filib++, a Fast Interval Library Supporting Containment Computations

MICHAEL LERCH, GERMAN TISCHLER, JÜRGEN WOLFF VON GUDENBERG
University of Würzburg, Germany
WERNER HOFSCHUSTER, WALTER KRÄMER
University of Wuppertal, Germany

`filib++` is an extension of the interval library `filib` originally developed at the University of Karlsruhe. The most important aim of `filib` is the fast computation of guaranteed bounds for interval versions of a comprehensive set of elementary functions. `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 NaNs known from the IEEE floating-point standard 754, to intervals. In this mode, the so-called containment sets are computed to enclose the topological closure of a range of a function over an interval. Second, our new state of the art design uses templates and traits classes to obtain an efficient, easily extendable and portable C++ library.

Categories and Subject Descriptors: G.4 [**Mathematical Software**]: Portable and efficient library design; G.1.0 [**Numerical Analysis**]: General-Interval arithmetic

General Terms: Interval Computations, Containment Sets, Validated Numerics, Exception Free Computations, C++ Class Library, Templates, Traits Class

Additional Key Words and Phrases: **`filib++, interval arithmetic, validation, guaranteed numerical results, containment computations`**

1. INTRODUCTION AND MOTIVATION

Let us first give some striking examples of why validated numerics (based on interval computations) may be needed. The following figure shows some graphical results produced with Maple's `implicitplot` command. The first two plots in Fig. 1. show Maple results when plotting the curve defined by $(x^2 + y^2)^3 - 27x^2y^2 = 0$, the second two plots in Fig. 2. show $(x^2 + y^2 - 6x)^2 - x^2 - y^2 = 0$, and the plots in Fig. 3. show $x^4 + y^4 - 8x^2 - 10y^2 + 16 = 0$. In the plots in the right column, the parameter `numpoints` of the `implicitplot` command is set explicitly, whereas in the left column Maple's default setting is used.

The difficulty in plotting these implicitly defined simple curves is not the lack of precision in the computations. The erroneous results come from the interpolation process. At least, near intersection points or near singularities, this process is difficult to control automatically.

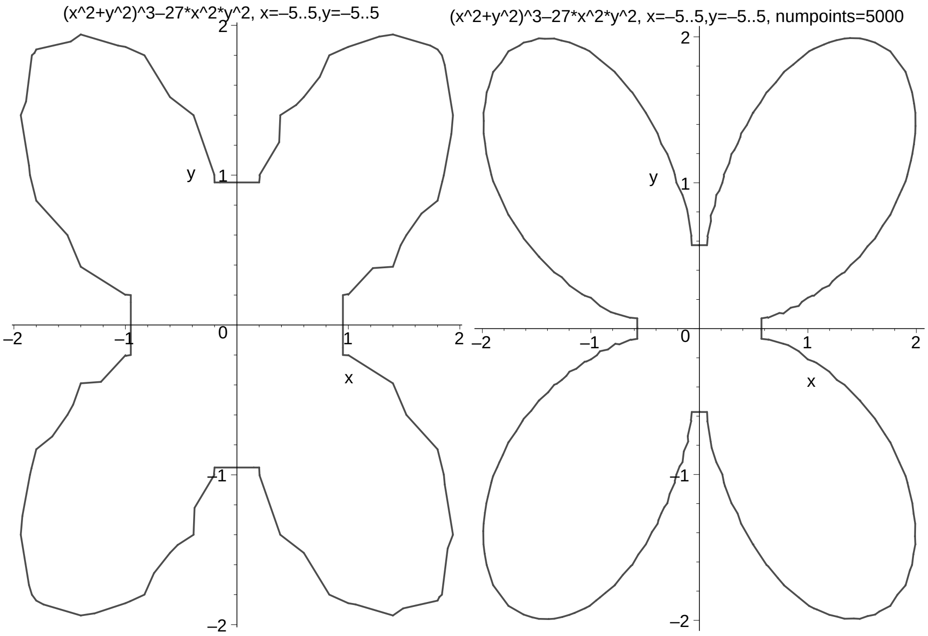

Fig. 1.

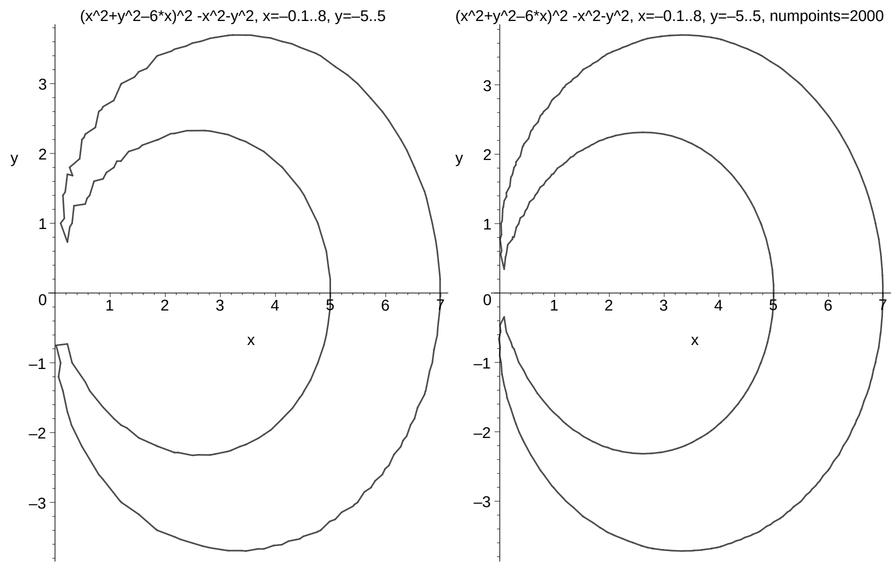

Fig. 2.

Using interval tools, as for example provided by our new library `filib++`, the results in Figures 4. to 6. are obtained without any intervention by the user. These results are verified to be correct.

