Interval Arithmetic. A practical implementation.

By Henrik Vestermark (hve@hvks.com)

Abstract:

You can find example of interval arithmetic template class for the C++ programming language but none that is easy to understand and easy to implement and that is the reason why this paper was written. The paper highlights the implementation of a general interval template class supporting the float and double type of the C++ programming language.

Introduction:

Building an interval template software class that can handle all interval arithmetic for IEEE754 floating point is just down to simple math. This paper describe the practical implementation aspect of building an interval template class and discuss the tricks and math that is behind the implementation. The implementation spans from the basic operators like **_,-,*,/** to elementary functions like sqrt(), log(), pow(), exp() to trigonometric functions of sin(), cos(), tan(), asin(), acos() and atan(). Of course it also support interval constants like π, ln2 and ln10. The interval arithmetic support the basic C++ types of *float* & *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 for some deeper insight into interval arithmetic and use of IEEE754 floating point format.

Contents

Interval Arithmetic

In order 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 of with an interval instead of an approximation of the result. The interval width represent the uncertainty of the result but we would know for sure that the correct result will be within this interval.

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

$[a, b]$ Where $a \leq b$

The 4 basic operators +,-,* and / is 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 multiplication 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 ^ b = 0
Mixed	a<0 ^ b $\geq$ 0
Positive	a>0
Negative	b<0

Instead of the 8 multiplication we can get by with two in most cases and 4 multiplication 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 "as is". Since 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 language and CPU's 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 for 32bits float and t=53 for 64bits 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 +∞

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

Controlling Rounding

To control rounding process, the Intel CPU architecture offer 4 ways of rounding.

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

It is all controlled in the Intel CPU FPU control word where bit 10-11 determine the rounding mode.

RC: Rounding control bits 10-11
00 - rounding to nearest (default rounding mode)
01 - rounding down (toward - infinity)
10 - rounding up (toward + infinity)
11 - rounding toward zero (truncating)

Intel has two instruction for loading (**FLDCW**) and storing this control word (**FSTCW**). However we usually can avoid dropping into assembler instruction by using the function _controlfp87() or the more platform independent version _controlfp() or the secure version in MicroSoft Visual Studio C++ 2013 _controlfp_s(). Any of these will work and we can now define 3 useful function. One for rounding down, one for rounding up and one for rounding to nearest (reset to default behavior). The following three C functions does exactly that.

```
#include <floath.h>

void fpdown() // Set the rounding mode to rounding down
      {
      unsigned int currentControl;
      _controlfp_s( &currentControl, _RC_DOWN, _MCW_RC);
      }

void fpup()// Set the rounding mode to rounding up
      {
      unsigned int currentControl;
      _controlfp_s( &currentControl, _RC_UP, _MCW_RC);
      }

void fpnear()// Set the rounding mode to rounding to nearest (default)
      {
      unsigned int currentControl;
      _controlfp_s( &currentControl, _RC_NEAR, _MCW_RC);
      }
```

However for more modern Intel CPU Architecture, Intel also have added the SSE2 instructions set that is controlled by a different control word. Since we don't want to eliminate or restrict the uses of the new instructions we have added the rounding control for these instructions as well using the __control87_2() function in Microsoft Visual studio vs 2013 that sets the round mode in both the legacy Intel Architecture and the SSE2 architecture. Our final version then becomes:

```
void fpnear()
	{
	unsigned int f87_cw, sse2_cw;
	int cc;
	cc=__control87_2(_RC_NEAR, _MCW_RC, &f87_cw, &sse2_cw );
	}

void fpdown()
	{
	unsigned int f87_cw, sse2_cw;
	int cc;
	cc=__control87_2(_RC_DOWN, _MCW_RC, &f87_cw, &sse2_cw);
	}

void fpup()
	{
	unsigned int f87_cw, sse2_cw;
	int cc;
	cc=__control87_2(_RC_UP, _MCW_RC, &f87_cw, &sse2_cw);
	}
```

Implementing the basic operators

Now that we can control rounding it is pretty straightforward to implement interval arithmetic.

Interval Addition:

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

```
double add_lower( double a, c )
	{
	double result;
	fpdown();
	result = a + c;
	fpnear();
	return result;
	}
```

```
double add_upper( double b, d )
        {
        double result;
        fpup();
        result = b + d;
        fpnear();
        return result;
        }
```

Interval Subtraction

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

```
double sub_lower( double a, d )
        {
        double result;
        fpdown();
        result = a - d;
        fpnear();
        return result;
        }

double sub_upper( double b, c )
        {
        double result;
        fpup();
        result = b - c;
        fpnear();
        return result;
        }
```

Interval Multiplication

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

```
double mul_lower( double a, b, c, d )
        {
        double result;
        fpdown();
        result = a + c;
        fpnear();
        return result;
```

```
        }

double mul_upper( double a, b, c, d )
        {
        double result;
        fpup();
        result = b + d;
        fpnear();
        return result;
        }
```

Interval Division

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

```
double div_lower( double a, b, c, d )
        {
        double div_low, div_up;
        fpdown();
        div_low = 1/d;
        fpup();
        div_up=1/c;
        fpnear();
        return mul_lower(a,b,div_low,div_up);
        }

double div_upper( double a, b, c, d )
        {
        double div_low, div_up;
        fpdown();
        div_low=1/d;
        fpup();
        div_up=1/c;
        fpnear();
        return mul_upper(a,b,div_low,div_up);
        }
```

Of course we can see that we have repeat the same code twice for finding div_low and div_up in the div_lower() and div_upper() function. In practical implementation you want to combine this into one function and one calculation for each variable.

Using interval arithmetic for complex numbers

Now that we can do all elementary interval operators. We can now apply our interval class with the use of standard c++ complex template class to create complex number arithmetic using interval arithmetic. This is surprisingly simple. To create a complex number using interval arithmetic you simple declare the complex number like the following:

```
complex<interval<double> > cd;
```

And initialization of the complex interval object is also straight forward. E.g.

```
complex<interval<double> > cid1(1);
complex<interval<double> > cid2(1, 2);
complex<interval<double> > cid3({ 1, 2 }, { 3, 4 });
complex<interval<double> > cid4({ 1 }, { 2 });
```

cid1 = [1,1]+i[0,0];
cid2 = [1,1]+i[2,2];
cid3 = [1,2]+i[3,4];
cid4 = [1,1]+i[2,2];

Where [] denotes the interval.

And now we have support of complex arithmetic using the interval class as the underlying object type.

A classic example is the evaluation of a polynomial at a complex point using interval arithmetic to bound the error in the evaluation of the polynomial function p(z). The result of the evaluation is a complex interval number that bound the error of the polynomial evaluation.

```
// This example show how to evaluate a polynomial with complex coefficients a, at a
// complex point z and output the result as a complex interval that is the bound of the
// evaluation error. The evaluation is done using Horner algorithm
#include "intervaldouble.h"
#include <complex>

complex<interval<double> > horner(int n, complex<double> a[], complex<double> z)
	{
	complex<interval<double> > fval;
	complex<interval<double> > zi=z;

	fval = a[0];
	for (int i = 1; i <= n; i++)
		fval=fval*zi+complex<interval<double> >(a[i]);
	return fval;
	}
```

When trying it with the polynomial: p(z)=+1x^5-1.5x^4+2.5x^3-3.5x^2+4.5x-5.5 there is a zero found using a real arithmetic root finder to be x=1.364018313559659

When using above example of the Horner algorithm with the root 1.364018313559659 we get:

fval= [-1.7763568394002505e-015, 2.6645352591003757e-015]

As we can see we got an interval around zero of the polynomial evaluation at that root and we can therefore not expect any improvement in the root by continuing the iterations.

The same evaluation using real arithmetic yield an fval~1.8e-15 and as we can see the real arithmetic result is within the bounds of the interval evaluation of [-1.78E-15, 2.67E-15].

Implementation & Source code

We are now ready to create our little template interval code in form of an interval header file “intervaldouble.h”

```
// Interval class.
// realistically the class Type can be float, double. Any other type is not supported.
// since float and double are done using the Intel CPU (H/W) and using "specialization”.
//
template<class _IT> class interval
	{
	_IT low, high;
	public:
		typedef _IT value_type;
	};
```

Of course as usually we need to define the Constructors, Coordinate functions and Operators.

Interval Constructors

Very simple the interval class contains two variable the lower bound *low* and the upper bound *high* of the interval. By definition we need to ensure that the *low* is ≤ *high.* We need some constructors that can do that. Now when we create an interval variable in our program we can initialize it with either zero element (uninitialized), one scalar element or an initial interval. See below.

```
//Example of an interval declaration
interval<double> id0;
interval<double> id1(1.0);
interval<double> id2(1.0,2.0);
```

The constructors that can do that and also ensure that *low* ≤ *high* is listed below:

```
 // constructor. zero, one or two arguments for type _IT
interval()                              {low=_IT(0); high=_IT(0);}
interval( const _IT& d )                {low=_IT(d); high=_IT(d);}
interval( const _IT& l, const _IT& h){
	if( l < h ) {low=l; high=h;} else {low=h; high=l;} }
```

However we also need to be able handle constructors for mixed types where the interval<class> type is different from the argument type. E.g.

```
interval<double> idint(5);

// Constructor for mixed type _IT != _X (base types).
// Allows auto construction of e.g. interval<float_precision> x(float)
template<class _X>interval(const _X& x) {low=_IT(x); high=_IT(x);}
```

And finally we need a mixed type constructor for handling initialization with another interval<class> type that has a different argument type.

```
// constructor for any other type to _IT. Both up and down conversion possible
template<class X> interval( const interval<X>& a )
      {
      If(a.lower()<a.upper())
            {fpdown(); low=_IT(a.lower()); fpup();
            high=_IT(a.upper()); fpnear();}
      else
            { fpdown(); low=_IT(a.upper()); fpup();
            high=_IT(a.lower()); fpnear();}
      }
```

Interval Coordinate functions

We would also have a need for some coordinating functions. E.g. get the lower or upper value of the interval or change them for that matter. Likely we need to find the width of the interval and the center when converting an interval to a scalar element.

Lastly we would also need some Boolean function to test if an interval contains zero or if a number is fully contained in the interval or the interval is a subset of another interval. When you look at different implementation you will see that the naming conversion differs a lot. When I use **lower()** and **upper()** for the interval ends other use left() and right() and instead of **width()** other could use diameter etc. It is all the same just different names for the same thing.

```
// Coordinate functions
_IT upper() const          {return high;}
_IT lower() const          {return low;}
_IT upper( const _IT& u ) {return (high=u);}
_IT lower( const _IT& l ) {return (low=l);}

_IT center() const         {return (high+low)/_IT(2);}
_IT radius() const         {_IT r; r=(high-low)/_IT(2);
                           if(r<_IT(0)) r=-r; return r;}
_IT width() const          {_IT r; r=high-low;
                           if(r<_IT(0)) r=-r; return r;}

bool contain(const _IT& f=_IT(0))
                           { return low<=f&&f<= high;}
bool contain(const interval<_IT>& i)
                           { return low<=i.lower()&&i.upper() <= high;}
```

Previously I have introduced the terms interval class as a way to classify a given interval type. E.g. zero, positive, negative and mixed. This can also be a helpful function to support.

```
/// The four different interval classification
/// # ZERO            a=0 && b=0
/// # POSITIVE        a>=0 && b>0
/// # NEGATIVE        a<0 && b<=0
```

```
/// # MIXED          a<0 && b>0
enum int_class { NO_CLASS, ZERO, POSITIVE, NEGATIVE, MIXED };
```

And the coordinating function that return the classification of the interval.

```
 enum int_class is_class() const
      {
      if(low==_IT(0)&&high==_IT(0)) return ZERO;
      if(low>=_IT(0)&&high>_IT(0)) return POSITIVE;
      if(low<_IT(0)&&high<=_IT(0)) return NEGATIVE;
      if(low<_IT(0)&&high>_IT(0)) return MIXED;
      return NO_CLASS
      }
```

Interval Operators

We also need our typical operators for converting an interval to one of the basic types like *short*, *int*, *long*, and their unsigned counterpart plus *float* and *double*. To do this we simply use the center() method and cast it to the requested operator type.

```
// Conversion Operators
// Conversion to short
operator short() const {return (short)center();}

// Conversion to int
operator int() const {return (int)center();}

// Conversion to long
operator long() const {return (long)center();}

// Conversion to unsigned short
operator unsigned short() const {return (unsigned short)center();}

// Conversion to unsigned int
operator unsigned int() const {return (unsigned int)center();}

// Conversion to unsigned long
operator unsigned long() const {return (unsigned long)center();}

// Conversion to double
operator double() const {return (double)center();}

// Conversion to float
operator float() const  {return high==low?(float)low:(float)center();}
```

We are now ready to do the more serious operators that implement the arithmetic operation of +,-,*,/ and the assignment statement =. We notice that all of the arithmetic operations support the binary operators and few also the monadic version (+,-) and again all have their assignment counterpart <operator>=. We can use a trick here to avoid a lot of unneeded code by observing that the binary operations <+,-,*,/> b can be rewritten as:

Instead of returning a<operator +,->b, we rewrite it to:

```
c=a;
c <operator +,-,*,/>= b;
return c;
```

using the assignment operator to implement the binary version of the operator.

