Bergische Universität
Wuppertal

The Interval Library `filib++` 2.0

Design, Features and Sample Programs

M. Lerch, G. Tischler, J. Wolff von Gudenberg,
W. Hofschuster, W. Krämer

Preprint 2001/4

Wissenschaftliches Rechnen/
Softwaretechnologie

Impressum

Herausgeber:	Prof. Dr. W. Krämer, Dr. W. Hofschuster Wissenschaftliches Rechnen/Softwaretechnologie Fachbereich 7 (Mathematik) Bergische Universität Wuppertal Gaußstr. 20 D-42097 Wuppertal

Internet-Zugriff

Die Berichte sind in elektronischer Form erhältlich über die World Wide Web Seiten

`http://www.math.uni-wuppertal.de/wrswt/literatur.html`

Autoren-Kontaktadresse

Dipl.-Inf. M. Lerch
G. Tischler
Prof. Dr. J. Wolff von Gudenberg
Lehrstuhl für Informatik II
Universität Würzburg
Am Hubland
D-97074 Würzburg

Dr. W. Hofschuster
Prof. Dr. W. Krämer
Bergische Universität Wuppertal
Gaußstr. 20
D-42097 Wuppertal

`filib++` is freely available from
`http://www.math.uni-wuppertal.de/wrswt/software.html`

Contents

1 Interval Evaluation and Containment Evaluation **6**
- 1.1 Interval Evaluation . . . 6
- 1.2 Containment Evaluation . . . 7
- 1.3 Functional Specification of `filib++` – Overview . . . 11
  - 1.3.1 Internal Representation . . . 11
  - 1.3.2 Construction and Access . . . 11
  - 1.3.3 Arithmetic Operations . . . 11
  - 1.3.4 Relations . . . 12
  - 1.3.5 Set Theoretic Functions . . . 12
  - 1.3.6 Elementary Arithmetic Functions . . . 12
  - 1.3.7 Input and Output . . . 12

2 Instantiation and Options **13**
- 2.1 Namespace filib . . . 13
- 2.2 Two Modes and Two Versions . . . 13
- 2.3 Template Parameters of Class `interval<>` . . . 13
  - 2.3.1 Basic Number Type . . . 14
  - 2.3.2 Rounding Control . . . 14
- 2.4 Traits . . . 16
- 2.5 Alternative Macro Version . . . 17
- 2.6 Instantiation Examples . . . 17

3 The `fp_traits<>` class **19**
- 3.1 Template Arguments . . . 19
- 3.2 Utility Functions . . . 19

4 The `interval<>` class **22**
- 4.1 Basic Number Type . . . 22
- 4.2 Constructors . . . 22
- 4.3 Assignment . . . 23
- 4.4 Arithmetic Methods . . . 23
- 4.5 Access and Information Methods . . . 24
- 4.6 Set Theoretic Methods . . . 27
- 4.7 Interval Relational Methods . . . 28
  - 4.7.1 Set Relations . . . 28

4.7.2 Certainly Relations 29
4.7.3 Possibly Relations 30
4.8 Input and Output 31

5 Global Functions 32
5.1 Arithmetic Operators 32
5.2 Access and Information 34
5.3 Set Theoretic Functions 34
5.4 Interval Relational Functions 36
5.4.1 Set Relational Functions 36
5.4.2 Certainly Relational Functions 36
5.4.3 Possibly Relational Functions 37
5.5 Elementary Functions 37
5.6 Input and Output Operators 39

6 Sample Programs 40
6.1 Some Simple Containment Computations 40
6.2 Working with the `fp_traits<>` class 43
6.3 Template Version Versus Macro Library Version 46

7 Application: Verified Computation of all Solutions of a Nonlinear Equation 49
7.1 Verified Bisection Method 49
7.2 Extended Interval Newton Method 55

8 Installation 64
8.1 Compiler Requirements 64
8.2 Installation and Usage 64
8.2.1 Installation 64
8.2.2 Usage of the Template Library 65
8.2.3 Usage of the Macro Library 65
8.3 Organization of Subdirectories 65

Abstract

`filib++` is an extension of the interval library `filib` originally developed in Karlsruhe [3]. The most important aim of the latter was the fast computation of guaranteed bounds for interval versions of a comprehensive set of elementary function. `filib++` extends this library in two aspects. First, it adds a second mode, the "extended" mode, that extends the exception-free computation mode using special values to represent infinities and NotaNumber known from the IEEE floating-point standard 754 to intervals. In this mode so-called containment sets are computed to enclose the topological closure of a range of a function defined over an interval [6]. Second, our new state of the art design uses templates and traits classes in order to get an efficient, easily extendable and potable library, fully according to the C++ standard [1].

Overview

Chapter 1 presents the difference between the normal mode computing interval evaluations and the exception free extended mode computing containment sets. The functionality of the library is roughly sketched. Chapter 2 then shortly explains the inner structure and describes how to use it. Chapters 3, 4, 5 contain the specification of the complete interface and the following two chapters 6, 7 show sample programs and small but powerful applications using `filib++` . Finally, chapter 8 gives some installation hints and describes how to compile, to link and to run a program.

Acknowledgments, Origins

Many people have contributed to the design and the construction of `filib++`.The first version of the library has been published in Karlsruhe by W. Hofschuster and W. Krämer [3, 4]. It was a library with emphasis on the fast evaluation of the elementary functions. Only the normal mode was implemented. The code for calculating elementary functions in `filib++` has been taken from this library with changes only for the extended mode.

The implementation of the extended mode has largely been influenced by Bill Walster and his team at Sun Microsystems [5, 7]. They started with a Fortran extension and now also provide a C++ library [8].

Input and output routines of `filib++` have been adapted from the libI77 of the runtime system of the Gnu Fortran Compiler [9].

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

Parts of this manual have been published in earlier reports [10, 11, 12].

Chapter 1

Interval Evaluation and Containment Evaluation

1.1 Interval Evaluation

We assume that the reader is familiar with the basic ideas of interval arithmetic (for a good introduction and further references see [2]). In this introductory chapter we use bold face for continuous intervals, represented by two real bounds, i.e.

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

$I\!I\!R$ denotes the space of all finite intervals.

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

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

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

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

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

The interval evaluation is not defined, if $\mathbf{x}$ contains a point $y \notin D_f$. Division by an interval containing 0, e.g., is forbidden. But note, that even if $\mathbf{x} \subseteq D_f$, $\mathbf{f}$ may not be defined. The result depends on the syntactic formulation of the expression: For the function

$$f_1(x) = \frac{1}{x \cdot x + 2}$$

the call $\mathbf{f}_1([-2,2])$ is not defined, because

$$[-2,2]\cdot[-2,2]=[-4,4]$$

whereas

$$f_2(x)=\frac{1}{x^2+2}$$

yields $\mathbf{f}_2([-2,2])=[1/6,1/2]$. Of course for real arguments x it holds $f_1(x)=f_2(x)$

The result of an elementary function call over an interval is defined as the set, i.e. interval because of continuity, 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 $\mathbf{f}(\mathbf{x}):=f(\mathbf{x})=\{f(x)|x\in\mathbf{x}\subseteq D_f\}$, i.e. the result of an interval evaluation of such a function is by definition equal to the range of the function over the argument interval.

1.2 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 indeterminate forms involving zero and infinities.

Following G. W. Walster in [5, 6] we define the containment set:

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

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

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

Hence, the containment set of a function is the closure of the range including all limits and accumulation points.

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

Let $\mathbb{IR}^*$ denote the set of all extended intervals with endpoints in $\mathbb{R}^*$.

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

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

Table 1.1: extended addition

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

Table 1.2: extended subtraction

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

Table 1.3: extended multiplication

Theorem 2 *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. [5]. We apply the well known rules to compute with infinities. If we encounter an undefined operation like $0 \cdot \infty$ we deliver the set of all limits, i.e. $\mathbb{R}^*$. Note that negative values are also possible, since 0 can be approached from both sides.

We show the containment sets for the basic arithmetic operations in the following tables.

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.

