`filib++` - Interval Library Specification and Reference Manual

Michael Lerch, German Tischler,
Jürgen Wolff von Gudenberg

Report No. 279 August 2001

Lehrstuhl für Informatik II
Universität Würzburg
Am Hubland
97074 Würzburg

{wolff}@informatik.uni-wuerzburg.de

Parts of this report have been published in [8, 9]

Contents

Abstract

`filib++` is an extension of the interval library `filib` originally developed in Karlruhe [2]. The most important aim of the latter was the fast computation of guaranteed bounds for interval versions of a comprehensive set of elementary function. `filib++` extends this library in two aspects. First, it adds a second mode, the "extended" mode, that extends the exception-free computation mode using special values to represent infinities and NotaNumber known from the IEEE floating-point standard 754 to intervals. In this mode so-called containment sets are computed to enclose the topological closure of a range of a function defined over an interval [5]. Second, state of the art design uses templates and traits classes in order to get an efficient, easily extendable and protable library, fully according to the C++ standard [1].

Overview

Chapter 1 presents the difference between the normal mode computing interval evaluations and the extend mode computing containment sets. The functionality of the library is roughly sketched.
Chapter 2 then shortly explains the inner structure and describes how to use it. Some sample programs are listed.
Chapters 3,4,5 contain the specification of the complete interface.
Finally, chapter 6 gives some installation hints.

Chapter 1

Interval Evaluation and Containment Evaluation

1.1 Interval Evaluation

We assume that the reader is familiar with the basic ideas of interval arithmetic. In this introductory chapter we use bold face for continuous intervals, represented by two real bounds.

$$\mathbf{x} = [\underline{x}, \overline{x}] = \{x \in \mathbb{R} | \underline{x} \leq x \leq \overline{x}\}$$

$\mathbb{IR}$ denotes the space of all finite intervals.
Let us deal with the enclosure of a range of a function, one of the main topics of interval arithmetic. We restrict our consideration to the one-dimensional case, extensions to more dimensions are obvious. Given an arithmetic function $f : D_f \subseteq \mathbb{R} \rightarrow \mathbb{R}$, $f(\mathbf{x})$ denotes the range of values of f over the interval $\mathbf{x} \subseteq D_f$.

Definition 1 *:*
*The **interval evaluation** $\mathbf{f} : \mathbb{IR} \rightarrow \mathbb{IR}$ of f is defined as the function that is obtained by replacing every occurence of the variable x by the interval variable $\mathbf{x}$ and by replacing every operator by its interval arithmetic counterpart and every elementary function by its range. Note, that this definition only holds, if all operations are executable without exception.*

The following theorem is known as the fundamental theorem of interval arithmetic.

Theorem 1 *:*
If the interval evaluation is defined, we have

$$f(\mathbf{x}) \subseteq \mathbf{f}(\mathbf{x})$$

The interval evaluation is not defined, if $\mathbf{x}$ contains a point $y \notin D_f$. Division by an interval containing 0, e.g., is forbidden. But note, that even if $\mathbf{x} \subseteq D_f$, $\mathbf{f}$ may not be defined. The result depends on the syntactic formulation of the expression.
$f_1(x) = \frac{1}{x \cdot x + 2}$
$\mathbf{f}_1([-2, 2])$ is not defined, because $[-2, 2] \cdot [-2, 2] = [-4, 4]$

whereas $f_2(x) = \frac{1}{x^2+2}$
yields $\mathbf{f}_2([-2, 2]) = [1/6, 1/2]$.
The elementary functions f are defined as the set of all function values, that is an interval, because the functions are continuous over their domain. That means that the interval evaluation is equal to the range, if it is defined. The interval evaluation is not defined, if the argument interval contains a point outside the domain of the corresponding function.
$\mathbf{f}(\mathbf{x}) := f(\mathbf{x}) = \{f(x) | x \in \mathbf{x} \subseteq D_f\}$

1.2 Containment Evaluation

To overcome the difficulties with partially defined functions throwing exceptions, we introduce a second mode, the "extended" mode. Here, usually no exceptions are raised, but the domains of interval functions and ranges of interval results are consistently extended.
Following G. W. Walster in [4, 5] we define the containment set:

Definition 2 *:*
Let $f : D_f \subseteq \mathbb{R} \rightarrow \mathbb{R}$, then the containment set $f^* : \wp\mathbb{R}^* \mapsto \wp\mathbb{R}^*$ defined by

$$f^*(\mathbf{x}) := \{f(x) | x \in \mathbf{x} \cap D_f\} \cup \{\lim_{x \to x^*} f(x) | x \in D_f, x^* \in \mathbf{x}\} \subseteq \mathbb{R}^* \tag{1.1}$$

contains the extended range of f, where $\mathbb{R}^* = \mathbb{R} \cup \{-\infty\} \cup \{\infty\}$.

Hence, the containment set of a function is the closure of the range including all limits and accumulation points.
Our goal is now to define an analogon to the interval evaluation which encloses the containment set, and is easy to compute.
Let $\mathbb{IR}^*$ denote the set of all extendeded intervals with endpoints in $\mathbb{R}^*$.

Definition 3 *:*
*The **containment evaluation** $\mathbf{f}^* : \mathbb{IR}^* \rightarrow \mathbb{IR}^*$ of f is defined as the function that is obtained by replacing every occurence of the variable x by the interval variable $\mathbf{x}$ and by replacing every operator or function by its extended interval arithmetic counterpart.*

We then have

Theorem 2 *:*
The containment evaluation is always defined, and we have

$$f^*(\mathbf{x}) \subseteq \mathbf{f}^*(\mathbf{x})$$

For the proof of this theorem all arithmetic operators and elementary functions are extended to the closure of their domain. This can be done in a straight forward manner, cf. [4]. We apply the well known rules to compute with infinities. If we encounter an undefined operation like $0 \cdot \infty$ we deliver the set of all limits, i.e. $\mathbb{R}^*$. Note that negative values are also possible, since 0 can be approached from both sides. We show the containment sets for the basic arithmetic operations in the following tables.

$+$	$-\infty$	y	$+\infty$
$-\infty$	$-\infty$	$-\infty$	$\mathbb{R}^*$
x	$-\infty$	$x+y$	$+\infty$
$+\infty$	$\mathbb{R}^*$	$+\infty$	$+\infty$

Table 1.1: extended addition

$-$	$-\infty$	y	$+\infty$
$-\infty$	$\mathbb{R}^*$	$-\infty$	$-\infty$
x	$+\infty$	$x-y$	$-\infty$
$+\infty$	$+\infty$	$+\infty$	$\mathbb{R}^*$

Table 1.2: extended subtraction

$*$	$-\infty$	$y<0$	0	$y>0$	$+\infty$
$-\infty$	$+\infty$	$+\infty$	$\mathbb{R}^*$	$-\infty$	$-\infty$
$x<0$	$+\infty$	$x*y$	0	$x*y$	$-\infty$
0	$\mathbb{R}^*$	0	0	0	$\mathbb{R}^*$
$x>0$	$-\infty$	$x*y$	0	$x*y$	$+\infty$
$+\infty$	$-\infty$	$-\infty$	$\mathbb{R}^*$	$+\infty$	$+\infty$

Table 1.3: extended multiplication

$/$	$-\infty$	$y<0$	0	$y>0$	$+\infty$
$-\infty$	$[0,+\infty]$	$+\infty$	$\{-\infty,+\infty\}$	$-\infty$	$[-\infty,0]$
$x<0$	0	x/y	$\{-\infty,+\infty\}$	x/y	0
0	0	0	$\mathbb{R}^*$	0	0
$x>0$	0	x/y	$\{-\infty,+\infty\}$	x/y	0
$+\infty$	$[-\infty,0]$	$-\infty$	$\{-\infty,+\infty\}$	$+\infty$	$[0,+\infty]$

Table 1.4: extended division

$A=[\underline{a};\overline{a}]$	$B=[\underline{b};\overline{b}]$	Range	containment set
$0\in A$	$0\in B$	$\mathbb{R}^*$	$\mathbb{R}^*$
$0\in A$	$B=[0;0]$	$\{-\infty;+\infty\}$	$\mathbb{R}^*$
$\overline{a}<0$	$\underline{b}<\overline{b}=0$	$[\overline{a}/\underline{b},\infty)$	$[\overline{a}/\underline{b},\infty]$
$\overline{a}<0$	$\underline{b}<0<\overline{b}$	$(-\infty;\overline{a}/\overline{b}]\cup[\overline{a}/\underline{b},+\infty)$	$\mathbb{R}^*$
$\overline{a}<0$	$0=\underline{b}<\overline{b}$	$(-\infty;\overline{a}/\overline{b}]$	$[-\infty;\overline{a}/\overline{b}]$
$\underline{a}>0$	$\underline{b}<\overline{b}=0$	$(-\infty;\underline{a}/\underline{b}]$	$[-\infty;\underline{a}/\underline{b}]$
$\underline{a}>0$	$\underline{b}<0=\overline{b}$	$(-\infty;\underline{a}/\underline{b}]\cup[\underline{a}/\overline{b},+\infty)$	$\mathbb{R}^*$
$\underline{a}>0$	$0=\underline{b}<\overline{b}$	$[\underline{a}/\overline{b};+\infty)$	$[\underline{a}/\overline{b};+\infty]$

Table 1.5: extended interval division

From these tables the definition of extended interval arithmetic can easily be deduced. For addition, subtraction, and multiplication can be returned, if a corresponding operation is encountered.
Some examples:
$[2,\infty]+[3,\infty]=[5,\infty]$
$[2,\infty]-[3,\infty]=\mathbb{R}^*$
$[2,\infty]*[-3,3]=\mathbb{R}^*$

Division is a little bit more subtle. Table 1.5 shows the cases where the denominator contains 0.