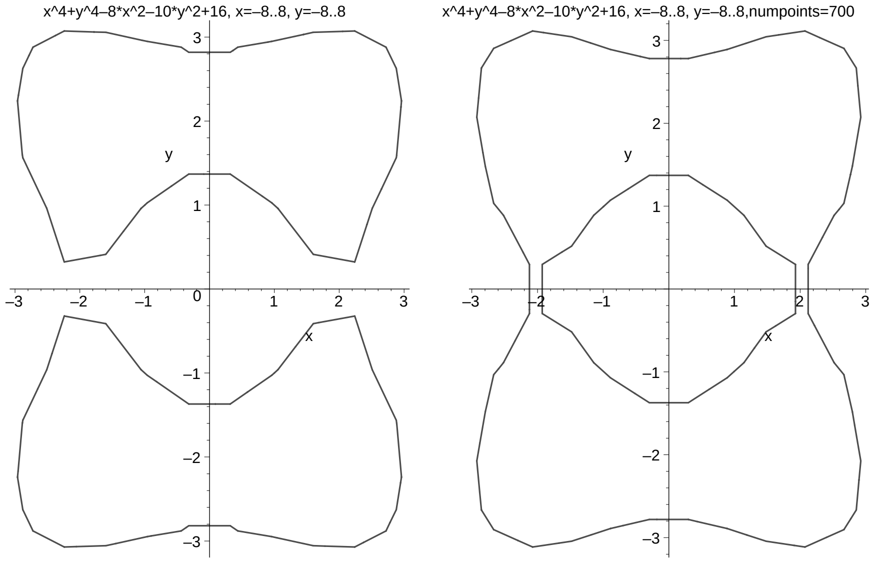

Fig. 3.

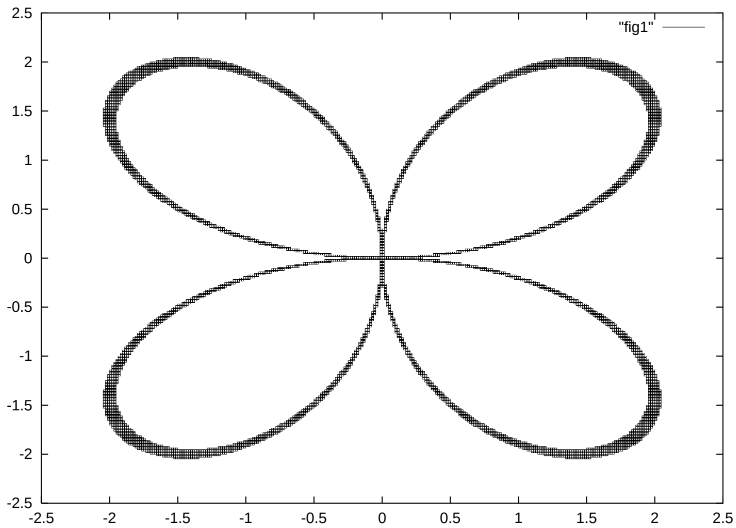

Fig. 4.

filib++ also provides containment computations to handle infinite intervals. Thus, we can, for example, compute enclosures of all zeros of a one-dimensional function over the complete real line (see below).

For the following, we assume that the reader is familiar with the basic ideas of interval arithmetic (for a good introduction and references see [Hammer et al.

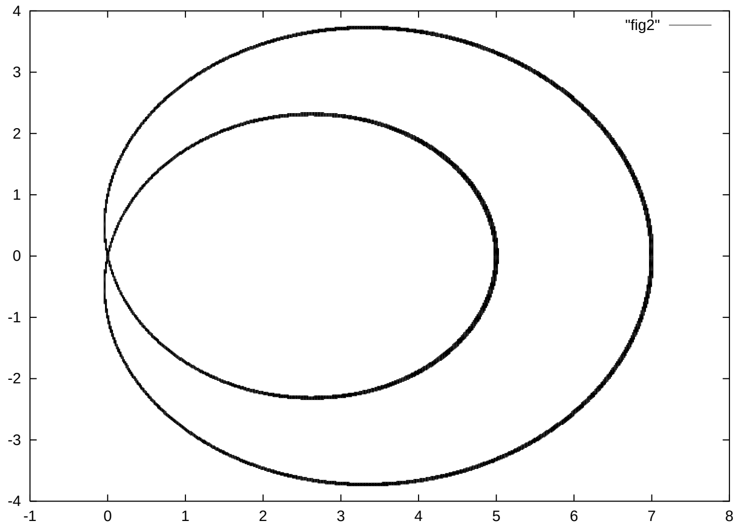

Fig. 5.

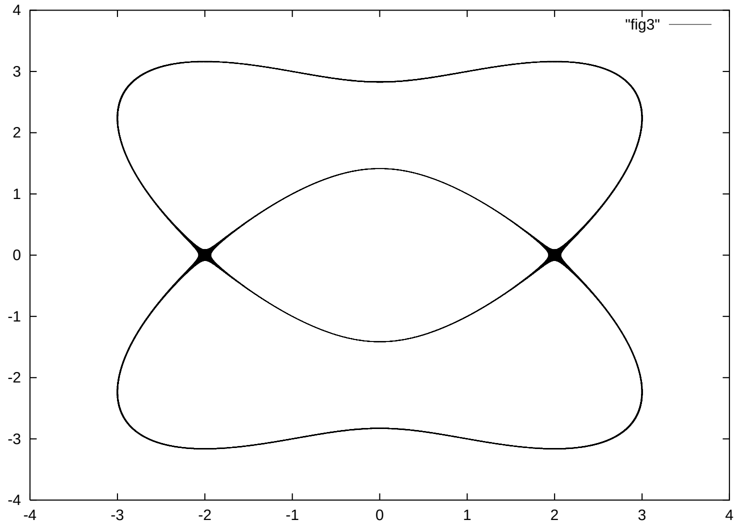

Fig. 6.