[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 1.4: extended division

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

Table 1.5: extended interval division

Some examples:

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

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

For the elementary functions Table 1.6 shows the extended domains and extended ranges. The containment set for an elementary function is computed by directly applying the definition $\mathbf{f}(\mathbf{x}) := \overline{\{f(x)|x\in\mathbf{x}\cap D_f\}}$. If the argument lies strictly outside the domain of the function, we obtain the empty set as result. If the argument $\mathbf{x}$ contains a singularity the corresponding values for $\pm\infty$ are produced. The functions in containment mode never produce an overflow or illegal argument error.
Some examples:

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

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

It has been shown in [6, 7], that using these 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	$[-\infty,-1] \cup [1,\infty]$	$\mathbb{R}^*$	$\operatorname{acoth}[-1,1] = [-\infty]$ $\operatorname{acoth}[1,1] = [\infty]$

Table 1.6: extended domains and ranges of elementary functions

1.3 Functional Specification of `filib++` – Overview

1.3.1 Internal Representation

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

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

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

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

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

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

the real numbers smaller than the negative overflow threshold are

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

and

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

means all real numbers.

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

1.3.2 Construction and Access

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

1.3.3 Arithmetic Operations

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

1.3.4 Relations

All three kinds of set-like, certainly or possibly comparisons [5] are provided as methods and as functions. For a certainly-relation to be true, every element of the operand intervals must satisfy the relation. A possibly-relation is true if it is satisfied by any elements of the operand intervals. The set-relations treat intervals as sets. All three classes of interval relational functions converge to the normal relational function on points if both operand intervals are degenerate.

For set-relations the operators `==, !=, >=, <=` are overloaded. We further supply the predicate y `interior x` $\iff \underline{x} < y < \overline{x}$.

1.3.5 Set Theoretic Functions

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

1.3.6 Elementary Arithmetic Functions

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

1.3.7 Input and Output

The standard input output operators are overloaded for intervals, but without appropriate rounding to the external decimal string format. Hexadecimal and binary input and output routines are also provided.

Chapter 2

Instantiation and Options

2.1 Namespace filib

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

2.2 Two Modes and Two Versions

One template version and one macro version of the library are supplied. In either version two modes can be used. In the default or "normal" mode traditional interval evaluations are computed, and the program terminates, if the argument interval is not contained in the domain of a function. Overflow treatment is done according to the chosen floating point option.

In the "extended" mode containment sets over $\mathbb{R}^*$ are computed, no exceptions are raised. This mode is obtained by setting the constant `FILIB_EXTENDED` during compilation.

For the macro library, this constant `FILIB_EXTENDED` has to be set for the compilation of the library as well as the application program. For the template version, the constant is not used during the building of the static parts of the library and thus only has be provided when compiling application programs. That essentially means that you do not have to recompile the template version for switching the extended mode on or off.

In the following we describe the template version. Since the interface of both versions are essentially identical, a few statements about the macro version in Section 2.5 are sufficient.

2.3 Template Parameters of Class `interval<>`

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

2.3.1 Basic Number Type

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

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

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

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

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

2.3.2 Rounding Control

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

The second parameter can have the following values

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

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

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

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

[1] this implementation is the original `filib` code, see [3]

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

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

  $\mathrm{low} : R \to R$

  $$\mathrm{low}(a) := \begin{cases} a \geq 0: & a \odot \mathrm{pred}(1) \\ a < 0: & a \odot \mathrm{succ}(1) \end{cases} \tag{2.1}$$

  and $\mathrm{high} : R \to R$

  $$\mathrm{high}(a) := \begin{cases} a \geq 0: & a \odot \mathrm{succ}(1) \\ a < 0: & a \odot \mathrm{pred}(1). \end{cases} \tag{2.2}$$

  where $\odot$ means *round-to-nearest*-multiplication and where $\mathrm{succ}(x) = \min\{y | y \in R, y > x\}$ or $\mathrm{pred}(x) = \max\{y | y \in R, y < x\}$, respectively.

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

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

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

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

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

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

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

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

  For the proof, see [11].

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

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

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

  IMPORTANT: it is necessary to call the method

`filib::fp_traits<N,K>::setup()`

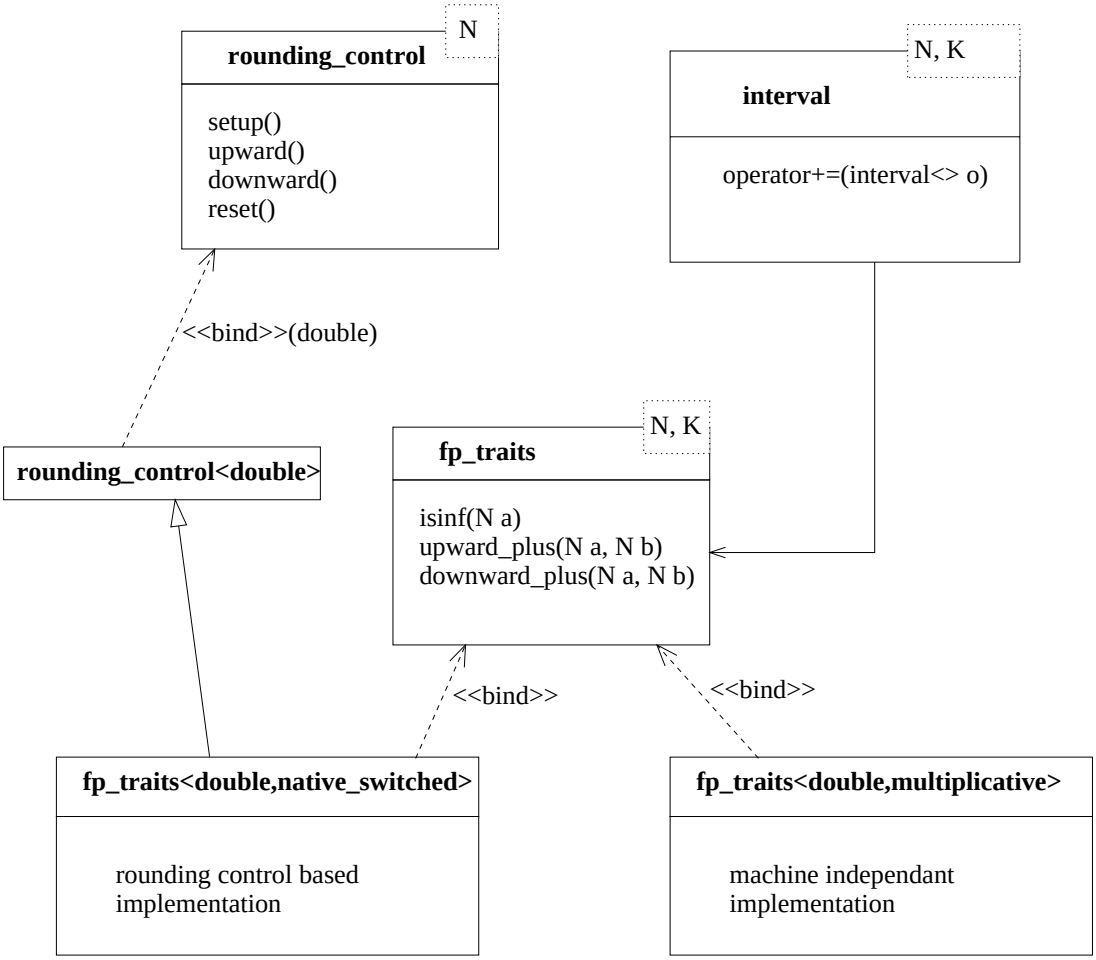
whenever you use instances of a different instantiation of the `interval<>` class with template parameters `N,K`. This is especially true at program start. When starting the usage of the `native_onesided_global` mode, the correct sequence is first calling `setup` and then `downward`.

2.4 Traits

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

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

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



2.5 Alternative Macro Version

For the mainly used data type `interval<double>` a non generic version that is highly optimized in speed is provided. It only supports the data type Interval, i.e. an interval of doubles. The switching between the various rounding modes is implemented via compile time constants. The arithmetic operations are defined as macros. This design certainly is not up to date concerning modern software engineering principles, but benchmarks showed, that the arithmetic was considerably faster, see [10]. The interface of the methods and functions is identical to the templated version.

2.6 Instantiation Examples

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

- `filib::interval<double> A;`
  This is the default instantiation. `A` is an interval over the floating-point type

`double`. The second parameter is set to its default *filib::native_switched*

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

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

Chapter 3

The fp_traits<> class

3.1 Template Arguments

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

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

3.2 Utility Functions

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

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

- `N const &` **infinity**`()`
  returns positive infinity
- `N const &` **ninfinity**`()`
  returns negative infinity
- `N const &` **quiet_NaN**`()`
  returns a quiet (non-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.

