FILIB++, A Fast Interval Library Supporting Containment Computations

MICHAEL LERCH, GERMAN TISCHLER, and JÜRGEN WOLFF VON GUDENBERG
University of Würzburg, Germany
and
WERNER HOFSCHUSTER and 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 design uses templates and traits classes to obtain an efficient, easily extendable, and portable C++ library.

Categories and Subject Descriptors: G.4 [**Mathematical Software**]; G.1.0 [**Numerical Analysis**]: General—*Interval arithmetic*

General Terms: Algorithms, Reliability, Standardization

Additional Key Words and Phrases: `filib++, interval arithmetic, validation, guaranteed numerical results, containment computations`, interval computations, containment sets, validated numerics, exception free computations, C++ class library, templates, traits class

1. INTRODUCTION AND MOTIVATION

Let us first give some illustrative examples of why validated numerics (based on interval computations) may be needed. Then, in the following two sections, we introduce the two modes of the library using interval evaluation or containment evaluation of a function, respectively. Section 4 describes the design of `filib++`

Authors' addresses: M. Lerch, G. Tischler, and J. Wolff von Gudenberg, University of Würzburg, Germany; W. Hofschuster and W. Krämer, University of Wuppertal, Germany; email: kraemer@math.uni-wuppertal.de.

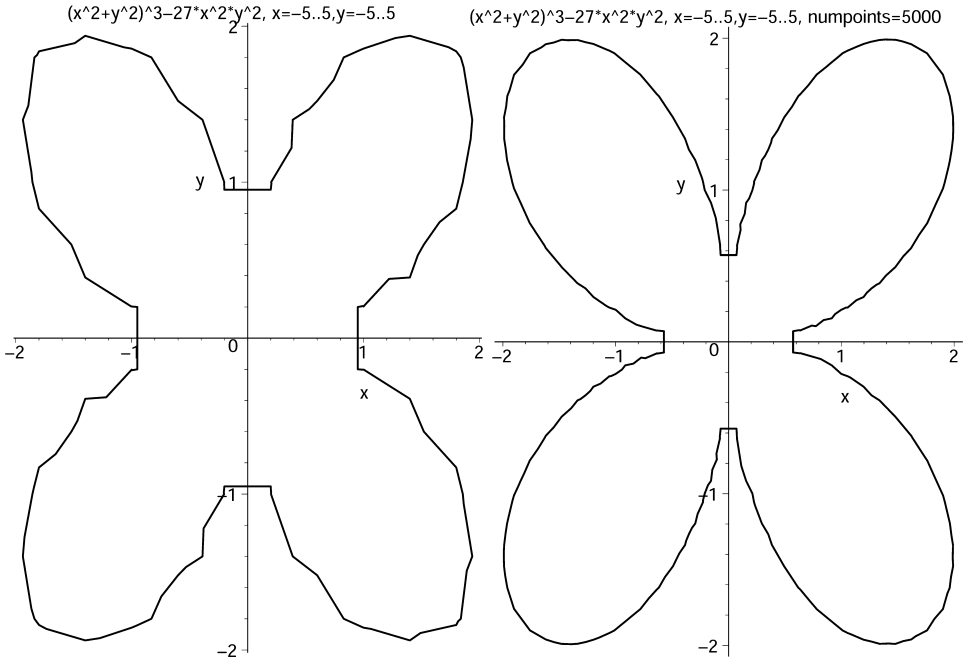

Fig. 1.

as a template library and provides its interface. In Section 5, an application is given, while Section 6 gives some performance figures and compares `filib++` with other interval libraries.

The following figures show some graphical results produced with Maple's `implicitplot` command. The first two plots in Figure 1 show Maple results when plotting the curve defined by $(x^2 + y^2)^3 - 27x^2y^2 = 0$, the second two plots in Figure 2 show $(x^2 + y^2 - 6x)^2 - x^2 - y^2 = 0$, and the plots in Figure 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.

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.

`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.

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. [1995]). We use bold face for continuous intervals, represented by two