1995]). We use bold face for continuous intervals, represented by two real bounds. That is,

$$\mathbf{x} = [\underline{x}, \overline{x}] = \{x \in I\!R \mid \underline{x} \leq x \leq \overline{x}\}.$$

$I\!I\!R$ denotes the space of all finite closed intervals, and $f(\mathbf{x}) = \{f(x) \mid x \in \mathbf{x}\}$ denotes the range of values of the function $f : D_f \subseteq I\!R \to I\!R$ over the interval

$\mathbf{x} \subseteq D_f$. In this paper, we restrict our consideration to the one-dimensional case. Extensions to more dimensions are more or less obvious.

For those who are interested in recent developments in the fields of interval mathematics and applications of it, we refer to [Krämer and Wolff von Gudenberg 2001; Kulisch et al. 2001].

2. INTERVAL EVALUATION

We consider the enclosure of the range of a function, one of the main topics of interval arithmetic.

DEFINITION 1. *The **interval evaluation** $\mathbf{f} : I\!I\!R \to I\!I\!R$ of f is defined as the function that is obtained by replacing every occurrence of the variable x by the interval variable $\mathbf{x}$, by replacing every operator by its interval arithmetic counterpart and by replacing every elementary function by its range. Note that this definition holds, if all operations are executable without exceptions.*

The following theorem is known as the fundamental theorem of interval arithmetic [Alefeld and Herzberger 1983].

THEOREM 2.1. *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, for example, is forbidden. 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 for f. For the function $f_1(x) = 1/(x \cdot x + 2)$ the call $\mathbf{f}_1([-2, 2])$ is not defined, because $[-2, 2] \cdot [-2, 2] = [-4, 4]$, whereas $f_2(x) = 1/(x^2 + 2)$ yields $\mathbf{f}_2([-2, 2]) = [1/6, 1/2]$. Of course, for real arguments x, $f_1(x) = f_2(x)$.

The result of an elementary function call over an interval is defined as the interval of all function values. Such function calls are not defined, if the argument interval contains a point outside the domain of the corresponding function. For elementary functions $f \in \{\sin, \cos, \exp \ldots\}$, it holds that $\mathbf{f}(\mathbf{x}) := f(\mathbf{x}) = \{f(x) | x \in \mathbf{x} \subseteq D_f\}$.That is, the result of an interval evaluation of such a function is by definition equal to the range of the function over the argument interval.

3. CONTAINMENT EVALUATION

To overcome the difficulties with partially defined functions throwing exceptions, we introduce a second mode, the “extended” mode. Here, no exceptions are raised, but the domains of interval functions and ranges of interval results are consistently extended. In the extended mode, intervalarithmetic operations and mathematical functions form a closed mathematical system. This means that valid results are produced for any possible operator-operand combination, including division by zero and other undetermined forms involving zero and infinities.

Following G. W. Walster in [Chiriaev and Walster 1999; Walster et al. 2000b] we define the containment set:

DEFINITION 2. *Let $f : D_f \subseteq I\!R \to I\!R$. Then the containment set $f^* : \wp I\!R^* \mapsto$*

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

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

Table I. Extended addition and 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 II. Extended multiplication

$\wp I\!\!R^*$ defined by

$$f^*(\mathbf{x}) := \{f(x) | x \in \mathbf{x} \cap D_f\} \cup \{\lim_{D_f \ni x \to x^*} f(x) | x^* \in \mathbf{x}\} \subseteq \mathbb{R}^* \tag{1}$$

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

Hence, the containment set of a function encloses the range of the function, as well as all limits and accumulation points.

Our goal is now to define an analogue to the interval evaluation, which encloses the containment set, and is easy to compute.

Let $\mathbb{IR}^*$ denote the set of all extended intervals with endpoints in $\mathbb{R}^*$, supplemented by the empty set (empty interval).

DEFINITION 3. *The **containment evaluation** $\mathbf{f}^* : \mathbb{IR}^* \to \mathbb{IR}^*$ of f is defined as the function that is obtained by replacing every occurrence 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 [Walster et al. 2000]

THEOREM 3.1. *The containment evaluation is always defined[1], and it holds*

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

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. [Chiriaev and Walster 1999]. 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, which is $\mathbb{R}^*$. Note that negative values are also possible, since 0 can be approached from both sides.

[1] in contrast to the interval evaluation

/	$-\infty$	$y<0$	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 III. Extended division

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

Table IV. Extended interval division

Tables I to IV show the containment sets for the basic arithmetic operations. From these tables, the definition of extended interval arithmetic can easily be deduced. For addition, subtraction, and multiplication infinite intervals can be returned, if a corresponding operation is encountered. Some typical computations are $[2,\infty]+[3,\infty]=[5,\infty]$, $[2,\infty]-[3,\infty]=\mathbb{R}^*$, $[2,\infty]*[-3,3]=\mathbb{R}^*$. Division is more subtle. Table IV shows the cases where the denominator contains 0.

For the elementary functions, Table V shows the extended domains and extended ranges. The containment set for an elementary function is computed by directly applying the definition of a containment set. 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. Their realizations on a computer never throw an exception. Some typical examples are $\log[-1,1]=[-\infty,0]$, $\sqrt{[-1,1]}=[0,1]$, $\log[-2,-1]=\emptyset$, $\coth[-1,1]=I\!\!R^*$.

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) 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 $\mathrm{power}(\mathbf{x},k)\subseteq\mathrm{pow}(\mathbf{x},[k,k])$ does not hold for negative $\mathbf{x}$.

It has been shown in [Walster et al. 2000b; 2000a] that, using extended operations, the containment evaluation can be computed without exceptions.

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	$[-\infty,-1] \cup [1,\infty]$	$\mathbb{R}^*$	$\mathrm{acoth}[-1,-1] = [-\infty]$ $\mathrm{acoth}[1,1] = [\infty]$

Table V. Extended domains and ranges for the elementary functions

4. FILIB++ BUILT AS A TEMPLATE LIBRARY

4.1 Traits and Templates

To describe the up-to-date design of filib++, we assume that the reader has some knowledge of C++ templates.