For the elementary functions Table 1.6 shows the extended domains and extended ranges.
The containment evaluation for an elementary function is computed by directly applying the definition of the containment set.

> $\mathbf{f}^*(\mathbf{x}) := \lozenge(\{f(x)|x\in\mathbf{x}\cap D_f\}\cup\{\lim_{x\to x^*} f(x)|x\in D_f, x^*\in\mathbf{x}\})$
>
> Here $\lozenge$ denotes the interval hull.
>
> If the argument lies strictly outside the domain of the function, we obtain the empty set as result.
>
> If the argument $\mathbf{x}$ contains a singularity the corresponding values for $\pm\infty$ are produced.
>
> The functions in containment mode never produce an overflow or illegal argument error.

Some examples:

name	domain	range	special values
sqr	$\mathbb{R}^*$	$[0, \infty]$	
power	$\mathbb{R}^* \times \mathbb{Z}$	$\mathbb{R}^*$	power([0,0],0) = [1,1]
pow	$[0, \infty] \times \mathbb{R}^*$	$[0, \infty]$	pow([0,0],[0,0]) $=[0, \infty]$
sqrt	$[0, \infty]$	$[0, \infty]$	
exp, exp10, exp2	$\mathbb{R}^*$	$[0, \infty]$	
expm1	$\mathbb{R}^*$	$[-1, \infty]$	
log, log10, log2	$[0, \infty]$	$\mathbb{R}^*$	log ([0, 0]) $= [-\infty]$
log1p	$[-1, \infty]$	$\mathbb{R}^*$	log1p $([-1, -1]) = [-\infty]$
sin	$\mathbb{R}^*$	$[-1, 1]$	
cos	$\mathbb{R}^*$	$[-1, 1]$	
tan	$\mathbb{R}^*$	$\mathbb{R}^*$	$\tan(\mathbf{x}) = \mathbb{R}^*$, if $\pi/2 + k\pi \in \mathbf{x}, k \in \mathbb{Z}$
cot	$\mathbb{R}^*$	$\mathbb{R}^*$	$\cot(\mathbf{x}) = \mathbb{R}^*$, if $k\pi \in \mathbf{x}, k \in \mathbb{Z}$
asin	$[-1, 1]$	$[-\pi/2, \pi/2]$	
acos	$[-1, 1]$	$[0, \pi]$	
atan	$\mathbb{R}^*$	$[-\pi/2, \pi/2]$	
acot	$\mathbb{R}^*$	$[0, \pi]$	
sinh	$\mathbb{R}^*$	$\mathbb{R}^*$	
cosh	$\mathbb{R}^*$	$[1, \infty]$	
tanh	$\mathbb{R}^*$	$[-1, 1]$	
coth	$\mathbb{R}^*$	$[-\infty, -1] \cup [1, \infty]$	coth[0,0] $=\mathbb{R}^*$
asinh	$\mathbb{R}^*$	$\mathbb{R}^*$	
acosh	$[1, \infty]$	$[0, \infty]$	
atanh	$[-1, 1]$	$\mathbb{R}^*$	
acoth	$[-\infty, -1] \cup [1, \infty]$	$\mathbb{R}^*$	$\text{acoth}[-1, -1] = [-\infty]$ $\text{acoth}[1, 1] = [\infty]$

Table 1.6: extended domains and ranges of elementary functions

$\log[-1, 1] = [-\infty, 0]$, $\sqrt{[-1, 1]} = [0, 1]$, $\log[-2, -1] = \emptyset$, $\coth[-1, 1] = \mathbb{R}^*$

The special values column shows the results of the interval version at points on the border of the open domain. In all cases the lim construction in (1.1) is applied and containment is guaranteed. Note that for the power function x^k only $\lim_{x \to 0} x^0$ is to be considered whereas x^y is calculated as $e^{y \ln x}$ in the pow function. We intentionally chose 2 different names, since $\text{power}(\mathbf{x}, k) \subseteq \text{pow}(\mathbf{x}, [k, k])$ does not hold for negative $\mathbf{x}$.

It has been shown in [5, 6], that using these extended operations the containment evaluation can be computed without exceptions.

1.3 Functional Specification of `filib++` – Overview

1.3.1 Internal Representation

In the normal mode of the library continuous real intervals are represented by two floating-point bounds, in fact a more general instantiation is possible, see 2.3. If a function or interval evaluation is not defined, the exception handling for the floating-point type is activated. That should terminate the program with an error message.

To cope with the closed set of real numbers in the extended mode, we accept the IEEE representation of $-\infty$, or ∞ as left or right hand bound of an extended interval, respectively. Thus we introduce one-sided open intervals:

$$\mathbf{x} = [\underline{x}, \infty] = \{x \in \mathbb{R} | x \geq \underline{x}\}$$

$$\mathbf{x} = [-\infty, \overline{x}] = \{x \in \mathbb{R} | x \leq \overline{x}\}$$

The real numbers larger than the overflow threshold `M` are

$$\mathbf{x} = [\mathtt{M}, \infty] = \{x \in \mathbb{R} | x \geq \mathtt{M}\}$$

$$\mathbf{x} = [-\infty, \mathtt{-M}] = \{x \in \mathbb{R} | x \leq \mathtt{-M}\}$$

.

$$[-\infty, \infty] = R^*$$

means all numbers.

The emtpy interval $\emptyset$ is represented as `[NaN,NaN]` . There are, however, no point intervals $[-\infty, -\infty]$ or $[\infty, \infty]$, we use the closed exterior intervals instead. This trick helps in a clear set theoretical interpretation and also facilitates the implementation. If we consider R^* as the base set all the open intervals can be interpreted as closed, and the usual formulae for interval arithmetic extended with obvious rules for $\pm\infty$ can be applied.

1.3.2 Construction and Access

The interval constructor expects two or one floating-point values as arguments with a default value for the point interval [0.0, 0.0]. Inf and sup are accessible via methods. There are checks for point interval (`isPoint`), empty interval (`isEmpty`) and unbounded interval (`isInfinite`). To check for sharpness of an interval the method `hasUlpAcc(n)` is provided, it is fulfilled, if both bounds differ at most by n ulps (unit last place).

1.3.3 Arithmetic Operations

The extended arithmetic operations for this data type, abbreviated as `I` and the base type `double` = `D` are accessible as overloaded operators. Operand combinations `I x I` `D x I`, and `I x D` are available for all operations $+, -, *, /$. Assignment operators $+=, -=, *=, /=$ are provided for `I x I` and `I x D` as methods of the class `interval`.

1.3.4 Relations

All 3 kinds of set-like, certainly or possibly comparisons [4] are provided as methods and as functions. The operators `==, !=, >=, <=` are overloaded for the set-like relations. We further supply the predicate y `interior x` $\iff \underline{x} < y < \overline{x}$.

1.3.5 Set Theoretic Functions

Utility methods and functions like midpoint, radius, diameter of an interval, its mignitude or magnitude, the interval of all absolute values, minima, maxima are provided. Lattice operations as intersection (`intersect`) or interval hull (`hull`)can be performed and the Hausdorff distance (`dist`) between two intervals can be computed.

1.3.6 Elementary Arithmetic Functions

The provided elementary functions and their (closed) domains and ranges are listed in table 1.6. In the normal mode the true domains of the functions are valid, i.e. $\log[0, 1.0]$ yields an error. The implemented elementary functions do not return least-bit accurate results, but an almost enclosure within a few ulps of the range is always guaranteed.

1.3.7 Input and Output

The standard input output operators are overloaded for intervals, but without appropriate rounding to the external decimal string format.

Chapter 2

Instantiation and Options

2.1 Namespace filib

The library is contained in the namespace `filib`. Hence, it is required to qualify each identifier of the library with `filib::` or to use the directive `using namespace filib`.

2.2 Two Modes and Two Versions

One template version and one macro version of the library are supplied. In either version two modes can be used. In the default or "normal" mode traditional interval evaluations are computed, and the program terminates, if the argument interval is not contained in the domain of a function. Overflow treatment is done according to the chosen floating point option.
In the "extended" mode containment sets over $\mathbb{R}^*$ are computed, no exceptions are raised. This mode is obtained by setting the constant `FILIB_EXTENDED` during compilation.
For the macro library, this constant `FILIB_EXTENDED` has to be set for the compilation of the library as well as the application program. For the template version, the constant is not used during the building of the static parts of the library and thus only has be provided when compiling application programs. That essentially means that you do not have to recompile the template version for switching the extended mode on or off.
In the following we describe the template version. Since the interface of both versions are essentially identical, a few statements about the macro version in section 2.5 are sufficient.