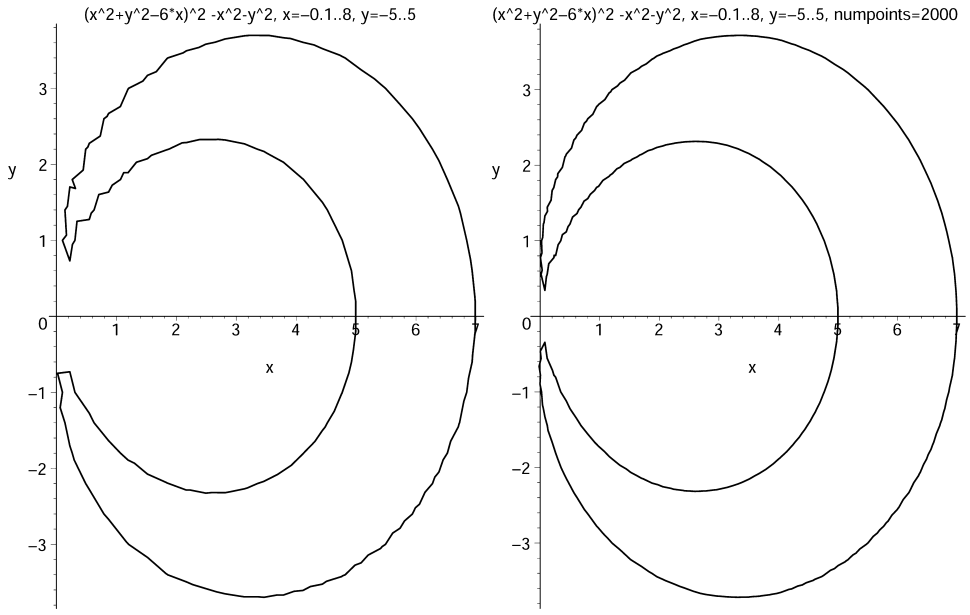
(x^2+y^2−6*x)^2 -x^2-y^2, x=−0.1..8, y=−5..5
(x^2+y^2−6*x)^2 -x^2-y^2, x=−0.1..8, y=−5..5, numpoints=2000
x
y

Fig. 2.

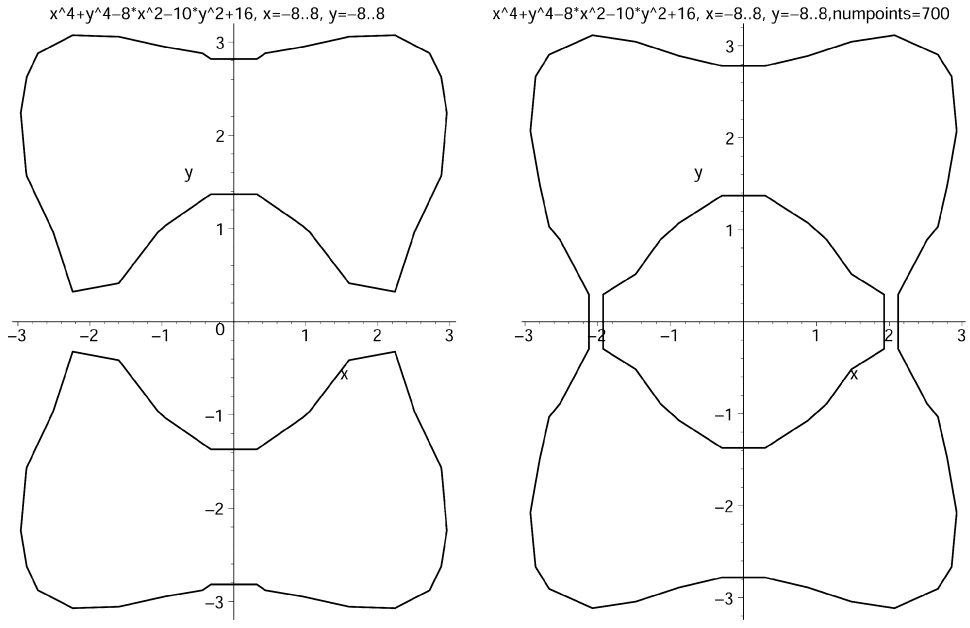
x^4+y^4−8*x^2−10*y^2+16, x=−8..8, y=−8..8
x^4+y^4−8*x^2−10*y^2+16, x=−8..8, y=−8..8,numpoints=700
x
y

Fig. 3.