This simplifies the matter and we need only to support the essential operators of the following and then just rewrite a <+,-,*,/> to any of the below assignment statement.

```
// 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>&);
```

And finally we would also need to support the intersection of two intervals, the union and the set minus. As outline in below essential operators. Again we rewrite any of the binary version of these operators

Instead of returning a<operator &,|,^>b, we rewrite it to:

```
c=a;
c <operator *,|,^>= b;
return c;
```

```
// More Essential operators
interval<_IT>& operator&=(const interval<_IT>&);
interval<_IT>& operator|=(const interval<_IT>&);
interval<_IT>& operator^=(const interval<_IT>&);
```

Source code:

I usually structure my source code around the class definition but try only to include the most essential stuff in there and then define the remaining stuff outside the class definition.

After the class definition we will list the code for handling of all the assignment operators which include =,+=,-=,*=,/=,&=,|=,^= followed by the unary and binary operators, which include unary version of +,- and binary version of +,-,*,/,&,|,^.

As usually I postponed the code to handle mixed type operator until the basic of the template class has been implemented. The reason is that it is not trivial to handle mixed types support in a template class. You usually need to write a lot of specialization

overloading specific mixed of types for all the arithmetic operators. So for now when an operator takes two arguments. E.g. *a* <*op*> *b* then both *a* and *b* most be of the same type otherwise an compiler error will be generated telling that it can find any suitable code for handling this mixed of operators. See example below.

```
interval<float> f;
interval<double> d;

d += f;                          // Generate an error.
d += interval<double>(f);        // OK

f -= d;                          // Generate an error
f -= interval<float>(d);         // OK
```

Then we do the Boolean version of == and !=. Notice for interval arithmetic the other Boolean operators like <,>,<=,>= is not defined for intervals.

Thereafter it is the interval version of the manifest constant like π, log2 and log10 and then we turn our attention to the elementary interval functions, Abs(), Sqrt(), Exp() and Log().

Lastly we establish the interval version of all the trigonometric functions, like Sin(), CoA(), Tan(), ArcSin(), ArcCos() and ArcTan().

- Class definition
- Essential Interval Arithmetic operators
- Unary, Binary & Mixed interval operators
- Interval Boolean operators
- Interval constants: π, Log2, Log10
- Elementary Interval functions
- Trigonometric Interval functions

Source code: Class definition

We are now done with our class definition and can turn our attention to the actual coding of the interval template class.

```
// the four different interval classification
// # ZERO             a=0 && b=0
// # POSITIVE         a>=0 && b>0
// # NEGATIVE         a<0 && b<=0
// # MIXED            a<0 && b>0
enum int_class { NO_CLASS, ZERO, POSITIVE, NEGATIVE, MIXED };

//
// Interval class.
// realistically the class Type can be float, double. Any other type is not supported.
// since float and double are done using the Intel CPU (H/W) and using "specialization".
//
```

```
template<class _IT> class interval
{
      _IT low, high;
public:
     typedef _IT value_type;

// constructor. zero, one or two arguments for type _IT
interval()                              {low=_IT(0); high=_IT(0);}
interval( const _IT& d )                {low=_IT(d); high=_IT(d);}
interval( const _IT& l, const _IT& h){
      if( l < h ) {low=l; high=h;} else {low=h; high=l;} }

// Constructor for mixed type _IT != _X (base types).
// Allows auto construction of e.g. interval<float_precision> x(float)
template<class _X>interval(const _X& x) {low=_IT(x); high=_IT(x);}

// constructor for any other type to _IT. Both up and down conversion possible
template<class X> interval( const interval<X>& a )
      {
      If(a.lower()<a.upper())
            {fpdown(); low=_IT(a.lower()); fpup();
            high=_IT(a.upper()); fpnear();}
      else
            { fpdown(); low=_IT(a.upper()); fpup();
            high=_IT(a.lower()); fpnear();}
      }


// Coordinate functions
_IT upper() const         {return high;}
_IT lower() const         {return low;}
_IT upper( const _IT& u ) {return (high=u);}
_IT lower( const _IT& l ) {return (low=l);}

_IT center() const        {return (high+low)/_IT(2);}
_IT radius() const        {_IT r; r=(high-low)/_IT(2);
                           if(r<_IT(0)) r=-r; return r;}
_IT width() const         {_IT r; r=high-low;
                           if(r<_IT(0)) r=-r; return r;}

bool contain(const _IT& f=_IT(0))
                           { return low<=f&&f<= high;}
bool contain(const interval<_IT>& i)
                           { return low<=i.lower()&&i.upper() <= high;}

// Return the classification of the interval
enum int_class is_class() const
      {
      if(low==_IT(0)&&high==_IT(0)) return ZERO;
      if(low>=_IT(0)&&high>_IT(0)) return POSITIVE;
      if(low<_IT(0)&&high<=_IT(0)) return NEGATIVE;
      if(low<_IT(0)&&high>_IT(0)) return MIXED;
      return NO_CLASS
      }
// Conversion Operators
// Conversion to short
operator short() const {return (short)center();}
```

```
// Conversion to int
operator int() const {return (int)center();}

// Conversion to long
operator long() const {return (long)center();}

// Conversion to unsigned short
operator unsigned short() const {return (unsigned short)center();}

// Conversion to unsigned int
operator unsigned int() const {return (unsigned int)center();}

// Conversion to unsigned long
operator unsigned long() const {return (unsigned long)center();}

// Conversion to double
operator double() const {return (double)center();}

// Conversion to float
operator float() const  {return high==low?(float)low:(float)center();}

// 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>&);

// More Essential operators
interval<_IT>& operator&=(const interval<_IT>&);
interval<_IT>& operator|=(const interval<_IT>&);
interval<_IT>& operator^=(const interval<_IT>&);
};
```

Source code: Essential Arithmetic operators

The assignment operator is the simple's one, as we don't lose precision in the assignment.

```
// Assignment operator. Works for all class types
//
template<class _IT> inline interval<_IT>&
interval<_IT>::operator=(const interval<_IT>& a )
   {
   low = a.lower();
   high = a.upper();
   return *this;
   }
```

For the += operator is also straight forward

```
// += operator. Works all other classes.
//
template<class _IT> inline interval<_IT>&
interval<_IT>::operator+=(const interval<_IT>&a)
   {
   fpdown(); low+=a.lower();
```

```
    fpup();   high+=a.upper();
    fpnear();
    return *this;
    }
```

Same goes for -= operator is also straight forward

```
// -= operator. Works all other classes.
//
template<class _IT> inline interval<_IT>& interval<_IT>::operator-=(
const interval<_IT>&a)
    {
    fpdown(); low -= a.high;
    fpup();   high -= a.low;
    fpnear();
    return *this;
    }
```

The multiply operator requires some more but still easy to implement following the definition of interval multiply outline earlier.

```
// Works all other classes.
// *= operator
// Instead of doing the mindless low = MIN(low*a.high, low*a.low,high*a.low,high*a.high)
// and
// high = MAX(low*a.high, low*a.low,high*a.low,high*a.high)
// which requires a total of 8 multiplication
//
//   low, high, a.low, a.high    result
//    +     +     +     +       +  +  [ low*a.low, high*a.high ]
//    +     +     -     +       -  +  [ high*a.low, high*a.high ]
//    +     +     -     -       -  -  [ high*a.low, low*a.high ]
//    -     +     +     +       -  +  [ low*a.high, high*a.high ]
//    -     +     -     +       -  +  [ MIN(low*a.high,high*a.low),
//                                      MAX(low*a.low,high*a.high) ]
//    -     +     -     -       -  -  [ high*a.low, low*a.low ]
//    -     -     +     +       -  -  [ low*a.high, high,a.low ]
//    -     -     -     +       -  -  [ low*a.high, low*a.low ]
//    -     -     -     -       +  +  [ high*a.high, low * a.low ]
//
template<class _IT> inline interval<_IT>& interval<_IT>::operator*=(
const interval<_IT>&a)
    {
    _IT l, h, t;

    if(low >= 0) //
       { // both low and high >= 0
       if(a.lower()>=0)
          { // a.low >=0, a.high >= 0
          fpdown();
          l=low*a.lower();
          fpup();
          h=high*a.upper();
          }
       else
          if(a.upper()>=0)
             {  //  a.low < 0, a.high >= 0
             fpdown();
             l=high*a.lower();
             fpup();
```

```
            h=high*a.upper();
            }
         else
            { // a.low and a.high < 0
            fpdown();
            l=high*a.lower();
            fpup();
            h=low*a.upper();
            }
      }
   else
      if(high>=0)
         {  // low < 0, high >= 0
         if(a.lower()>=0)
            { // a.low >=0, a.high >= 0
            fpdown();
            l=low*a.upper();
            fpup();
            h=high*a.upper();
            }
         else
            if(a.upper()>=0)
               {  //  a.low < 0, a.high >= 0
               fpdown();
               l=low*a.upper(); if(l>(t=high*a.lower())) l=t;
               fpup();
               h=high*a.upper(); if(h<(t=low*a.lower())) h = t;
               }
            else
               { // a.low and a.high < 0
               fpdown();
               l=high*a.lower();
               fpup();
               h=low*a.lower();
               }
         }
      else
         { // low and high are < 0
         if(a.lower()>=0)
            { // a.low >=0, a.high >= 0
            fpdown();
            l=low*a.upper();
            fpup();
            h=high*a.lower();
            }
         else
            if(a.upper()>=0)
               {  //  a.low < 0, a.high >= 0
               fpdown();
               l=low*a.upper();
               fpup();
               h=low*a.lower();
               }
            else
               {
               fpdown();
               l=high*a.upper();
```

```
                fpup();
                h=low*a.lower();
                }
        }

        low = l;
        high = h;
        fpnear();
        return *this;
        }


// Works for all classes
// /= operator
//
template<class _IT> inline interval<_IT>& interval<_IT>::operator/=(
const interval<_IT>&b)
    {
    interval<_IT> a, c;

    fpdown();
    c.low=(_IT)1/b.upper();
    fpup();
    c.high=(_IT)1/b.lower();
    fpnear();

    a=interval(low,high);
    c*=a;

    low=c.lower();
    high=c.upper();

    return *this;
    }


// Works on all classes.
// Return the intersection
//
template<class _IT> inline interval<_IT>&
interval<_IT>::operator&=(const interval<_IT>&a)
    {
    if(a.lower()>low)
        low = a.lower();
    if(a.upper()<high)
        high = a.upper();
    if(low>high)  // Empty set
        {
        low = 0; high = 0;
        }

    return *this;
    }

// Works on all classes.
// Return the union
//
```

```
template<class _IT> inline interval<_IT>&
interval<_IT>::operator|=(const interval<_IT>& a)
   {
   if(low>a.upper()||high<a.lower())
      {
      if(a.upper()-a.lower()>high-low)
            { // return the largest set
            low=a.lower();
            high=a.upper();
            }
         }
      else
         { // non empty intersection
         if(a.lower()<low)
            low = a.lower();
         if(a.upper()>high)
            high=a.upper();
         }
      }


// Works on all classes.
// Return the set minus
//
template<class _IT> inline interval<_IT>&
interval<_IT>::operator^=(const interval<_IT>& a)
   {
   if(a.lower()<high&&a.upper()>low) // intersection is not empty
      {
      if(a.upper()<=low)
         low=a.upper();
      else
         if(a.high()>=high)
            high = a.lower();
      }

   return *this;
   }
```

Source code: Unary, Binary & Mixed interval operators

With all the essential assignment operators done it is very easy to handle the Unary version of +,- and the binary version of ,+,-,*,/,|,&,^.