2.3 Template Parameters of Class `interval<>`

An interval is defined as a template. There are 2 template parameters, the underlying basic (floating-point) number type `N`, and the method, how to implement the directed roundings `rounding_control`.

2.3.1 Basic Number Type

An interval is given by 2 computer representable bounds, the lower bound or infimum and the upper bound or supremum

$$\mathbf{x} = [\underline{x}, \overline{x}]$$

It represents the continous set of all numbers from the mathematical set $\mathbb{S}$ that is approximated by the basic number type `N`, e.g. $\S = \mathbb{R}$ and `N=double`.

$$\mathbf{x} = [\underline{x}, \overline{x}] = \{x \in \mathbb{S} | \underline{x} \leq x \leq \overline{x}\}$$

The type `N` has to be an arithmetic type, i.e. all the operators have to be provided. In the extended mode constants for $\pm\infty$ und `NaN` are needed. These constants are supplied by the `fp_traits<>` class. Currently the only reasonable choices are `double` or `float`. Hence, only $\mathbb{R}$ can be taken for $\mathbb{S}$. The elementary functions for a basic type cannot be generated by instantiation of a template, but have to be implemented by suitable algorithms. In the current version of `filib++` only `double` functions are implemented. [1]

2.3.2 Rounding Control

Rounding, the use of the directed roundings in particular, is controlled by the second template parameter. It addresses the low-level, machine dependent part of the implementation.
The second parameter can have the following values

- *native_switched:* Before an operation computing a floating-point bound of the interval is executed, the rounding mode is switched via an assembler statement that changes the floating-point control word. This is an expensive operation, since the pipelines have to be cleared. After the interval operation the rounding mode is switched back to the default. This is our default mode for interval operations.

- *native_directed:* The same as *native_switched* but the rounding mode is not switched back. Note, that this mode influences the non-interval operations of the program.

- *native_onesided_switched:* Since $\Delta(x) = -(\nabla(-x))$, one directed rounding mode suffices for interval operations, where ∇ and Δ denote the rounding to $-\infty$ or ∞, respectively. After the interval operation the rounding mode is switched back to the default.

- *native_onesided_global:* Here, the rounding mode is set to ∇ and never changed. Note, that this mode influences the non-interval operations of the program. Before using this mode the user has to switch the rounding mode to rounding to $-\infty$. (`fp_traits<T>::downward()`)

[1] this implementation is the original `filib` code, see [2]

- *multiplicative:* If the architecture does not support directed rounding modes, they can be simulated by a multiplication of the result.

  We define two functions (R is a floating-point screen):

  low : $R \to R$
  $$\text{low}(a) := \begin{cases} a \geq 0: & a \odot \text{pred}(1) \\ a < 0: & a \odot \text{succ}(1) \end{cases} \tag{2.1}$$
  and high : $R \to R$
  $$\text{high}(a) := \begin{cases} a \geq 0: & a \odot \text{succ}(1) \\ a < 0: & a \odot \text{pred}(1). \end{cases} \tag{2.2}$$
  where $\odot$ means *round-to-nearest*-multiplication and where $\text{succ}(x) = \min\{y | y \in R, y > x\}$ or $\text{pred}(x) = \max\{y | y \in R, y < x\}$, respectively.

  Note that, because $\text{high}(a) = -\text{low}(-a)$, one function suffices.

  For a binary floating-point system $R = R(2, n, emin, emax)$ we have

  $$\text{pred}(1) = 1 - 2^{-n} = 1 - \frac{1}{2}\varepsilon^*$$

  $$\text{succ}(1) = 1 + 2^{1-n} = 1 + \varepsilon^*$$

  where $\varepsilon^* = 2^{1-n}$ denotes the bound for the relative rounding error ($|\varepsilon| \leq \varepsilon^*$).

  Theorem 3 *:*
  Let $R = R(2, n, emin, emax)$ be a binary floating-point system and $\bigcirc : \mathbb{R} \to R$ the rounding to the nearest. Then for all $x \in \mathbb{R}$ not in the over- or underflow range, i.e. $\mathtt{M} \geq |x| \geq 2^{emin-1}$ or $x = 0$, we have

  $$\text{low}(\bigcirc x) \leq x$$

  $$\text{high}(\bigcirc x) \geq x$$

  For the proof, see [9].

- *pred_succ_rounding:* Another way to simulate the directed roundings is to manipulate the representation of a floating-point number in order to obtain the predecessor or successor of that number. This is usually done using integer arithmetic. It can be sped up, if a table of $\text{ulp}(x)$ is stored containing the unit in last place with respect to the exponent of x.

- *no_rounding:* This mode is only for testing and tuning. Do NOT use it in applications. It does NOT compute enclosures.

  The one-sided rounding mode seems to be very appealing, since it minimizes switches of the rounding control. But note, that it currently does not work in the case of gradual underflow. For i386 architectures the rounding of values in the overflow range to $\pm\infty$ have to be forced by an intermediate storing of the value and, hence, the predicted performance gain is lost.

  IMPORTANT: it is necessary to call the method filib::fp_traits<N,K>::setup()

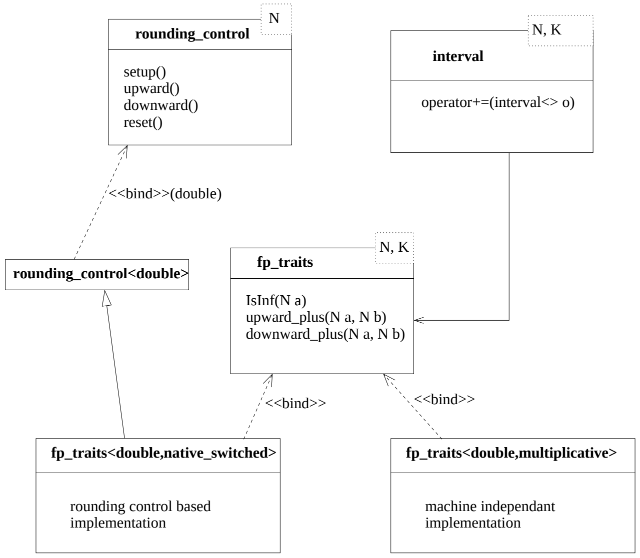
whenever you use instances of a different instantiation of the `interval<>` class with template parameters `N,K`. This is especially true at program start. When starting the usage of the `native_onesided_global` mode, the correct sequence is first calling `setup` and then `downward`.

2.4 Traits

Let us have a closer look into the design of the library. The `interval<>` class implements its operations relying on functions for directed floating-point arithmetic operations and on a function to reset the rounding mode. For example a simplified version of the `+=` operator looks like:

```
interval<N,K> & interval<N,K>::operator +=
(interval<N,K> const & o)
{
  INF=fp_traits<N,K>::downward_plus(INF,o.INF);
  SUP=fp_traits<N,K>::upward_plus(SUP,o.SUP);
  fp_traits<N,K>::reset();
  return *this;
}
```

These type and rounding mode specific operations are provided by a traits class `fp_traits<>` that handles all the operations depending on the type `N` and rounding control `K`. Specializations of this traits class for `double` and `float` and each of the described rounding control mechanisms are instantiated in the library. The specializations for rounding modes that rely on machine specific rounding control methods inherit these methods from an instantiation of the class `rounding_control`. That is illustrated in the following diagram.



2.5 Alternative Macro Version

For the mainly used data type `interval<double>` a non generic version that is highly optimized in speed is provided. It only supports the data type Interval, i.e. an interval of doubles. The switching between the various rounding modes is implemented via compile time constants. The arithmetic operations are defined as macros. This design certainly is not up to date concerning modern software engineering principles, but benchmarks showed, that the arithmetic was considerably faster, see [8]. The interface of the methods and functions is identical to the templated version.

2.6 Instantiation Examples

Some examples may help to use the library. Another example can be found in the `examples` directory of the distribution.

- `filib::interval<double> A;`
  This is the default instantiation. `A` is an interval over the floating-point type `double`. The second parameter is set to its default *filib::native_switched*

- `filib::interval<double,filib::multiplicative> A;`
  `A` is an interval over `double`. Multiplicative rounding is used. The hardware need not support directed roundings.

- `filib::interval<double,filib::native_onesided_global> A;`
  This is probably the fastest mode for most of the currently available machines. But it changes the floating-point semantics of the program.