Chapter 4

The `interval<>` class

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

4.1 Basic Number Type

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

4.2 Constructors

The following constructors are provided for the interval class:

- **interval**`()`:
  The interval $[0, 0]$ is constructed.
- **interval**`(N const & a)`:
  The interval $[a, a]$ is constructed.The point intervals for $+\infty$ and $-\infty$ are given by $[M, +$`INFTY`$]$ or $[-$`INFTY`$, -M]$, respectively.
- **interval**`(N const & a, N const & b)`:
  If $a \leq b$ 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 rounded down and the supremum is rounded up. If the strings cannot be parsed by `strtod`, an exception of type `filib::interval_io_exception` is thrown.

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

4.3 Assignment

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

4.4 Arithmetic Methods

The following methods are provided for updating arithmetic operations. Note that the usual operators are available as global functions (see 5.1).

The special cases of the extended mode are not explicitly mentioned here, see tables 1.1,1.2,1.3,1.4 for details.

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

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

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

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

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

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

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

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

- `interval<> &` **operator-=**`(N const & a)`(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.

4.5 Access and Information Methods

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

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

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

$*$ `bool isEmpty() const`:
returns `true`, iff `T` is the empty interval.

$*$ `bool isInfinite() const`:
returns `true`, iff `T` has at least one infinite bound.

$*$ `static interval<> EMPTY()` :
returns the empty interval.

$*$ `static interval<> ENTIRE()` :
returns $\mathbb{R}^*$.

* `static interval<> NEG_INFTY()` :
  returns the point interval $-\infty = [-\texttt{INFTY}, -M]$.

* `static interval<> POS_INFTY()`
  returns the point interval $+\infty = [M, +\texttt{INFTY}]$.

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

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

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

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

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

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

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

  In the extended mode the following cases are distinguished:

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

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

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

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

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

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

In the extended mode the following cases are distinguished:

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

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

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

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

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

  In the extended mode the following cases are considered:

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

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

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

  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:

$$\texttt{T.abs()} = \begin{cases} \emptyset & \text{for } \texttt{T == } \emptyset \\ [M, +\texttt{INFTY}] & \text{if } \texttt{T.isInfinite()} \end{cases}$$

4.6 Set Theoretic Methods

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

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

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

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

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

  In the extended mode return

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

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

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

  In the extended mode return

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

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

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

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

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

  In the extended mode return

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

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

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

  In the extended mode return

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

  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 `x` $\in$ `T`

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

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

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

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

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

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

4.7 Interval Relational Methods

4.7.1 Set Relations

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

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

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

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

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

- `bool` **sgt**`(interval<> const & X) const`:
  returns `true`, iff 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 return `false`, if `T` $== \emptyset$ and `X` $== \emptyset$.

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

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

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

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

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

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

4.7.2 Certainly Relations

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

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

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

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

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

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

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

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

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

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

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

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

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

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

  In the extended mode return `false`, if `T == ∅` or `X == ∅`.

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

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

  In the extended mode return `false`, if `T == ∅` or `X == ∅`.

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

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

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

  In the extended mode return `false`, if `T == ∅` or `X == ∅`.

4.7.3 Possibly Relations

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

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

  In the extended mode return `false`, if `T == ∅` or `X == ∅`.

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

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

  In the extended mode return `true`, if `T == ∅` or `X == ∅`.

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

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

  In the extended mode return `false`, if `T == ∅` or `X == ∅`.

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

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

  In the extended mode return `false`, if `T == ∅` or `X == ∅`.

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

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

  In the extended mode return `false`, if `T == ∅` or `X == ∅`.

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

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

  In the extended mode return `false`, if `T == ∅` or `X == ∅`.

4.8 Input and Output

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

- `std::ostream &` **hexImage**`(std::ostream & out) const`:
  output a hexadecimal representation. 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 `<<`. (see 5.6)

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

Chapter 5

Global Functions

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

5.1 Arithmetic Operators

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.2 Access and Information

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

5.3 Set Theoretic Functions

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

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

- `interval<>` **blow**`(interval<> const & A, N const & eps)`:
  equivalent to `A.blow(eps)`.

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

- `interval<>` **hull**`(interval<> const & A, interval<> const & B)`:
  equivalent to `A.hull(B)`, also available as `interval_hull()`.

- `interval<>` **hull**`( N const & b, interval<> const & A)`:
  equivalent to `A.hull(b)`, also available as `interval_hull()`.

- `interval<>` **hull**`( N const & a, N const & b)`:
  returns the interval hull of the 2 numbers `a` and `b`, also available as `interval_hull()`.

  In the extended mode returns $\emptyset$, if `x == y == NaN`

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

- `bool` **in**`(N & a, interval<> const & B)`:
  equivalent to `B.contains(a)`.

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

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

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

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

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

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

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

5.4 Interval Relational Functions

5.4.1 Set Relational Functions

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

5.4.2 Certainly Relational Functions

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

5.4.3 Possibly Relational Functions

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

5.5 Elementary Functions

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

- `interval<>` **acos**`(interval<> const & A)`:
  inverse cosine
- `interval<>` **acosh**`(interval<> const & A)`:
  inverse hyperbolic cosine
- `interval<>` **acot**`(interval<> const & A)`:
  inverse cotangent
- `interval<>` **acoth**`(interval<> const & A)`:
  inverse hyperbolic cotangent
- `interval<>` **asin**`(interval<> const & A)`:
  inverse sine
- `interval<>` **asinh**`(interval<> const & A)`:
  inverse hyperbolic sine
- `interval<>` **atan**`(interval<> const & A)`:
  inverse tangent
- `interval<>` **atanh**`(interval<> const & A)`:
  inverse hyperbolic tangent

- `interval<>` **cos**`(interval<> const & A)`:
  cosine
- `interval<>` **cosh**`(interval<> const & A)`:
  hyperbolic cosine
- `interval<>` **cot**`(interval<> const & A)`:
  cotangent
- `interval<>` **coth**`(interval<> const & A)`:
  hyperbolic cotangent
- `interval<>` **exp**`(interval<> const & A)`:
  exponential e^A
- `interval<>` **exp10**`(interval<> const & A)`:
  exponential to base 10. 10^A
- `interval<>` **exp2**`(interval<> const & A)`:
  exponential to base 2. 2^A
- `interval<>` **expm1**`(interval<> const & A)`:
  $e^A - 1$
- `interval<>` **log**`(interval<> const & A)`:
  logarithm to base e
- `interval<>` **log10**`(interval<> const & A)`:
  logarithm to base 10
- `interval<>` **log1p**`(interval<> const & A)`:
  $\log(A + 1)$
- `interval<>` **log2**`(interval<> const & A)`:
  logarithm to base 2
- `interval<>` **power**`(interval<> const & A, int const & p)`:
  power to an integer A^p
- `interval<>` **pow**`(interval<> const & A, interval<> const & B)`:
  general power function $\{a^b : a \in A, b \in B\}$.
- `interval<>` **sin**`(interval<> const & A)`:
  sine
- `interval<>` **sinh**`(interval<> const & A)`:
  hyperbolic sine
- `interval<>` **sqr**`(interval<> const & A)`:
  square

- `interval<>` **sqrt**`(interval<> const & A)`:
  square root
- `interval<>` **tan**`(interval<> const & A)`:
  tangent
- `interval<>` **tanh**`(interval<> const & A)`:
  hyperbolic tangent

5.6 Input and Output Operators

- `std::ostream &` **operator**`<<(std::ostream & out, interval<> const & A)`:
  outputs the interval `A` to the stream `out`. According to the output precision the usual format is `[` $\underline{a}$`,` $\overline{a}$ `]`. Note that the bounds are NOT rounded directly to the string format, but the standard output method is used instead. We recommend to use the `bitImage` method (see 4.8) for a detailed view. In case of an erroneous interval `[ UNDEFINED ]` is output in the normal mode. In the extended mode there are several special cases:
  - `[ EMPTY ]` for the empty interval
  - `[ -INFTY ]` for $[-\infty, -\mathtt{M}]$
  - `[ +INFTY ]` for $[\mathtt{M}, \infty]$
  - `[ ENTIRE ]` for $\mathbb{R}^*$
- `std::istream &` **operator**`>>(std::istream & in, interval<> & A) throw(filib::interval_io_exception)` :
  reads the interval `A` from the stream `in`. If the input cannot be parsed as an interval, an exception of type `filib::interval_io_exception` is thrown. Note that the input is converted to the used arithmetic type by using the standard function `strtod()`. That means that there is no care taken for directed rounding in the case that the provided numbers do not have an exact machine representation. We recommend using the method `readBitImage()` if there is need for a perfectly predictable input method.

