety interval around the result to ensure correct bound
for the computation
       asin_min = nextafter(asin_min, -infi);
       asin_max = nextafter(asin_max, +infi);

       // Since arcsin is monotonically increasing in its domain, we directly
return the interval
       return interval<IT>(IT(asin_min), IT(asin_max));
}

// acos(x)
// This function works for both the build in types: float, double or long double
// but also for the float_precision class. (arbitrary precision).
// This is done by ensure that
// local float_precision declaration is performed at the precision of x.
// Implemented via the use of constexpr (requires c++17)
// lambda functions.
//
template<class IT> inline interval<IT> acos(const interval<IT>& x)
{
       // Check if the input interval exceeds the domain of acos
       if (x.inf() < IT(-1) || x.sup() > IT(1))
              throw std::domain_error("acos is undefined for values outside the
interval [-1, 1].");

       const IT infi(infinity_interval<IT>());// infi(INFINITY);
       // Calculate acos values at the interval's endpoints
       const IT acos_l(acos(x.sup())); // Note: we use sup here
       const IT acos_r(acos(x.inf())); // Note: we use inf here

       // Ensure the interval is correctly oriented
       IT acos_min(min(acos_l, acos_r));
       IT acos_max(max(acos_l, acos_r));

       // Established a safety interval around the result to ensure correct bound
for the computation
       acos_min = nextafter(acos_min, -infi);
       acos_max = nextafter(acos_max, +infi);

       // Since acos is monotonically decreasing in its domain, the min and max
are computed differently than in asin
       return interval<IT>(IT(acos_min), IT(acos_max));
}


// atan(x)
// This function works for both the build in types: float, double or long double
// but also for the float_precision class. (arbitrary precision).
// This is done by ensure that
// local float_precision declaration is performed at the precision of x.
// Implemented via the use of constexpr (requires c++17)
// lambda functions.
//
template<class IT> inline interval<IT> atan(const interval<IT>& x)
{
       const IT infi(infinity_interval<IT>());// infi(INFINITY);
       // Calculate atan values at the interval's endpoints
       const IT atan_l(atan(x.inf()));
       const IT atan_r(atan(x.sup()));
```

```
        // Ensure the interval is correctly oriented
        IT atan_min(min(atan_l, atan_r));
        IT atan_max(max(atan_l, atan_r));

        // Established a safety interval around the result to ensure correct bound 
for the computation
        atan_min = nextafter(atan_min, -infi);
        atan_max = nextafter(atan_max, +infi);

        // Since atan is monotonically increasing in its domain, we directly return 
the interval
        return interval<IT>(IT(atan_min), IT(atan_max));
}
```

As for the Hyperbolic functions, we can take advantage that we already have efficient exp() and log() implemented and just use the identity of these functions.

$$\sinh(x) = \frac{1}{2}(e^x - e^{-x}) = \frac{1}{2}\left(e^x - \frac{1}{e^x}\right)$$

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

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1}$$

$$\mathrm{Arcsinh}(x) = \ln\left(x + \sqrt{x^2 + 1}\right)$$

$$\mathrm{Arccosh}(x) = \ln\left(x + \sqrt{x^2 - 1}\right), \quad \text{where } x \geq 1$$

$$\mathrm{Arctanh}(x) = \frac{1}{2}\ln\left(\frac{1 + x}{1 - x}\right), \quad \text{where } |x| < 1$$

```
// Use the identity. sinh(x)=0.5*(exp(x)-1/exp(x))
template<class IT> inline interval<IT> sinh(const interval<IT>& x)
{
        const interval<IT> one(1);
        IT e = exp(x);
        return interval<IT>(0.5) * (e - 1 / e);
}

// Use the identity. cosh(x)=0.5*(exp(x)+1/exp(x))
template<class IT> inline interval<IT> cosh(const interval<IT>& x)
{
        const interval<IT> one(1);
        IT e = exp(x);
        return interval<IT>(0.5) * (e + one / e);
}
```

```
//Use the identity. tanh(x)=(exp(x)-1/exp(x))/(exp(x)+1/exp(x))=(exp(x)^2-
1)/(exp(x)^2+1)
template<class IT> inline interval<IT> tanh(const interval<IT>& x)
{
	const interval<IT> one(1);
	IT e = exp(x);
	e *= e;
	return (e-one) /(e+one);
}