In analogy to the `basic_string` template of the C++ standard, we use a concept called a traits class for our implementation. A traits class is a template class that allows accessing features and operations of its template parameter(s). In the example case of the `basic_string` template, a character traits class is used. This traits class can be invoked for several character types, as the well-known `char` primitive type, but usually also for some kind of a wide character type. It brings functions like assigning characters (an operation on characters) and functions that return some special value of the parameter type like the `eos` symbol (end of string, a special feature value of the parameter type). The methods of the `basic_string` template can work exclusively with the functions of the traits class, without directly using any special properties of the template parameter, and thus, without need for a specialization for template arguments.

In our case, we use a traits class `fp_traits` for the basic number type on which we build interval arithmetic on. It provides all the functions we need for basic interval operations, like directed addition, subtraction, multiplication and division, as well as some functions returning special values, like returning the maximum finite value of the type. It also has functions for testing properties of the number type, like testing for infinity.

Our traits class has two template parameters: the first selects the basic number type, for example `float` or `double`, and the second selects the implementation of the directed rounding, for example hardware based or software based, by computing the predecessor or successor of a number.

We currently have the following rounding methods:

—`native_switched`: operations are based on hardware support. After using directed operations, we switch the rounding mode back to the default (round to nearest).

—`native_directed`: is like `native_switched`, but without switching back to the default mode. This changes the floating point semantics of the rest of the program, since usually the default floating point rounding mode is round to nearest. Because on most processors switching the rounding mode is a very expensive operation, this may speed up the computation, if only interval arithmetic is used or the traits class is called to reset the rounding mode to the default explicitly, when needed.

—`multiplicative`: rounding is implemented by multiplication with the predecessor or successor of 1. This relies on the hardware implementation to produce results that were rounded with the round to nearest rounding scheme.

—`no_rounding`: used for debugging reasons, not useable for applications.

—`native_onesied_switched`: use only rounding in one direction and compute the other side by factoring out -1 and then using negation. On machines where switching the rounding mode is expensive, it may be faster though, as only one switch to a directed rounding mode is used.

—`native_onesided_global`: like `native_onesided_switched` plus no switching back to the default rounding mode after interval operations. As the rounding mode only needs to be set at the beginning of the programs, this mode will bring the fastest computation on machines, where switching the rounding mode is expensive. But it has the problems described in `native_directed` plus it is necessary to also set the rounding mode to downward again, if we want to continue using this mode, after having gone back to the default rounding.

—`pred_succ_rounding`: do rounding by computing the predecessor or successor after performing a computation. The machine may not compute results that are off by more then one ulp (unit in the last place) for this mode to be able to work correctly.

These seven rounding modes are provided, since the degree of conforming to or exploiting the IEEE arithmetic differs from machine to machine and compiler to compiler. Therefore, we suggest to test the performance for any rounding mode on the target machine, see 6.1 for our test results.

The availablity of various rounding modes further enhances the portability of the libray. Whereas some modes need assembler statements to access the directed roundings, the multiplicative rounding always works on IEEE architectures. Specializations of this traits class for `double` and `float` are provided in the library, as listed in Table VI.

The specializations for rounding modes that rely on machine specific rounding control methods inherit these methods from an instantiation of the template class `rounding_control`. That is illustrated in Fig. 7., which shows part of the inheritence hierarchy used in the library (note that this is a simplified diagram).

Our implementation can be easily extended for new number types by producing new specializations of the `fp_traits` template. At the current stage of the library, this is not true for the standard functions, as they are derived from `fi_lib` and have not been converted to using the traits concept, for speed and complexity reasons.

Examples showing how to call the `fp_traits` functions directly can be found in the interval example code shown below.

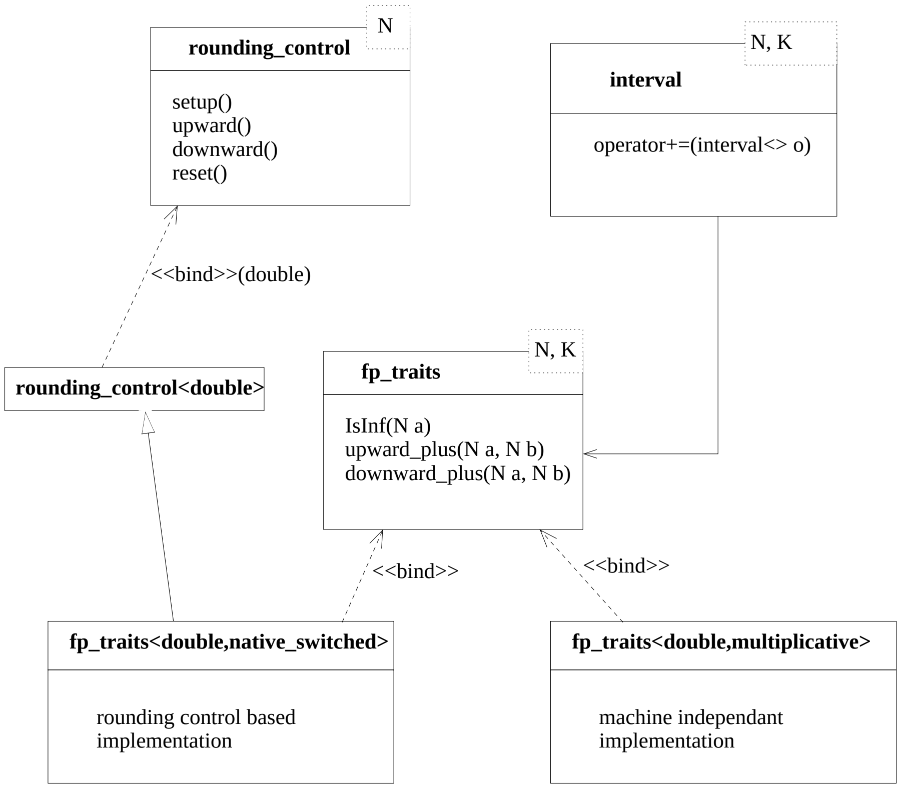

Fig. 7. Part of the Inheritence Hierarchy

The filib++ interval class is realized as a template class. Currently there are two template parameters, the underlying basic (floating-point) number type `N`, and the method, implementing the directed roundings `rounding_control`. Here, the basic number type `N` may be `float` or `double`, and `rounding_control` may be `native_switched`, `native_directed` and `multiplicative` (see Table VI).