Chapter 6

Sample Programs

This section demonstrates important features of the library `filib++` by small and easy to read sample programs. Some simple containment computations are shown, the problem of data conversion is demonstrated, the usage of the `fp_traits<>` class is considered, and binary and hexadecimal input and output of intervals is presented. An example for the usage of the template version as well as the usage of the macro version of `filib++` is given.

6.1 Some Simple Containment Computations

The following simple program shows how to use the (extended) interval data type `filib::interval<double>` of the library `filib++` to perform containment computations. To access the operations of the closed extended real interval system the compiler option `-DFILIB_EXTENDED` has to be used.

```
#include <interval/interval.hpp> //Use filib++
#include <iostream>

//Simplify instantiation of intervals
typedef filib::interval<double> interval;

using std::cout;
using std::endl;

int main()
{
   filib::fp_traits<double>::setup(); //Do initializations
   interval x(-1.5, 3); //Constructor call to create the interval [-1.5,3]
   cout << "x= " << x << endl;
   cout << "cos(x)= " << cos(x) << endl;
   cout << "log(x)= " << log(x) << endl;
   cout << "atan(log(x))= " << atan(log(x)) << endl;
   cout << "log([-2,-1])= " << log(interval(-2,-1)) << endl;
   return 0;
}
```

The output shows the results of the (exception free) containment computations:

```
x= [-1.5, 3]
cos(x)= [-0.99, 1]
log(x)= [-Infinity, 1.1]
atan(log(x))= [-1.57, 0.832]
log([-2,-1])= [ EMPTY ]
```

Predefined interval constants like `PI()` or `ENTIRE()` may be used. Again the typedef `typedef filib::interval<double> interval;` simplifies instantiation of intervals and access of interval constants.

```
#include <interval/interval.hpp> //Use filib++
#include <iostream>

//Simplify instantiation of intervals
typedef filib::interval<double> interval;

using std::cout;
using std::endl;

int main()
{
   filib::fp_traits<double>::setup(); //Do initializations
   interval x(0.5);                   //Point interval [0.5, 0.5]
   cout << "x= " << x << endl;
   cout << "isPoint(x): " << isPoint(x) << endl;

   x= interval::PI(); //Predefined constant interval containing Pi
   interval::precision(16);     //Increase precision for output
   cout << "x= " << x << endl << "sin(x)= " << sin(x) << endl;
   cout << "x.hasUlpAcc(1): " << x.hasUlpAcc(1)  << endl;

   x= interval::ENTIRE(); //x represents the complete real line
   cout << "x= " << x << "  rad(x)= " << rad(x) << endl;
   cout << "isInfinite(x): " << isInfinite(x) << endl;
   cout << "cosh(sqrt(log(x-5.0))/sin(x))= "
        <<  cosh(sqrt(log(x-5.0))/sin(x)) << endl;
   return 0;
}
```

The output of the program is:

```
x= [0.5, 0.5]
isPoint(x): 1
x= [3.141592653589793, 3.141592653589794]
sin(x)= [-3.216245299353279e-16, 1.224646799147355e-16]
x.hasUlpAcc(1): 1
x= [ ENTIRE ]  rad(x)= Infinity
```

```
isInfinite(x): 1
cosh(sqrt(log(x-5.0))/sin(x))= [1, Infinity]
```

To get proper enclosures of non representable real numbers like 0.1 you have to be careful. Simply writing `interval(0.1)` gives no such enclosure (for a detailed discussion of the data conversion problem which is a common problem in numerical computing see [2]):

```
#include <interval/interval.hpp> //Use filib++
#include <iostream>

//Simplify instantiation of intervals
typedef filib::interval<double> interval;

using std::cout;
using std::endl;

int main()
{
   filib::fp_traits<double>::setup(); //Do initializations
   //Note: interval(0.1) gives no enclosure for 0.1!
   //The real value 0.1 is not machine representable
   interval x= interval(0.1);
   cout << "x.hasUlpAcc(0): " << x.hasUlpAcc(0) << endl;

   //Compute an enclosure of the real value 0.1
   interval y;
   y= interval(1)/interval(10); //Use interval division!
   //Check the accuracy of the enclosure:
   cout << "y.hasUlpAcc(1): " << y.hasUlpAcc(1) << endl;
   cout << "y.hasUlpAcc(0): " << y.hasUlpAcc(0) << endl;

   cout << "Internal representation of interval x:" << endl;
   x.hexImage(cout);
   cout << "Internal representation of interval y:" << endl;
   y.hexImage(cout);
   return 0;
}
```

The output shows that the interval `x` is a point interval (the method call `x.hasUlpAcc(0)` gives `true`). The real number 0.1 is **not** an element of `x`. The interval `y` which is computed via an interval division is a correct enclosure of 0.1 (indeed it is the best possible using double precision interval bounds). The hexadecimal output of `y` shows this fact. As usual, printing 1 means `true` and 0 means `false` for an expression resulting a boolean value.

```
x.hasUlpAcc(0): 1
y.hasUlpAcc(1): 1
y.hasUlpAcc(0): 0
```

```
Internal representation of interval x:
[ 0:3fb:999999999999a ,
  0:3fb:999999999999a ]
Internal representation of interval y:
[ 0:3fb:9999999999999 ,
  0:3fb:999999999999a ]
```

Note, that the correct handling of the data conversion problem is critical to achieve the validation interval computations promise!

6.2 Working with the fp_traits<> class

We can also use directed rounded operations to compute an enclosure of e. g. the real number 0.1. Such operations are supplied by the class filib::fp_traits<interval::value_type> where value_type is the scalar numeric data type used to represent interval bounds (e. g. double).

```
#include <interval/interval.hpp> //Use filib++
#include <iostream>

//Simplify instantiation of intervals
typedef filib::interval<double> interval;

//Simplify access to methods of class fp_traits<>
typedef filib::fp_traits<interval::value_type> traits;

using std::cout;
using std::endl;

int main()
{
   filib::fp_traits<double>::setup(); //Do initializations
   //Compute an enclosure of the real value 0.1:
   interval x( traits::downward_divides(1,10),
               traits::upward_divides(  1,10) );
   cout << "Internal representation of the enclosure of 0.1:"  << endl;
   x.hexImage(cout);

   cout << "Largest floating point number: ";
   cout << traits::max() << endl;

   return 0;
}
```

The output shows the hexadecimal representation of the computed enclosure. By the way, also the result of filib::fp_traits<interval::value_type>::max() is shown, i. e. the value of the largest finite upper interval bound (which depends on the actual numeric data type used for interval bounds).

```
Internal representation of the enclosure of 0.1:
[ 0:3fb:9999999999999 ,
  0:3fb:999999999999a ]
Largest floating point number: 1.79769e+308
```

Internal representations of some intervals including infinite intervals are shown as output of the following program.

```
#include <iostream>
#include <fstream>

#include <interval/interval.hpp> //filib++

using std::cout;
using std::endl;

//Simplify instantiation of intervals
typedef filib::interval<double> interval;

//Simplify access to methods of class fp_traits<>
typedef filib::fp_traits<interval::value_type> traits;

void output(const string s, const interval x)
{
   cout << s << " = " << x  << " = " << endl;
   x.bitImage(cout);  //Internal representation
}

int main()
{
   filib::fp_traits<double>::setup(); //Do initializations
   interval x(2,3);
   output("x", x); //Decimal output and output of internal representation
   x=interval(1.7,2.9);
   output("x", x);

   x= interval::POS_INFTY();
   output("POS_INFTY()", x);
   cout << "mid(x)= " << mid(x) << endl;
   cout << "mid(x)==traits::infinity(): "
        << ( mid(x)==traits::infinity() ) << endl;
   cout << "in(mid(x), x): " << in(mid(x), x) << endl;
   output("Empty set", intersect(interval(), interval(1)) );
   output("Point interval [1,1]", interval( 1) );
   output("sin(interval::PI())", sin(interval::PI()) );
   cout << "Please input BitImage for x! ";
   x=interval::readBitImage(cin);
   output("Input was", x);
   cout << "and in hexadecimal representation:" << endl;
```

```
   x.hexImage(cout);
   cout << "Please input HexImage for x! ";
   x=interval::readHexImage(cin);
   cout << "Hexadecimal input was:" << endl;
   x.hexImage(cout);
   return 0;
}
```