// Use the identity. asinh(x)=Ln(x+sqrt(x^2+1))
template<class IT> inline interval<IT> asinh(const interval<IT>& x)
{
	const interval<IT> one(1);
	IT xsq = x;
	x *= x;
	return log(x + sqrt(xsq + one));
}

// Use the identity. acosh(x)=Ln(x+sqrt(x^2-1))
template<class IT> inline interval<IT> acosh(const interval<IT>& x)
{
	const interval<IT> one(1);
	IT xsq = x;
	x *= x;
	return log(x + sqrt(xsq - one));
}

// Use the identity. atanh(x)=0.5*Ln((1+x)/(1-x))
template<class IT> inline interval<IT> atanh(const interval<IT>&x)
{
	const interval<IT> one(1);
	return interval<IT>(0.5) * log((one + x)/(one-x));
}
```

This concludes the phase 3 implementation.

Phase Four implementation

We now turn to the ability to be able to let the template type be of an arbitrary precision floating point type. The difference between the built-in types like float (6-7 decimal digits) and double (15-16 decimal digits) is that the arbitrary precision class can be of any precision up to millions or even billions of decimal digits accuracy. Now that we can deal with unlimited precision, we cannot just rely on the simple template class type to handle it since the precision can also vary.

One of the first issues is to expand on the template factory function allowing it to also handle arbitrary precision types using the following expansion of the Phase Two implementation of these constants.

```
// Get PI at the precision for IT (float_precision also based on the precision)
template<typename IT> constexpr interval<IT> pi_interval(const size_t precision =
float_precision_ctrl.precision())
{
	if constexpr (std::is_same<IT, float>::value)
		return interval<IT>(IT(3.141'592'50), IT(3.141'592'74));
	else if constexpr (std::is_same<IT, double>::value)
		return interval<IT>(IT(3.141'592'653'589'793'1),
IT(3.141'592'653'589'793'6));
	else if  (std::is_same<IT, float_precision> ::value)
	{
		float_precision pi(0, precision, ROUND_DOWN);
		// Get PI with one higher decimal accuracy to be able to get the
left side of the interval correctly
		pi=_float_table(_PI, precision+1);
		return interval<float_precision>(pi, nextafter(pi,FP_INFINITY));
	}
	else

	static_assert(is_floating_point<IT>::value||is_same<IT,float_precision>::va
lue, "Unsupported type for pi_interval.Type must be float, double, long double or
float_precision.");
}

// Get e at the precision for IT.(float_precision also based on the precision)
template<typename IT> constexpr interval<IT> e_interval(const size_t precision =
float_precision_ctrl.precision())
{
	if constexpr (std::is_same<IT, float>::value)
		return interval<IT>(IT(2.718'281'75), IT(2.718'281'98));
	else if constexpr (std::is_same<IT, double>::value)
		return interval<IT>(IT(2.718'281'828'459'045'1),
IT(2.718'281'828'459'045'5));
	else if (std::is_same<IT, float_precision> ::value)
	{
		float_precision e1(0, precision, ROUND_DOWN);
		// Get e with one higher decimal accuracy to be able to get the left
side of the interval correctly
		e1 = _float_table(_EXP1, precision + 1);
		return interval<float_precision>(e1, nextafter(e1, FP_INFINITY));
	}
	else
		static_assert(is_floating_point<IT>::value || is_same<IT,
float_precision>::value, "Unsupported type for pi_interval.Type must be float,
double, long double or float_precision.");
}

// Get e at the precision for IT.(float_precision also based on the precision)
template<typename IT> constexpr interval<IT> ln2_interval(const size_t precision
= float_precision_ctrl.precision())
{
	if constexpr (std::is_same<IT, float>::value)
		return interval<IT>(IT(0.693'147'123), IT(0.693'147'182));
	else if constexpr (std::is_same<IT, double>::value)
		return interval<IT>(IT(0.693'147'180'559'945'29),
IT(0.693'147'180'559'945'40));
	else if (std::is_same<IT, float_precision> ::value)
	{
		float_precision ln2(0, precision, ROUND_DOWN);
		// Get LN2 with one higher decimal accuracy to be able to get the
left side of the interval correctly
		ln2 = _float_table(_LN2, precision + 1);
		return interval<float_precision>(ln2, nextafter(ln2, FP_INFINITY));
	}
	else
```