2.7 Sample Programs

2.7.1 Evaluation of a Polynomial – Template Version

```
// Typical usage of the library filib++ (template version)

#include <interval/interval.hpp>
#include <vector> // STL container vector
#include <iostream>

using filib::interval;
using std::vector;
using std::cout;
using std::endl;

// Evaluation of a polynomial using Horner's rule
interval<double> horner
(
   // interval coefficients in STL container vector
   vector< interval<double> > const & pol,
   // interval argument
   interval<double> x
)
{
   // result
   interval<double> res = interval<double>();  // res is [0,0]

   vector< interval<double> >::const_iterator p= pol.begin();

   while ( p != pol.end() )
   {
      res *= x;
      res += *(p++);
   }
   return res; // now res == pol(x)
 }

int main()
{
   filib::fp_traits<double>::setup();
```



```
   interval<double> coeff2(2), coeff1(5), coeff0(3), x(0,1);
   vector< interval<double> > pol;
   pol.push_back(coeff2);
   pol.push_back(coeff1);
   pol.push_back(coeff0);
   // horner(pol,x) computes coeff2*x*x + coeff1*x + coeff0
   cout << "pol(x)= " << horner(pol,x) << endl;
   interval<double> y(-1,1);
   cout << "pol(y)= " << horner(pol,y) << endl;
   return 0;
}
```

2.7.2 Evaluation of a Polynomial – Macro Version

```
#include <Interval.h>
#include <vector>
#include <iostream>

/* Evaluation of a Polynomial  */
/* using Horner's rule */
Interval horner
(
  /*  interval coefficients in STL container vector*/
  std::vector< Interval > const & v,
  /* interval argument */
  Interval A
)
{
  /* result */
  Interval R = Interval();

  std::vector< Interval >::const_iterator
    a = v.begin(), b = v.end();

  while ( a != b )
  {
    R *= A;
    R += *(a++);
  }

  return R;
}
```

Chapter 3

The `fp_traits<>` class

3.1 Template Arguments

The `fp_traits<>` class is a template class with two template arguments. The first argument is supposed to be a numeric type, where there are currently implementations for `float` and `double`. The second parameter is a non-type parameter of type `rounding_strategy` as described in section 2.3.2. The following table shows the currently available combinations.

first param	second param
`double`	`native_switched`
`double`	`native_directed`
`double`	`multiplicative`
`double`	`no_rounding`
`double`	`native_onesided_switched`
`double`	`native_onesided_global`
`double`	`pred_succ_rounding`
`float`	`native_switched`
`float`	`native_directed`
`float`	`multiplicative`
`float`	`no_rounding`
`float`	`native_onesided_switched`
`float`	`native_onesided_global`

3.2 Utility Functions

The following static member functions are mandatory for all implementations of the `fp_traits<>` class (where `N` denotes the first template parameter):

- `bool` **IsNaN**`(N const & a)`
  test if `a` is not a number
- `bool` **IsInf**`(N const & a)`
  test if `a` is infinite

- `N const &` **infinity**`()`
  returns positive infinity
- `N const &` **ninfinity**`()`
  returns negative infinity
- `N const &` **quiet_NaN**`()`
  returns a quiet (non-signalling) **NaN**
- `N const &` **max**`()`
  returns the maximum finite value possible for `N`
- `N const &` **min**`()`
  returns the minimum finite positive non-denormalized value possible for `N`
- `N const &` **l_pi**`()`
  returns a value that is no bigger than π
- `N const &` **u_pi**`()`
  returns a value that is no smaller than π
- `int const &` **precision**`()`
  returns the current output precision
- `N` **abs**`(N const & a)`
  returns the absolute value of `a`
- `N` **upward_plus**`(N const & a, N const & b)`
  returns a value of type `N`. It shall be as close to $a+b$ as possible and no smaller than $a+b$.
- `N` **downward_plus**`(N const & a, N const & b)`
  returns a value of type `N`. It shall be as close to $a+b$ as possible and no bigger than $a+b$.
- `N` **upward_minus**`(N const & a, N const & b)`
  returns a value of type `N`. It shall be as close to $a-b$ as possible and no smaller than $a-b$.
- `N` **downward_minus**`(N const & a, N const & b)`
  returns a value of type `N`. It shall be as close to $a-b$ as possible and no bigger than $a-b$.
- `N` **upward_multiplies**`(N const & a, N const & b)`
  returns a value of type `N`. It shall be as close to $a \cdot b$ as possible and no smaller than $a \cdot b$.
- `N` **downward_multiplies**`(N const & a, N const & b)`
  returns a value of type `N`. It shall be as close to $a \cdot b$ as possible and no bigger than $a \cdot b$.
- `N` **upward_divides**`(N const & a, N const & b)`
  returns a value of type `N`. It shall be as close to a/b as possible and no smaller than a/b.

- N **downward_divides**(N const & a, N const & b)
  returns a value of type N. It shall be as close to a/b as possible and no bigger than a/b.

Chapter 4

The interval<> class

Let $\underline{x}$ or $\overline{x}$ denote infimum or supremum of the interval X, the interval this* is written as T = $[\underline{t}\ ,\ \overline{t}]$. N denotes the underlying basic number type, i.e the type of the bounds (see 2.3.1). Furthermore M is the largest representable number of type N and $\pm$INFTY denotes an internal constant for $\pm\infty$. [NaN, NaN] represents the empty interval where NaN denotes an internal representation for "Not a Number".

4.1 Basic Number Type

- The typename value_type is defined for the basic number type.
- The type of traits used by the class is introduced as traits_type.

4.2 Constructors

The following constructors are provided for the interval class:

- **interval**():
  The interval $[0, 0]$ is constructed.
- **interval**(N const & a):
  The interval $[a, a]$ is constructed.The point intervals for $+\infty$ and $-\infty$ are given by $[M, +$INFTY$]$ or $[-$INFTY$, -M]$, respectively.
- **interval**(N const & a, N const & b):
  If $a \le b$ the interval $[a, b]$ is constructed, otherwise the empty interval.
- **interval**(std::string const & infs, std::string const & sups) throw(filib::interval_io_exception):
  Construct an interval using the strings infs and sups. The bounds are first transformed to the primitive double type by the standard function strtod and then the infimum is rounded down and the supremum is rounded up. If the strings cannot be parsed by strtod, an exception of type filib::interval_io_exception is thrown.

- **interval**(`interval<> const & o`):
  Copy constructor, an interval equal to the interval `o` is constructed.

4.3 Assignment

- `interval<> &` **operator=**(`interval<> const & o`):
  The interval `o` is assigned.

4.4 Arithmetic Methods

The following methods are provided for updating arithmetic operations. Note that the usual operators are available as global functions (see 5.1).
The special cases of the extended mode are not explicitly mentioned here, see tables 1.1,1.2,1.3,1.4 for details.

- `interval<> const &` **operator+**`() const` (unary plus):
  The unchanged interval is returned.

- `interval<>` **operator-**`() const` (unary minus):
  $[-\overline{t}, -\underline{t}]$ is returned.

- `interval<> &` **operator+=**(`interval<> const & A`)(updating addition):

$$\underline{t} := \underline{t} + \underline{a},\ \overline{t} := \overline{t} + \overline{a}$$

- `interval<> &` **operator+=**(`N const & a`)(updating addition):

$$\underline{t} := \underline{t} + a,\ \overline{t} := \overline{t} + a$$

- `interval<> &` **operator-=**(`interval<> const & A`)(updating subtraction):

$$\underline{t} := \underline{t} - \overline{a},\ \overline{t} := \overline{t} - \underline{a}$$

- `interval<> &` **operator-=**(`N const & a`)(updaing subtraction):

$$\underline{t} := \underline{t} - a,\ \overline{t} := \overline{t} - a$$

- `interval<> &` **operator*=**(`interval<> const & A`)(updating multiplication):

$$\underline{t} := \min\{\underline{t} * \underline{a}, \overline{t} * \underline{a}, \underline{t} * \overline{a}, \overline{t} * \overline{a}\},\ \overline{t} := \max\{\underline{t} * \underline{a}, \overline{t} * \underline{a}, \underline{t} * \overline{a}, \overline{t} * \overline{a}\}$$

- `interval<> &` **operator*=**(`N const & a`)(updating multiplication):

$$\underline{t} := \min\{\underline{t} * a, \overline{t} * a\},\ \overline{t} := \max\{\underline{t} * a, \overline{t} * a\}$$

- `interval<> &` **operator/=**(`interval<> const & A`)(updating division):

$$\underline{t} := \min\{\underline{t}/\underline{a}, \overline{t}/\underline{a}, \underline{t}/\overline{a}, \overline{t}/\overline{a}\},\ \overline{t} := \max\{\underline{t}/\underline{a}, \overline{t}/\underline{a}, \underline{t}/\overline{a}, \overline{t}/\overline{a}\}$$

  The case $0 \in A$ throws an error in normal mode. $\mathbb{R}^*$ is returned in extended mode.

- `interval<> &` **operator/=**(`N const & a`)(updating division):

$$\underline{t} := \min\{\underline{t}/a, \overline{t}/a\},\ \overline{t} := \max\{\underline{t}/a, \overline{t}/a\}$$

  The case $a = 0$ throws an error in normal mode. $\mathbb{R}^*$ is returned in extended mode.