Please take note of the qualifying prefixes of constants like `POS_INFTY()` and `infinity`. The first one comes from the class `filib::interval<double>` and the second one comes from the corresponding traits class `filib::fp_traits<interval::value_type>`).

```
x = [2, 3] =
[ 0:10000000000:0000000000000000000000000000000000000000000000000000 ,
  0:10000000000:1000000000000000000000000000000000000000000000000000 ]
x = [1.7, 2.9] =
[ 0:01111111111:1011001100110011001100110011001100110011001100110011 ,
  0:10000000000:0111001100110011001100110011001100110011001100110011 ]
POS_INFTY() = [ +INFTY ] =
[ 0:11111111110:1111111111111111111111111111111111111111111111111111 ,
  0:11111111111:0000000000000000000000000000000000000000000000000000 ]
mid(x)= Infinity
mid(x)==traits::infinity(): 1
in(mid(x), x): 1
Empty set = [ EMPTY ] =
[ 0:11111111111:1000000000000000000000000000000000000000000000000000 ,
  0:11111111111:1000000000000000000000000000000000000000000000000000 ]
Point interval [1,1] = [1, 1] =
[ 0:01111111111:0000000000000000000000000000000000000000000000000000 ,
  0:01111111111:0000000000000000000000000000000000000000000000000000 ]
sin(interval::PI()) = [-3.22e-16, 1.22e-16] =
[ 1:01111001011:0111001011001110110011100110011100111010111010010000001001 ,
  0:01111001010:0001101001100010011000110011000101000101110000010000 ]
Please input BitImage for x!
[ 1:01111001011:0111001011001110110011100110011100111010111010010000001001 ,
  0:01111001010:0001101001100010011000110011000101000101110000010000 ]
Input was = [-3.22e-16, 1.22e-16] =
[ 1:01111001011:0111001011001110110011100110011100111010111010010000001001 ,
  0:01111001010:0001101001100010011000110011000101000101110000010000 ]
and in hexadecimal representation:
[ 1:3cb:72cece675d209 ,
  0:3ca:1a62633145c10 ]
Please input HexImage for x!
[ 1:3cb:72cece675d209 ,
  0:3ca:1a62633145c10 ]
Hexadecimal input was:
[ 1:3cb:72cece675d209 ,
  0:3ca:1a62633145c10 ]
```

6.3 Template Version Versus Macro Library Version

The following sample program uses the template version of `filib++` together with the C++ vector container from the standard template library (STL) to compute an interval polynomial. The interval data type used is `interval<double>`.

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

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

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


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

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

int main()
{
   filib::fp_traits<double>::setup(); //Do initializations
   interval<double> coeff2(2), coeff1(5), coeff0(2,3);
   vector< interval<double> > pol;
   pol.push_back(coeff2); //[2,2]
   pol.push_back(coeff1); //[5,5]
   pol.push_back(coeff0); //[2,3]
   //horner(pol,x) computes coeff2*x*x + coeff1*x + coeff0
```

```
   interval<double> x(0,1);
   cout << "pol(" << x << ")= " << horner(pol,x) << endl;
   x= interval<double>(-1,0.5);
   cout << "pol(" << x << ")= " << horner(pol,x) << endl;
   return 0;
}

pol([0, 1])= [2, 10]
pol([-1, 0.5])= [-4, 6]
```

Now a very similar program using the macro version of filib++is shown. Note the different header file Interval.h. The macro version only knows the interval data type Interval. The bounds of such intervals are double numbers.

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

#include <Interval.h> //Header file for macro version
#include <vector>     //STL container vector
#include <iostream>

using std::vector;
using std::cout;
using std::endl;


// Evaluation of a polynomial using Horner's rule
Interval horner
(
   //interval polynomial coefficients in STL container vector
   vector<Interval> const& pol,
   //interval argument
   const Interval& x
)
{
   Interval res(0);  //res is initialized with [0,0]
   vector<Interval>::const_iterator p= pol.begin();

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

int main()
{
   Interval coeff2(2), coeff1(5), coeff0(2,3);
   vector< interval<double> > pol;
```

```
   pol.push_back(coeff2); //[2,2]
   pol.push_back(coeff1); //[5,5]
   pol.push_back(coeff0); //[2,3]
   //horner(pol,x) computes coeff2*x*x + coeff1*x + coeff0
   Interval x(0,1);
   cout << "pol(" << x << ")= " << horner(pol,x) << endl;
   x= Interval(-1,0.5);
   cout << "pol(" << x << ")= " << horner(pol,x) << endl;
   return 0;
}
```

The output equals the output of the template version:

```
pol([0, 1])= [2, 10]
pol([-1, 0.5])= [-4, 6]
```

Chapter 7

Application: Verified Computation of all Solutions of a Nonlinear Equation

The following two application programs use containment computations to compute enclosures for all zeros of a one dimensional function over the entire field of real numbers. Functions composed of arithmetical operations and the elementary functions like $\sin, \cos, \log, \ldots$ can be computed using the containment mode over arbitrary finite and infinite (extended) intervals.

7.1 Verified Bisection Method

The following algorithm performs bisection steps until the final enclosures of the zeros are sharp enough. Regions (intervals) which certainly do not contain a zero are eliminated. Regions which still may contain a zero and which are still not sharp enough are stored in a list named `toDo`. The elements of this list are bisected in further steps. Intervals which are sharp enough and for which a range computation does not exclude the value 0 are stored in a second list called `done`. Such intervals possibly contain a zero of the function.

The resulting list `done` is constructed in such a way, that all interval elements are disjoint and ordered with respect to smaller lower bounds. This is done in the function `absorb`. If a candidate is to be stored in the list `done` a check is performed whether this candidate has common points with another candidate already in the list. If this is the case the list element is replaced by the interval hull of both candidates.

The algorithm terminates if the list `toDo` is empty. Than it is sure that all zeros of the function are enclosed by the elements stored in `done`. This result holds even if the function has e. g. infinitely many zeros or if e. g. the search range is the complete real line. Because containment computations are performed it is also allowed that the search range contains regions were the function (considered as a real-valued function with real arguments) is not defined in the usual mathematical sense.

```
//Bisection method to find enclosures of all zeros of a function
```

```
#include <iostream>
#include <list>
#include <interval/interval.hpp> //filib++

using std::cout;
using std::endl;
using std::list; //STL
using std::pair; //STL

//Simplify instantiation of intervals
typedef filib::interval<double> interval;

//Simplify access to methods of class fp_traits<>
typedef filib::fp_traits<interval::value_type> traits;

template<class T> T f(T x)
{
   return sin( T(1)/(x-T(1)) )*log( T(1.5)-abs(x) );
}

template <class I> pair<I,I> bisect(I x)
{
   typename I::value_type midx(mid(x));

   if ( midx == traits::ninfinity() )     //midx == -00?
   {
      midx= -traits::max();
   }
   else if( midx == traits::infinity() )  //midx == +00?
   {
      midx= traits::max();                //Set midx to largest number
   }
   pair<I,I> p;
   p.first = I(x.inf(), midx);
   p.second= I(midx, x.sup());
   return p;
}

template <class I> void absorb(list<I>& done, I x)
{
   list<I>::iterator p=done.begin();
   bool absorbed=false;
   while ( p!=done.end() )
   {
      if ( !disjoint(*p, x) )
      {  //Overlapping intervals
         *p= hull(*p, x);   //hull for filib++
```

```
         absorbed= true;
         break;
      }
      if ( x.inf() < (*p).inf() )
      {  //Sort from left to right
         done.insert(p,x);
         absorbed= true;
         break;
      }
      p++;
   }
   if (!absorbed) done.push_back(x);
   return;  //"done" is a sorted list without overlapping intervals
}

template <class function, class I>
list<I> findAllZeros(function f, const I searchRange, const double epsilon)
{
   int numberOfZeros(0);
   list<I> toDo;
   int toDoMaxSize(10000); //Upper bound for number of elements in toDo
   list<I> done;
   toDo.push_back(searchRange);

   while( ! toDo.empty() )
   {
      I range= toDo.front();
      toDo.pop_front();
      I fRange= f(range);
      if ( in(0., fRange) )
      {
         pair<I,I> p=bisect(range);
         if
         (
               (diam(range)  < epsilon)
            || (diam(fRange) < epsilon)
            || (range==p.first)
            || (range==p.second)
            || (toDo.size()>toDoMaxSize)
         )
         {
            //Interval is sharp enough and may contain a zero:
            if (++numberOfZeros % 1000 == 0) cout << numberOfZeros << endl;
            absorb(done, range);
            continue; //Proceed with next iteration of while-loop
         }
         //Use bisection to get sharper results
         toDo.push_back(p.first);
```

```
         toDo.push_back(p.second);
      }
   }
   cout << "Number of intervals possibly containing a zero:" << endl;
   cout << "  Without absorption: " << numberOfZeros << endl;
   cout << "  With absorption:    " << done.size() << endl;
   return done;
}

template<class T>
std::ostream& operator<< (std::ostream &os, const list<T>& toPrint)
{
   int numberOfElement=1;
   list<T>::const_iterator p=toPrint.begin();
   while (p!=toPrint.end())
   {
      os << "  " << numberOfElement++ << "  " <<  (*p++) << endl;
   }
   return os;
}

int main()
{
   filib::fp_traits<double>::setup(); //Do initializations
   interval::precision(8);            //Precision of interval output
   double epsilon=0;
   cout << "All zeros of sin(1/(x-1))*log(1.5-abs(x))" << endl;

   while (true)
   {
      interval x;
      x= interval::ENTIRE(); //Complete real line
      cout << endl << "Search range: " << x << endl;
      cout << "Epsilon (e.g. 1e-3; <ctr> <c> to terminate): ";
      cin >> epsilon;
      list<interval> zeros= findAllZeros(f<interval>, x, epsilon);
      if (zeros.size() > 0)
      {
         cout << "There may be zeros in the interval(s):" << endl << zeros;
         cout << "Epsilon was " << epsilon << endl;
      }
      else
      {
         cout << "There are definitely no zeros!"<< endl;
      }
   }
   return 0;
}
```