```
		static_assert(is_floating_point<IT>::value || is_same<IT,
float_precision>::value, "Unsupported type for pi_interval.Type must be float,
double, long double or float_precision.");
}

// Get e at the precision for IT.(float_precision also based on the precision)
template<typename IT> constexpr interval<IT> ln10_interval(const size_t precision
= float_precision_ctrl.precision())
{
	if constexpr (std::is_same<IT, float>::value)
		return interval<IT>(IT(2.302'584'89), IT(2.302'585'12));
	else if constexpr (std::is_same<IT, double>::value)
		return interval<IT>(IT(2.302'585'092'994'045'5),
IT(2.302'585'092'994'045'9));
	else if (std::is_same<IT, float_precision> ::value)
	{
		float_precision ln10(0, precision, ROUND_DOWN);
		// Get LN2 with one higher decimal accuracy to be able to get the
left side of the interval correctly
		ln10 = _float_table(_LN10, precision + 1);
		return interval<float_precision>(ln10, nextafter(ln10,
FP_INFINITY));
	}
	else
		static_assert(is_floating_point<IT>::value || is_same<IT,
float_precision>::value, "Unsupported type for pi_interval.Type must be float,
double, long double or float_precision.");
}
```

Now that we have templatized these functions we can call pi like this:

```
	pi_interval<float>();
	double: pi_interval<double>();
```

And

```
	float_precision like pi_interval<float_precision>().
```

Notice that in the last example, you can add an optional parameter for the wanted precision in decimal digits. If omitted the default precision of a float_precision variable will be used. Otherwise, it will return π with the requested precision. E.g. pi_interval<float_precision>(50); return pi with 50 decimal digits precision. With that in place, we can use this function directly in any template function to request the equivalent constant for the template type.

The need for specialization of operator overloading for arbitrary precision support

The essential operator as a template function (as it stands right now) is:

```
// += operator. Works for nearly all classes.
// Always return a "proper" and closed [] interval
// a:=a+[EMPTY] or b:=[EMPTY]+b or [EMPTY]:=[EMPTY]+[EMPTY]
template<class IT> inline interval<IT>& interval<IT>::operator+=( const
interval<IT>& a )
{
	// Handle EMPTY interval first
	if (a.type == EMPTY)
```

```
			return *this;
		if (type == EMPTY)
			return (*this = a);

		const IT infi(INFINITY);
		// Neither a or b is [EMPTY]
		std::pair<IT,IT> xleft = fasttwo_sum(inf(), a.inf());
		std::pair<IT,IT> xright = fasttwo_sum(sup(), a.sup());
		// Any adjustment?
		if (xleft.second < IT(0))
			xleft.first=nextafter(xleft.first, -infi);
		if (xright.second > IT(0))
			xright.first = nextafter(xright.first, +infi);
		left = xleft.first;
		right = xright.first;
		type = CLOSE;
		return *this;
}
```

However, that will not work using an arbitrary precision class. The issue is that *this and rhs parameter can be of different precision and for it to work you need to evaluate it using the maximum precision of both *this and rhs. If you do not do that then when called with *this equals 20 decimal precision and rhs=40 decimal digits the result will be truncated to 20 decimal precision. Well is that so bad when this is an assignment and you would need to round it to the left-hand side precision? That is correct but if you look at the following expression:

```
	float_precision a(1,20), b(2,40), c(3,20);
```

The expression like:

	b=a+b;

This will not be correct since the interim computation of a+b will yield a 20-decimal precision number while:

	b=b+a;

Here the interim computation of b + a will correctly yield a 40 decimal precision number. The way to fix this issue is to use specialization by making a special add function for float_precision number that can cope with different precision of *this and a.