General speaking you can turn a binary expression like a <operator> b into and two assignment operations.

```
c=a;
c<operator>=b;
```

```
// Binary + operator for same type
// Works for all classes
```

```
//
template<class _IT> inline interval<_IT>operator+(const interval<_IT>&a, const
interval<_IT>& b)
       {
       interval<_IT> c(a);

       c += b;
       return c;
       }

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

// Binary – operator for same type
// Works for all classes
//
template<class _IT> inline interval<_IT>operator-(const interval<_IT>&a, const
interval<_IT>&b)
   {
   interval<_IT> c(a);

   c -= b;
   return c;
   }


// Unary - operator
// Works for all classes
//
template<class _IT> inline interval<_IT>operator-(const interval<_IT>&a)
   {
   interval<_IT>c(0);

   c -= a;
   return c;
   }


// Binary * operator for same type
// Works for all classes
//
template<class _IT> inline interval<_IT> operator*(const interval<_IT>&a, const
interval<_IT>&b)
   {
   interval<_IT> c(a);

   c *= b;
   return c;
   }

// Binary / operator for same types
// Works for all classes
//
template<class _IT> inline interval<_IT> operator/(const interval<_IT>&a, const
interval<_IT>& b)
   {
   interval<_IT> c(a);

   if ( c == b && b.is_class() != ZERO )
```

```
         c = interval<_IT>(1,1);
   else
      c /= b;

   return c;
   }

// Binary & operator for same types
// Return intersection
// Works for all classes
//
template<class _IT> inline interval<_IT>operator&(const interval<_IT>&a, const
interval<_IT>&b)
       {
       interval<_IT> c(a);

       c &= b;
       return c;
       }

// Binary | operator for same types
// Return union
// Works for all classes
//
template<class _IT> inline interval<_IT>operator|(const interval<_IT>&a, const
interval<_IT>&b)
       {
       interval<_IT> c(a);

       c |= b;
       return c;
       }

// Binary ^ operator for same types
// Return set minus
// Works for all classes
//
template<class _IT> inline interval<_IT>operator^(const interval<_IT>&a, const
interval<_IT>&b)
       {
       interval<_IT> c(a);

       c ^= b;
       return c;
       }
```

Source code: Interval Boolean operators

Implementation of the Boolean operators is nearly self-explanatory.

```
// == operator
// Works for all classes
//
template<class _IT> inline bool operator==(const interval<_IT>&a, const
interval<_IT>&b)
       {
       return a.lower()==b.lower()&&a.upper()==b.upper();
       }

// != operator
// Works for all classes
```

```
//
template<class _IT> inline bool operator!=(const interval<_IT>&a, const
interval<_IT>&b)
	{
	return a.lower()!=b.lower()||a.upper()!=b.upper();
	}
```

Source code: Interval constants of π, LN2 & LN10

We are now in need to implement the interval versions of the manifest constants for π, LN2 & LN10. Luckily for us IEEE754 do support that and we need to use the same technic for loading them as we do for interval arithmetic. First we need some low level functions that can retrieve theses constants both rounded down and rounded up. All of Intel instructions for these constants always returned it as a double. In order to create the equivalent float version we would need a special conversion from double -> float that can be done using correct rounding.

```
// Return the PI as a double type either as a rounded down or rounded up.
// Intel has a dedicated instruction fldpi for that.
//
inline double pidouble(enum round_mode rm)
	{
	double res;

	switch (rm)
		{
		case ROUND_DOWN: fpdown(); break;
		case ROUND_UP: fpup(); break;
		}

	_asm
		{
		fldpi;                Load PI
		fstp qword ptr[res]; Store result in res
		}
	fpnear();
	return res;
	}



// Return log(2); LN2 as a double type either rounded down or rounded up
// Intel has an instruction fldln2 for that
//
inline double ln2double(enum round_mode rm)
	{
	double res;

	switch (rm)
		{
		case ROUND_DOWN: fpdown(); break;
		case ROUND_UP: fpup(); break;
		}

	_asm
		{
```

```
            fldln2;              Load ln2
            fstp qword ptr[res]; Store result in res
            }
      fpnear();
      return res;
      }


// Return Log(10); LN10 as a double type either rounded up or rounded down
// Intel does not have a direct instruction for that but we can multiply their
// FLDL2T and FLDLN2 to get our needed result.
//
inline double ln10double(enum round_mode rm)
      {
      double res;

      switch (rm)
            {
            case ROUND_DOWN: fpdown(); break;
            case ROUND_UP: fpup(); break;
            }

      _asm
            {
            ; ln10 = FLDL2T * FLDLN2
            fldl2t;                 Load log2(10)
            fldln2;                 Load LN2
            fmulp st(1),st;         Calculate LN2 * lOG2(10)
            fstp qword ptr[res];    Store ln10 in result
            }
      fpnear();
      return res;
      }


// Support function for correctly converting a double number back to a float number.
// By default IEE754 round to nearest and that will create incorrect result
// for interval arithmetic. So we need to have a special function that can do that
// correctly.
//
inline float tofloat(const double& d, enum round_mode rm)
      {
      float fres;

      switch (rm)
            {
            case ROUND_DOWN:  fpdown();  break;
            case ROUND_UP: fpup();  break;
            }

      fres = (float)d;
      fpnear();
      return fres;
      }
```

We can now create our higher level functions that return these constant as an interval by simply calling the lower level functions as rounded down and rounded up.

```
////////////////////////////////////////////////////////////////////////////
///
/// Interval constants like PI, LN2 and LN10
///
////////////////////////////////////////////////////////////////////////////

// Load manifest constant PI for double
//
inline interval<double> int_pidouble()
	{
	interval<double> pi;

	pi.lower( pidouble(ROUND_DOWN) );
	pi.upper( pidouble(ROUND_UP) );
	return pi;
	}

// Load manifest constant PI as a float
//
inline interval<float> int_pifloat()
	{
	interval<double> pid;
	interval<float> pif;

	pid = int_pidouble();
	pif.lower(tofloat(pid.lower(), ROUND_DOWN));
	pif.upper(tofloat(pid.upper(), ROUND_UP));
	return pif;;
	}

// Load manifest constant Ln2 as a double
//
inline interval<double> int_ln2double()
	{
	interval<double> ln2;

	ln2.lower(ln2double(ROUND_DOWN));
	ln2.upper(ln2double(ROUND_UP));
	return ln2;
	}

// Load manifest constant LN2 as a float
//
inline interval<float> int_ln2float()
	{
	interval<double> ln2d;
	interval<float> ln2f;

	ln2d = int_ln2double();
	ln2f.lower(tofloat(ln2d.lower(), ROUND_DOWN));
	ln2f.upper(tofloat(ln2d.upper(), ROUND_UP));
	return ln2f;;
	}

// Load manifest constant ln10 as a double
```

```
//
inline interval<double> int_ln10double()
	{
	interval<double> ln10;

	ln10.lower(ln10double(ROUND_DOWN));
	ln10.upper(ln10double(ROUND_UP));
	return ln10;
	}

// Load manifest constant LN10 as a float and return it as an interval
//
inline interval<float> int_ln10float()
	{
	interval<double> ln10d;
	interval<float> ln10f;

	ln10d = int_ln10double();
	ln10f.lower(tofloat(ln10d.lower(), ROUND_DOWN));
	ln10f.upper(tofloat(ln10d.upper(), ROUND_UP));
	return ln10f;;
	}
```

Source code: Elementary interval functions

Into this category belong the following functions: abs(), sqrt(), log10(), log2(), exp() and the power function pow(). The abs() function is straight forward.

```
////////////////////////////////////////////////////////////////////////////////
///
/// Interval abs()
///
////////////////////////////////////////////////////////////////////////////////
//
template<class _IT> inline interval<_IT> abs( const interval<_IT>& a )
	{
	if (a.lower() >= _IT(0) )
		return a;
	else
		if (a.upper() <= _IT(0) )
			return -a;

	return interval<_IT>(_IT(0), ( a.upper() > -a.lower() ? a.upper()
: -a.lower() ) );
	}
```

For sqrt(), log10() and log2() we again resort to the instructions support in Intel CPU and again as we did for the manifest constant we first need to retrieve this as rounded down or rounded up using below lower level functions.

```
// Return Sqrt(x) as a double rounded down or rounded up
//
inline double sqrtdouble( double d, enum round_mode rm )
```

```
	{
	double sq = d;

	switch( rm )
		{
		case ROUND_DOWN: fpdown(); break;
		case ROUND_UP: fpup(); break;
		}

	_asm
		{
		fld qword ptr[sq];  Load x onto the stack
		fsqrt;              Calculate sqrt(x)
		fstp qword ptr[sq]; Store result in sq
		}
	fpnear();
	return sq;
	}

// Return Sqrt(x)as a float
// Call sqrtdouble(x) for double and the convert it to a float
//
inline float sqrtfloat(float f, enum round_mode rm)
	{
	double sq = f;
	float fres;

	sq = sqrtdouble(sq, rm);
	fres = tofloat(sq, rm);
	return fres;
	}

// Return log(x) as a double rounded up or rounded down
//
inline double logdouble(double d, enum round_mode rm)
	{
	double lg = d;

	switch (rm)
		{
		case ROUND_DOWN: fpdown(); break;
		case ROUND_UP: fpup(); break;
		}

	_asm
		{
		fld qword ptr[lg];  Load x onto stack
		fldln2;             Load loge2
		fxch st(1);         Exchange stack top
		fyl2x;              Calculate y * ln2 x
		fstp qword ptr[lg]; Store result in lg
		}

	fpnear();
	return lg;
	}

// return log(x) as a  float
```

```
// call logdouble(x) and convert it down to a float
//
inline float logfloat(float f, enum round_mode rm)
	{
	double lg = (double)f;
	float fres;

	lg = logdouble( lg, rm );
	fres = tofloat(lg, rm);
	return fres;
	}

// Return log10(x) as a double
//
inline double log10double(double d, enum round_mode rm)
	{
	double lg = d;

	switch (rm)
		{
		case ROUND_DOWN: fpdown(); break;
		case ROUND_UP: fpup(); break;
		}

	_asm
		{
		fld qword ptr[lg];   Load x onto stack
		fldlg2;              Load log10(2)
		fxch st(1);          Exchange stack top
		fyl2x;               Calculate y * ln2 x
		fstp qword ptr[lg];  Store result in lg
		}

	fpnear();
	return lg;
	}

// Return log10(x) as a float
// Call log10double(x) and round it down to a float
//
inline float log10float(float f, enum round_mode rm)
	{
	double lg = (double)f;
	float fres;

	lg = log10double( lg, rm );
	fres = tofloat(lg, rm);
	return fres;
	}
```

With these lower level functions we can now implement the interval version of these functions for both interval<double> and the <interval>float type.

```
// Interval sqrt(x) for double.
//
inline interval<double> sqrt( const interval<double>& x )
   {
```

```
    double lower, upper;

    lower = sqrtdouble( x.lower(), ROUND_DOWN );
    upper = sqrtdouble( x.upper(), ROUND_UP );
    return interval<double>( lower, upper );
    }

// Interval sqrt(x) for float.
//
inline interval<float> sqrt( const interval<float>& x )
    {
    float lower, upper;

    lower = sqrtfloat( x.lower(), ROUND_DOWN );
    upper = sqrtfloat( x.upper(), ROUND_UP );
    return interval<float>( lower, upper );
    }

// interval log(x) for double.
//
inline interval<double> log( const interval<double>& x )
    {
    double lower, upper;

    lower = logdouble( x.lower(), ROUND_DOWN);
    upper = logdouble( x.upper(), ROUND_UP);
    return interval<double>( lower, upper );
    }


// interval log(x) for float.
//
inline interval<float> log( const interval<float>& x )
    {
    float lower, upper;

    lower = logfloat( x.lower(), ROUND_DOWN);
    upper = logfloat( x.upper(), ROUND_UP);
    return interval<float>( lower, upper );
    }

// Interval log10(x) for double.
//
inline interval<double> log10( const interval<double>& x )
    {
    double lower, upper;

    lower = log10double( x.lower(), ROUND_DOWN);
    upper = log10double( x.upper(), ROUND_UP);
    return interval<double>( lower, upper );
    }


// Interval log10(x) for float.
//
inline interval<float> log10( const interval<float>& x )
    {
    float lower, upper;
```

```
    lower = log10float( x.lower(), ROUND_DOWN);
    upper = log10float(x.upper(), ROUND_UP);
    return interval<float>( lower, upper );
    }
```

The only missing part is now the exp() and the pow() functions. Unfortunately the Intel instructions does not support these function directly and we would therefor need to resort to do it manually using the Taylor series for the exp() functions. We can't use the build in Microsoft C version since it does not allow us to control the rounding as we need.