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;
}
```

Some examples may help to use the library:

`filib::interval<double> A;`
This is the default instantiation of an interval: `A` is an interval over the floating-point type `double`, the second and third template parameters are set to their default
`filib::native_switched` or `filib::i_mode_normal` implicitly.

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

`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.
Another example can be found in the `examples` directory of the distribution.

Of course the interval class is compatible with the STL (standard template library). This means that we can easily use the data structures provided there for vectors, lists, queues, etc. as they are often needed in applications. As the STL is part of the C++ standard, it is portable and usually very efficient.

4.2 Template Arguments

The `fp_traits<>` class as well as the `interval<>` class are template classes with two or three template arguments. The first argument must be 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 the preceding section. Table VI shows all currently available combinations. In forthcoming filib++ versions the third parameter of `interval<>` will denote the interval evaluation mode. So it will be possible to switch from containment evaluations to interval evaluation mode.

4.3 Utility Functions Coming from the `fp_traits<>` class

The following static member functions are mandatory for all implementations of the `fp_traits<>` class (`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

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

Table VI. Currently existing `fp_traits` implementations

`N const &` **quiet_NaN**`()`: returns a quiet (non-signaling) **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.

`void` **reset**`()`: reset the rounding mode

4.4 The `interval<>` class

Let $\underline{x}$ and $\overline{x}$ denote the infimum and 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. 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".

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

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 $a \leq b$ 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 decreased to its predecessor and the supremum is increased to its successor. 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.

The assignment operator is

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

The following arithmetic methods are provided for updating arithmetic operations. Note that the usual operators are available as global functions (see V).

The special cases of the extended mode are not explicitly mentioned here, see Tables I, II, refdiv 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)`(updating 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.

The following access and information methods are provided. 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:

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

- `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. That is,

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

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:

$$\texttt{T.abs() = [T.mig(), T.mag()]}$$

In the extended mode the following cases are considered:

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

The set theoretic methods are

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

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

In the extended mode, it returns

$$\texttt{T.imin()} = \emptyset \quad \text{for } \texttt{T == } \emptyset \text{ or } \texttt{X == } \emptyset$$

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

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

In the extended mode this function returns

$$\texttt{T.imax()} = \emptyset \quad \text{for } \texttt{T == } \emptyset \text{ or } \texttt{X == } \emptyset.$$

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

$$\texttt{T.dist(X) == max } \{ \texttt{ abs(T.inf()-X.inf()), abs(T.sup()-X.sup()) } \}$$

In the extended mode this function returns

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

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

$$\texttt{T.blow(eps) == (1+eps)}\cdot\texttt{T - eps}\cdot\texttt{T}$$

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

`interval<>` **hull**`(interval<> const & X) const`: the interval hull.
In the extended mode, it returns

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

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

`interval<>` **hull**`(N const & X) const`: the interval hull.
In the extended mode, it returns

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

This function is also available under the `interval_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 $\texttt{x} \in \texttt{T}$

`bool` **interior**`(interval<> const & X) const`: returns `true` iff `T` is contained in the interior of `X`.
In the extended mode it returns `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`.

Three kinds of interval relational methods are provided: set relations, certainly relations, and possibly relations. The first character of the name of such a method is s for set relations, c for certainly relations, and p for possibly relations.

Set Relations

Set relational functions are true if the interval operands satisfy the underlying relation in the **ordinary set theoretic** sense.

`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, it returns `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, it returns `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, it returns `false` if `T ==` $\emptyset$ and `X ==` $\emptyset$.

Certainly Relations

Certainly relational functions are true if the underlying relation (e. g. less than) is true for **every element** of the operand intervals.

`bool` **`ceq`**`(interval<> const & X) const`: returns `true` iff the $=$ relation holds for all individual points from `T` and `X`, i.e. $\forall t \in \texttt{T}, \forall x \in \texttt{X} : t = x$
That implies that `T` and `X` are point intervals.
In the extended mode, it returns `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. $\forall t \in \texttt{T}, \forall x \in \texttt{X} : t \neq x$
That implies that `T` and `X` are disjoint.
In the extended mode, it returns `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. $\forall t \in \texttt{T}, \forall x \in \texttt{X} : t \geq x$
In the extended mode, it returns `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. $\forall t \in \texttt{T}, \forall x \in \texttt{X} : t > x$
That implies that `T` and `X` are disjoint.
In the extended mode, it returns `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. $\forall t \in \texttt{T}, \forall x \in \texttt{X} : t \leq x$
In the extended mode, it returns `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, it returns `false` if `T == ` $\emptyset$ or `X == ` $\emptyset$.

Possibly Relations
Possibly relational functions are true if **any element** of the operand intervals satisfy the underlying relation.

`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 , it returns `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, it returns `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, it returns `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, it returns `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, it returns `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, it returns `false` if `T == ` $\emptyset$ or `X == ` $\emptyset$.

Input and Output routines are

`std::ostream &` **bitImage**`(std::ostream & out) const`: output the bitwise internal representation.

`std::ostream &` **hexImage**`(std::ostream & out) const`: output a hexadecimal representation. This routine is not available for the macro version of `filib++`.

`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. This routine is not available for the macro version of `filib++`.

`static int const &` **precision**`()`: returns the output precision that is used by the output operator `<<`.

`static int` **precision**`(int const & p)`: set the output precision to `p`. The default value is 3.

The 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 (mathematical notation of expressions), but also more efficient.

The class `interval<double>` also provides the elementary mathematical functions sin, cos, acos, acosh, acot, acoth, asin, asinh, atan, atanh, acoth, exp, exp 10, exp 2, expm1, log, log 10, log $1p$, log 2, pow, sinh, sqr, sqrt, tan, tanh in both the normal and the extended interval mode. The typical interface for such a function with one argument is

`interval<> sin(interval<> const & A)`:
sine function with resulting set $\{\sin(a) : a \in A\}$.
For functions with two arguments we typically have
`interval<> pow(interval<> const & A, interval<> const & B)`:
general power function with resulting set $\{a^b : a \in A, b \in B\}$.

5. APPLICATION

The following program code demonstrates the use of the `filib++` library in connection with the standard template library (STL) of C++ to get verified graphics of implicitly defined curves as shown in the first chapter of this paper.