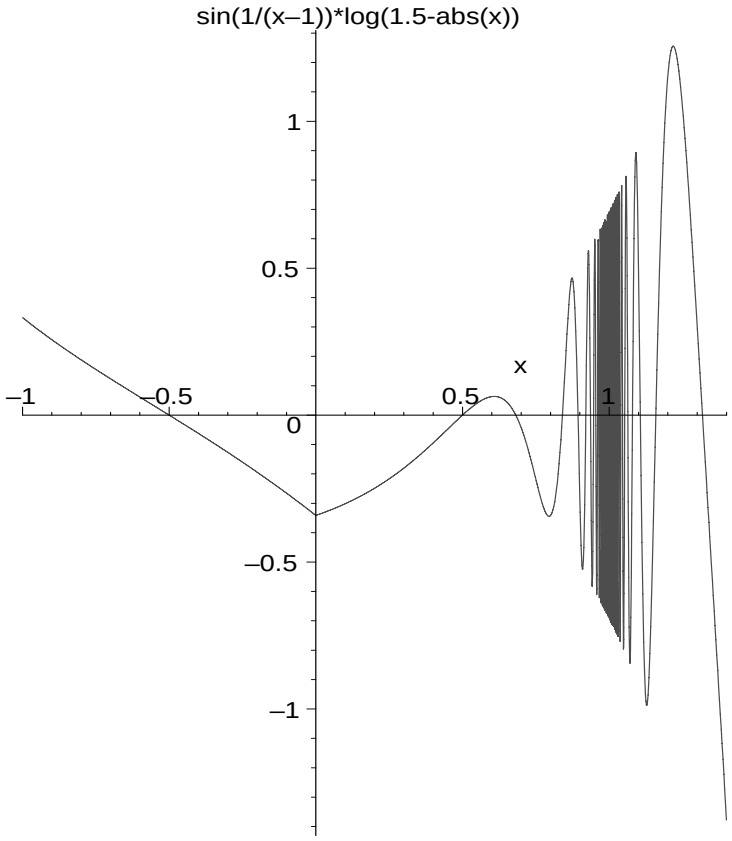


Figure 1 shows the function $\sin(\frac{1}{x-1})\log(1.5 - \mathrm{abs}(x))$. The natural domain of this function considered as a real valued function with a real argument is the union of the two open intervals $(-1.5, 1)$, and $(1, 1.5)$. Near the point 1 there are infinitely many zeros of the function.

Nevertheless our simple bisection method using containment computations finds regions containing all zeros. No exceptions are thrown. The resolution of the different regions becomes better for smaller values of the variable `epsilon` as may be seen by inspecting the output of the program (see below). Even though the natural domain of the considered function is finite the program starts with the entire real line as search region.

For the function shown in the figure and for different values of `epsilon` the program produces the following output:

```
All zeros of sin(1/(x-1))*log(1.5-abs(x))

Search range: [ ENTIRE ]
Epsilon (e.g. 1e-3; <ctr> <c> to terminate): 1e-2
Number of intervals possibly containing a zero:
  Without absorption: 28
  With absorption:    13
There may be zeros in the interval(s):
  1  [-0.515625, -0.484375]
  2  [0.4921875, 0.5078125]
  3  [0.6796875, 0.6875]
  4  [0.8359375, 0.84375]
```

```
  5  [0.890625, 0.8984375]
  6  [0.9140625, 0.921875]
  7  [0.9296875, 0.9375]
  8  [0.9453125, 1.0546875]
  9  [1.0625, 1.0703125]
  10  [1.078125, 1.0859375]
  11  [1.1015625, 1.109375]
  12  [1.15625, 1.1640625]
  13  [1.3125, 1.3203125]
Epsilon was 0.01

Search range: [ ENTIRE ]
Epsilon (e.g. 1e-3; <ctr> <c> to terminate): 1e-4
Number of intervals possibly containing a zero:
  Without absorption: 292
  With absorption:    121
There may be zeros in the interval(s):
  1  [-0.50012207, -0.49987793]
  2  [0.49993896, 0.50006104]
  3  [0.68164062, 0.68170166]
  4  [0.84082031, 0.84088135]
  5  [0.89385986, 0.8939209]
           ...       ...
  117  [1.0636597, 1.0637207]
  118  [1.0795288, 1.0795898]
  119  [1.1060791, 1.1061401]
  120  [1.1591187, 1.1591797]
  121  [1.3182983, 1.3183594]
Epsilon was 0.0001

Search range: [ ENTIRE ]
Epsilon (e.g. 1e-3; <ctr> <c> to terminate): 1e-6
1000
2000
Number of intervals possibly containing a zero:
  Without absorption: 2314
  With absorption:    959
There may be zeros in the interval(s):
  1  [-0.50000095, -0.49999905]
  2  [0.49999905, 0.50000095]
  3  [0.68168926, 0.68169022]
  4  [0.84084415, 0.84084511]
  5  [0.8938961, 0.89389706]
           ...       ...
  955  [1.0636616, 1.0636625]
  956  [1.0795774, 1.0795784]
  957  [1.1061029, 1.1061039]
  958  [1.1591549, 1.1591558]
```

```
  959  [1.3183098, 1.3183107]
Epsilon was 1e-06
```

Some additional remarks:

- For some functions it may be advantageously to use `relDiam` instead of `diam`.
- Functions like $f(x) \equiv 0$ or $f(x) = 1 - \sin^2(x) - \cos^2(x)$ should be used as test functions for a program searching for zeros. Some further stopping criteria may be necessary. Try it!
- The resulting list contains intervals which may contain a zero. It is possible (e.g. due to overestimations in interval function evaluations) that there is no zero in such an interval. It is also possible that such an interval contains several zeros or even a continuum of zeros. Additional checks (sign tests, monotonicity tests) should be performed. Only intervals certainly not containing a zero are eliminated by the algorithm.
- It is not necessary that the function is differentiable.
- Due to containment computations it is not necessary that the function is continuous. There may be singularities in the search region and the search region may be infinite.
- Due to containment computations the natural domain of the function may be rather complicated. The initial search region may or may not be a subset of the natural domain. In any case, roots of the function are never lost!