```
// += operator. Specialization for float_precision class
// Always return a "proper" and closed [] interval
// a:=a+[EMPTY] or b:=[EMPTY]+b or [EMPTY]:=[EMPTY]+[EMPTY]
inline interval<float_precision>& interval<float_precision>::operator+=(const
interval<float_precision>& a)
{
		// Handle EMPTY interval first
		if (a.type == EMPTY)
			return *this;
		if (type == EMPTY)
			return (*this = a);

		const float_precision infi(INFINITY);
		const float_precision c0(0);
		// Get maximum precision of operands a and b
		const size_t max_prec(max(
			max(left.precision(), left.precision()),
			max(a.leftinterval().precision(), a.rightinterval().precision())
		));
		// Neither a or b is [EMPTY]
```

```
	std::pair<float_precision, float_precision> xleft = fasttwo_sum(inf(), 
a.inf());
	std::pair<float_precision, float_precision> xright = fasttwo_sum(sup(), 
a.sup());
	// Any adjustment?
	if (xleft.second < c0 )
		xleft.first = nextafter(xleft.first, -infi);
	if (xright.second > c0)
		xright.first = nextafter(xright.first, +infi);
	left.precision(max_prec);
	right.precision(max_prec);
	left = xleft.first;
	right = xright.first;
	type = CLOSE;
	return *this;
}
```

The same need arose for both the other essential operators: -=, *=, and the /= operator.

However, we don't need to create a specialization function for all the template functions. E.g. sin(x) where x can be either float, double, or the floating-point class.

The standard interval function for sin(x) would be the following template class.

```
// Sin template class for float or double
template<class IT> inline interval<IT> sinsimpel(const interval<IT>& x)
{
	const IT infi(INFINITY);
	const IT pi = acos(IT(-1)); // More precise pi value
	const IT twopi(IT(2) * pi);
	IT l(fmod(x.inf(), twopi)); // Normalize l within a single period
	IT r(l + fmod(x.sup() - x.inf(), twopi)); // Calculate r based on l and the 
interval width

	// Normalize angles to be within [0, 2*pi)
	if (l < IT(0))
		l += twopi;
	if (r >= twopi)
		r -= twopi;

	// If the interval width is >= 2pi, the sine function covers the full range 
[-1, 1]
	if (x.sup() - x.inf() >= twopi)
		return interval<IT>(IT(-1), IT(1));

	// Calculate sine values at the interval's endpoints
	IT sin_l(sin(l));
	IT sin_r(sin(r));

	// Check for critical points within the interval
	IT sin_min(min(sin_l, sin_r));
	IT sin_max(max(sin_l, sin_r));
	if (l <= pi / IT(2) && pi / IT(2) <= r)
		sin_max = IT(1); // pi/2 is within interval
	if (l <= IT(1.5) * pi && IT(1.5) * pi <= r)
		sin_min = IT(-1); // 3*pi/2 is within interval

	// Established a safety interval around the result to ensure correct bound 
for the computation
	if (sin_min != IT(-1) && sin_min != IT(0))
```

```
		sin_min = nextafter(sin_min, -infi);
	if (sin_max != IT(1) && sin_max != IT(0))
		sin_max = nextafter(sin_max, +infi);

	// Create and return the interval based on calculated min and max sine 
values
	return interval<IT>(IT(sin_min), IT(sin_max));
}
```

We notice that we need to compute π and use the simple identity of π=acos(-1). That will technically not work for the float_precision class, since the -1 will be in default precision and not the precision of x. Instead, we can turn to the specialization of sin(x) for the float_precision type. A straightforward implementation can yield something like below.

```
inline interval<float_precision> sin(const interval<float_precision>& x)
{
	const float_precision infi(INFINITY);
	const float_precision pi = _float_table(_PI, 
max(x.leftinterval().precision(), x.rightinterval().precision()));
	const float_precision twopi(float_precision(2) * pi);
	float_precision l(fmod(x.inf(), twopi)); // Normalize l within a single 
period
	float_precision r(l + fmod(x.sup() - x.inf(), twopi)); // Calculate r based 
on l and the interval width

	// Normalize angles to be within [0, 2*pi)
	if (l < float_precision(0))
		l += twopi;
	if (r >= twopi)
		r -= twopi;

	// If the interval width is >= 2pi, the sine function covers the full range 
[-1, 1]
	if (x.sup() - x.inf() >= twopi)
		return interval<float_precision>(float_precision(-1), 
float_precision(1));

	// Calculate sine values at the interval's endpoints
	float_precision sin_l(sin(l));
	float_precision sin_r(sin(r));

