### 0.0.1 Interval Arithmetic (*Interval_nt*)

This section describes briefly what interval arithmetic is, its implementation in CGAL, and its possible use by geometric programs. The main reason for having interval arithmetic in CGAL is its integration into the filtered robust and fast predicates scheme, but we also provide a number type so that you can use it separately if you find any use for it (such as interval analysis, or to represent data with tolerance).

The purpose of interval arithmetic is to provide an efficient way to bound the roundoff errors made by floating point computations. You can choose the behaviour of your program depending on these errors; that is what is done for the filtered robust predicates (see Section 0.0.3). You can find more theoretical information on this topic in [**?**].

#### Definition

Interval arithmetic is a large concept and we will only consider here a simple arithmetic based on intervals whose bounds are *double*s. So each variable is an interval representing any value inside the interval. All arithmetic operations (+, -, $*$, /, $\sqrt{}$, *square()*, *min()*, *max()* and *abs()*) on intervals preserve the inclusion. This property can be expressed by the following formula ($x$ and $y$ are reals, $X$ and $Y$ are intervals, $\mathcal{OP}$ is an arithmetic operation):

$$\forall\ x \in X, \forall\ y \in Y, (x\ \mathcal{OP}\ y) \in (X\ \mathcal{OP}\ Y)$$

For example, if the final result of a sequence of arithmetic operations is an interval that does not contain zero, then you can safely determine its sign.

*#include <CGAL/Interval_arithmetic.h>*

#### Creation

*Interval_nt ( double d);* — introduces the interval $[d;d]$.

*Interval_nt ( double i, double s);*

introduces the interval $[i;s]$.

*Interval_nt convert_to<Interval_nt>( NT)*

Several such functions provide a cast from the following numerical types to an *Interval_nt* containing the value. The following types are supported: *leda_real*, *leda_rational*, *leda_integer*, *leda_bigfloat*, *Gmpz*, *Fixed_precision_nt*, all built-in types that fit exactly in a *double* (which excludes *long double* and 64 bits integers, which must be treated separately), and all *Quotient<RT>* where RT is a type listed above. The user can add such functions for his own number types, see the files *CGAL/Interval_arithmetic/IA_*.h* for examples.

**Operations**

All functions required by a class to be considered as a CGAL number type (see **??**) are present, as well as the utility functions, sometimes with a particular semantic which is described below. There are also a few additional functions.

| | | |
|---|---|---|
| *Interval_nt* | *I / J* | returns $[-\infty;+\infty]$ when the denominator contains 0. |
| *Interval_nt* | *sqrt( I)* | returns $[0;\sqrt{upper\_bound(I)}]$ when only the lower bound is negative (expectable case with roundoff errors), and is unspecified when the upper bound also is negative (unexpected case). |
| *double* | *to_double( I)* | returns the middle of the interval, as a double approximation of the interval. |
| *double* | *I.inf()* | returns the lower bound of the interval. |
| *double* | *I.sup()* | returns the upper bound of the interval. |
| *bool* | *I.is_point()* | returns whether both bounds are equal. |
| *bool* | *I.is_same( J)* | returns whether both intervals have the same bounds. |
| *bool* | *I.overlap( J)* | returns whether both intervals have a non empty intersection. |

The two following operators can be used for interval analysis, they are not directly useful for *Interval_nt* as a number type.

| | | |
|---|---|---|
| *Interval_nt* | *I* \|\| *J* | returns the smallest interval containing the two intervals. |
| *Interval_nt* | *I* && *J* | returns the biggest interval contained in the two intervals. The result is unspecified if the two intervals don't intersect. |

The comparison operators ($<$, $>$, $<=$, $>=$, $==$, $!=$, *sign()* and *compare()*) have the following semantic: it is the intuitive one when for all couples of values in both intervals, the comparison is identical (case of non-overlapping intervals). This can be expressed by the following formula ($x$ and $y$ are reals, $X$ and $Y$ are intervals, $\mathcal{OP}$ is a comparison operator):

$$(\forall x \in X, \forall y \in Y, (x\ \mathcal{OP}\ y) = true) \Rightarrow (X\ \mathcal{OP}\ Y) = true$$

and

$$(\forall x \in X, \forall y \in Y, (x\ \mathcal{OP}\ y) = false) \Rightarrow (X\ \mathcal{OP}\ Y) = false$$

Otherwise, the comparison is not safe, and we first increment the counter *Interval_nt_advanced::number_of_failures*, and then throw the exception *Interval_nt_advanced::unsafe_comparison*.