7.2 Extended Interval Newton Method

In this subsection an alternative method to find all zeros of a univariate function is considered. For this purpose, the following application program implements an extended interval Newton method. Under mild assumptions on the function under consideration this method has for simple zeros locally the quadratic convergence property. In its simplest form the interval Newton Method computes an enclosure of a zero of a real function $f(x)$. It is assumed that the derivative $f'(x)$ is continuous in $[a, b]$, and that

$$0 \notin \{f'(x), x \in [a, b]\}, \text{ and } f(a) \cdot f(b) < 0.$$

If X_n is an inclusion of the zero, then an improved inclusion X_{n+1} may be computed by

$$X_{n+1} := \left(m(X_n) - \frac{f(m(X_n))}{f'(X_n)} \right) \cap X_n,$$

where $m(X)$ is a point within the interval X, usually the midpoint. The mathematical theory of the Interval Newton method appears in [2].

In the extended mode the interval Newton operator returns the entire real axis whenever the range of derivatives (the denominator interval in the Newton operator)

contains 0. In such cases the intersection reproduces the starting interval and no improvement is achieved. Therefore, a bisection step is introduced and the interval Newton operator is applied recursively to both subintervals. Intervals which do certainly not contain a root are eliminated by a range computation with additional check whether the range contains 0.

```
// Interval Newton method using bisection to avoid division by intervals
// containing zero in the interval Newton operator.
// In the extended mode (compiler option -DFILIB_EXTENDED) of filib++
// containment computations are performed. No exceptions are raised
// due to a consistent extension of domains and ranges of
// interval functions.

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

using std::cout;
using std::endl;

//Simplify instantiation of intervals
typedef filib::interval<double> interval;

//Simplify access to methods of class fp_traits<>
typedef filib::fp_traits<interval::value_type> traits;

//Data type for univariate interval functions
typedef interval (*function) (const interval&);

//Function pol
interval pol(const interval& x)
{
  return (x-1.0)*(x+2.0)*(x-3.0);
}

//Derivative dpol of function pol
interval dpol(const interval& x)
{
   return (x+2.0)*(x-3.0)+(x-1.0)*(x-3.0)+(x-1.0)*(x+2.0);
}


//Function f
interval f(const interval& x)
{
   return sin(1.0/x);
}

//Derivative df of function f
```

```
interval df(const interval& x)
{
   return -cos(1.0/x)/sqr(x);
}

//Function g
interval g(const interval& x)
{
   return 0.5 + sin(1.0/x);
}

//Derivative dg of function g
interval dg(const interval& x)
{
   return -cos(1.0/x)/sqr(x);
}

//Function h only partially defined
interval h(const interval & x)
{
   return 1.0-sqrt(x-4.5);
}

//Derivative dh of function h
interval dh(const interval & x)
{
   return -0.5/sqrt(x-4.5);
}

//Function r with multiple zeros
interval r(const interval& x)
{
   return power(x-1.0,3)*power(x-2.0,2)*(x-3.0);
}

//Derivative dr of function r
interval dr(const interval& x)
{
   return 6.0*power(x,5) - 50.0*power(x,4) + 160.0*power(x,3)
                         - 246.0*power(x,2) + 182.0*x - 52.0;
}

static unsigned int ZeroCounter; //Counts verified simple zeros

void inewton(function f, function df, const interval& x)
{
   static double relDiamMax(1e-5);
   if ( !in(0.0, f(x)) ) return; //Definitely no root
```

```
   interval::value_type midx(mid(x));
   if ( midx == traits::ninfinity() )    //midx == -00?
   {
      midx= -traits::max();
   }
   else if( midx == traits::infinity() ) //midx == +00?
   {
      midx= traits::max();               //Set midx to largest number
   }
   interval fmidx(f(interval(midx))), dfx(df(x));
   if ( in(0.0, dfx) //Avoid 0 in denominator interval
        && x!=interval::NEG_INFTY()
        && x!=interval::POS_INFTY()
        && relDiam(x) > relDiamMax ) //Splitting only if x is still too wide
   {  //Split interval
      inewton(f, df, interval(x.inf(), midx));  //Left part
      inewton(f, df, interval(midx, x.sup()));  //Right part
      return;
   }
   interval xNew;
   xNew= (midx - fmidx/dfx).intersect(x); //Interval Newton Operator
   if (xNew==interval::EMPTY()) return;   //Definitely no root
   if (xNew.interior(x)) //Verification ok, one simple root
   {
      cout << "*** Verified "
           << ++ZeroCounter << "  " << xNew << endl;
      return;
   }
   if (xNew==x)  //No further improvement
   {
      cout << "Possibly containing a zero: " << xNew << endl;
      return;
   }
   else //One bound improved, try further improvement to get validation
   {
      inewton(f, df, xNew);
   }
}

int main()
{
   filib::fp_traits<double>::setup(); //Do initializations
   cout << __FILE__ << endl << endl;
   interval::precision(8); //Set output precision to 8

   ZeroCounter=0;          //Counts only simple zeros
   interval searchRange= interval::ENTIRE();
   cout << "All roots of (x-1.0)*(x+2.0)*(x-3.0) in the range "
```

```
        << searchRange << ":" << endl;
   inewton(pol, dpol, searchRange);   //Interval Newton Method

   ZeroCounter=0;
   cout << "All roots of  sin(1/x) in the range "
        << searchRange << ":" << endl;
   inewton(f, df, searchRange); //Interval Newton Method

   ZeroCounter=0;
   cout << "All roots of 1/2 + sin(1/x) in the range "
        << searchRange << ":" << endl;
   inewton(g, dg, searchRange); //Interval Newton Method

   ZeroCounter=0;
   cout << "All roots of 1-sqrt(x-4.5) in the range "
        << searchRange << ":" << endl;
   inewton(h, dh, searchRange); //Interval Newton Method

   ZeroCounter=0;
   cout << "All roots of (x-1)^3*(x-2)^2*(x-3) in the range "
        << searchRange << ":" <<endl;
   inewton(r, dr, searchRange); //Interval Newton Method

   return 0;
}
```

What follows is the runtime output of this program separated by the mathematical formulas of the functions treated as test functions. Note that the search range is in all cases the complete real line i. e. the set $(-\infty, \infty)$.

The first function is the simple polynomial

$$\mathrm{pol}(x) = (x-1)(x+2)(x-3)$$

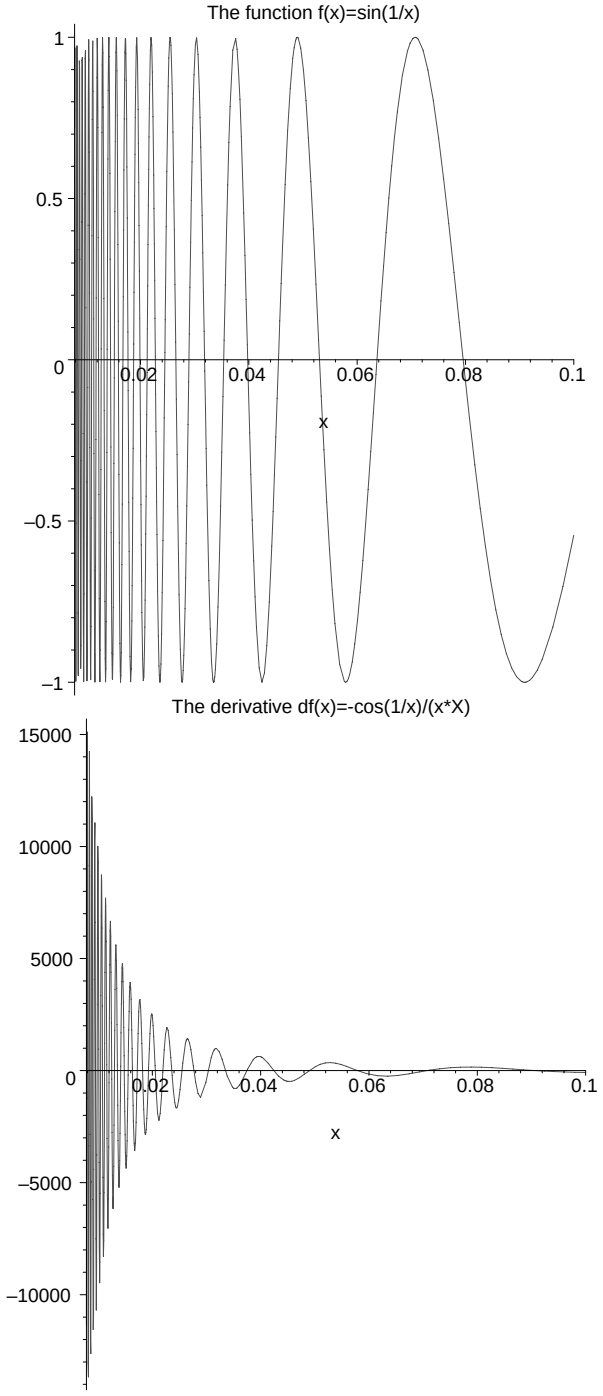
with simple zeros 1, -2, and 3. Running our program yields

```
All roots of (x-1.0)*(x+2.0)*(x-3.0) in the range [ ENTIRE ]:
============================================================
*** Verified 1  [-2, -2]
*** Verified 2  [1, 1]
*** Verified 3  [3, 3]
```

The second test function is

$$f(x) = \sin(1/x)$$

This function has infinitely many zeros near 0 and it tends to 0 for x approaching $-\infty$ or x approaching ∞. Figure 2 shows the function $\sin(\frac{1}{x})$ itself and Figure 3 its derivative $-\cos(\frac{1}{x})/x^2$ in the subrange $[\frac{1}{128}, 0.1]$.