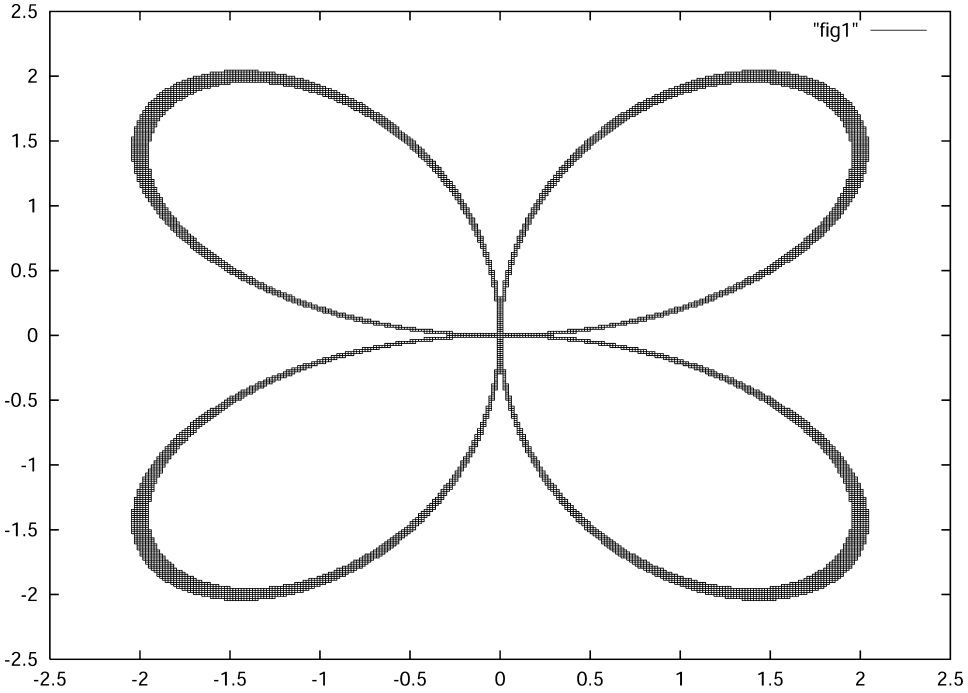
"fig1"
2.5
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2.5
-2.5
-2
-1.5
-1
-0.5
0
0.5
1
1.5
2
2.5

Fig. 4.

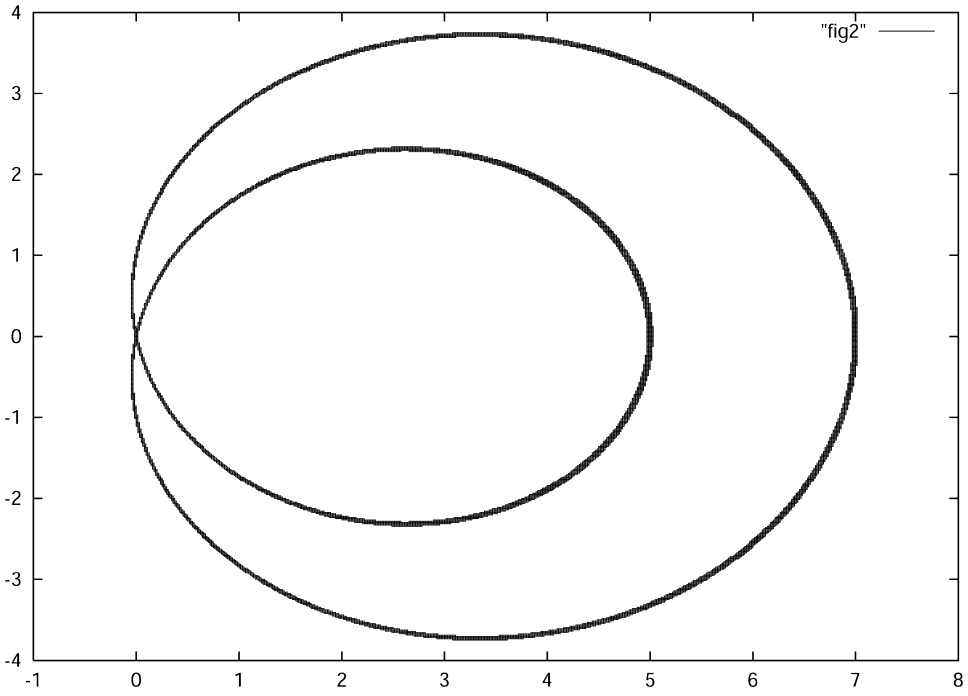
"fig2"
4
3
2
1
0
-1
-2
-3
-4
-1
0
1
2
3
4
5
6
7
8

Fig. 5.