4.5 Access and Information Methods

Methods only available in extended mode are marked with the specific item marker *.

- `N const &` **inf**`() const`:
  returns the lower bound.
- `N const &` **sup**`() const`:
  returns the upper bound.

* `bool` **isEmpty**`() const`:
  returns `true`, iff `T` is the empty interval.
* `bool` **isInfinite**`() const`:
  returns `true`, iff `T` has at least one infinite bound.
* `static interval<>` **EMPTY**`() :`
  returns the empty interval.
* `static interval<>` **ENTIRE**`() :`
  returns $\mathbb{R}^*$.
* `static interval<>` **NEG_INFTY**`() :`
  returns the point interval $-\infty = [-\texttt{INFTY}, -M]$.
* `static interval<>` **POS_INFTY**`()`
  returns the point interval $+\infty = [M, +\texttt{INFTY}]$.

- `static interval<>` **ZERO**`()` :
  returns the point interval $0 = [0.0, 0.0]$

- `static interval<>` **ONE**`()` :
  returns the point interval $1 = [1.0, 1.0]$

- `static interval<>` **PI**`()` :
  returns an enclosure of π.

- `bool` **isPoint**`() const`:
  returns `true`,iff `T` is a point interval.

- `static bool` **isExtended**`() const`:
  returns `true`,iff the library has been compiled in the extended mode.

- `bool` **hasUlpAcc**`(unsigned int const & n) const`:
  returns `true`, iff the distance of the bounds $\overline{t} - \underline{t} \leq n$ `ulp`, i.e. the interval contains at most $n+1$ machine representable numbers.

- `N` **mid**`() const`:
  returns an approximation of the midpoint of `T`, that is contained in `T`

  In the extended mode the following cases are distinguished:

$$\texttt{T.mid()} = \begin{cases} \texttt{NaN} & \text{for } \texttt{T == } \emptyset \\ 0.0 & \text{for } \texttt{T == } \mathbb{R}^* \\ +\texttt{INFTY} & \text{for } \texttt{T == [}a, +\texttt{INFTY]} \\ -\texttt{INFTY} & \text{for } \texttt{T == [}-\texttt{INFTY}, a\texttt{]} \end{cases}$$

- `N` **diam**`() const`:
  returns the diameter or width of the interval (upwardly rounded). The method is also available under the alias `width`. In the extended mode the following cases are distinguished:

$$\texttt{T.diam()} = \begin{cases} \texttt{NaN} & \text{if } \texttt{T == } \emptyset \\ +\texttt{INFTY} & \text{if } \texttt{T.isInfinite()} \end{cases}$$

- `N` **relDiam**`() const`:
  returns an upper bound for the relative diameter of `T`:

  `T.relDiam() == T.diam()` if `T.mig()` is less than the smallest positive normalized floating-point number,

  `T.relDiam() == T.diam()/T.mig()` otherwise.

  In the extended mode the following cases are distinguished:

$$\texttt{T.relDiam()} = \begin{cases} \texttt{NaN} & \text{if } \texttt{T == } \emptyset \\ +\texttt{INFTY} & \text{if } \texttt{T.isInfinite()} \end{cases}$$

- `N` **rad**`() const`:
  returns the radius of `T` (upwardly rounded) In the extended mode the following cases are considered:

$$\texttt{T.rad()} = \begin{cases} \texttt{NaN} & \text{if } \texttt{T == } \emptyset \\ +\texttt{INFTY} & \text{if } \texttt{T.isInfinite()} \end{cases}$$

- `N` **mig**`() const`:
  returns the mignitude, i.e.

$$\texttt{T.mig() == min}\{\texttt{abs(t) t}\in \texttt{T}\}$$

  In the extended mode the following cases are considered:

$$\texttt{T.mig()} = \texttt{NaN} \quad \text{if } \texttt{T == } \emptyset$$

- `N` **mag**`() const`:
  returns the magnitude, the absolute value of `T`. also

$$\texttt{T.mag() == max(}\{\texttt{abs(t) t}\in \texttt{T}\}\texttt{)}$$

  In the extended mode the following cases are considered:

$$\texttt{T.mag()} = \begin{cases} \texttt{NaN} & \text{if } \texttt{T == } \emptyset \\ +\texttt{INFTY} & \text{if } \texttt{T.isInfinite()} \end{cases}$$

- `interval<>` **abs**`() const`:
  returns the interval of all absolute values (moduli) of `T`:

```
T.abs() = [ T.mig(), T.mag()]
```

  In the extended mode the following cases are considered:

$$\texttt{T.abs()} = \begin{cases} \emptyset & \text{for } \texttt{T == } \emptyset \\ [\texttt{T.mig()}, +\texttt{INFTY}] & \text{if } \texttt{T.isInfinite()} \text{ and one bound is finite} \\ [M, +\texttt{INFTY}] & \text{if both bounds are infinite} \end{cases}$$

4.6 Set Theoretic Methods

- `interval<>` **imin**`(interval<> const & X)`:
  returns an enclosure of the interval of all minima of `T` and `X`, i.e.

$$\texttt{T.imin(X) == } \{ \texttt{ z: z == min(a,b): a } \in \texttt{ T, b } \in \texttt{ X } \}$$

$$\texttt{T.imin()} = \emptyset \qquad \text{für } \texttt{T == } \emptyset \text{ or } \texttt{X == } \emptyset$$

- `interval<>` **imax**`(interval<> const & X)`:
  returns an enclosure of the interval of all minima of `T` and `X`, i.e.

$$\texttt{T.imax(X) == \{ z: z == max(a,b): a} \in \texttt{T, b} \in \texttt{X \}}$$

In the extended mode return

$$\texttt{T.imax()} = \emptyset \qquad \text{für } \texttt{T == } \emptyset \text{ or } \texttt{X == } \emptyset$$

- `N` **dist**`(interval<> const & X)`:
  returns an upper bound of the Hausdorff-distance of `T` and `X`, i.e.

  `T.dist(X) == max { abs(T.inf()-X.inf()), abs(T.sup()-X.sup()) }`

  In the extended mode return

  $$\texttt{T.dist(X)} = \texttt{NaN} \qquad \text{für } \texttt{T == } \emptyset \text{ or } \texttt{X == } \emptyset$$

- `interval<>` **blow**`(N const & eps) const`:
  return the ε-inflation:

  `T.blow(eps) == (1+eps)·T - eps·T`

- `interval<>` **intersect**`(interval<> const & X) const`:
  returns the intersection of the intervals `T` and `X`. If `T` and `X` are disjoint return $\emptyset$ in the extended mode and an error in the normal mode.

- `interval<>` **hull**`(interval<> const & X) const`:
  the interval hull

  In the extended mode return

  $$\texttt{T.hull()} = \emptyset \qquad \text{if } \texttt{T == X == } \emptyset$$

  This function is also available under the `intervall_hull()` alias.

- `interval<>` **hull**`(N const & X) const`:
  the interval hull.

  In the extended mode return

  $$\texttt{T.hull()} = \emptyset \qquad \text{if } \texttt{T == } \emptyset \text{ and } \texttt{X == NaN}$$

  This function is also available under the `intervall_hull()` alias.

- `bool` **disjoint**`(interval<> const & X) const`:
  returns `true`, iff `T` and `X` are disjoint, i.e. `T.intersect(X) ==` $\emptyset$.

- `bool` **contains**`(N x) const`:
  returns `true`, iff `x` $\in$ `T`

- `bool` **interior**`(interval<> const & X) const`:
  returns `true`, iff `T` is contained in the interior of `X`.

  In the extended mode return `true`, if `T ==` $\emptyset$

- `bool` **proper_subset**`(interval<> const & X) const`:
  returns `true`, iff `T` is a proper subset of `X`.

- `bool` **subset**`(interval<> const & X) const`:
  returns `true`, iff `T` is a subset of `X`.

- `bool` **proper_superset**`(interval<> const & X) const`:
  returns `true`, iff `T` is a proper superset of `X`.

- `bool` **superset**`(interval<> const & X) const`:
  returns `true`, iff `T` is a superset of `X`.

4.7 Interval Relational Methods

4.7.1 Set Relations

- `bool` **seq**`(interval<> const & X) const`:
  returns `true`, iff `T` and `X` are equal sets.

- `bool` **sne**`(interval<> const & X) const`:
  returns `true`, iff `T` and `X` are not equal sets.

- `bool` **sge**`(interval<> const & X) const`:
  returns `true`, iff the $\geq$ relation holds for the bounds

  $$\texttt{T.sge(X) == T.inf()} \geq \texttt{X.inf() \&\& T.sup()} \geq \texttt{X.sup()}$$

  In the extended mode return `true`, if `T ==` $\emptyset$ and `X ==` $\emptyset$.

- `bool` **sgt**`(interval<> const & X) const`:
  returns `true`, iff the $>$ relation holds for the bounds

  $$\texttt{T.sgt(X) == T.inf()} > \texttt{X.inf() \&\& T.sup()} > \texttt{X.sup()}$$

  In the extended mode return `false`, if `T ==` $\emptyset$ and `X ==` $\emptyset$.