To find the exponential for x we use the Taylor series for e^x:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} \ldots$$

We eliminate x< 0 by using the identity: $e^{-x} = \frac{1}{e^x}$

Again to make the Taylor series converge more quickly we reduced the argument x≤0.5 using the identity: $e^x = e^{\left(\frac{x}{2}\right)^2}$ or more general we reduced the argument x for some *k*:

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

```
// MSC exp(x does not allow rounding control
// We have to do it manually
// Use a taylor series until their is no more change in the result
// exp(x) == 1 + x + x^2/2!+x^3/3!+....
// Equivalent with the same standard C function call
// use argument reduction via exp(x)=(exp(x/2^k)2^k
// And use Brent enhancement using the double formula:
// expm(x)=exp(x)-1 && expm(2x)=expm(x)(2+expm(x)) on the backend to preseve
// loss of significance digits
//
inline interval<double> exp(const interval<double>& x)
    {
    int  i, k = 0;
    interval<double> c, res, p0, old;
    const interval<double> c1(1), c2(2);

    c = x;
    if (x.is_class() == NEGATIVE)
       c = abs(c);

    // Determine Reduction factor
    k = int((log(2) + log(abs(c.center()))) / log(2));
    k = std::min(k, 10);
    if (k > 0)
       {
       i = 2 << (k - 1);
       c /= interval<double>( (double)i );
       }
```

```
   p0 = c;
   old = c1;
   res = old + p0;
   for(i=2; i<100&&(res.lower()!=old.lower()&&
res.upper()!=old.upper()); i++)
      {
      old = res;
      p0 *= ( c / interval<double>((double)i));
      res += p0;
      }

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

   if (x.is_class() == NEGATIVE)
      res = c1 / res;

   return res;
   }

// MSC exp(x) does not allow rounding control for the exp()
// interval float exp() call the double version of interval exp(x)
//  and convert it back to interval<float>
//
inline interval<float> exp(const interval<float>& x)
   {
   interval<double> exp(const interval<double>&);
   interval<double> fx(x);
   float lower, upper;

   fx = exp(fx);
   lower = tofloat(fx.lower(), ROUND_DOWN);
   upper = tofloat(fx.upper(), ROUND_UP);
   return interval<float>(lower, upper);
   }
```

Lastly to calculate x^y =pow(x) we use a combination of the exponential and logarithm function using the equation:

$$x^y = e^{y*\log_e(x)}$$

```
// MSC pow(x) does not allow rounding control
// We have to do it manually
// x^y == exp( y * ln( x ) ) );
//
inline interval<double> pow(const interval<double>& x, const double y)
      {
      interval<double> c;
```

```
	c = log(x);
	c *= interval<double>(y);
	c = exp(c);

	return c;
	}

// MSC pow(x) does not allow rounding control
// We have to do it manually
// x^y == exp( y * ln( x ) ) );
// To avoid loss of precision we actually perform the operation using double and then
// convert the result back to float. This is consistent with the pow(x) that only takes
// double as an argument.
//
inline interval<float> pow(const interval<float>& x, const float y)
	{
	interval<double> c(x);
	float upper, lower;

	c = log(c);
	c *= interval<double>(y);
	c = exp(c);
	lower = tofloat(c.lower(), ROUND_DOWN);
	upper = tofloat(c.upper(), ROUND_UP);
	return interval<float>(lower, upper);
	}
```

Source code: Trigonometric interval functions

Lastly we need to handle the Trigonometric interval functions. Which is Sin(), Cos(), Tan() and the equivalent Arc version ArcSin(x), ArcCos(x) & ArcTan(x). Again we have only partial support for these functions in the Intel CPU hardware, the functions that can be supported directly is Sin(x), Cos(x), Tan(x) and ArcTan(x). However ArcSin(x) and ArcCos(x) need to be found using a Taylor expansion using interval arithmetic.

First the simple ones of sin(x), cos(x), tan(x) and Arctan(x) that just return either the upper or lower end of the interval

```
////////////////////////////////////////////////////////////////////////////////
///
/// Interval sin(), cos(), tan(), asin(), acos(), atan()
///
////////////////////////////////////////////////////////////////////////////////

// Sin() for double interval
//
inline double sindouble(double d, enum round_mode rm)
	{
	double res = d;

	switch (rm)
		{
		case ROUND_DOWN: fpdown(); break;
```

```
            case ROUND_UP: fpup(); break;
            }

      _asm
            {
            fld qword ptr[res];  Load lower into floating point stack
            fsin;                Calculate sin
            fstp qword ptr[res]; Store result in lower
            }
      fpnear();

      return res;
      }

// Sin() for float interval
//
inline float sinfloat(float f, enum round_mode rm)
      {
      double res = f;
      float fres;

      res = sindouble(res, rm);
      fres = tofloat(res, rm);
      return fres;
      }

// cos() for double
//
inline double cosdouble(double d, enum round_mode rm)
      {
      double res = d;

      switch (rm)
            {
            case ROUND_DOWN: fpdown(); break;
            case ROUND_UP: fpup(); break;
            }

      _asm
            {
            fld qword ptr[res];  Load lower into floating point stack
            fcos;                Calculate cos
            fstp qword ptr[res]; Store result in lower
            }
      fpnear();

      return res;
      }

// Cos(x) for float interval
//
inline float cosfloat(float f, enum round_mode rm)
      {
      double res = f;
      float fres;

      res = cosdouble( res, rm );
      fres = tofloat(res, rm);
```

```
	return fres;
	}

// tan() for double  interval
//
inline double tandouble(double d, enum round_mode rm)
	{
	double res = d;

	switch (rm)
		{
		case ROUND_DOWN: fpdown(); break;
		case ROUND_UP: fpup(); break;
		}

	_asm
		{
		fld qword ptr[res];  Load lower into floating point stack
		fptan;               Calculate tan
		fstp qword ptr[res]; Pop ST(0) and ignore
		fstp qword ptr[res]; Store result
		}
	fpnear();

	return res;
	}

// Tan(x) for float interval
//
inline float tanfloat(float f, enum round_mode rm)
	{
	double res = f;
	float fres;

	res = tandouble( res, rm );
	fres = tofloat( res, rm );
	return fres;
	}

// atan() for double interval
//
inline double atandouble(double d, enum round_mode rm)
	{
	double res = d;

	switch (rm)
		{
		case ROUND_DOWN: fpdown(); break;
		case ROUND_UP: fpup(); break;
		}

	_asm
		{
		fld qword ptr[res];  Load lower into floating point stack
		fld1;                Load 1.0 on top of stack
		fpatan;              Calculate tan
		fstp qword ptr[res]; Store result
		}
```

```
	fpnear();

	return res;
	}
// Return atan(x) rounded up or rounded down
//
inline float atanfloat(float f, enum round_mode rm)
	{
	double res = f;
	float fres;

	res = atandouble( res, rm );
	fres = tofloat(res, rm);
	return fres;
	}
```

We can now complete the first set of trigonometric functions returning a true interval<double> or interval<float>

```
// Interval sin(x) for float
//
inline interval<float> sin(const interval<float>& x)
	{
	float lower, upper;

	lower = sinfloat(x.lower(), ROUND_DOWN);
	upper = sinfloat(x.upper(), ROUND_UP);
	return interval<float>(lower, upper);
	}

// Interval Sin(x) for double
//
inline interval<double> sin(const interval<double>& x)
	{
	double lower, upper;

	lower = sindouble(x.lower(), ROUND_DOWN);
	upper = sindouble(x.upper(), ROUND_UP);
	return interval<double>(lower, upper);
	}



// Interval cos(x) for float.
//
inline interval<float> cos(const interval<float>& x)
	{
	float lower, upper;

	lower = cosfloat(x.lower(), ROUND_DOWN);
	upper = cosfloat(x.upper(), ROUND_UP);
	return interval<float>(lower, upper);
	}

// Interval Cos(x) for double.
//
```

```
inline interval<double> cos(const interval<double>& x)
	{
	double lower, upper;

	lower = cosdouble(x.lower(), ROUND_DOWN);
	upper = cosdouble(x.upper(), ROUND_UP);
	return interval<double>(lower, upper);
	}



// Interval tan(x) for float.
//
inline interval<float> tan(const interval<float>& x)
	{
	float lower, upper;

	lower = tanfloat(x.lower(), ROUND_DOWN);
	upper = tanfloat(x.upper(), ROUND_UP);
	return interval<float>(lower, upper);
	}

// Interval Tan(x) for double.
//
inline interval<double> tan(const interval<double>& x)
	{
	double lower, upper;

	lower = tandouble(x.lower(), ROUND_DOWN);
	upper = tandouble(x.upper(), ROUND_UP);
	return interval<double>(lower, upper);
	}

// Interval Arctan(x) for float.
//
inline interval<float> atan(const interval<float>& x)
	{
	float lower, upper;

	lower = atanfloat(x.lower(), ROUND_DOWN);
	upper = atanfloat(x.upper(), ROUND_UP);
	return interval<float>(lower, upper);
	}



// Interval ArcTan(x) for double.
//
inline interval<double> atan(const interval<double>& x)
	{
	double lower, upper;

	lower = atandouble(x.lower(), ROUND_DOWN);
	upper = atandouble(x.upper(), ROUND_UP);
	return interval<double>(lower, upper);
	}
```

Lastly we need to deal with the two functions that has no hardware support by Intel CPU.

The first one ArcSin(x) can be handle by using the Taylor Series for ArcSin(x):

$$ArcSin(x) = x + \frac{x^3}{2*3} + \frac{3x^5}{2*4*5} + \frac{(3*5)x^7}{2*4*6*7}...$$

To find ArcCos(x) we used the identity:

$$ArcCos(x) = \frac{\pi}{2} - ArcSin(x)$$

```
// MSC asin() does not allow rounding control
// So we have to do it manually
/// Description:
///   Use a taylor series until there is no more change in the result
///   asin(x) == x + x^3/(2*3)+(1*3)x^5/(2*4*5)+(1*3*5)x^7/(2*4*6*7)....
///   Use argument reduction via the identity arcsin(x)=2arcsin(x)/(sqrt(2)+sqrt(1-x*x))
//       This function replace the other function using newton iteration.
//
inline interval<double> asin(const interval<double>& x)
      {
      int k, sign;
      interval<double> r, u, v, v2, sqrt2, lc, uc;
      const double c1(1), c2(2);

      if (x.lower() >= c1 || x.upper() <= -c1)
            {
            throw interval<double>::domain_error(); return x;
            }

      v = x;
      if (v.lower() < -c1)
            v.lower(-c1);
      if (v.upper() > c1)
            v.upper(c1);

      sign = v.is_class();
      if (sign == NEGATIVE)
            v = -v;

      // Now use the identity arcsin(x)=2arcsin(x)/(sqrt(2)+sqrt(1-x*x))
      // until argument is less than dlimit
      // Reduce the argument to below 0.5 to make the newton run faster
      sqrt2 = interval<double>(c2); // Ensure correct number of digits
      sqrt2 = sqrt(sqrt2);
      for (k = 0; v.lower() > 0.5; k++)
            v /= sqrt2 * sqrt(interval<double>(c1) +
            sqrt(interval<double>(c1) - v * v));

      v2 = v * v;
      r = v;
      u = v;
      // Now iterate using Taylor expansion
      for (unsigned int j = 3;; j += 2)
            {
```

```
            uc = interval<double>((j - 2) * (j - 2));
            lc = interval<double>(j * j - j);
            v = uc * v2 / lc;
            r *= v;
            if( u.lower() + r.lower() == u.lower() || u.upper() +
r.upper() == u.upper())
                  break;
            u += r;
            }

      if (k > 0)
            u *= interval<double>(1 << k);

      if (sign == NEGATIVE )
            u = -u;

      return u;
      }

// MSC acos() does not allow rounding control
// So we have to do it manually by using the entity acos()=π-asin()
//
inline interval<double> acos(const interval<double>& x)
      {
      interval<double> pi, res;
      const double c1(1);

      if (x.lower() >= c1 || x.upper() <= -c1)
            {
            throw interval<double>::domain_error(); return x;
            }

      pi.lower( pidouble(ROUND_DOWN) );
      pi.upper( pidouble(ROUND_UP) );
      res = pi * interval<double>(0.5) - asin( x );
      return res;
      }

// MSC asin() does not allow rounding control for the asin()
// Since we dont normally do it using float arithmetic (as for sqrt(), log() and log10())
// we simply just call the interval<double> version
// of asin() and convert back to float preserving as much accuracy as possible
//
inline interval<float> asin(const interval<float>& x)
      {
      interval<double> asin(const interval<double>&);
      interval<double> fx(x);
      float lower, upper;

      fx = asin(fx);
      lower = tofloat(fx.lower(), ROUND_DOWN);
      upper = tofloat(fx.upper(), ROUND_UP);
      return interval<float>(lower, upper);
      }


// MSC acos() does not allow rounding control for the acos()
// Since we dont normally do it using float arithmetic (as for sqrt(), log() and log10())
// we simply just call the interval<double> version
```

```
// of acos() and convert back to float preserving as much accuracy as possible
//
inline interval<float> acos(const interval<float>& x)
	{
	interval<double> acos(const interval<double>&);
	interval<double> fx(x);
	float lower, upper;

	fx = acos(fx);
	lower = tofloat(fx.lower(), ROUND_DOWN);
	upper = tofloat(fx.upper(), ROUND_UP);
	return interval<float>(lower, upper);
	}

////////////////////////////////////////////////////////////////////////////////
///
/// END Interval sin(), cos(), tan(), asin(), acos(), atan()
///
////////////////////////////////////////////////////////////////////////////////
```

Source code: cin & cout operators

The standard output **cout** and input **cin** is straight forward. **cout** output the interval in the format of

[lower , upper]

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

[lower, upper]
[interval] which is equivalent with [interval, interval]

Or just:

interval which is equivalent with [interval, interval]

```
// Output Operator <<
//
template<class _Ty> inline std::ostream& operator<<( std::ostream&
strm, interval<_Ty>& a )
	{ return strm << "[" << a.lower() << "," << a.upper() << "]"; }

// Input operator >>
//
template<class _Ty> inline std::istream& operator>>( std::istream&
strm, interval<_Ty>& c )
   {
   _Ty l, u; char ch;
   if( strm >> ch && ch != '[')
      strm.putback(ch), strm >> l, u = l;
      else
      if( strm >> l >> ch && ch != ',')
            if( ch == ']')
               u = l;
            else
            strm.putback( ch ); // strm.setstate(std::ios::failbit);
```

```
        else
        if( strm >> u >> ch && ch != ']')
              strm.putback( ch ); // strm.setstate(ios_base::failbit);

if(!strm.fail())
        c = interval<_Ty>( l, u );

return strm;
}
```

Source code: intervaldouble.h

Now we are ready to list the full source. It can easily be copying and pasted into a application