The (drastically shortened) runtime output of our interval Newton program is as follows:

```
All roots of  sin(1/x) in the range [ ENTIRE ]:
================================================
Possibly containing a zero: [ -INFTY ]
*** Verified 1  [-0.33679641, -0.3113457]
*** Verified 2  [-0.15917363, -0.15913636]
*** Verified 3  [-0.10610816, -0.10609895]
*** Verified 4  [-0.079602173, -0.079558769]
```

```
*** Verified 5  [-0.063685996, -0.063649611]
*** Verified 6  [-0.053242997, -0.053026195]
     ...      ...             ...
*** Verified 41720  [-7.6296714e-06, -7.6296414e-06]
*** Verified 41721  [-7.6294899e-06, -7.6294884e-06]
Possibly containing a zero: [-7.6293945e-06, 0]
Possibly containing a zero: [0, 7.6293945e-06]
*** Verified 41722  [7.6294884e-06, 7.6294899e-06]
*** Verified 41723  [7.6296414e-06, 7.6296714e-06]
     ...      ...             ...

*** Verified 83437  [0.053026195, 0.053242997]
*** Verified 83438  [0.063649611, 0.063685996]
*** Verified 83439  [0.079558769, 0.079602173]
*** Verified 83440  [0.10609895, 0.10610816]
*** Verified 83441  [0.15913636, 0.15917363]
*** Verified 83442  [0.3113457, 0.33679641]
Possibly containing a zero: [ +INFTY ]
```

Only the enclosures of the first six zeros, the enclosures near the point 0 and the enclosures of the last six zeros are shown. Indeed, 83442 enclosures of roots are computed and automatically verified by the program. The asymptotic behavior of $f(x)$ leads to the first and the last line shown in the output. The two lines

```
Possibly containing a zero: [-7.6293945e-06, 0]
Possibly containing a zero: [0, 7.6293945e-06]
```

reflect the fact that the function has infinitely many zeros near the point 0.

The test function

$$g(x) = \frac{1}{2} + \sin(1/x)$$

is very similar to the previous function but it tends to $1/2$ for x approaching $-\infty$ or ∞. It produces the output

```
All roots of 1/2 + sin(1/x) in the range [ ENTIRE ]:
=================================================
*** Verified 1  [-1.9292209, -1.8871211]
*** Verified 2  [-0.40066845, -0.37767046]
*** Verified 3  [-0.1472081, -0.1462089]
*** Verified 4  [-0.11235794, -0.11232511]
*** Verified 5  [-0.07649634, -0.076357255]
*** Verified 6  [-0.065858203, -0.065856122]
     ...      ...             ...

*** Verified 41721  [-7.6296417e-06, -7.6296405e-06]
*** Verified 41722  [-7.629519e-06, -7.6295188e-06]
Possibly containing a zero: [-7.6293945e-06, 0]
Possibly containing a zero: [0, 7.6293945e-06]
```

```
*** Verified 41723  [7.6294543e-06, 7.6294593e-06]
*** Verified 41724  [7.6297005e-06, 7.6297324e-06]
     ...      ...             ...

*** Verified 83438  [0.054564438, 0.054569839]
*** Verified 83439  [0.061578931, 0.061635973]
*** Verified 83440  [0.082842379, 0.083672968]
*** Verified 83441  [0.1004532, 0.10057939]
*** Verified 83442  [0.17313413, 0.17499143]
*** Verified 83443  [0.26538306, 0.27550698]
```

The output shows that $g(x)$ does not have zeros smaller than -1.92923 or greater than 0.27551.

The next test function

$$h(x) = 1 - \sqrt{x - 4.5}$$

is only partially defined. Its natural domain is $x \geq 4.5$. The only zero 5.5 is simple. The program produces

```
All roots of 1-sqrt(x-4.5) in the range [ ENTIRE ]:
===================================================
*** Verified 1  [5.5, 5.5]
```

The last test function

$$r(x) = (x - 1)^3(x - 2)^2(x - 3)$$

is a polynomial with multiple zeros at the points 1 and 2 and a simple zero at the point 3. For this function we get

```
All roots of (x-1)^3*(x-2)^2*(x-3) in the range [ ENTIRE ]:
===========================================================
Possibly containing a zero: [1, 1.0000076]
Possibly containing a zero: [2, 2.0000153]
*** Verified 1  [3, 3]
```

Multiple zeros cannot be verified by the implemented method.

Some general remarks:

- Due to the bisection mechanism, the program can also be used to get enclosures of multiple zeros.
- Zeros cannot be lost by the method.
- Further rigorous existence and nonexistence tests should be applied. Such tests may be based on sign changes.
- The program should be improved by an additional stopping criterion to avoid infinite recursions. E. g. a check for a maximum recursion level should be applied or a bound for the maximum number of enclosures to be computed should be given.

- If at a certain state of the algorithm a root is verified (the Newton operator maps the starting interval in its interior) some more iterations may be performed to get a sharper enclosure of the root. The convergence is fast (locally quadratic convergence rate).

- The program is already rather sophisticated with respect to robustness. Nevertheless it is short, easy to understand and it demonstrates powerful extended interval computations.

- As search regions infinite intervals may be used. The extended mode of filib++ allows the (exception free) computation of enclosures of all(!) zeros of a given function considering the complete(!) real line $\mathbb{IR}$.

- Using extended interval operations and infinite intervals allows the propagation of information in such a way that the algorithm may finally give back verified results even if intermediate overflows (again exception free) occur.

- All test functions use an explicit formula for their corresponding derivative function. Of course, automatic differentiation or automatic slope computations may be used (and should be used) instead.

- To get a list or a vector containing all enclosures of zeros the Standard Template Library can be used (as we have done when implementing the verified bisection method in the previous subsection)

Chapter 8

Installation

8.1 Compiler Requirements

A compiler conforming to ISO 14882 (ISO C++) is sufficient, but currently not available. We have used GNU C++ Compiler (version 2.95.2) and KAI C++ Compiler. The code also compiles with version 3 of the GNU C++ Compiler.

Furtheron a unix compatible `make` utility is needed (e.g. GNU make or BSD make), GNU Binutils (version 2.9.5 or better) and a BSD compatible `install` program.

8.2 Installation and Usage

8.2.1 Installation

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

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

or

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

For other compilers the makefile has to be adapted.

The command

```
make libs
```

compiles and builds the library.

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

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

8.2.2 Usage of the Template Library

The source file has to contain the directive

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

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

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

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

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

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

8.2.3 Usage of the Macro Library

The source file has to contain the directive

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

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

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

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

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

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

8.3 Organization of Subdirectories

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

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

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

Bibliography

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

[2] Hammer, R., Hocks, M., Kulisch, U., Ratz, D.: *C++ Toolbox for Verified Computing*. Basic Numerical Problems. Springer-Verlag, Berlin, 1995

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

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

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

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

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

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

[9] libI77: see `http://www.eecs.lehigh.edu/~mschulte/compiler`

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

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

[12] Lerch, M., Tischler, G., Wolff von Gudenberg, J.: `filib++`-Interval Library, Specification and Reference Manual, Techn. Report 279, Informatik, Universität Würzburg, August 2001