	// Check for critical points within the interval
	float_precision sin_min(min(sin_l, sin_r));
	float_precision sin_max(max(sin_l, sin_r));
	if (l <= pi / float_precision(2) && pi / float_precision(2) <= r)
		sin_max = float_precision(1); // pi/2 is within interval
	if (l <= float_precision(1.5) * pi && float_precision(1.5) * pi <= r)
		sin_min = float_precision(-1); // 3*pi/2 is within interval

	// Established a safety interval around the result to ensure correct bound 
for the computation
	if (sin_min != float_precision(-1) && sin_min != float_precision(0))
		sin_min = nextafter(sin_min, -infi);
	if (sin_max != float_precision(1) && sin_max != float_precision(0))
		sin_max = nextafter(sin_max, +infi);

	// Create and return the interval based on calculated min and max sine 
values
```

```
	return interval<float_precision>(float_precision(sin_min),
float_precision(sin_max));
}
```

We have simply just replaced all IT to float_precision and called the arbitrary precision packages _float_table(_PI) to get the desired precision.

We typically would need to inquire about the actual precision of the parameter we called the sin(x) function and therefore will need a specialization function for that. However, by using C++17's new features (constexpr lambda function) we can avoid that and write a single template function handling the sin(x) for all types.

```
// sin(x)
// The function for calculating the sine over an interval can be significantly
optimized and simplified
// to handle the periodic nature of the sine function and ensure it correctly
covers the range of sine
// values within the specified interval.
// Here's an optimized approach that considers the sine function's properties :
// The sine function is periodic with a period of 2π, and its range is between −
1 and 1.
// For any input interval, the sine function's output interval might wrap around
this range.
// If the interval's width is greater than or equal to 2π, the sine function
covers its entire range of [−1,1].
// For intervals smaller than 2π, calculate the exact sine values at the
interval's endpoints
// and check for any critical points (multiples of 2π / 2) within the interval to
determine the maximum
// and minimum sine values.
// This function works for both the build-in types: float, double or long double
// but also for the float_precision class. (arbitrary precision). This is done by
ensuring that
// local float_precision declaration is performed at the precision of x.
Implemented via the use of constexpr (requires c++17)
// lambda functions. the variables l and r inherit the precision from the call to
fmod()
template<class IT> inline interval<IT> sin(const interval<IT>& x)
{
	const IT infi(INFINITY);
	const IT pi = [&]()->IT {
		if constexpr (std::is_floating_point<IT>::value)
		{ // for floating point type
			return acos(IT(-1)); // More precise pi value
		}
		else
		{ // For float_precision class.
		  // Use maximum precision of the left or right interval
			return _float_table(_PI,
max(x.leftinterval().precision(),x.rightinterval().precision()));
		}
	}(); // The Lambda is immediately invoked
	const IT twopi(IT(2) * pi);
	IT l(fmod(x.inf(), twopi)); // Normalize l within a single period
	IT r(l + fmod(x.sup() - x.inf(), twopi)); // Calculate r based on l and the
interval width

	// Normalize angles to be within [0, 2*pi)
```

```
       if (l < IT(0))
              l += twopi;
       if (r >= twopi)
              r -= twopi;

       // If the interval width is >= 2pi, the sine function covers the full range
[-1, 1]
       if (x.sup() - x.inf() >= twopi)
              return interval<IT>(IT(-1), IT(1));

       // Calculate sine values at the interval's endpoints
       IT sin_l(sin(l));
       IT sin_r(sin(r));

       // Check for critical points within the interval
       IT sin_min(min(sin_l, sin_r));
       IT sin_max(max(sin_l, sin_r));
       if (l <= pi / IT(2) && pi / IT(2) <= r)
              sin_max = IT(1); // pi/2 is within interval
       if (l <= IT(1.5) * pi && IT(1.5) * pi <= r)
              sin_min = IT(-1); // 3*pi/2 is within interval

       // Established a safety interval around the result to ensure the correct
bound for the computation
       if (sin_min != IT(-1)  && sin_min != IT(0) )
              sin_min = nextafter(sin_min, -infi);
       if( sin_max != IT(1) && sin_max != IT(0) )
              sin_max= nextafter(sin_max, +infi);