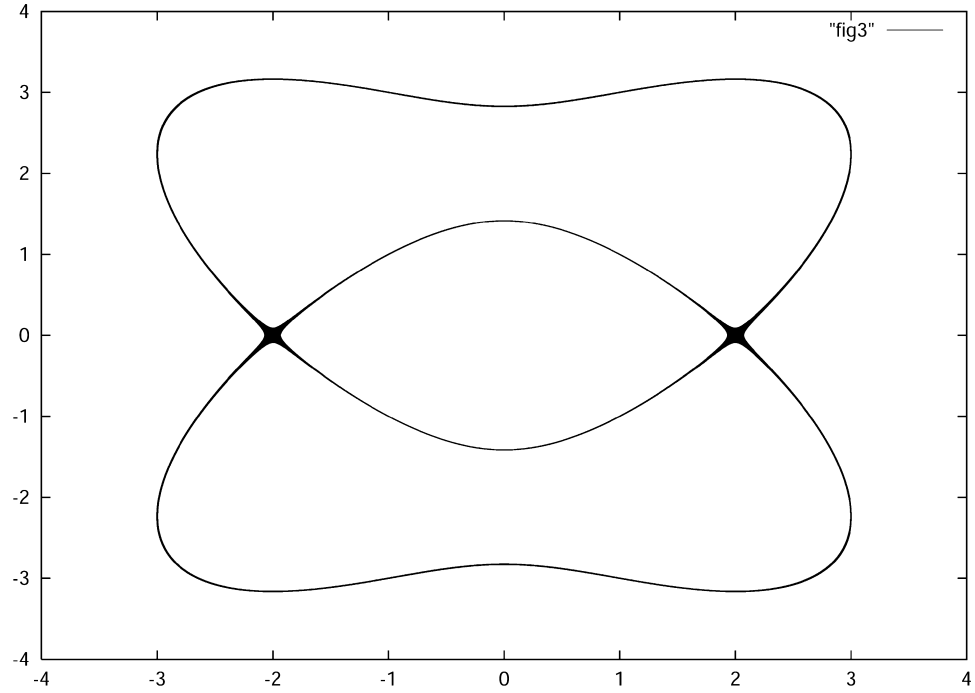

Fig. 6.

real bounds. That is,

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

$\mathbb{IR}$ 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 $f : D_f \subseteq \mathbb{R} \to \mathbb{R}$ over the interval $\mathbf{x} \subseteq D_f$. In this article, we restrict our consideration to the one-dimensional case. Extensions to more dimensions are straightforward.

For those who are interested in recent developments in the fields of interval mathematics and applications of it, refer to Krämer and Wolff von Gudenberg [2001] and 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 $\mathbf{f} : \mathbb{IR} \to \mathbb{IR}$ of f is defined as the function that is obtained by replacing every occurrence of a variable x by an interval variable $\mathbf{x}$, by replacing every operator by its interval arithmetic counterpart and by replacing every elementary function by its range under the assumption that 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 first (normal) mode of the library uses interval evaluation. 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 $\mathbf{f}(\mathbf{x}) := f(\mathbf{x}) = \{f(x) \mid x \in \mathbf{x} \subseteq D_f\}$. 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, interval arithmetic 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.

Let $\mathbb{R}^* = \mathbb{R} \cup \{-\infty\} \cup \{\infty\}$. Following G.W. Walster [Chiriaev and Walster 1999; Walster et al. 2000b], we define the containment set:

Definition 2. Let $f : D_f \subseteq \mathbb{R} \to \mathbb{R}$. Then the containment set $f^* : \wp\mathbb{R}^* \to \wp\mathbb{R}^*$ defined by

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

contains the extended range of f.

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]

Table I. Extended Addition and Subtraction

$+$	$-\infty$	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 II. Extended Multiplication

$*$	$-\infty$	$y<0$	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 III. Extended Division

$/$	$-\infty$	$y<0$	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 IV. Extended Interval Division

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

THEOREM 3.1. *The containment evaluation of f is always defined*[1]*, and*

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

For the proof of this theorem, all arithmetic operators and elementary functions are extended to the closure of their domains. This can be done in a straightforward manner, see 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.

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 $[2,\infty]+[3,\infty]=[5,\infty]$, $[2,\infty]-[3,\infty]=\mathbb{R}^*$, $[2,\infty]*[-3,3]=\mathbb{R}^*$.

[1]This is in contrast to the interval evaluation.

Table V. Extended Domains and Ranges for the Elementary Functions

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

Division is more subtle. Table IV shows the cases when 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 the 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 $\log[-1, 1] = [-\infty, 0]$, $\log[-2, -1] = \emptyset$, $\coth[-1, 1] = \mathbb{R}^*$. Note that ordinary intervals may be obtained, even if the interval evaluation is not defined, $\sqrt{[-1, 1]} = [0, 1]$, for example.

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 considered, whereas x^y is calculated as $e^{y \ln x}$ in the pow function. We intentionally chose 2 different names, since power($\mathbf{x}, k$) $\subseteq$ pow($\mathbf{x}$, $[k, k]$) does not hold for negative $\mathbf{x}$.

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

4. FILIB++ 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.

Analogous to the `basic_string` template of the C++ standard, we use a concept called a traits class [Myers 1995] for our implementation. A traits class is a template class that allows access to data fields and the 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 on several character types, as the well-known `char` primitive type, but often also for some kind of a wide character type. The traits class 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. The `fp_traits` class 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 the maximum finite value of the type. It also has functions for testing properties of the number type such as testing for infinity, for example.

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.

The following rounding methods are supported.

—`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`. It 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 rounded to the 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 are rounded with rounding to the nearest.

—`no_rounding`. It uses the default rounding, hence not useable for applications. We used it for performance tests.

Table VI. Currently Existing fp_traits Implementations

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

—native_onesided_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 than native_switched though as only one switch to a directed rounding mode is used.