File: intervaldouble.h

```
#ifndef INC_INTERVAL
#define INC_INTERVAL

/*
 ****************************************************************************
 *
 *
 *                      Copyright (c) 2002-2015
 *                      Future Team Aps
 *                      Denmark
 *
 *                      All Rights Reserved
 *
 *   This source file is subject to the terms and conditions of the
 *   Future Team Software License Agreement which restricts the manner
 *   in which it may be used.
 *   Mail: hve@hvks.com
 *
 ****************************************************************************
*/

/*
 ****************************************************************************
 *
 *
 * Module name     :   intervaldouble.h
 * Module ID Nbr   :
 * Description     :   Interval arithmetic template class
 *                     Works with both float and double
 * ---------------------------------------------------------------------------
 * Change Record   :
 *
 * Version       Author/Date              Description of changes
 * -------  -----------           ----------------------
 * 01.01 HVE/28dec14              Initial release
 *
 * End of Change Record
 * ---------------------------------------------------------------------------
*/


/* define version string */
static char _VinterP_[] = "@(#)intervaldouble.h 01.01 -- Copyright (C) Future Team Aps";

#include <float.h>
#include <algorithm>

// By default HARDWARE_SUPPORT controlled if IEEE754 floating point control can be used for interval
arithmetic.
// The intervaldouble.h requires this to be defined.
#define HARDWARE_SUPPORT

/// The four different interval classification
/// # ZERO                       a=0 && b=0
/// # POSITIVE            a>=0 && b>0
/// # NEGATIVE            a<0 && b<=0
/// # MIXED                       a<0 && b>0
enum int_class { NO_CLASS, ZERO, POSITIVE, NEGATIVE, MIXED };

/// The four different ronding modes
/// # ROUND_NEAR  Rounded result is the closest to the infinitely precise result.
/// # ROUND_DOWN  Rounded result is close to but no greater than the infinitely precise result.
/// # ROUND_UP    Rounded result is close to but no less than the infinitely precise result.
/// # ROUND ZERO  Rounded result is close to but no greater in absolute value than the infinitely
precise result.
enum round_mode { ROUND_NEAR, ROUND_UP, ROUND_DOWN, ROUND_ZERO };

//
```

```
// Interval class
// Realistically the class Type can be float, double. Any other type is not supported
// Since float and double are done unsing the Intel cpu (H/W) and using "specilization" .
//
template<class _IT> class interval {
   _IT low, high;
   public:
      typedef _IT value_type;

      // constructor. zero, one or two arguments for type _IT
      interval()                                                    { low = _IT(0); high = _IT(0); }
           interval( const _IT& d )                                 { low = _IT(d); high = _IT(d); }
           interval( const _IT& l, const _IT& h) { if( l < h ) { low =l; high = h; } else { low = h;
high = l; } }
           // Constrcutor for mixed type _IT != _X (base types). Allows auto construction of e.g.
interval<float_precision> x(float)
          template <class _X> interval(const _X& x) { low = _IT(x); high = _IT(x); }

      // constructor for any other type to _IT. Both up and down conversion possible
      template<class X> interval( const interval<X>& a )
                {
                if (a.lower() < a.upper()) { fpdown();  low = _IT(a.lower()); fpup();  high =
_IT(a.upper()); fpnear();  }
                else { fpdown();  low = _IT(a.upper()); fpup();  high = _IT(a.lower()); fpnear(); }
                }

      // Coordinate functions
      _IT upper() const                                         { return high; }
      _IT lower() const                                         { return low; }
      _IT upper( const _IT& u )                         { return ( high = u ); }
      _IT lower( const _IT& l )                         { return ( low = l ); }

      _IT center() const                                { return ( high + low ) / _IT(2); }
      _IT radius() const                                { _IT r; r =( high - low ) / _IT(2); if( r < _IT(0)
) r = -r; return r; }
           _IT width() const                                        { _IT r; r = high - low; if (r <
_IT(0)) r = -r; return r; }

           bool contain_zero() const                            { return low <= _IT(0) && _IT(0) <= high;
}  // Obsolete. use contains() instead.
           bool contain( const _IT& f=_IT(0)){ return low <= f && f <= high;  }
           bool contain(const interval<_IT>& i) { return low <= i.lower() && i.upper() <= high; }
           bool is_empty() const                                { return high < low; }

           enum int_class is_class() const {
                                                                                  if (low == _IT(0)
&& high == _IT(0)) return ZERO;
                                                                                  if (low >= _IT(0)
&& high > _IT(0)) return POSITIVE;
                                                                                  if (low < _IT(0)
&& high <= _IT(0)) return NEGATIVE;
                                                                                  if (low < _IT(0)
&& high > _IT(0)) return MIXED;
                                                                                  return NO_CLASS;
                                                                                  }

           // Operators
           operator short() const                   { return (short)((high + low) / _IT(2)); }
                 // Conversion to short
           operator int() const                                 { return (int)center() /*(( high + low ) /
_IT(2) )*/; }                     // Conversion to int
           operator long() const                                { return (long)((high + low) / _IT(2)); }
                           // Conversion to long
           operator unsigned short() const { return (unsigned short)((high + low) / _IT(2)); }
         // Conversion to unsigned short
           operator unsigned int() const            { return (unsigned int)((high + low) / _IT(2)); }
         // Conversion to unsigned int
           operator unsigned long() const   { return (unsigned long)((high + low) / _IT(2)); }
         // Conversion to unsigned long
           operator double() const                  { return (double)( ( high + low ) / _IT(2) ); }
                 // Conversion to double
           operator float() const                   { return high == low? (float)low : (float)((high +
low) / _IT(2)); }                          // Conversion to float

      _IT *ref_lower()                                          { return &low; }
      _IT *ref_upper()                                          { return &high; }

      // 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<_IT>& operator&=( const interval<_IT>& );
           interval<_IT>& operator|=( const interval<_IT>& );
           interval<_IT>& operator^=( const interval<_IT>& );

           // Exception class. No used
           class bad_int_syntax {};
           class bad_float_syntax {};
           class out_of_range   {};
           class divide_by_zero {};
           class domain_error   {};
           class base_error               {};
   };


// Unary and Binary arithmetic
// Arithmetic + Binary and Unary
template <class _IT, class _X> interval<_IT> operator+( const interval<_IT>&, const _X&);
template <class _IT, class _X> interval<_IT> operator+( const _X&, const interval<_IT>&);
template<class _IT> interval<_IT> operator+( const interval<_IT>&, const interval<_IT>& );
template<class _IT> interval<_IT> operator+( const interval<_IT>& );
                                // Unary

// Arithmetic - Binary and Unary
template <class _IT, class _X> interval<_IT> operator-(const interval<_IT>&, const _X&);
template <class _IT, class _X> interval<_IT> operator-(const _X&, const interval<_IT>&);
template<class _IT> interval<_IT> operator-( const interval<_IT>&, const interval<_IT>& );
template<class _IT> interval<_IT> operator-( const interval<_IT>& );
                                // Unary

// Arithmetic * Binary
template <class _IT, class _X> interval<_IT> operator*(const interval<_IT>&, const _X&);
template <class _IT, class _X> interval<_IT> operator*(const _X&, const interval<_IT>&);
template<class _IT> interval<_IT> operator*( const interval<_IT>&, const interval<_IT>& );

// Arithmetic / Binary
template <class _IT, class _X> interval<_IT> operator/(const interval<_IT>&, const _X&);
template <class _IT, class _X> interval<_IT> operator/(const _X&, const interval<_IT>&);
template<class _IT> interval<_IT> operator/( const interval<_IT>&, const interval<_IT>& );

// Boolean Comparison Operators
template <class _IT, class _X> bool operator==(const interval<_IT>&, const _X&);
template <class _IT, class _X> bool operator==(const _X&, const interval<_IT>&);
template <class _IT, class _X> bool operator==(const interval<_IT>&, const interval<_X>&);
//template<class _IT> bool operator==(const interval<_IT>&, const interval<_IT>&);

template <class _IT, class _X> bool operator!=(const interval<_IT>&, const _X&);
template <class _IT, class _X> bool operator!=(const _X&, const interval<_IT>&);
template <class _IT, class _X> bool operator!=(const interval<_IT>&, const interval<_X>&);
//template<class _IT> bool operator!=(const interval<_IT>&, const interval<_IT>&);

// Other functions
template<class _IT> interval<_IT> abs(const interval<_IT>&);

// Manifest Constants like PI, LN2 and LN10
inline interval<float> int_pifloat();
inline interval<double> int_pidouble();
inline interval<float> int_ln2float();
inline interval<double> int_ln2double();
inline interval<float> int_ln10float();
inline interval<double> int_ln10double();

// Elementary functions
inline interval<float> sqrt(const interval<float>&);
inline interval<double> sqrt(const interval<double>&);
inline interval<float> log( const interval<float>& );
inline interval<double> log(const interval<double>&);
inline interval<float> log10(const interval<float>&);
inline interval<double> log10(const interval<double>&);
inline interval<float> exp(const interval<float>&, const float);
inline interval<double> exp(const interval<double>&, const double);
inline interval<float> pow(const interval<float>&, const float );
inline interval<double> pow( const interval<double>&, const double );

// Trigonometric functions
inline interval<float> sin(const interval<float>&);
inline interval<double> sin(const interval<double>&);
inline interval<float> cos(const interval<float>&);
inline interval<double> cos(const interval<double>&);
inline interval<float> tan(const interval<float>&);
```

```
inline interval<double> tan(const interval<double>&);
inline interval<float> asin(const interval<float>&);
inline interval<double> asin(const interval<double>&);
inline interval<float> acos(const interval<float>&);
inline interval<double> acos(const interval<double>&);
inline interval<float> atan(const interval<float>&);
inline interval<double> atan(const interval<double>&);

////////////////////////////////////////////////////////////////////////////////////////////////////
///
//
//
//       Floating point control for the IEEE754 hardware. Only fo non managed application
//       Enable by defined #define HARDWARE_SUPPORT
//  Currently it only works with HARDWARE_SUPPORT defined.
//
//
////////////////////////////////////////////////////////////////////////////////////////////////////
///

#ifdef HARDWARE_SUPPORT
inline void fpnear()
         {
         unsigned int f87_cw, sse2_cw;
         int cc;
         //_controlfp_s( &currentControl, _RC_NEAR, _MCW_RC);
         cc=__control87_2(_RC_NEAR, _MCW_RC, &f87_cw, &sse2_cw );
         }

inline void fpdown()
         {
         unsigned int f87_cw, sse2_cw;
         int cc;
         //_controlfp_s( &currentControl, _RC_DOWN, _MCW_RC);
         cc=__control87_2(_RC_DOWN, _MCW_RC, &f87_cw, &sse2_cw);
         }

inline void fpup()
         {
         unsigned int f87_cw, sse2_cw;
         int cc;
         //_controlfp_s( &currentControl, _RC_UP, _MCW_RC);
         cc=__control87_2(_RC_UP, _MCW_RC, &f87_cw, &sse2_cw);
         }
#else
inline void fpnear()      {}
inline void fpdown()      {}
inline void fpup()                {}
#endif

////////////////////////////////////////////////////////////////////////////////////////////////////
///
//
//
//   End Floating point control for the IEEE754 hardware
//
//
////////////////////////////////////////////////////////////////////////////////////////////////////
///

// Output Operator <<
//
template<class _Ty> inline std::ostream& operator<<( std::ostream& strm, interval<_Ty>& a )
         { return strm << "[" << a.lower() << "," << a.upper() << "]"; }

// Input operator >>
//
template<class _Ty> inline std::istream& operator>>( std::istream& strm, interval<_Ty>& c )
   {
   _Ty l, u; char ch;
   if( strm >> ch && ch != '[')
      strm.putback(ch), strm >> l, u = l;
        else
      if( strm >> l >> ch && ch != ',')
               if( ch == ']')
                  u = l;
               else
             strm.putback( ch ); // strm.setstate(std::ios::failbit);
            else
          if( strm >> u >> ch && ch != ']')
                   strm.putback( ch ); //, strm.setstate(ios_base::failbit);
```

```
   if(!strm.fail())
            c = interval<_Ty>( l, u );

   return strm;
   }


////////////////////////////////////////////////////////////////////////////////////////////////////
///
//
//
//    Essential Operators =,+=,-=,*=,/=
//
//
////////////////////////////////////////////////////////////////////////////////////////////////////
///


// Assignment operator. Works for all class types
//
template<class _IT> inline interval<_IT>& interval<_IT>::operator=( const interval<_IT>& a )
   {
   low = a.lower();
   high = a.upper();
   return *this;
   }

// += operator. Works all other classes.
// Please note that this is for all integer classes. interval<int>, interval<long>
// were there os no loss of precision
//
template<class _IT> inline interval<_IT>& interval<_IT>::operator+=( const interval<_IT>& a )
   {
   fpdown();
   low += a.lower();
   fpup();
   high += a.upper();
   fpnear();
   return *this;
   }

// -= operator. Works all other classes.
// Please note that this is for all integer classes. interval<int>, interval<long>
// were there is no loss of precision
//
template<class _IT> inline interval<_IT>& interval<_IT>::operator-=( const interval<_IT>& a )
   {
        fpdown();
   low -= a.high;
   fpup();
   high -= a.low;
   fpnear();
   return *this;
   }

// Works all other classes.
// Please note that this is for all interger classes. interval<int>, interval<long>,
// were there is no loss of precision
// Instead of doing the mindless low = MIN(low*a.high, low*a.low,high*a.low,high*a.high) and
// high = MAX(low*a.high, low*a.low,high*a.low,high*a.high) requiring a total of 8 multiplication
//
//   low, high, a.low, a.high    result
//    +     +     +     +        +  +  [ low*a.low, high*a.high ]
//    +     +     -     +        -  +  [ high*a.low, high*a.high ]
//    +     +     -     -        -  -  [ high*a.low, low*a.high ]
//    -     +     +     +        -  +  [ low*a.high, high*a.high ]
//    -     +     -     +        -  +  [ MIN(low*a.high,high*a.low), MAX(low*a.low,high*a.high) ]
//    -     +     -     -        -  -  [ high*a.low, low*a.low ]
//    -     -     +     +        -  -  [ low*a.high, high,a.low ]
//    -     -     -     +        -  -  [ low*a.high, low*a.low ]
//    -     -     -     -        +  +  [ high*a.high, low * a.low ]
//
 template<class _IT> inline interval<_IT>& interval<_IT>::operator*=( const interval<_IT>& a )
        {
        _IT l, h, t;

        if (low >= 0) //
        { // both low and high >= 0
                if (a.lower() >= 0)
                { // a.low >=0, a.high >= 0
```

```
                        fpdown();
                        l = low * a.lower();
                        fpup();
                        h = high * a.upper();
                }
                else
                if (a.upper() >= 0)
                {  //  a.low < 0, a.high >= 0
                        fpdown();
                        l = high * a.lower();
                        fpup();
                        h = high * a.upper();
                }
                else
                { // a.low and a.high < 0
                        fpdown();
                        l = high * a.lower();
                        fpup();
                        h = low * a.upper();
                }
        }
        else
        if (high >= 0)
        {  // low < 0, high >= 0
                if (a.lower() >= 0)
                { // a.low >=0, a.high >= 0
                        fpdown();
