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 a geometric predicate pre-compiler, but we also provide a number type so that the users can use it separately if they find any use for it. The purpose of interval arithmetic is to provide an efficient way to control the errors made by floating point computations.

This type is nothing more than doubles, but you can, when you ask for it, have a control on the computation error. By default, when the flag *CGAL_IA_NO_WARNINGS* is not defined, warnings will be emitted if you try to compare two overlapping intervals. This means that if you do the same computation with doubles, there is a chance that the comparison may return a false result, otherwise the computation with doubles will be safe.

You can select the behaviour of the algorithm in such cases, when you program filtered predicates for example. You can find more 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. For example, if the final result of a sequence of arithmetic operations is an interval that does not contain zero, then you can surely decide the sign of the result.

#include <CGAL/Interval_arithmetic.h>

Creation

CGAL_Interval_nt (double d = 0);

introduces an interval whose bounds are d.

CGAL_Interval_nt (double d, double e);

introduces the interval [d;e].

Several functions *CGAL_to_interval_nt()* provide a cast from any numerical type used by CGAL, to an interval *CGAL_Interval_nt*.

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*

is defined with the following meaning for the degenerate case where the denominator is an interval containing zero: it returns the interval $]-\infty;+\infty[$ (the 'division by zero' exception can be raised, but the default is to ignore it, see next paragraph for details).

CGAL_Interval_nt	*sqrt(I)*	is defined and produces an error if the lower bound of the interval is a negative number.
bool	$I < J$	returns *true* when any value in I is inferior to any value in J.
bool	$I == J$	returns *true* when I and J overlap (or intersect).
bool	$I > J$	is defined logically from the previous ones
bool	$I <= J$	idem
bool	$I >= J$	idem
bool	$I\ != J$	idem
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.

Implementation

CGAL_Interval_nt derives from *CGAL_Interval_nt_advanced*, see Section 0.0.2.

Example

Here a simple example is given showing how to make safe and fast predicates. This kind of problem is going to be resolved automatically by a predicate compiler, but before it is available, you can do it by hand. Let's consider the *CGAL_orientation* predicate:

```
template < class FT >
CGAL_Orientation CGAL_orientation(const FT &px, const FT &py,
                                  const FT &qx, const FT &qy,
                                  const FT &rx, const FT &ry)
{
  FT d1 = (px-qx)*(qy-ry);
  FT d2 = (py-qy)*(qx-rx);
  return  (d1 < d2) ? CGAL_RIGHTTURN
       : ((d1 == d2)? CGAL_COLLINEAR : CGAL_LEFTTURN);
}
```

This function becomes approximately the following, if you want it to be exact :

```
#include <CGAL/Interval_arithmetic.h>
typedef CGAL_Interval_nt Dis;

CGAL_Orientation CGAL_orientation(const FT &px, const FT &py,
                                  const FT &qx, const FT &qy,
                                  const FT &rx, const FT &ry)
{
  Dis d1 = ((Dis)px-(Dis)qx)*((Dis)qy-(Dis)ry);
  Dis d2 = ((Dis)py-(Dis)qy)*((Dis)qx-(Dis)rx);

  if (d1 == d2) {
    return CGAL_orientation((leda_real) px, (leda_real) py,
    (leda_real) qx, (leda_real) qy,
    (leda_real) rx, (leda_real) ry);
  } else {
    return (d1 < d2) ? CGAL_RIGHTTURN : CGAL_LEFTTURN;
  };
}
```

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. See for details above, and in the test programs.

#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 ($+, -, *, /, \surd$) to specify the rounding mode of each operation instead of using the default, which is usually 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 different from one architecture to another. That's why assembly versions are used as often as possible. Another trick is to compute the opposite of the lower bound, instead of the normal lower bound, which allows us to change the rounding mode once less. 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.

Note also that NaNs are not handled, so be careful with that (especially if you 'divide by zero').

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 ported to each non-supported platform, because of the use of directed rounding modes. Here is the current state of supported platforms:

- Sparc
  - g++ uses the assembly versions, independent of the OS (Solaris tested).
  - CC (Solaris) uses the C library functions to change the rounding modes.
- Intel
  - g++ uses the assembly versions, independent of the OS (tested for Linux).
  - For Windows (NT, 95), Solaris, and others the code is not (yet) ported.
- Alpha
  - g++ 2.8 supports the "-mfp-rounding-mode=d" command line option to set the dynamic rounding mode.
  - For CC the situation is unknown (try command line option like "-fprm d").
- Mips
  - g++ and CC use the C library functions.

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.