—native_onesided_global. It is like native_onesided_switched but there is 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. However, it has the problems described in native_directed, and it is necessary to 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 unit in the last place (ulp) 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 testing the performance for any rounding mode on the target machine.

The availability of various rounding modes further enhances the portability of the library. Whereas the native 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. This is illustrated in Figure 7 which shows part of the inheritance hierarchy used in the library (note that this is a simplified diagram).

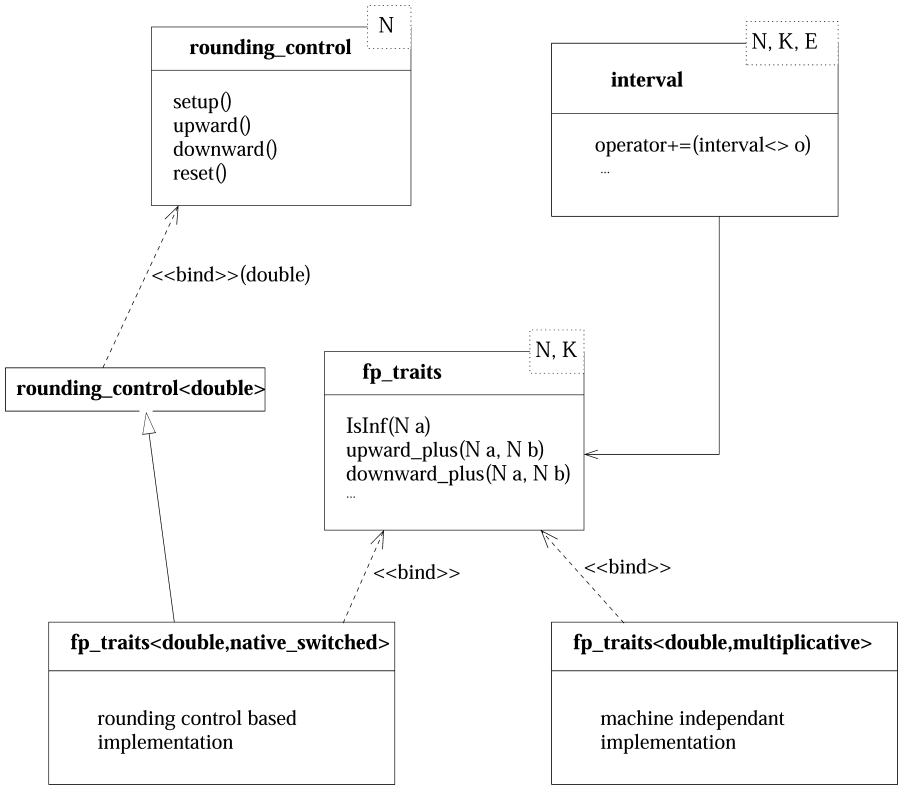

Fig. 7. Part of the inheritance hierarchy.

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

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

The filib++ interval class is realized as a template class. Currently there are three template parameters: the underlying basic (floating-point) number type `N`, the method of implementing the directed roundings `rounding_control`, and the computation mode `interval_mode`. Here, the basic number type `N` may be `float` or `double`, and `rounding_control K` may be one of the rounding modes listed in Table VI. The parameter `interval_mode` may be one of

— `i_mode_normal`: computation is performed by interval evaluation of expressions, floating-point exceptions are not masked;
— `i_mode_extended`: floating point exceptions are masked and evaluation is performed in containment mode;
— `i_mode_extended_flag`: as `i_mode_extended`, but a global flag is set, if an exception would have occurred in normal mode.