                        l = low * a.upper();
                        fpup();
                        h = high * a.upper();
                }
                else
                if (a.upper() >= 0)
                {  //  a.low < 0, a.high >= 0
                        fpdown();
                        l = low * a.upper(); if (l > (t = high * a.lower())) l = t;
                        fpup();
                        h = high * a.upper(); if (h < (t = low * a.lower())) h = t;
                }
                else
                { // a.low and a.high < 0
                        fpdown();
                        l = high * a.lower();
                        fpup();
                        h = low * a.lower();
                }
        }
        else
        { // low and high are < 0
                if (a.lower() >= 0)
                { // a.low >=0, a.high >= 0
                        fpdown();
                        l = low * a.upper();
                        fpup();
                        h = high * a.lower();
                }
                else
                if (a.upper() >= 0)
                {  //  a.low < 0, a.high >= 0
                        fpdown();
                        l = low * a.upper();
                        fpup();
                        h = low * a.lower();
                }
                else
                { // a.low and a.high < 0
                        fpdown();
                        l = high * a.upper();
                        fpup();
                        h = low * a.lower();
                }
        }

        low = l;
        high = h;
        fpnear();

        return *this;
        }

// Works for all other classes
```

```
// Please note that this is for all interger classes. interval<int>, interval<long>
// were there is no loss of precision
// Actually there is specialization for both <int>
template<class _IT> inline interval<_IT>& interval<_IT>::operator/=( const interval<_IT>& b )
   {
   interval<_IT> a, c;

   fpdown();
   c.low = (_IT)1 / b.upper();
   fpup();
   c.high = (_IT)1 / b.lower();
   fpnear();
   a = interval( low, high );
   c *= a;

   low = c.lower();
   high = c.upper();

   return *this;
   }

// Specialization for int and /=
//
inline interval<int>& interval<int>::operator/=( const interval<int>& b )
   {
   double tlow, thigh;
   interval<int> a;
   interval<double> c;

   tlow = 1 / (double)b.upper();
   thigh = 1 / (double)b.lower();

   a = interval( low, high );
   c = interval<double>( tlow, thigh );
   c *= a;

   low = (int)floor( c.lower() );
   high = (int)ceil( c.upper() );

   return *this;
   }

// Works on all classes.
// Return the intersection
//
template<class _IT> inline interval<_IT>& interval<_IT>::operator&=(const interval<_IT>& a)
        {
        if (a.lower() > low )
                low = a.lower();
        if (a.upper() < high)
                high = a.upper();
        if (low > high)  // Empty set
                {
                low = 0; high = 0;
                }

        return *this;
        }

// Works on all classes.
// Return the union
//
template<class _IT> inline interval<_IT>& interval<_IT>::operator|=(const interval<_IT>& a)
        {
        if (low > a.upper() || high < a.lower())
                {
                if (a.upper() - a.lower() > high - low)
                        { // return the largest set
                        low = a.lower();
                        high = a.upper();
                        }
                }
        else
                { // non empty intersection
                if (a.lower() < low)
                        low = a.lower();
                if (a.upper() > high)
                        high = a.upper();
                }
        }
```

```
// Works on all classes.
// Return the set minus
//
template<class _IT> inline interval<_IT>& interval<_IT>::operator^=(const interval<_IT>& a)
        {
        if ( a.lower() < high && a.upper() > low ) // intersection is not empty
                {
                if (a.upper() <= low)
                        low = a.upper();
                else
                        if (a.high() >= high)
                                high = a.lower();
                }

        return *this;
        }

////////////////////////////////////////////////////////////////////////////////
///
/// END Essential Operators
///
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////////////////////////////
///
//
//
//    Binary and Unary Operators +,-,*,/
//
//
////////////////////////////////////////////////////////////////////////////////////////////////////////
///

// Binary + operator
// Works for all classes
//
template<class _IT,class _X> inline interval<_IT> operator+(const interval<_IT>& a, const _X& b)
        {
        interval<_IT> c(a);

        c += interval<_IT>(_IT(b));
        return c;
        }

// Binary + operator
// Works for all classes
//
template<class _IT,class _X> inline interval<_IT> operator+( const _X& a, const interval<_IT>& b)
        {
        interval<_IT> c(b);

        c += interval<_IT>(_IT(a));
        return c;
        }

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

        c += b;
        return c;
        }


// 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, class _X> inline interval<_IT> operator-(const interval<_IT>& a, const _X& b)
        {
```

```
        interval<_IT> c(a);

        c -= interval<_IT>(_IT(b));
        return c;
        }

// Binary - operator
// Works for all classes
//
template<class _IT, class _X> inline interval<_IT> operator-(const _X& a, const interval<_IT>& b)
        {
        interval<_IT> c(a);

        c -= b;
        return c;
        }

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

   c -= b;
   return c;
   }


// Unary - operator
// Works for all classes
//
template<class _IT> inline interval<_IT> operator-( const interval<_IT>& a )
   {
   interval<_IT> c(0);

   c -= a;
   return c;
   }


// Binary * operator
// Works for all classes
//
template<class _IT, class _X> inline interval<_IT> operator*(const interval<_IT>& a, const _X& b)
        {
        interval<_IT> c(a);

        c *= interval<_IT>(_IT(b));
        return c;
        }

// Binary * operator
// Works for all classes
//
template<class _IT, class _X> inline interval<_IT> operator*(const _X& a, const interval<_IT>& b)
        {
        interval<_IT> c(b);

        c *= interval<_IT>(_IT(a));
        return c;
        }

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

   c *= b;
   return c;
   }

// Binary / operator
// Works for all classes
//
template<class _IT, class _X> inline interval<_IT> operator/(const interval<_IT>& a, const _X& b)
        {
        interval<_IT> c(a);
```

```
        c /= interval<_IT>(_IT(b));
        return c;
        }

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

        c /= b;
        return c;
        }

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

   if ( c == b && b.is_class() != ZERO )
         c = interval<_IT>(1,1);
   else
      c /= b;

   return c;
   }

// Binary & operator
// Return intersection
// Works for all classes
//
template<class _IT> inline interval<_IT> operator&( const interval<_IT>& a, const interval<_IT>& b )
        {
        interval<_IT> c(a);

        c &= b;
        return c;
        }

// Binary | operator.
// Return union
// Works for all classes
//
template<class _IT> inline interval<_IT> operator|(const interval<_IT>& a, const interval<_IT>& b)
        {
        interval<_IT> c(a);

        c |= b;
        return c;
        }

// Binary ^ operator
// Return set minus
// Works for all classes
//
template<class _IT> inline interval<_IT> operator^(const interval<_IT>& a, const interval<_IT>& b)
        {
        interval<_IT> c(a);

        c ^= b;
        return c;
        }

////////////////////////////////////////////////////////////////////////////////
///
/// END Binary and Unary Operators
///
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
///
/// Boolean Interval for == and !=
///
////////////////////////////////////////////////////////////////////////////////


// Binary == operator
// Works for all mixed classes
```

```
//
template<class _IT, class _X> inline bool operator==(const interval<_IT>& a, const _X& b)
        {
        interval<_IT> c(b);
        return c.lower() == a.lower() && c.upper() == a.upper();
        }

// Binary == operator
// Works for all mixed classes
//
template<class _IT, class _X> inline bool operator==(const _X& a, const interval<_IT>& b)
        {
        interval<_IT> c(a);
        return c.lower() == b.lower() && c.upper() == b.upper();
        }

// == operator
// Works for all classes
//
template<class _IT, class _X> inline bool operator==(const interval<_IT>& a, const interval<_X>& b)
        {
        return a.lower() == b.lower() && a.upper() == b.upper();
        }

// == operator
// Works for all classes
//
//template<class _IT> inline bool operator==(const interval<_IT>& a, const interval<_IT>& b)
//{
//      return a.lower() == b.lower() && a.upper() == b.upper();
//}


// Binary != operator
// Works for all mixed classes
//
template<class _IT, class _X> inline bool operator!=(const interval<_IT>& a, const _X& b)
        {
        interval<_IT> c(b);
        return c.lower() != a.lower() || c.upper() != a.upper();
        }

// Binary != operator
// Works for all mixed classes
//
template<class _IT, class _X> inline bool operator!=(const _X& a, const interval<_IT>& b)
        {
        interval<_IT> c(a);
        return c.lower() != b.lower() || c.upper() != b.upper();
        }

// != operator
// Works for all classes
//
template<class _IT, class _X> inline bool operator!=(const interval<_IT>& a, const interval<_X>& b)
        {
        return a.lower() != b.lower() || a.upper() != b.upper();
        }

// != operator
// Works for all classes
//
//template<class _IT> inline bool operator!=(const interval<_IT>& a, const interval<_IT>& b)
//      {
//      return a.lower() != b.lower() || a.upper() != b.upper();
//      }

////////////////////////////////////////////////////////////////////////////////
///
/// END Boolean operators
///
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
///
/// Interval abs()
///
////////////////////////////////////////////////////////////////////////////////

template<class _IT> inline interval<_IT> abs( const interval<_IT>& a )
        {
```

```
        if (a.lower() >= _IT(0) )
                return a;
        else
                if (a.upper() <= _IT(0) )
                        return -a;

        return interval<_IT>(_IT(0), ( a.upper() > -a.lower() ? a.upper() : -a.lower() ) );
        }

////////////////////////////////////////////////////////////////////////////////
///
/// END interval functions
///
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
///
/// Interval sqrt(), log10(), log(), exp() and pow()
///
////////////////////////////////////////////////////////////////////////////////

// Support function for correctly converting and double number back to a float
// By default IEE754 round to nearest and that will create incorrect result for interval arithmetic
//

// Support function for correctly converting and double number back to a float
// By default IEE754 round to nearest and that will create incorrect result for interval arithmetic
//
inline float tofloat(const double& d, enum round_mode rm)
        {
        float fres;

        switch (rm)
                {
                case ROUND_DOWN:  fpdown();  break;
                case ROUND_UP: fpup();  break;
                }

        fres = (float)d;
        fpnear();
        return fres;
        }

// If H/W Support allows us to control the rounding mode then we can do it directly.
// Log(2) for double
inline double ln2double(enum round_mode rm)
        {
        double res;

        switch (rm)
                {
                case ROUND_DOWN: fpdown(); break;
                case ROUND_UP: fpup(); break;
                }

        _asm
                {
                fldln2;                             Load ln2
                fstp qword ptr[res]; Store result in res
                }
        fpnear();

        return res;
        }

// Log(10) for double
inline double ln10double(enum round_mode rm)
        {
        double res;

        switch (rm)
                {
                case ROUND_DOWN: fpdown(); break;
                case ROUND_UP: fpup(); break;
                }

        _asm
                {
                ; ln10 = FLDL2T * FLDLN2
                fldl2t;                 Load log2(10)
                fldln2;                                         Load LN2
```

```
                fmulp st(1),st;                 Calculate LN2 * lOG2(10)
                fstp qword ptr[res];      Store ln10 in result
                }
        fpnear();

        return res;
        }

// PI for double
inline double pidouble(enum round_mode rm)
        {
        double res;

        switch (rm)
                {
                case ROUND_DOWN: fpdown(); break;
                case ROUND_UP: fpup(); break;
                }

        _asm
                {
                fldpi;                              Load PI
                fstp qword ptr[res]; Store result in res
                }
        fpnear();

        return res;
        }

// Sqrt() for double
inline double sqrtdouble( double d, enum round_mode rm )
        {
        double sq = d;

        switch( rm )
        {
        case ROUND_DOWN: fpdown(); break;
        case ROUND_UP: fpup(); break;
        }

        _asm
                {
                fld qword ptr[sq];  Load lower into floating point stack
                fsqrt;              Calculate sqrt
                fstp qword ptr[sq]; Store result in sq
                }
        fpnear();

        return sq;
        }

// Sqrt() for float
inline float sqrtfloat(float f, enum round_mode rm)
        {
        double sq = f;
        float fres;

        sq = sqrtdouble(sq, rm);
        fres = tofloat(sq, rm);
        return fres;
        }

// log() for double
inline double logdouble(double d, enum round_mode rm)
        {
        double lg = d;

        switch (rm)
                {
                case ROUND_DOWN: fpdown(); break;
                case ROUND_UP: fpup(); break;
                }

        _asm
                {
                fld qword ptr[lg];      Load lower into floating point stack
                fldln2;                 Load loge2
                fxch st(1);             Exchange stack top
                fyl2x;                  Calculate y * ln2 x
                fstp qword ptr[lg];     Store result in lower
                }
```

```
        fpnear();

        return lg;
        }

// log() for float
inline float logfloat(float f, enum round_mode rm)
        {
        double lg = (double)f;
        float fres;

        lg = logdouble( lg, rm );
        fres = tofloat(lg, rm);
        return fres;
        }

// log10() for double
inline double log10double(double d, enum round_mode rm)
        {
        double lg = d;

        switch (rm)
                {
                case ROUND_DOWN: fpdown(); break;
                case ROUND_UP: fpup(); break;
                }

        _asm
                {
                fld qword ptr[lg];      Load lower into floating point stack
                fldlg2;                 Load log10(2)
                fxch st(1);             Exchange stack top
                fyl2x;                  Calculate y * ln2 x
                fstp qword ptr[lg];     Store result in lg
                }

        fpnear();

        return lg;
        }

// log10 for float
inline float log10float(float f, enum round_mode rm)
        {
        double lg = (double)f;
        float fres;

        lg = log10double( lg, rm );
        fres = tofloat(lg, rm);
        return fres;
        }


// sqrt for float using managed code.
//
inline interval<float> sqrt( const interval<float>& x )
   {
   float lower, upper;
#ifdef HARDWARE_SUPPORT
   lower = sqrtfloat( x.lower(), ROUND_DOWN );
   upper = sqrtfloat( x.upper(), ROUND_UP );
#else
#endif
   return interval<float>( lower, upper );
   }

// sqrt for double using managed code.
//
inline interval<double> sqrt( const interval<double>& x )
   {
   double lower, upper;

        lower = sqrtdouble( x.lower(), ROUND_DOWN );
        upper = sqrtdouble( x.upper(), ROUND_UP );
   return interval<double>( lower, upper );
   }


// log for float using manged code.
```

```
//
inline interval<float> log( const interval<float>& x )
         {
         float lower, upper;

         lower = logfloat( x.lower(), ROUND_DOWN);
         upper = logfloat( x.upper(), ROUND_UP);
   return interval<float>( lower, upper );
   }

// log for double using managed code.
//
inline interval<double> log( const interval<double>& x )
         {
         double lower, upper;

         lower = logdouble( x.lower(), ROUND_DOWN);