```
//Verified computation of level curves using filib++ and routines from the STL
#include <fstream>
#include <list>
#include <interval/interval.hpp> //filib++
//...
using namespace filib;
//Simplify instantiation of intervals
typedef filib::interval<double> I;
//...
//Data type for two dimensional boxes
typedef pair<I,I> rectangle;

void levelCurve(
                 ofstream& out,     //output file
                 I(*f)(rectangle),  //function
                 rectangle x,
                 I level,
                 double epsilon
               )
{
   list<rectangle> toDo;
   toDo.push_back(x);
   while( !toDo.empty() )
   {
      rectangle box= toDo.front();
      toDo.pop_front();
      I fRange= f(box);
      if (!disjoint(level, fRange)) //box may contain points of level curve
      {
         if ( diam(box) < epsilon )
            plotBox(out, box);      //write data to plot box using gnuplot
```

```
         else
         {
            if( diam(box.first) > diam(box.second) )
            {
               pair<I,I> p = bisect(box.first);
               toDo.push_back(make_pair(p.first,  box.second));
               toDo.push_back(make_pair(p.second, box.second));
            }
            else
            {
               pair<I,I> p = bisect(box.second);
               toDo.push_back(make_pair(box.first, p.first));
               toDo.push_back(make_pair(box.first, p.second));
            }
         }
      }
   }
}

I f(rectangle box) //two dimensional (interval) function
{
    I x(box.first), y(box.second);
    I h= sqr(x) + sqr(y);      //x^2+y^2
    return sqr(h-I(6)*x) - h;  //(x^2+y^2-6x)^2 - (x^2+y^2)
}
```

With the data and function calls

```
double epsilon=0.01;
I level(0,0);
rectangle x(make_pair(I(-8, 8), I(-8, 8)));
levelCurve(out, f, x, level, epsilon);    //fine
levelCurve(out, f, x, level, 60*epsilon); //coarse
```

we get the verified graphic in Fig. 8. The boxes are a coarse coverage of the curve. Note, that the variable `level` is of type `interval`. This allows plotting level curves simultaneously for a given range of levels. The figure 9. has been produced using `level(-1,20)` and `epsilon`=0.05.

Containment computations also allow the treatment of functions over infinite domains, functions with singularities, or functions that are only defined partially. For example, a coverage of the graph of $f(x) := x + \arctan(\log(\sin(10x)/(x+1)))$ may be computed using the program from above with a function $g(x, y) := f(x) - y$ without any modification. The result is shown in Figure 10 (left part).

The corresponding (unreliable) Maple plot using `numpoints=10000` is also shown (right figure). Here, e. g. the part of the graph of f close to $x = -1$ is missing. Note that the natural domain of definition D_f is the union of infinitely many disjoint open intervals. The interval plot routine never loses points of the graph.

Using the bisection algorithm described in [Lerch et al. 2001] *verified enclosures* for *all zeros* of f over the *entire real line* (stored in a list named `zeros`) may be computed using

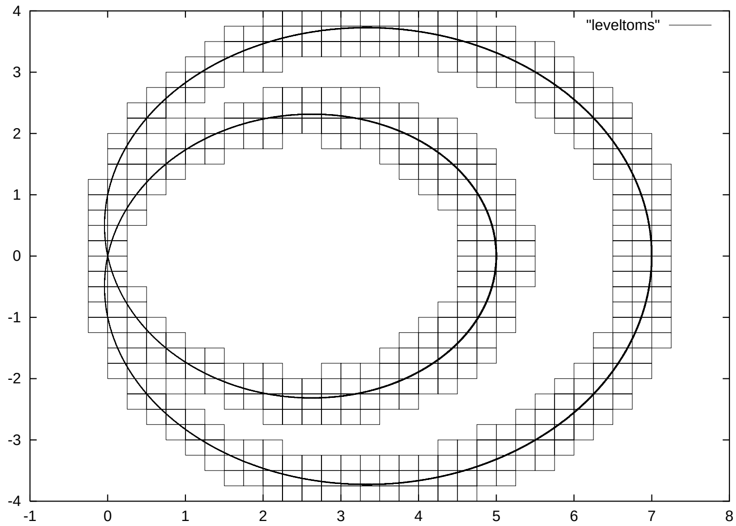

Fig. 8.

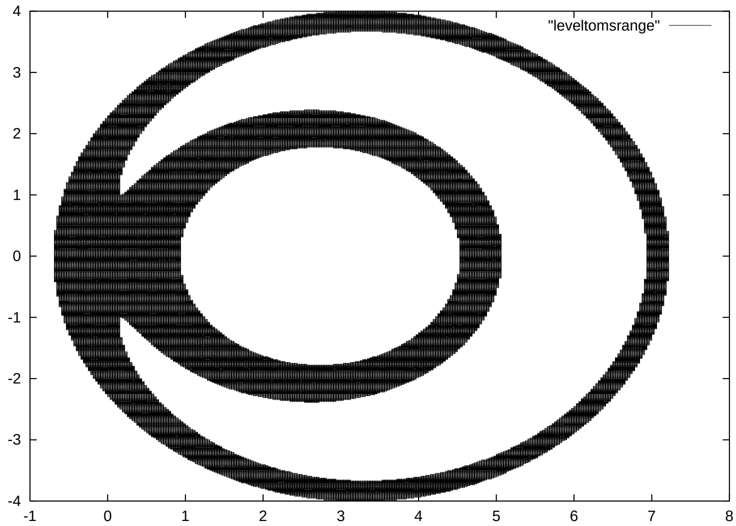

Fig. 9.