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,E> & interval<N,K,E>::operator +=
                                (interval<N,K,E> 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;
}
```

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 efficient.

4.2 Template Instantiation

Some examples of declaring an interval may help in using the library.

`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 native_switched or i_mode_normal implicitly.

`interval<double,multiplicative,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.

`interval<double,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 since directed rounding is used for floating-point operations.

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

4.3 Utility Functions 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): tests if a is not a number
- bool **IsInf**(N const & a): tests if a is infinite
- N const & **infinity**(): returns positive infinity
- N const & **ninfinity**(): returns negative infinity
- N const & **quiet_NaN**(): returns a quiet (nonsignaling) **NaN**
- N const & **max**(): returns the maximum finite value for N
- N const & **min**(): returns the minimum finite positive nondenormalized value for N

`N const &` **l_pi**`()`: returns a value that is as close as possible but no bigger than π

`N const &` **u_pi**`()`: returns a value that is as close as possible but 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 as close to $a + b$ as possible and no bigger than $a + b$.

`N` **upward_minus**`(N const & a, N const & b)`:

`N` **downward_minus**`(N const & a, N const & b)`:

`N` **upward_multiplies**`(N const & a, N const & b)`:

`N` **downward_multiplies**`(N const & a, N const & b)`:

`N` **upward_divides**`(N const & a, N const & b)`:

`N` **downward_divides**`(N const & a, N const & b)`:
Correspondingly for $-, \cdot, /$

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

4.4 The `interval<>` Class

Denote by $\underline{x}$ and $\overline{x}$ 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, that is, 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(interval_io_exception)`: constructs 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 `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, III 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. $\mathbb{R}^* = [-$ `INFTY`$, +$ `INFTY`$]$ 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. $\mathbb{R}^* = [-$ `INFTY`$, +$ `INFTY`$]$ 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 of `T`.

`N const &` **sup**`() const`: returns the upper bound of `T`.

$*$ `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 $-\infty = [-$ `INFTY`$, -M]$.
* `static interval<>` **POS_INFTY**`()` returns the point interval $+\infty = [M, +$ `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` if and only if `T` is a point interval.

`bool` **hasUlpAcc**`(unsigned int const & n) const`: returns `true` if and only if the distance of the bounds $\overline{t} - \underline{t} \leq n$ `ulp`, that is, a 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:

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

`N` **mig**`() const`: returns the mignitude, that is,

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

In the extended mode, the following cases are considered:

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

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

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

In the extended mode, the following cases are considered:

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

`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:

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

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 **diam**() const: returns the diameter or width of an 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:

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

The set theoretic methods are

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

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

In the extended mode, it returns

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

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

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

In the extended mode, this function returns

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

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

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

In the extended mode, this function returns

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

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

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

interval<> **intersect**(interval<> const & X) const: returns the intersection of the intervals T and X. If T and X are disjoint, 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

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

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

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

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

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

`bool` **contains**`(N x) const`: returns `true` if and only if `x` $\in$ `T`

`bool` **interior**`(interval<> const & X) const`: returns `true` if and only if `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` if and only if `T` is a proper subset of `X`.

`bool` **subset**`(interval<> const & X) const`: returns `true` if and only if `T` is a subset of `X`.

`bool` **proper_superset**`(interval<> const & X) const`: returns `true` if and only if `T` is a proper superset of `X`.

`bool` **superset**`(interval<> const & X) const`:returns `true` if and only if `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` if and only if `T` and `X` are equal sets.

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

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

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

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

`bool` **sgt**`(interval<> const & X) const`: returns `true` if and only if the $>$ relation holds for the bounds

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

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

`bool` **sle**`(interval<> const & X) const`: returns `true` if and only if the $\leq$ relation holds for the bounds

```
T.sle(X) == T.inf() ≤ X.inf() && T.sup() ≤ X.sup().
```

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

`bool` **slt**`(interval<> const & X) const`: returns `true` if and only if the $<$ relation holds for the bounds

```
T.slt(X) == T.inf() < X.inf() && T.sup() < 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` if and only if the $=$ relation holds for all individual points from `T` and `X`, that is, $\forall t \in$ `T`, $\forall x \in$ `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` if and only if the $\neq$ relation holds for all individual points from `T` and `X`, that is, $\forall t \in$ `T`, $\forall x \in$ `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` if and only if the $\geq$ relation holds for all individual points from `T` and `X`, that is, $\forall t \in$ `T`, $\forall x \in$ `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` if and only if the $>$ relation holds for all individual points from `T` and `X`, that is, $\forall t \in$ `T`, $\forall x \in$ `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` if and only if the $\leq$ relation holds for all individual points from `T` and `X`, that is, $\forall t \in$ `T`, $\forall x \in$ `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` if and only if the $<$ relation holds for all individual points from `T` and `X`, that is, $\forall t \in$ `T`, $\forall x \in$ `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`, if and only if there exist points that the $=$ relation holds, that is, $\exists t \in$ `T`, $\exists x \in$ `X` : $t = x$
In the extended mode, it returns `false` if `T == ∅` or `X == ∅`.

`bool` **pne**`(interval<> const & X) const`: returns `true` if and only if there exist points that the $\neq$ relation holds, that is, $\exists t \in$ `T`, $\exists x \in$ `X` : $t \neq x$
In the extended mode, it returns `true` if `T == ∅` or `X == ∅`.

`bool` **pge**`(interval<> const & X) const`: returns `true` if and only if there exist points that the $\geq$ relation holds, that is, $\exists t \in$ `T`, $\exists x \in$ `X` : $t \geq x$
In the extended mode, it returns `false` if `T == ∅` or `X == ∅`.

`bool` **pgt**`(interval<> const & X) const`: returns `true` if and only if there exist points that the $>$ relation holds, that is, $\exists t \in$ `T`, $\exists x \in$ `X` : $t > x$
In the extended mode, it returns `false` if `T == ∅` or `X == ∅`.

