0.0.1 Interval Arithmetic (*CGAL_Interval_nt*)

This chapter 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.

The purpose of interval arithmetic is to provide an efficient way to control the roundoff errors made by floating point computations. This control is provided via the comparison operators. When there is an unsafe comparison between intervals, either a warning or an exception (see below) is raised. Otherwise, it means that if you do the same computation with doubles, your program will have the correct output (with the exact same input data).

You can select the behaviour of the algorithm in such cases, when you program filtered predicates for example. You can find more theoretical information on this topic in [BBP98].

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 made on intervals give a result that contains all possible values: the inclusion is preserved. 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

CGAL_Interval_nt (double d);

introduces the interval [d;d].

CGAL_Interval_nt (double i, double s);

introduces the interval [i;s].

Several functions *CGAL_to_Interval_nt()* provide a cast from any numerical type used by CGAL, to an interval *CGAL_Interval_nt*, containing the value. The user can add such functions for his own number types.

Operations

All functions required by a class to be considered a CGAL number type are present, sometimes with a particular semantic, notably :

CGAL_Interval_nt *I / J* returns $[-\infty;+\infty]$ when the denominator contains 0.

CGAL_Interval_nt	*sqrt(I)*	returns [0;sqrt(upper_bound(I))] when only the lower bound is negative (expectable case with round-off errors), and is unspecified when the upper bound is also negative (unexpected case).
double	*CGAL_to_double(I)*	returns the middle of the interval, as a double approximation of the interval.
double	*I.lower_bound()*	returns the lower bound of the interval.
double	*I.upper_bound()*	returns the upper bound of the interval.

The 6 comparison operators have the following semantic : it is the logical expected one when for all couples of values in both intervals, the comparison is identical (case of non-overlapping intervals). Otherwise, the comparison is not safe, and the default behaviour is to emit a warning, unless one of the flags *CGAL_IA_NO_WARNINGS* or *CGAL_NO_WARNINGS* has been set at compile time, the return value is unspecified.

The same remark applies to the utility functions *CGAL_sign()* and *CGAL_compare()*. This is not the case for *CGAL_min()*, *CGAL_max()* and *CGAL_abs()*, which have the logical meaning and no side effect (they return an interval).

Implementation

CGAL_Interval_nt derives from *CGAL_Interval_nt_advanced*, see Section 0.0.2. Nearly all operations on *CGAL_Interval_nt* need the rounding mode to be set towards the nearest, and leave it the same way if they have to modify it internally.

0.0.2 Advanced Interval Arithmetic (*CGAL_Interval_nt_advanced*)

CGAL_Interval_nt derives from *CGAL_Interval_nt_advanced*. It allows you to make faster computations, but you need to set the correct rounding mode of the FPU (towards infinity) before doing any computation with this number type, and each function leaves the rounding mode the same way if if needs to modify it internally. See for details above, and in the test programs.

The other noticable difference is that it throws the exception *CGAL_Interval_nt_advanced::unsafe_comparison* when you make unsafe comparisons. This mechanism is used internally by the filtered predicate scheme described in the next section.

The function *CGAL_to_Interval_nt()* is replaced by *CGAL_to_Interval_nt_advanced()* and has the same requirements concerning the rounding mode as other functions.

#include <CGAL/Interval_arithmetic.h>

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 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 infinite, and $c.s = a.s + b.s$ rounded towards plus infinite, 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 *CGAL_Interval_nt_advanced* assume that the rounding mode is set to positive infinite, and everything works with this correctly set. The class *CGAL_Interval_nt* takes care of this, but is a bit slower.

FPU rounding modes :

We provide the following functions (in C or assembly) to change the rounding mode.

- *CGAL_FPU_set_rounding_to_zero()*
- *CGAL_FPU_set_rounding_to_nearest()*
- *CGAL_FPU_set_rounding_to_infinity()*
- *CGAL_FPU_set_rounding_to_minus_infinity()*

Platform support:

This part of CGAL must be explicitly ported to each non yet supported platform, because of the use of directed rounding modes. For Intel, Sparc and Mips, the GNU compilers only use assembly code. SunPro and MipsPro compilers use the slow C code instead.

0.0.3 Robust filtered predicates (*CGAL_Filtering<CT,ET>*)

This class is a wrapper type for the number type CT, with the difference that all predicates (currently Cartesian 2 dimensional only) are specialized such that they are guaranted to be exact. Speed is achieved via a filtering scheme using interval arithmetic 0.0.1. Here are the necessary requirements :

- CT is the construction and storage type, it is the only member of the class *CGAL_Filtering<CT,ET>*. All arithmetic operations, such has the one involved in intermediate constructions, that are called outside the predicates will be executed with this number type. You can prevent this from hapenning by compiling with the flag -*DCGAL_DENY_INEXACT_OPERATIONS_ON_FILTER*.
- The ET type must do the operations involved in the predicates called exactly.
- A *CGAL_to_Interval_nt_advanced(CT)* function must be provided, that returns an interval containing the value of the argument of type CT. This "cast" function has to be as fast as possible, and exact (in the sense that the interval must contain surely the initial value). This function is provided for the usual types double, Gmpz, leda_real, leda_rational... The user can add his own types. This might become an additional requirement for number types.

- A *CGAL_to_exact_type<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 when the filter fails. This converison has to be exact.

Examples :

You might use at the beginning of your program :

```
#include<CGAL/Filter.h>
#include<CGAL/leda_real.h>
typedef CGAL_Filtering<double, leda_real> NT;
```

And then use *NT* as the number type for all your computations. This is the only combinaison that works at the moment.

```
#include<CGAL/Filter.h>
#include<CGAL/Gmpz.h>
typedef CGAL_Filtering<int, Gmpz> NT;
```

This one might be ok if you are sure that the predicates involved do not use divisions nor square roots.

```
#include<CGAL/Filter.h>
#include<CGAL/Gmpz.h>
typedef CGAL_Filtering<double, Gmpz> NT;
```

This one is false, since there is no exact conversion from a double to a Gmpz, except when the double has in fact an integer value.

Bibliography

[BBP98] H. Brönnimann, C. Burnikel, and S. Pion. Interval arithmetic yields efficient dynamic filters for computational geometry. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 165–174, 1998.