**Implementation**

*Interval_nt* derives from *Interval_nt_advanced*, see Section 0.0.2. Nearly all operations on *Interval_nt* need the rounding mode to be set to 'round to nearest' (which is the default), and leave it in this state if they have to modify it internally.

### 0.0.2 Advanced Interval Arithmetic (*Interval_nt_advanced*)

*Interval_nt* derives from *Interval_nt_advanced*. The advanced class allows you to make faster computations, but you need to set the rounding mode of the FPU to 'round to infinity' (see below for how to do that) before doing any computation with this number type, and each function (arithmetic operators and conversion functions) leaves the rounding mode in this state if it needs to modify it internally. See below for details.

The function *convert_to<Interval_nt>()* is replaced by the function *convert_to<Interval_nt_advanced>()* and has the same requirements concerning the rounding mode as other functions.

*#include <CGAL/Interval_arithmetic.h>*

We provide the following functions to change the rounding mode:

*enum FPU_CW_t { FPU_cw_near, FPU_cw_zero, FPU_cw_up, FPU_cw_down};*

The different selectors for the rounding mode (respectively to the nearest, to zero, to plus infinity, and to minus infinity).

*void* *FPU_set_cw( FPU_CW_t R)*

To set the rounding mode to $R$.

*FPU_CW_t*

*FPU_get_cw( void)* To get the current rounding mode.

*FPU_CW_t*

*FPU_get_and_set_cw( FPU_CW_t R)*

To set the rounding mode to $R$ and get the old one.

**Example**

The correct way to protect an area of code that uses operations on the class *Interval_ nt_ advanced* is the following:

```
FPU_CW_t backup = FPU_get_cw();
FPU_set_cw(FPU_cw_up);
... // The code to be protected.
FPU_set_cw(backup);
```

**Implementation**

The basic idea is to use the directed rounding modes specified by the *IEEE 754* standard, which are implemented by almost all processors nowadays. It states that you have the possibility, concerning the basic floating point operations $(+, -, *, /, \sqrt{})$ to specify the rounding mode of each operation instead of using the default, which is set to 'round to the nearest'. This feature allows us to compute easily on *intervals*. For example, to add the two intervals [a.i;a.s] and [b.i;b.s], compute $c.i = a.i + b.i$ rounded towards minus infinity, and $c.s = a.s + b.s$ rounded towards plus infinity, and the result is the interval [c.i;c.s]. This method can be extended easily to the other operations.

The problem is that we have to change the rounding mode very often, and the functions of the C library doing this operation are slow and not portable. That's why assembly versions are used as often as possible. Another trick is to store the opposite of the lower bound, instead of the lower bound itself, which allows us to never change the rounding mode inside simple operations. Therefore, all basic operations, which are in the class *Interval_ nt_ advanced* assume that the rounding mode is set to 'round to infinity', and everything works with this correctly set. The class *Interval_ nt* takes care of this, but is a bit slower.

So, if the user needs the speed of *Interval_ nt_ advanced*, he must take care of setting the rounding mode to 'round to infinity' before each operations on this number type. And if other operations might be affected by this, he must take care to reset it to 'round to the nearest' before them.

Platform support:
This part of CGAL must be explicitly ported to each non yet supported platform. For Intel, Sparc and Mips, only the GNU compilers use assembly code. The MipsPro compiler uses slower C code instead, and does not support the construction *convert_ to<>()*, but a workaround *convert_from_ to(target, source)*.

### 0.0.3 Robust filtered predicates (*Filtered_ exact<CT, ET, Cache_ t = No_ Filter_ Cache>*)

The class *Filtered_ exact<CT,ET>* is a wrapper type for the number type *CT*, with the difference that all predicates are specialized such that they are guaranteed to be exact. Speed is achieved via a filtering scheme using interval arithmetic (see Section 0.0.2). Here are the necessary requirements:

- *CT* is the construction and storage type. The only data member of the class *Filtered_ exact<CT,ET>* is the *value* of type *CT*. All arithmetic operations called outside the predicates will be executed with this number type. You can disable this feature by compiling with the flag *-DCGAL_DENY_INEXACT_OPERATIONS_ON_FILTER* (it allows the programmer to spot the inexact operations that should be incorporated in the predicates). The arithmetic operations called from the predicates are always computed exactly.