- `bool` **sle**`(interval<> const & X) const`:
  returns `true`, iff the $\leq$ relation holds for the bounds

  $$\texttt{T.sle(X) == T.inf()} \leq \texttt{X.inf() \&\& T.sup()} \leq \texttt{X.sup()}$$

  In the extended mode return `true`, if `T ==` $\emptyset$ and `X ==` $\emptyset$.

- `bool` **slt**`(interval<> const & X) const`:
  returns `true`, iff the $<$ relation holds for the bounds

  $$\texttt{T.slt(X) == T.inf()} < \texttt{X.inf() \&\& T.sup()} < \texttt{X.sup()}$$

  In the extended mode return `false`, if `T ==` $\emptyset$ and `X ==` $\emptyset$.

4.7.2 Certainly Relations

- `bool` **ceq**`(interval<> const & X) const`:
  returns `true`, iff the $=$ relation holds for all individual points from `T` and `X`, i.e.

  $$\forall t \in \mathtt{T}, \forall x \in \mathtt{X} : t = x$$

  That implies that `T` and `X` are point intervals.

  In the extended mode return `false`, if `T ==` $\emptyset$ or `X ==` $\emptyset$.

- `bool` **cne**`(interval<> const & X) const`:
  returns `true`, iff the $\neq$ relation holds for all individual points from `T` and `X`, i.e.

  $$\forall t \in \mathtt{T}, \forall x \in \mathtt{X} : t \neq x$$

  That implies that `T` and `X` are disjoint.

  In the extended mode return `true`, if `T ==` $\emptyset$ or `X ==` $\emptyset$.

- `bool` **cge**`(interval<> const & X) const`:
  returns `true`, iff the $\geq$ relation holds for all individual points from `T` and `X`, i.e.

  $$\forall t \in \mathtt{T}, \forall x \in \mathtt{X} : t \geq x$$

  In the extended mode return `false`, if `T ==` $\emptyset$ or `X ==` $\emptyset$.

- `bool` **cgt**`(interval<> const & X) const`:
  returns `true`, iff the $>$ relation holds for all individual points from `T` and `X`, i.e.

  $$\forall t \in \mathtt{T}, \forall x \in \mathtt{X} : t > x$$

  That implies that `T` and `X` are disjoint.

  In the extended mode return `false`, if `T ==` $\emptyset$ or `X ==` $\emptyset$.

- `bool` **cle**`(interval<> const & X) const`:
  returns `true`, iff the $\leq$ relation holds for all individual points from `T` and `X`, i.e.

  $$\forall t \in \mathtt{T}, \forall x \in \mathtt{X} : t \leq x$$

  In the extended mode return `false`, if `T ==` $\emptyset$ or `X ==` $\emptyset$.

- `bool` **clt**`(interval<> const & X) const`:
  returns `true`, iff the $<$ relation holds for all individual points from `T` and `X`, i.e.

  $$\forall t \in \mathtt{T}, \forall x \in \mathtt{X} : t < x$$

  That implies that `T` and `X` are disjoint.

  In the extended mode return `false`, if `T ==` $\emptyset$ or `X ==` $\emptyset$.

4.7.3 Possibly Relations

- `bool` **peq**`(interval<> const & X) const`:
  returns `true`, iff the $=$ relation holds for any points from `T` and `X`, i.e.

  $$\exists t \in \mathtt{T}, \exists x \in \mathtt{X} : t = x$$

  In the extended mode return `false`, if `T ==` $\emptyset$ or `X ==` $\emptyset$.

- `bool` **pne**`(interval<> const & X) const`:
  returns `true`, iff the $\neq$ relation holds for any points from `T` and `X`, i.e.

  $$\exists t \in \mathtt{T}, \exists x \in \mathtt{X} : t \neq x$$

  In the extended mode return `true`, if `T ==` $\emptyset$ or `X ==` $\emptyset$.

- `bool` **pge**`(interval<> const & X) const`:
  returns `true`, iff the $\geq$ relation holds for any points from `T` and `X`, i.e.

  $$\exists t \in \mathtt{T}, \exists x \in \mathtt{X} : t \geq x$$

  In the extended mode return `false`, if `T ==` $\emptyset$ or `X ==` $\emptyset$.

- `bool` **pgt**`(interval<> const & X) const`:
  returns `true`, iff the $>$ relation holds for any points from `T` and `X`, i.e.

  $$\exists t \in \mathtt{T}, \exists x \in \mathtt{X} : t > x$$

  In the extended mode return `false`, if `T ==` $\emptyset$ or `X ==` $\emptyset$.

- `bool` **ple**`(interval<> const & X) const`:
  returns `true`, iff the $\leq$ relation holds for any points from `T` and `X`, i.e.

  $$\exists t \in \mathtt{T}, \exists x \in \mathtt{X} : t \leq x$$

  In the extended mode return `false`, if `T ==` $\emptyset$ or `X ==` $\emptyset$.

- `bool` **plt**`(interval<> const & X) const`:
  returns `true`, iff the $<$ relation holds for any points from `T` and `X`, i.e.

  $$\exists t \in \mathtt{T}, \exists x \in \mathtt{X} : t < x$$

  In the extended mode return `false`, if `T ==` $\emptyset$ or `X ==` $\emptyset$.

4.8 Input and Output

- `std::ostream & ` **bitImage**`(std::ostream & out) const`:
  output the bitwise internal representation.
- `std::ostream & ` **hexImage**`(std::ostream & out) const`:
  output a hexadecimal representation.
- `static interval<N,K> ` **readBitImage**`(std::istream & in) throw(filib::interval_io_exception)`: read a bit representation of an interval from `in` and return it. If the input cannot be parsed as a bit image, an exception of type `filib::interval_io_exception` is thrown.
- `static interval<N,K> ` **readHexImage**`(std::istream & in) throw(filib::interval_io_exception)`: read a hex representation of an interval from `in` and return it. If the input cannot be parsed as a hex image, an exception of type `filib::interval_io_exception` is thrown.
- `static int const & ` **precision**`()`:
  returns the output precision that is used by the output operator `<<`. (see 5.6)
- `static int ` **precision**`(int const & p)`:
  set the output precision to `p`. The default value is 3.

Chapter 5

Global Functions

Let R denote the interval $[\underline{r}, \overline{r}]$. All operations which have been specified as updating methods of the class `interval<>` are available as global functions as well. This interface to the operations is not only more familiar and convenient for the user, but also more efficient.

5.1 Arithmetic Operators

- `interval<> ` **operator+**`(interval<> const & A, interval<> const & B)`:
  returns the interval `R` with

$$\underline{r} := \underline{a} + \underline{b},\ \overline{r} := \overline{a} + \overline{b}$$

- `interval<> & ` **operator+**`(interval<> const & A, N const & b)`:
  returns the interval `R` with

$$\underline{r} := \underline{a} + b,\ \overline{r} := \overline{a} + b$$

- `interval<> ` **operator+**`(N const & A, interval<> const & B)`:
  returns the interval `R` with

$$\underline{r} := a + \underline{b},\ \overline{r} := a + \overline{b}$$

- `interval<> ` **operator-**`(interval<> const & A, interval<> const & B)`:
  returns the interval `R` with

$$\underline{r} := \underline{a} - \overline{b},\ \overline{r} := \overline{a} - \underline{b}$$

- `interval<> & ` **operator-**`(interval<> const & A, N const & b)`:
  returns the interval `R` with

$$\underline{r} := \underline{a} - b,\ \overline{r} := \overline{a} - b$$

- `interval<> ` **operator-**`(N const & A, interval<> const & B)`:
  returns the interval `R` with

$$\underline{r} := a - \overline{b}, \overline{r} := a - \underline{b}$$

- `interval<>` **cancel**`(interval<> const & A, interval<> const & B)`:
  returns the interval `R` with

$$\underline{r} := \underline{a} - \underline{b}, \overline{r} := \overline{a} - \overline{b}$$

  if $\underline{a} - \underline{b} \leq \overline{a} - \overline{b}$. Otherwise an error is thrown in the normal mode, or the empty interval is returned in the extended mode.

- `interval<>` **operator***`(interval<> const & A, interval<> const & B)`:
  returns the interval `R` with

$$\underline{r} := \min\{\underline{a} * \underline{b}, \overline{a} * \underline{b}, \underline{a} * \overline{b}, \overline{a} * \overline{b}\}, \overline{r} := \max\{\underline{a} * \underline{b}, \overline{a} * \underline{b}, \underline{a} * \overline{b}, \overline{a} * \overline{b}\}$$

- `interval<> &` **operator***`(interval<> const & A, N const & b)`:
  returns the interval `R` with

$$\underline{r} := \min\{\underline{a} * b, \overline{a} * b\}, \overline{r} := \max\{\underline{a} * b, \overline{a} * b\}$$

- `interval<>` **operator***`(N const & A, interval<> const & B)`:
  returns the interval `R` with

$$\underline{r} := \min\{a * \underline{b}, a * \underline{b}, a * \overline{b}, a * \overline{b}\}, \overline{r} := \max\{a * \underline{b}, a * \underline{b}, a * \overline{b}, a * \overline{b}\}$$

- `interval<>` **operator/**`(interval<> const & A, interval<> const & B)`:
  returns the interval `R` with

$$\underline{r} := \min\{\underline{a}/\underline{b}, \overline{a}/\underline{b}, \underline{a}/\overline{b}, \overline{a}/\overline{b}\}, \overline{r} := \max\{\underline{a}/\underline{b}, \overline{a}/\underline{b}, \underline{a}/\overline{b}, \overline{a}/\overline{b}\}$$

  $0 \in a$ produces an error in the normal mode.