```
list<interval> zeros= findAllZeros(f<interval>, interval::ENTIRE(), epsilon);
```

For `epsilon=0.0001` we get the enclosures

```
There may be zeros in the interval(s):
  1  [-1.00006103515624978, -0.99999999999999988]
  2  [-0.95947265624999989, -0.95941162109374988]
  3  [-0.52752685546874989, -0.52740478515624988]
```

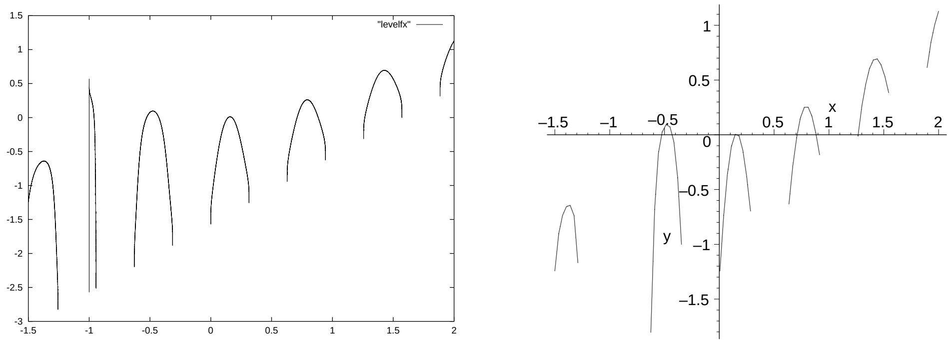

Fig. 10. $f(x) := x + \arctan(\log(\sin(10x)/(x+1)))$: verified (left)/Maple (right)