`bool` **ple**`(interval<> const & X) const`: returns `true` if and only if there exist points that the $\leq$ relation holds, that is, $\exists t \in$ `T`, $\exists x \in$ `X` : $t \leq x$
In the extended mode, it returns `false` if `T == ∅` or `X == ∅`.

`bool` **plt**`(interval<> const & X) const`: returns `true` if and only if there exist points that the $<$ relation holds, that is, $\exists t \in$ `T`, $\exists x \in$ `X` : $t < x$
In the extended mode, it returns `false` if `T == ∅` or `X == ∅`.

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.

`static interval<N,K,E>` **readBitImage**`(std::istream & in) throw(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 `interval_io_exception` is thrown.

`static interval<N,K,E>` **readHexImage**`(std::istream & in) throw(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 `interval_io_exception` is thrown.

`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 obtain verified graphics of implicitly defined curves as shown in the first chapter of this article.

```
//Verified computation of level curvesusing filib++ and routines from the STL
#include <fstream>
#include <list>
#include <interval/interval.hpp> //filib++
//...
using namespace filib;
//Simplify instantiation of intervals
typedef 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 obtain the verified graphic in Figure 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. Figure 9 is 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 defined partially. For example, a coverage of the graph of $f(x) := x + \arctan(\log(\sin(10x)/(x+1)))$ may be computed using the previous program 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, the part of the graph of f close to $x = -1$ is missing.

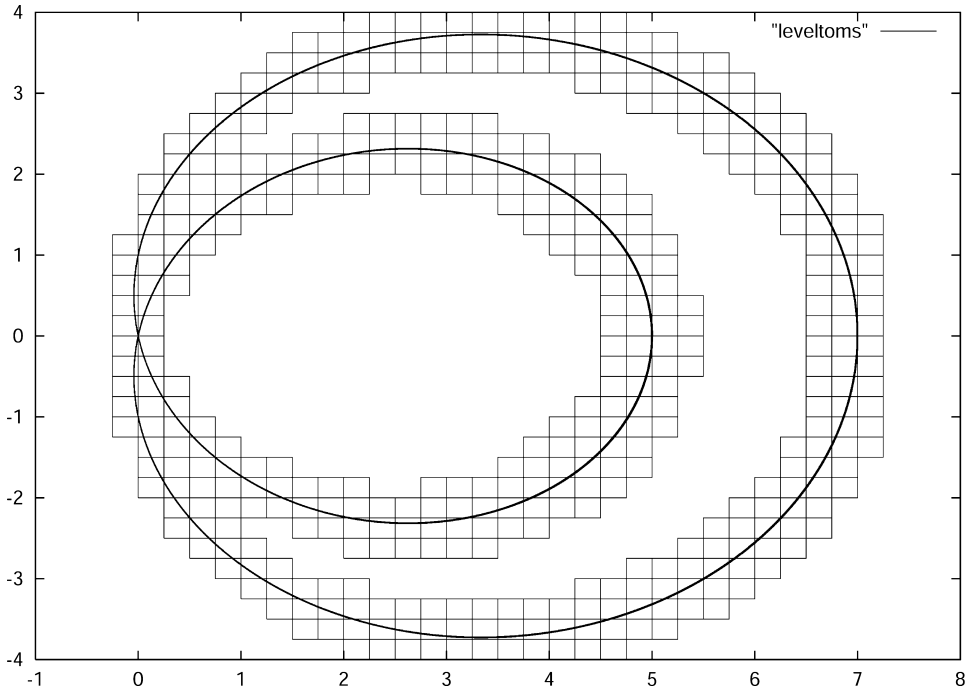
"leveltoms"
4
3
2
1
0
-1
-2
-3
-4
-1
0
1
2
3
4
5
6
7
8

Fig. 8.

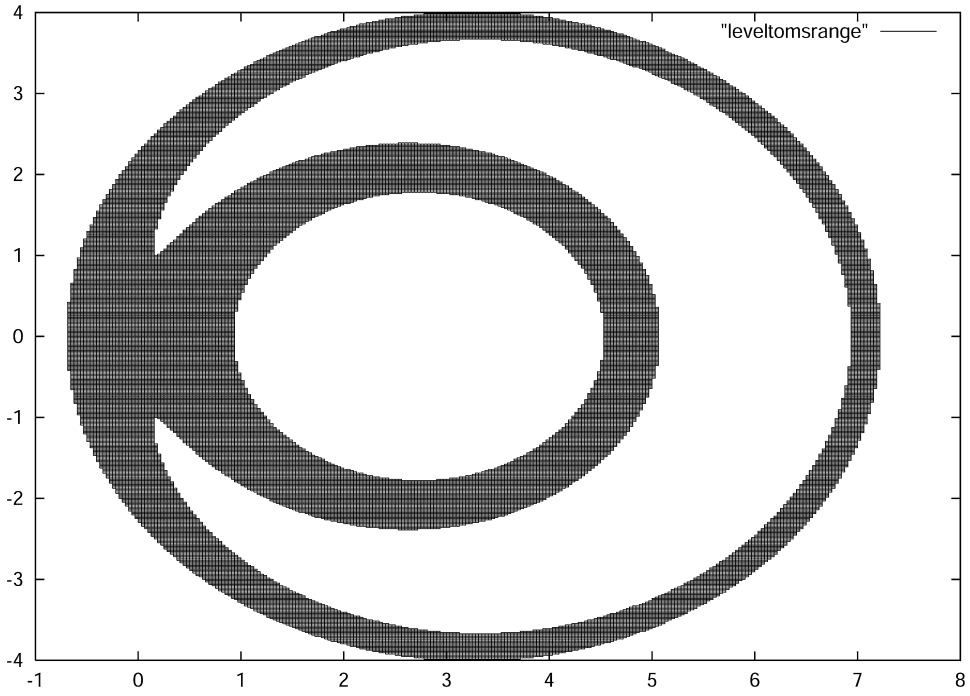
"leveltomsrange"
4
3
2
1
0
-1
-2
-3
-4
-1
0
1
2
3
4
5
6
7
8

Fig. 9.

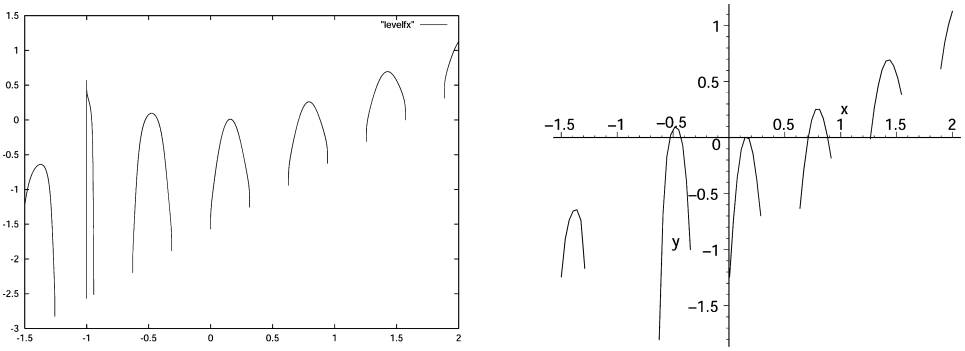

Fig. 10. $f(x) := x + \arctan(\log(\sin(10x)/(x+1)))$: verified (left)/Maple (right).

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

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

For `epsilon=0.0001`, we compute 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]
  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 perform differently. The performance heavily depends on the hardware 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. Tables VII and VIII show the performance in MIOPs (million interval operations per second) for the normal and the extended mode.

Table VII. Performance for Normal Interval 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

Table VIII. Performance for Extended 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