- `interval<> &` **operator/**`(interval<> const & A, N const & b)`:
  returns the interval `R` with

$$\underline{r} := \min\{\underline{a}/b, \overline{a}/b\}, \overline{r} := \max\{\underline{a}/b, \overline{a}/b\}$$

  $b = 0$ produces an error in the normal mode.

- `interval<>` **operator/**`(N const & A, interval<> const & B)`:
  returns the interval `R` with

$$\underline{r} := \min\{a/\underline{b}, a/\underline{b}, a/\overline{b}, a/\overline{b}\}, \overline{r} := \max\{a/\underline{b}, a/\underline{b}, a/\overline{b}, a/\overline{b}\}$$

  $0 \in a$ produces an error in the normal mode.

5.2 Access and Information

- `N const &` **inf**`(interval<> const & A)`:
  equivalent to `A.inf()`.
- `N const &` **sup**`(interval<> const & A)`:
  equivalent to `A.sup()`.
- `N` **inf_by_value**`(interval<> const & A)`:
  return a copy of `A.inf()`.
- `N` **sup_by_value**`(interval<> const & A)`:
  return a copy of `A.sup()`.
- `bool` **isPoint**`(interval<> const & A)`:
  equivalent to `A.isPoint()`.
- `bool` **hasUlpAcc**`(interval<> const & A)`:
  equivalent to `A.hasUlpAcc()`.
- `N` **mid**`(interval<> const & A)`:
  equivalent to `A.mid()`.
- `N` **diam**`(interval<> const & A)`:
  equivalent to `A.diam()`. An alias named `width` is available.
- `N` **relDiam**`(interval<> const & A)`:
  equivalent to `A.relDiam()`.
- `N` **rad**`(interval<> const & A)`:
  equivalent to `A.rad()`.
- `N` **mig**`(interval<> const & A)`:
  equivalent to `A.mig()`.
- `N` **mag**`(interval<> const & A)`:
  equivalent to `A.mag()`.
- `interval<>` **abs**`(interval<> const & A)`:
  equivalent to `A.abs()`.

5.3 Set Theoretic Functions

- `interval<>` **imin**`(interval<> const & A, interval<> const & B)`:
  equivalent to `A.imin(B)`.
- `interval<>` **imax**`(interval<> const & A, interval<> const & B)`:
  equivalent to `A.imax(B)`.
- `N` **dist**`(interval<> const & A, interval<> const & B)`:
  equivalent to `A.dist(B)`.

- `interval<>` **blow**`(interval<> const & A, N const & eps)`:
  equivalent to `A.blow(eps)`.
- `interval<>` **intersect**`(interval<> const & A, interval<> const & B)`:
  equivalent to `A.intersect(B)`.
- `interval<>` **hull**`(interval<> const & A, interval<> const & B)`:
  equivalent to `A.hull(B)`, also available as `intervall_hull()`.
- `interval<>` **hull**`( N const & b, interval<> const & A)`:
  equivalent to `A.hull(b)`, also available as `intervall_hull()`.
- `interval<>` **hull**`( N const & a, N const & b)`:
  returns the interval hull of the 2 numbers `a` and `b`, also available as `intervall_hull()`.

  In the extended mode returns $\emptyset$, if `x == y == NaN`
- `bool` **disjoint**`(interval<> const & A, interval<> const & B)`:
  equivalent to `A.disjoint(B)`.
- `bool` **in**`(N & a, interval<> const & B)`:
  equivalent to `B.contains(a)`.
- `bool` **interior**`(interval<> const & A, interval<> const & B)`:
  equivalent to `A.interior(B)`.
- `bool` **proper_subset**`(interval<> const & A, interval<> const & B)`:
  equivalent to `A.proper_subset(B)`.
- `bool` **subset**`(interval<> const & A, interval<> const & B)`:
  equivalent to `A.subset(B)`.
- `bool` **operator**<=`(interval<> const & A, interval<> const & B)`:
  equivalent to `A.subset(B)`.
- `bool` **proper_superset**`(interval<> const & A, interval<> const & B)`:
  equivalent to `A.proper_superset(B)`.
- `bool` **superset**`(interval<> const & A, interval<> const & B)`:
  equivalent to `A.superset(B)`.
- `bool` **operator**>= `(interval<> const & A, interval<> const & B)`:
  equivalent to `A.superset(B)`.

5.4 Interval Relational Functions

5.4.1 Set Relational Functions

- `bool` **seq**`(interval<> const & A, interval<> const & B)`:
  equivalent to `A.seq(B)`.
- `bool` **operator**==`(interval<> const & A, interval<> const & B)`:
  equivalent to `A.seq(B)`.

- `bool` **sne**`(interval<> const & A, interval<> const & B)`:
  equivalent to `A.sne(B)`.
- `bool` **operator**! =`(interval<> const & A, interval<> const & B)`:
  equivalent to `A.sne(B)`.
- `bool` **sge**`(interval<> const & A, interval<> const & B)`:
  equivalent to `A.sge(B)`.
- `bool` **sgt**`(interval<> const & A, interval<> const & B)`:
  equivalent to `A.sgt(B)`.
- `bool` **sle**`(interval<> const & A, interval<> const & B)`:
  equivalent to `A.sle(B)`.
- `bool` **slt**`(interval<> const & A, interval<> const & B)`:
  equivalent to `A.slt(B)`.

5.4.2 Certainly Relational Functions

- `bool` **ceq**`(interval<> const & A, interval<> const & B)`:
  equivalent to `A.ceq(B)`.
- `bool` **cne**`(interval<> const & A, interval<> const & B)`:
  equivalent to `A.cne(B)`.
- `bool` **cge**`(interval<> const & A, interval<> const & B)`:
  equivalent to `A.cge(B)`.
- `bool` **cgt**`(interval<> const & A, interval<> const & B)`:
  equivalent to `A.cgt(B)`.
- `bool` **cle**`(interval<> const & A, interval<> const & B)`:
  equivalent to `A.cle(B)`.
- `bool` **clt**`(interval<> const & A, interval<> const & B)`:
  equivalent to `A.clt(B)`.

5.4.3 Possibly Relational Functions

- `bool` **peq**`(interval<> const & A, interval<> const & B)`:
  equivalent to `A.peq(B)`.
- `bool` **pne**`(interval<> const & A, interval<> const & B)`:
  equivalent to `A.pne(B)`.
- `bool` **pge**`(interval<> const & A, interval<> const & B)`:
  equivalent to `A.pge(B)`.
- `bool` **pgt**`(interval<> const & A, interval<> const & B)`:
  equivalent to `A.pgt(B)`.

- `bool` **ple**`(interval<> const & A, interval<> const & B)`:
  equivalent to `A.ple(B)`.

- `bool` **plt**`(interval<> const & A, interval<> const & B)`:
  equivalent to `A.plt(B)`.

5.5 Elementary Functions

The elementary functions return enclosures of the ranges. In general, they are not 1-ulp accurate, but reasonably fast. These functions are only implemented for intervals based on the `double` type.