```
 4  [-0.42694091796874995, -0.42681884765624995]
 5  [0.142700195312499972,  0.14282226562499998]
 6  [0.174560546874999972,  0.17468261718749998]
 7  [0.709289550781249889,  0.70941162109374989]
 8  [0.883361816406249889,  0.88342285156249989]
 9  [1.26605224609374978,   1.26611328124999978]
10  [1.57073974609374978,   1.57080078124999978]
```

Note that $f(-1)$ is not defined (division by zero) and $\lim_{x \to \frac{\pi}{2}} f(x) = 0$.

6. PERFORMANCE

6.1 Internal Comparison

Instantiations of the `interval<>` class with different rounding strategies or evaluation modes differently perform. The performance heavily depends on the harware architecture and the compiler version.

We tested the arithmetic operations in a loop, the numbers (`double`) were randomly generated into vectors of different lengths. The processor was a 2GHz Pentium 4 running under Linux. We used the gcc 3.2.1 compiler with optimization level O3. The following two tables show the performance in MIOPs (million interval operations per second) for the normal and the extended mode.

Rounding mode	+	-	*	/
native	22.4	22.2	11.4	8.8
native-switch	3.9	3.9	3.5	3.0
native-onesided	20.9	21.2	13.9	8.2
native-onesided-switch	19.2	19.3	8.9	6.3
no-switch	24.7	24.6	16.4	9.2
multiplicative	8.8	8.9	6.1	6.2
pred-succ	7.5	7.8	1.5	1.7

normal interval mode

Rounding mode	+	-	*	/
native	18.7	18.9	4.5	8.5
native-switch	3.6	3.6	2.5	2.8
native-onesided-switch	11.9	11.9	7.9	6.3
native-onesided-switch	10.5	10.6	4.5	5.0
no-switch	22.0	22.1	10.6	9.1
multiplicative	8.5	8.5	4.6	5.6
pred-succ	6.8	7.0	0.5	0.9

extended interval mode

6.2 External Comparison

The same test scenario has been applied to compare filib++ with Profil/BIAS [Knüppel 1994]. Note that the latter could only be compiled using the gnu 2.95.2 compiler.

Library	+	-	*	/
filib++	22.4267	22.1937	11.3699	8.85632
profil	11.6108	11.274	7.63891	9.76969

We further compared our portable open source library filib++ on a sun solaris workstation with the native interval support in the nonportable commercial sun forte environment [Sun Microsystems 2001] (so the comparison is not realy fair for filib++). Nevertheless, the results are interesting.

Note, that the results depend significantly on the optimization level chosen for the compilation. In all cases we use optimization level O3.

The expression we use for the time measurement of basic interval operations is $(x * (x + y) - (x * y - z) - x)/(z * y)$. This expression is evaluated within a loop (1000000 repetitions). filib++ on a sun solaris compiled with the gnu 2.95.2 compiler takes 3130 msec. Using the native interval operations of the sun forte compiler on the same machine takes 3370 msec.

For the time measurement of elementary interval function calls we compute the expression $\log(\exp(\arctan(\sin(y) * \cos(x))))$. Again 1000000 repetitions are performed. Here, filib++ on a sun solaris compiled with the gnu 2.95.2 compiler takes 8470 msec whereas the native interval functions of the sun forte compiler take 5240 msec.

To facilitate tests on other machines, the source code of the program for the time measurement is part of the filib++ distribution.

7. CONCLUSION

`filib++` is an efficient, powerful, portable, publicly available C++ interval library supporting containment computations. Its design using template classes in combination with traits classes is flexible and up to date. The library can be used with any C++ compiler conforming to the C++ standard from 1998.

In this paper we payed only little attention to the use of the library. Only some small applications have been discussed to show the quality of mathematical results that may be achieved using `filib++`. The numerical results are guaranteed to be correct in a mathematical sense. The library is designed to be of maximum value in the field of *Validated Computing*. Containment computations allow writing robust code with only little programming effort. Some additional applications (for example, the verified computation of all solutions of a nonlinear equation) may be found in [Lerch et al. 2001]. The complete source codes are available in the `examples` directory of the `filib++` installation.

In a forthcoming version the interval class `interval<>` of filib++ will have an additional third template parameter to enable the user to do computations using containment sets as well as comutations using ordinary interval operations. This may be helpful for the verified computation of roots and fixed-points of systems of equations using Brouwer's fixed-point theorem [Hammer et al. 1995].

There are a number of public domain and commercial interval libraries [Hofschuster et al. 2001; Knüppel 1994; Kearfott et al. 1994; Kearfott 1996; Rump 1998; Sun Microsystems 2001]. Beside Sun Forte C++ with interval support [Sun Microsystems 2001], which is not portable and only commercially available, the new library `filib++` is the only one providing extended interval arithmetic based on containment sets. `filib++` sources are available from
`http://www.math.uni-wuppertal.de/wrswt/software/filib.html`.

REFERENCES

ALEFELD, G. AND HERZBERGER, J. 1983. *Introduction to Interval Computations.* Academic Press, New York.

CHIRIAEV, D. AND WALSTER, G. 1999. Interval arithmetic specification. `www.mscs.mu.edu/~globsol/walster-papers.html`.

HAMMER, R., HOCKS, M., KULISCH, U., AND RATZ, D. 1995. *C++ Toolbox for Verified Computing – Basic Numerical Problems.* Springer-Verlag, Berlin.

HOFSCHUSTER, W. AND KRÄMER, W. 1998a. FI_LIB, eine schnelle und portable Funktionsbibliothek für reelle Argumente und reelle Intervalle im IEEE-double-Format. Preprint 98/7, Institut für Wissenschaftliches Rechnen und Mathematische Modellbildung, Universität Karlsruhe. `http://www.math.uni-wuppertal.de/wrswt/preprints/prep987.ps`.

HOFSCHUSTER, W. AND KRÄMER, W. 1998b. `fi_lib` sources. `http://www.math.uni-wuppertal.de/WRSWT/software.html`.

HOFSCHUSTER, W., KRÄMER, W., WEDNER, S., AND WIETHOFF, A. 2001. C-xsc 2.0 – a c++ class library for extended scientific computing. Preprint 01/1, Wissenschaftliches Rechnen/Software Technologie, Universität Wuppertal. `http://www.math.uni-wuppertal.de/wrswt/preprints/prep_01_1.ps`.

KEARFOTT, R. B. 1996. Algorithm 763: Interval arithmetic, a fortran 90 module for an interval data type. *ACM Trans. Math. Software 22,* 4, 385–392.

KEARFOTT, R. B., DAWANDE, M., DU, K., AND HU, C. 1994. Algorithm 737: Intlib, a portable fortran-77 elementary function library. *ACM Trans. Math. Software 20,* 4 (December), 447–459.

KEARFOTT, R. B. AND NOVOA, M. 1990. Algorithm 681: Intbis, a portable interval newton/bisection package. *ACM Trans. Math. Software 16,* 2 (June), 152–157.

KLATTE, R., KULISCH, U., LAWO, C., RAUCH, M., AND WIETHOFF, A. 1993. *C-XSC – A C++ Class Library for Scientific Computing.* Springer-Verlag, Berlin.

KNÜPPEL, O. 1994. Profil/bias – a fast interval library. *Computing 53*, 277–287.

KRÄMER, W. 2002. Advanced software tools for validated computing. In *Proceedings of the Thirty First Spring Conference of the Union of Bulgarian Mathematicians.* Union of Bulgarian Mathematicians, Borovets, 344–355.

KRÄMER, W. AND WOLFF VON GUDENBERG, J. 2001. *Scientific Computing, Validated Numerics, Interval Methods.* Kluwer Academic/Plenum Publishers, New York, Boston.

KULISCH, U., LOHNER, R., AND FACIUS, A. 2001. *Perspectives on Enclosure Methods.* Springer-Verlag, Wien.

LERCH, M., TISCHLER, G., WOLFF VON GUDENBERG, J., HOFSCHUSTER, W., AND KRÄMER, W. 2001. The interval library filib++ 2.0 - design, features and sample programs. Preprint BUGHW-WRSWT 2001/4, Universität Wuppertal.

LERCH, M. AND WOLFF VON GUDENBERG, J. 2000. `fi_lib++` : Specification, implementation and test of a library for extended interval arithmetic. In *RNC4 proceedings.* 111–123.

RUMP, S., M. 1998. Intlab - interval laboratory. In *Developments in Reliable Computing*, T. Csendes, Ed. KluwerAcademic Publisher, Dordrecht, 77–104.

RUMP, S., M. 1999. Fast and parallel interval arithmetic. *Bit 39,* 3 (Sept.), 534–554.

STROUSTRUP, B. 2000. *The C++ Programming Language*, Special Edition ed. Addison-Wesley, Reading, Mass.

Sun Microsystems 2001. *C++ Interval Arithmetic Programming Reference (Forte Developer 6 update 2).* Sun Microsystems. `http://www.sun.com/forte/cplusplus/interval/index.html`.

WALSTER, G. ET AL. 2000a. The "simple" closed interval system. Tech. rep., Sun Microsystems.

WALSTER, G. W. ET AL. 2000b. Extended real intervals and the topological closure of extended real numbers. Tech. rep., Sun Microsystems.

WALSTER, W. G., HANSEN, E. R., AND D., P. J. 2000. Extended real intervals and the topological closure of extended real relations. Technical report, Sun Microsystems.

WOLFF VON GUDENBERG, J. 2000. Interval arithmetic and multimedia architectures. Techn. Report 265, Informatik, Universität Würzburg.

...