       // Create and return the interval based on calculated min and max sine
values
       return interval<IT>(IT(sin_min), IT(sin_max));
}
```

As it turns out we only need to use the constexpr with a lambda function plus the four specializations for float_precision as mentioned before.

That concludes Phase 4 of the project and all sources can be found on my website. The first 3 phases can be found on Interval Arithmetic (hvks.com). The one including the arbitrary precision type (float_precision) can be found on Arbitrary Precision C++ Packages (hvks.com)

Error Handling and Exceptions

I try not to throw an exception if the equivalent function in C++ does not throw an exception under similar conditions. This is also true for interval division by zero.

Performance Considerations

Surprisingly enough use of software emulation of rounding does not degrade the performance as already pointed out by [7] and also the technique used to emulate the interval version of the trigonometric, logarithm or exponential functions is faster and has tighter bounds that implemented these functions using e.g. the Taylor series or other identities for these functions.

Comparison with Other Libraries

I have not provided any direct comparison with other interval libraries although it could be interesting to do such a comparison.

Testing and Validation

As usual, we need to perform regular unit testing of the interval class by testing each variant of a constructor, conversion operators, essential operators, mixed type, and the standard built-in functions. See the output below from my test run.

The output of the unit testing.

```
START Double & float INTERVAL TEST
Interval Constructors - No Argument-> OK
Interval Constructors - 1 Argument-> OK
Interval Constructors - 2 Arguments-> OK
Interval Constructors - 3 Arguments-> OK
Interval Constructors - 3 Arguments: Float-> OK
Interval Conversion () operators-> OK
Interval Arithmetic: ADD-> OK
Interval Arithmetic: SUB-> OK
Interval Arithmetic: MUL-> OK
Interval Arithmetic: DIV-> OK
Interval Arithmetic (FLOAT): ADD-> OK
Interval Arithmetic (FLOAT): SUB-> OK
Interval Arithmetic (FLOAT): MUL-> OK
Interval Arithmetic (FLOAT): DIV-> OK
Interval Arithmetic: SQR()==x^2-> OK
Interval Arithmetic (FLOAT): SQR()==x^2-> OK
Interval Arithmetic: SQRT(x)-> OK
Interval Arithmetic (FLOAT): SQRT(x)-> OK
Interval boolean operators (==,!=)-> OK
Interval Constructors - Mixed Argument-> OK
Interval Constructors - Mixed Argument (Interval)-> OK
Interval Assignment - Mixed Intervals-> OK
Interval mixed Arithmetic '+,-' operators-> OK
Interval mixed Arithmetic '*' operators-> OK
Interval mixed Arithmetic '/' operators-> OK
Interval mixed Boolean  operators-> OK
Interval Functions log(x)-> OK
Interval Functions log10(x)-> OK
Interval Functions exp(x)-> OK
Interval Functions pow(x,y)-> OK
Interval Functions sin(x)-> OK
Interval Functions cos(x)-> OK
Interval Functions tan(x)-> OK
END Double & Float INTERVAL TEST: Passed
```

Future Directions

A future direction could be to enable the type to be of arbitrary precision instead of limiting to the built-in types of float, double, or long double. E.g. to support the use of the author's own arbitrary precision floating point class.

The full source and this paper can be found on my website at Interval Arithmetic (hvks.com).

Reference

1. Intel 64 and IA32 Architectures Software Developers Manual. Volume 2. June 2014
2. Wilkinson, J H, Rounding errors in Algebraic Processes, Prentice-Hall Inc, Englewood Cliffs, NJ 1963
3. 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
4. T. Hickey & M.H. van Emden. Interval Arithmetic: from Principles to Implementation
5. Hend Dawood, Theories of Interval Arithmetic, Mathematical Foundations and Applications, Lambert Academic Publishing. 2011. ISBN: 978-3-8465-0152-2
6. Wikipedia, Interval Arithmetic, https://en.wikipedia.org/wiki/Interval_arithmetic,
7. Daniel Pfeffer, Interval arithmetic using round-to-nearest, https://www.codeproject.com/Articles/1040839/Interval-arithmetic-using-round-to-nearest-mode-pa
8. N.T. Hayes, February 2013, Standard for Modal Interval Arithmetic.
9. J.G. Role, Interval Arithmetic and Analysis: An Introduction