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.

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 really 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 C++ compiler 2.95.2 takes 3130 milliseconds. Using the native interval operations of the Sun Forte compiler on the same machine takes 3370 milliseconds.

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 milliseconds, whereas the native interval functions of the Sun Forte compiler take 5240 milliseconds.

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.

Table IX. Comparison with Profil/BIAS

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

In this article, we payed little attention to the use of the library. Only some small applications have been discussed to show the quality of mathematical results that can 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 (e.g., 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.

The third template parameter `interval_mode` enables the user to do computations using containment sets as well as computations 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; Hofschuster and Krämer 2004; 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, NY.

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, Germany.

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. AND KRÄMER, W. 2004. C-XSC 2.0—a C++ class library for extended scientific computing. In *Numerical Software with Result Verification*, R. Alt, A. Frommer, B. Kearfott, and W. Luther, Eds. Springer Lecture Notes in Computer Science, Vol. 2991, 15–35.

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. Soft. 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. Soft. 20*, 4 (Dec.) 447–459.

KEARFOTT, R. B. AND NOVOA, M. 1990. Algorithm 681: Intbis, a portable interval newton/bisection package. *ACM Trans. Math. Soft. 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, Germany.

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 31st Spring Conference of the Union of Bulgarian Mathematicians*. Union of Bulgarian Mathematicians, Borovets, Bulgaria, 344–355.

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

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

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.

MYERS, N. 1995. Traits: a new and useful template technique. *C++ Report*.

RUMP, S., M. 1998. Intlab–interval laboratory. In *Developments in Reliable Computing*, T. Csendes, Ed. KluwerAcademic Publisher, New York, NY.

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

STROUSTRUP, B. 2000. *The C++ Programming Language*, Special Ed. Addison-Wesley, Reading, MA.

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. Tech. rep., Sun Microsystems.

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

Received June 2002; accepted September 2005