- The *ET* type must be able to compute exactly the operations involved in the predicates called.
- A *convert_to<Interval_nt_advanced>(CT)* function must be provided, that returns an interval containing the value of the argument of type *CT*, see Section 0.0.1.
- A *convert_to<ET>(CT)* function must also be provided, that returns a number of type *ET* representing exactly the argument of type *CT*. It's a conversion function that is used for the exact computation, when the filter fails. This conversion has to be done exactly to ensure robustness.

The following member functions are used to access the numerical value for the different number types:

| | | |
|---|---|---|
| *CT* | *.value()* | returns the wrapped value. |
| *ET* | *.exact()* | returns the converted value to *ET*. |
| *Interval_nt_advanced* | | |
| | *.interval()* | returns the converted value to *Interval_nt_advanced*. |

For efficiency reasons, the conversion of the *value* to an *Interval_nt_advanced* may be cached using *Filter_Cache* as optionnal third template parameter of the class. This is only wishful if *CT* is a complicated type for which the conversion to an interval takes a somewhat long time, and is asked for the same variable several times. You might want to experiment with this feature, depending on your needs.

**Example**

You might use at the beginning of your program a *typedef* as follows:

```
#include<CGAL/Arithmetic_filter.h>
#include<CGAL/leda_real.h>
#include<CGAL/double.h>
typedef Filtered_exact<double, leda_real> NT;
```

Or if you are sure that the predicates involved do not use divisions nor square roots:

```
#include<CGAL/Arithmetic_filter.h>
#include<CGAL/Gmpz.h>
#include<CGAL/int.h>
typedef Filtered_exact<int, Gmpz> NT;
```

And if you know that the double variables contain integer values, you can use:

```
#include<CGAL/Arithmetic_filter.h>
#include<CGAL/Gmpz.h>
#include<CGAL/double.h>
typedef Filtered_exact<double, Gmpz> NT;
```

As a general rule, we advise the use of *Filtered_ exact<double, leda_ real>*.

If you need exact computations also outside the predicates, you can use the following:

```
#include<CGAL/Arithmetic_filter.h>
#include<CGAL/lera_real.h>
typedef Filtered_exact<leda_real, leda_real, Filter_Cache> NT;
```

### Implementation

The template definition of the low level predicates of CGAL are overloaded for the type *Filtered_ exact<CT,ET>*. It is a partial specialisation, which implies that this is not supported by the compilers that do not support this C++ feature. A workaround is in place that allows to use one such filtered number type at once, by explicitely specializing the predicates for it. To do so, just define the macros *CGAL_ IA_ CT* and *CGAL_ IA_ ET* to the corresponding number types before any inclusion of CGAL files, as well as *CGAL_ IA_ CACHE*. For example :

```
#define CGAL_IA_CT double
#define CGAL_IA_ET leda_real
#define CGAL_IA_CACHE No_Filter_Cache
```

For each predicate file, the overloaded code is generated automatically by a *PERL* script (*examples/Interval_ arithmetic/filtered_ predicate_ converter*) that you can use for your own predicates (see *examples/Interval_ arithmetic/README*). This script parses the template declaration of the functions and generates the overloaded code the following way:

- convert the entries to intervals using *convert_ to<Interval_ nt_ advanced>(CT)*, using the *interval()* member function,
- call the original template function with the type *Interval_ nt_ advanced*,
- if no exception is thrown, return the value,
- if an exception is thrown (the filter failed), convert the original entries using *convert_ to<ET>(CT)*, using the *exact()* member function,
- and call the original template function with the type *ET*.

### Example

The low level template predicates of CGAL are in files named *CGAL/predicates_ on_ ftC2.h* (resp. ftC3), the script is used to produce the files *CGAL/Arithmetic_ filter/predicates_ on_ ftC2.h* (resp. ftC3).

For the moment, only the cartesian predicates of the kernel are supported, as well as the power tests used by the regular triangulation.

# Bibliography