- `interval<>` **acos**`(interval<> const & A)`:
  inverse cosine
- `interval<>` **acosh**`(interval<> const & A)`:
  inverse hyperbolic cosine
- `interval<>` **acot**`(interval<> const & A)`:
  inverse cotangent
- `interval<>` **acoth**`(interval<> const & A)`:
  inverse hyperbolic cotangent
- `interval<>` **asin**`(interval<> const & A)`:
  inverse sine
- `interval<>` **asinh**`(interval<> const & A)`:
  inverse hyperbolic sine
- `interval<>` **atan**`(interval<> const & A)`:
  inverse tangent
- `interval<>` **atanh**`(interval<> const & A)`:
  inverse hyperbolic tangent
- `interval<>` **cos**`(interval<> const & A)`:
  cosine
- `interval<>` **cosh**`(interval<> const & A)`:
  hyperbolic cosine
- `interval<>` **cot**`(interval<> const & A)`:
  cotangent
- `interval<>` **coth**`(interval<> const & A)`:
  hyperbolic cotangent
- `interval<>` **exp**`(interval<> const & A)`:
  exponential e^A
- `interval<>` **exp10**`(interval<> const & A)`:
  exponential to base 10. 10^A

- `interval<>` **exp2**`(interval<> const & A)`:
  exponential to base 2. 2^A
- `interval<>` **expm1**`(interval<> const & A)`:
  $e^A - 1$
- `interval<>` **log**`(interval<> const & A)`:
  logarithm to base e
- `interval<>` **log10**`(interval<> const & A)`:
  logarithm to base 10
- `interval<>` **log1p**`(interval<> const & A)`:
  $\log(A + 1)$
- `interval<>` **log2**`(interval<> const & A)`:
  logarithm to base 2
- `interval<>` **power**`(interval<> const & A, int const & p)`:
  power to an integer A^p
- `interval<>` **pow**`(interval<> const & A, interval<> const & B)`:
  general power function $\{a^b : a \in A, b \in B\}$.
- `interval<>` **sin**`(interval<> const & A)`:
  sine
- `interval<>` **sinh**`(interval<> const & A)`:
  hyperbolic sine
- `interval<>` **sqr**`(interval<> const & A)`:
  square
- `interval<>` **sqrt**`(interval<> const & A)`:
  square root
- `interval<>` **tan**`(interval<> const & A)`:
  tangent
- `interval<>` **tanh**`(interval<> const & A)`:
  hyperbolic tangent

5.6 Input and Output

- `std::ostream &` **operator<<**`(std::ostream & out, interval<> const & A)`:
  outputs the interval `A` to the stream `out`. According to the output precision the usual format is `[` $\underline{a}$, $\overline{a}$ `]`. Note that the bounds are NOT rounded directly to the string format, but the standard output method is used instead. We recommend to use the `bitImage` method (see 4.8) for a detailed view. In case of an erroneous interval `[ UNDEFINED ]` is output in the normal mode. In the extended mode there are several special cases:

- [EMPTY] for the empty interval
- [-INFTY] for $[-\infty, -\mathtt{M}]$
- [+INFTY] for $[\mathtt{M}, \infty]$
- [ENTIRE] for $\mathbb{R}^*$

- `std::istream & `**operator**`>>(std::istream & in, interval<> & A) throw(filib::interval_io_exception)` :
  reads the interval `A` from the stream `in`. If the input cannot be parsed as an interval, an exception of type `filib::interval_io_exception` is thrown. Note that the input is converted to the used arithmetic type by using the standard function `strtod()`. That means that there is no care taken for directed rounding in the case that the provided numbers do not have an exact machine representation. We recommend using the method `readBitImage()` if there is need for a perfectly predictable input method.

Chapter 6

Installation

6.1 Compiler Requirements

A compiler conforming to ISO 14882 (ISO C++) is sufficient, but currently not available. We have used GNU C++ Compiler (version 2.95.2) and KAI C++ Compiler. The code also compiles with version 3 of the GNU C++ Compiler.
Furtheron a unix compatible `make` utility is needed (e.g. GNU make or BSD make), GNU Binutils (version 2.9.5 or better) and a BSD compatible `install` program.

6.2 Installation and Usage

6.2.1 Installation

The library is delivered as a gziped tar file. If you unpack it, the source code is put into a subdirectory `interval`. The compilation with the `GCC` or `KCC` is controlled by the included makefiles. A convenient way is to set the appropriate link

```
ln -s makefiles/Makefile.gcc Makefile
```

or

```
ln -s makefiles/Makefile.kcc Makefile
```

For other compilers the makefile has to be adapted.
The command

```
make libs
```

compiles and builds the library.
The target directory, e.g. `/usr/local/filib` is set to the variable `PREFIX` and then the library is installed by the command

```
make install OWN=<user> GRP=<group> PREFIX=/usr/local/filib
```

6.2.2 Usage of the Template Library

The source file has to contain the directive

```
#include <interval/interval.hpp>
```

in order to declare the identifiers of the library. It has to work in the namespace `filib`. When compiling source files, it is necessary to inform the compiler about the path of the `include` files. For `GCC` or `KCC` the compiler option `-I`*PREFIX*`/include` is given.

```
c++ -c -I/usr/local/filib/include source.cpp -o source.o
```

Linking is controlled by another option telling the location of the library during linking and at runtime. An example for the KAI compiler:

```
KCC source.o -o source --no_abstract_float -L/usr/local/filib/lib
         -lprim -Wl,-rpath=/usr/local/filib/lib
```

`--no_abstract_float` is used for calling the KAI compiler for correctness reasons.

6.2.3 Usage of the Macro Library

The source file has to contain the directive

```
#include <Interval.h>
```

in order to declare the identifiers of the macro library. When compiling source files, it is necessary to inform the compiler about the path of the `include` files. For `GCC` or `KCC` the compiler option `-I`*PREFIX*`/include` is given.

```
c++ -c -I/usr/local/filib/include source.cpp -o source.o
```

Linking is controlled by another option telling the location of the library during linking. An example for the KAI compiler:

```
KCC source.o -o source -L/usr/local/filib/lib -lfi -lieee -lm
-Wl,-rpath=/usr/local/filib/lib --no_exceptions --no_abstract_float
```

As above `--no_abstract_float` is used for the KAI compiler. In addition the macro library is by default compiled with `--no_exceptions` for performance reasons.

6.3 Organization of Subdirectories

We finally describe the structure of the directories in form of a directory tree:


```
interval
 |
 |--- doc (this manual)
 |   |
 |   |--- tex (PS/PDF documentation files)
 |
 |--- examples (a few tiny examples)
 |
 |--- fp_traits (traits classes for fp types)
 |
 |--- ieee (code for handling IEEE 754 types)
 |
 |--- interval (interval arithmetics)
 |   |
 |   |--- stdfun (standard functions)
 |      |
 |      |--- interval (interval versions)
 |      |
 |      |--- point (point versions)
 |
 |--- macro (macro based library)
 |   |
 |   |-- config (compile time switches)
 |   |
 |   |-- doc (old documentation)
 |   |
 |   |-- example (example code)
 |   |
 |   |-- include (c++ header files)
 |   |
 |   |-- src (static/non-inline interval code)
 |
 |-- licenses (license (GPL))
 |
 |-- makefiles (various makefiles for GCC/KAI/etc.)
 |
 |-- readme (some information on installations)
 |
 |-- rounding_control (low-level machine rounding control)
```

The installation copies files from the directories `interval`, `fp_traits`, `ieee`, `rounding_control` and `macro/include` to the installation `include` directory. The installation `lib` directory is after installating filib++ populated by libraries built on the target machine. (We currently do not support cross-compilation.)

Bibliography

[1] The C++ Programming Language, ISO 14882, 1998

Acknowledgements, Origins

Many people have contributed to the design and the construction of filib++. The first version of the library has been published in Karlsruhe by Werner Hofschuster and Walter Krämer. It was a library with emphasis on the fast evaluation of the elementary functions. The normal mode was implemented.

[2] Hofschuster, W.; Krämer, W.: *FI_LIB, eine schnelle und portable Funktionsbibliothek für reelle Argumente und reelle Intervalle im IEEE-double-Format*, Preprint Nr. 98/7 des Instituts für Wissenschaftliches Rechnen und Mathematische Modellbildung, Universität Karlsruhe, July 1998. `ftp://ftp.iam..uni-karlsruhe.de` `/pub/iwrmm/preprints/prep987.ps`

[3] Hofschuster, W.; Krämer, W.: Quellen der `fi_lib` `ftp://iamk4515.mathematik.uni-karlsruhe.de/pub/iwrmm/software/fi_lib.tgz` Now in Wuppertal ! `http://www.math.uni-wuppertal.de/org/WRST/software.html`

The code for the calculation of the elementary functions has been taken from this library with changes only for the extended mode.

The implementation of the extended mode has largely been influenced by Bill Walster and his team at Sun Microsystems.

[4] Chiriaev,D., Walster, G.W.:*Interval Arithmetic Specification*, www.mscs.mu.edu/ globsol/walster-papers.html

[5] Walster, G.W. et al.: *Extended Real Intervals and the Topological Closure of Extended Real Numbers*, Sun Microsystems, Feb 2000

[6] Walster, G.W. et al.: *The "Simple" Closed Interval System*, Sun Microsystems, Feb 2000

They started with a Fortran extension and now also provide a C++ library.

[7] C++ Interval Arithmetic Programming Reference, Sun Microsystems, Oct 2000 `http://www.sun.com/forte/cplusplus/interval/index.html`

Input and output routines of filib++ have been adapted from the libI77 of the runtime system of the Gnu Fortran Compiler // (see http://www.eecs.lehigh.edu/ mschulte/compiler)

We further thank Jens Maurer for fruitful discussions on the design of a template library conforming to the C++ standard.

Parts of this manual have been published in earlier reports.

[8] Michael Lerch, Jürgen Wolff von Gudenberg, `fi_lib++` : Specification, Implementation and Test of a Library for Extended Interval Arithmetic, RNC4 proceedings, pp. 111-123, April 2000

[9] Jürgen Wolff von Gudenberg, Interval Arithmetic and Multimedia Architectures, Techn. Report 265, Informatik, Universität Würzburg, Oct 2000