         upper = logdouble( x.upper(), ROUND_UP);
   return interval<double>( lower, upper );
   }

// log10 for float using manged code.
//
inline interval<float> log10( const interval<float>& x )
   {
   float lower, upper;

   lower = log10float( x.lower(), ROUND_DOWN);
   upper = log10float(x.upper(), ROUND_UP);
   return interval<float>( lower, upper );
   }

// log10 for double using managed code.
//
inline interval<double> log10( const interval<double>& x )
   {
         double lower, upper;

         lower = log10double( x.lower(), ROUND_DOWN);
         upper = log10double( x.upper(), ROUND_UP);

   return interval<double>( lower, upper );
   }


// MSC exp() does not allow rounding control
// So we have to do it manually
// Use a taylor series until their is no more change in the result
// exp(x) == 1 + x + x^2/2!+x^3/3!+....
// Equivalent with the same standard C function call
// use argument reduction via exp(x)=(exp(x/2^k)2^k
// And use Brent enhancement using the double formula:
// expm(x)=exp(x)-1 && expm(2x)=expm(x)(2+expm(x)) on the backend to preseve
// loss of significance digits
//
inline interval<double> exp(const interval<double>& x)
         {
         int  i, k = 0;
         interval<double> c, res, p0, old;
         const interval<double> c1(1), c2(2);

         c = x;
         if (x.is_class() == NEGATIVE)
                  c = abs(c);

         // Determine Reduction factor
         k = int((log(2) + log(abs(c.center()))) / log(2));
         k = std::min(k, 10);
         if (k > 0)
                  {
                  i = 2 << (k - 1);
                  c /= interval<double>( (double)i );
                  }

         p0 = c;
         old = c1;
         res = old + p0;
         for (i = 2; i < 100 && (res.lower() != old.lower() && res.upper() != old.upper()); i++)
                  {
                  old = res;
                  p0 *= ( c / interval<double>((double)i));
```

```
                res += p0;
                }

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

        if (x.is_class() == NEGATIVE)
                res = c1 / res;

        return res;
        }

// MSC exp() does not allow rounding control for the exp()
// Since we dont normally do it using float arithmetic (as for sqrt(), log() and log10()) we simply
just call the interval<double> version
// of exp() and convert back to float preserving as much accuracy as possible
//
inline interval<float> exp(const interval<float>& x)
        {
        interval<double> exp(const interval<double>&);
        interval<double> fx(x);
        float lower, upper;

        fx = exp(fx);
        lower = tofloat(fx, ROUND_DOWN);
        upper = tofloat(fx, ROUND_UP);
        return interval<double>(lower, upper);
        }


// MSC pow() does not allow rounding control
// So we have to do it manually
// x^y == exp( y * ln( x ) ) );
// To avoid loss of precision we actually perform the operation using double and then
// convert the result back to float. This is consistent with the pow() that only takes double as an
argument.
//
inline interval<float> pow(const interval<float>& x, const float y)
        {
        interval<double> c(x);
        float upper, lower;

        c = log(c);
        c *= interval<double>(y);
        c = exp(c);
        lower = (float)c.lower();
        upper = (float)c.upper();
        if (lower > c.lower() )
                lower -= lower * 0.5f * FLT_EPSILON;
        if (upper < c.upper() )
                upper += upper * 0.5f * FLT_EPSILON;

        return interval<float>(lower, upper);
        }

// MSC pow() does not alllow rounding control
// So we have to do it manually
// x^y == exp( y * ln( x ) ) );
//
inline interval<double> pow(const interval<double>& x, const double y)
        {
        interval<double> c;

        c = log(x);
        c *= interval<double>(y);
        c = exp(c);

        return c;
        }

////////////////////////////////////////////////////////////////////////////////
///
/// END Interval sqrt(), log10(), log(), exp(), pow()
///
////////////////////////////////////////////////////////////////////////////////
```

```
////////////////////////////////////////////////////////////////////////////////
///
/// Interval constants like PI, LN2 and LN10
///
////////////////////////////////////////////////////////////////////////////////

// Load manifest constant PI for double
//
inline interval<double> int_pidouble()
         {
         interval<double> pi;

         pi.lower( pidouble(ROUND_DOWN) );
         pi.upper( pidouble(ROUND_UP) );
         return pi;
         }

// Load manifest constant PI for float
//
inline interval<float> int_pifloat()
         {
         interval<double> pid;
         interval<float> pif;

         pid = int_pidouble();
         pif.lower(tofloat(pid.lower(), ROUND_DOWN));
         pif.upper(tofloat(pid.upper(), ROUND_UP));
         return pif;;
         }



// Load manifest constant lN 2 for double
//
inline interval<double> int_ln2double()
         {
         interval<double> ln2;

         ln2.lower(ln2double(ROUND_DOWN));
         ln2.upper(ln2double(ROUND_UP));
         return ln2;
         }

// Load manifest constant LN2 for float
//
inline interval<float> int_ln2float()
         {
         interval<double> ln2d;
         interval<float> ln2f;

         ln2d = int_ln2double();
         ln2f.lower(tofloat(ln2d.lower(), ROUND_DOWN));
         ln2f.upper(tofloat(ln2d.upper(), ROUND_UP));
         return ln2f;;
         }

// Load manifest constant ln10 for double
//
inline interval<double> int_ln10double()
         {
         interval<double> ln10;

         ln10.lower(ln10double(ROUND_DOWN));
         ln10.upper(ln10double(ROUND_UP));
         return ln10;
         }

// Load manifest constant LN10 for float
//
inline interval<float> int_ln10float()
         {
         interval<double> ln10d;
         interval<float> ln10f;

         ln10d = int_ln10double();
         ln10f.lower(tofloat(ln10d.lower(), ROUND_DOWN));
         ln10f.upper(tofloat(ln10d.upper(), ROUND_UP));
         return ln10f;;
         }
```

```
////////////////////////////////////////////////////////////////////////////////
///
/// END Interval constants
///
////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////
///
/// Interval sin(), cos(), tan(), asin(), acos(), atan()
///
////////////////////////////////////////////////////////////////////////////////


// Sin() for double interval
//
inline double sindouble(double d, enum round_mode rm)
        {
        double res = d;

        switch (rm)
                {
                case ROUND_DOWN: fpdown(); break;
                case ROUND_UP: fpup(); break;
                }

        _asm
                {
                fld qword ptr[res];  Load lower into floating point stack
                fsin;                               Calculate sin
                fstp qword ptr[res]; Store result in lower
                }
        fpnear();

        return res;
        }

// Sin() for float interval
//
inline float sinfloat(float f, enum round_mode rm)
        {
        double res = f;
        float fres;

        res = sindouble(res, rm);
        fres = tofloat(res, rm);
        return fres;
        }


// cos() for double interval
//
inline double cosdouble(double d, enum round_mode rm)
        {
        double res = d;

        switch (rm)
                {
                case ROUND_DOWN: fpdown(); break;
                case ROUND_UP: fpup(); break;
                }

        _asm
                {
                fld qword ptr[res];  Load lower into floating point stack
                fcos;                               Calculate cos
                fstp qword ptr[res]; Store result in lower
                }
        fpnear();

        return res;
        }

// Cos(x) for float interval
//
inline float cosfloat(float f, enum round_mode rm)
        {
        double res = f;
        float fres;
```

```
        res = cosdouble( res, rm );
        fres = tofloat(res, rm);
        return fres;
        }

// tan() for double interval
//
inline double tandouble(double d, enum round_mode rm)
        {
        double res = d;

        switch (rm)
                {
                case ROUND_DOWN: fpdown(); break;
                case ROUND_UP: fpup(); break;
                }

        _asm
                {
                fld qword ptr[res];  Load lower into floating point stack
                fptan;               Calculate tan
                fstp qword ptr[res]; Pop ST(0) and ignore
                fstp qword ptr[res]; Store result
                }
        fpnear();

        return res;
        }

// Tan(x) for float interval
//
inline float tanfloat(float f, enum round_mode rm)
        {
        double res = f;
        float fres;

        res = tandouble( res, rm );
        fres = tofloat( res, rm );
        return fres;
        }

// atan() for double interval
//
inline double atandouble(double d, enum round_mode rm)
        {
        double res = d;

        switch (rm)
                {
                case ROUND_DOWN: fpdown(); break;
                case ROUND_UP: fpup(); break;
                }

        _asm
                {
                fld qword ptr[res];              Load lower into floating point stack
                fld1;                                    Load 1.0 on top of stack
                fpatan;                                  Calculate tan
                fstp qword ptr[res];     Store result
                }
        fpnear();

        return res;
        }

// Return atan(x) rounded up or rounded down
//
inline float atanfloat(float f, enum round_mode rm)
        {
        double res = f;
        float fres;

        res = atandouble( res, rm );
        fres = tofloat(res, rm);
        return fres;  // or alternatively return tofloat(atandouble((double)f, rm ), rm );
        }


// Interval sin(x) for float
//
inline interval<float> sin(const interval<float>& x)
```

```
        {
        float lower, upper;

        lower = sinfloat(x.lower(), ROUND_DOWN);
        upper = sinfloat(x.upper(), ROUND_UP);
        return interval<float>(lower, upper);
        }

// Interval Sin(x) for double
//
inline interval<double> sin(const interval<double>& x)
        {
        double lower, upper;

        lower = sindouble(x.lower(), ROUND_DOWN);
        upper = sindouble(x.upper(), ROUND_UP);
        return interval<double>(lower, upper);
        }



// Interval cos(x) for float.
//
inline interval<float> cos(const interval<float>& x)
        {
        float lower, upper;

        lower = cosfloat(x.lower(), ROUND_DOWN);
        upper = cosfloat(x.upper(), ROUND_UP);
        return interval<float>(lower, upper);
        }

// Interval Cos(x) for double.
//
inline interval<double> cos(const interval<double>& x)
        {
        double lower, upper;

        lower = cosdouble(x.lower(), ROUND_DOWN);
        upper = cosdouble(x.upper(), ROUND_UP);
        return interval<double>(lower, upper);
        }



// Interval tan(x) for float.
//
inline interval<float> tan(const interval<float>& x)
        {
        float lower, upper;

        lower = tanfloat(x.lower(), ROUND_DOWN);
        upper = tanfloat(x.upper(), ROUND_UP);
        return interval<float>(lower, upper);
        }

// Interval Tan(x) for double.
//
inline interval<double> tan(const interval<double>& x)
        {
        double lower, upper;

        lower = tandouble(x.lower(), ROUND_DOWN);
        upper = tandouble(x.upper(), ROUND_UP);
        return interval<double>(lower, upper);
        }

// Interval Arctan(x) for float.
//
inline interval<float> atan(const interval<float>& x)
        {
        float lower, upper;

        lower = atanfloat(x.lower(), ROUND_DOWN);
        upper = atanfloat(x.upper(), ROUND_UP);
        return interval<float>(lower, upper);
        }



// Interval ArcTan(x) for double.
```

```
//
inline interval<double> atan(const interval<double>& x)
        {
        double lower, upper;

        lower = atandouble(x.lower(), ROUND_DOWN);
        upper = atandouble(x.upper(), ROUND_UP);
        return interval<double>(lower, upper);
        }

// MSC asin() does not allow rounding control
// So we have to do it manually
/// Description:
///   Use a taylor series until their is no more change in the result
///   asin(x) == x + x^3/(2*3)+(1*3)x^5/(2*4*5)+(1*3*5)x^7/(2*4*6*7)....
///   Use argument reduction via the identity arcsin(x)=2arcsin(x)/(sqrt(2)+sqrt(1-x*x))
//        This function replace the other function using newton iteration. Taylor series is
significant
//    faster e.g 50% for 10 digits, 3x for 100 digits and 5x for 1000 digits.
//
inline interval<double> asin(const interval<double>& x)
        {
        int k, sign;
        interval<double> r, u, v, v2, sqrt2, lc, uc;
        const double c1(1), c2(2);

        if (x.lower() >= c1 || x.upper() <= -c1)
                {
                throw interval<double>::domain_error(); return x;
                }

        v = x;
        if (v.lower() < -c1)
                v.lower(-c1);
        if (v.upper() > c1)
                v.upper(c1);

        sign = v.is_class();
        if (sign == NEGATIVE)
                v = -v;

        // Now use the identity arcsin(x)=2arcsin(x)/(sqrt(2)+sqrt(1-x*x))
        // until argument is less than dlimit
        // Reduce the argument to below 0.5 to make the newton run faster
        sqrt2 = interval<double>(c2);                                   // Ensure correct number of digits
        sqrt2 = sqrt(sqrt2);      // Now calculate sqrt2 with precision digits
        for (k = 0; v.lower() > 0.5; k++)
                v /= sqrt2 * sqrt(interval<double>(c1) + sqrt(interval<double>(c1) - v * v));

        v2 = v * v;
        r = v;
        u = v;
        // Now iterate using taylor expansion
        for (unsigned int j = 3;; j += 2)
                {
                uc = interval<double>((j - 2) * (j - 2));
                lc = interval<double>(j * j - j);
                v = uc * v2 / lc;
                r *= v;
                if( u.lower() + r.lower() == u.lower() || u.upper() + r.upper() == u.upper())
                        break;
                u += r;
                }

        if (k > 0)
                u *= interval<double>(1 << k);

        if (sign == NEGATIVE )
                u = -u;

        return u;
        }

// MSC acos() does not allow rounding control
// So we have to do it manually
/// Description:
///   Use a taylor series until their is no more change in the result
///   asin(x) == x + x^3/(2*3)+(1*3)x^5/(2*4*5)+(1*3*5)x^7/(2*4*6*7)....
///   Use argument reduction via the identity arcsin(x)=2arcsin(x)/(sqrt(2)+sqrt(1-x*x))
//        This function replace the other function using newton iteration. Taylor series is
significant
```

```
//    faster e.g 50% for 10 digits, 3x for 100 digits and 5x for 1000 digits.
//
inline interval<double> acos(const interval<double>& x)
        {
        interval<double> pi, res;
        const double c1(1);

        if (x.lower() >= c1 || x.upper() <= -c1)
                {
                throw interval<double>::domain_error(); return x;
                }

        pi.lower( pidouble(ROUND_DOWN) );
        pi.upper( pidouble(ROUND_UP) );
        res = pi * interval<double>(0.5) - asin( x );
        return res;
        }

// MSC asin() does not allow rounding control for the asin()
// Since we dont normally do it using float arithmetic (as for sqrt(), log() and log10()) we simply
just call the interval<double> version
// of asin() and convert back to float preserving as much accuracy as possible
//
inline interval<float> asin(const interval<float>& x)
        {
        interval<double> asin(const interval<double>&);
        interval<double> fx(x);
        float lower, upper;

        fx = asin(fx);
        lower = tofloat(fx.lower(), ROUND_DOWN);
        upper = tofloat(fx.upper(), ROUND_UP);
        return interval<float>(lower, upper);
        }


// MSC acos() does not allow rounding control for the acos()
// Since we dont normally do it using float arithmetic (as for sqrt(), log() and log10()) we simply
just call the interval<double> version
// of acos() and convert back to float preserving as much accuracy as possible
//
inline interval<float> acos(const interval<float>& x)
        {
        interval<double> acos(const interval<double>&);
        interval<double> fx(x);
        float lower, upper;

        fx = acos(fx);
        lower = tofloat(fx.lower(), ROUND_DOWN);
        upper = tofloat(fx.upper(), ROUND_UP);
        return interval<float>(lower, upper);
        }


////////////////////////////////////////////////////////////////////////////////
///
/// END Interval sin(), cos(), tan(), asin(), acos(), atan()
///
////////////////////////////////////////////////////////////////////////////////

#endif
```

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 Airthmetic: from Principles to Implementation