Chapter 1

Interval Arithmetic

Sylvain Pion

1.1 Introduction

This chapter describes quickly 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 GPC (Geometric Predicate Compiler), but we also provide a class so that the users can use it separately if they find any use for it.

The purpose of interval arithmetic is here to provide an efficient way to control the errors made by floating point computations.

Interval arithmetic is a large concept and we will only consider here a simple arithmetic based on intervals which 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 contain 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.

1.2 CGAL classes

1.2.1 Class CGAL_Interval_nt

#include <CGAL/Interval_arithmetic.h>

Types

The two following classes provide the same functionality (in fact, the first one derives from the second one).

class	*CGAL_Interval_nt;*	Represents the interval number type.
class	*CGAL_Interval_nt_advanced;*	Is faster but requires more work from the user.

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.

Creation

CGAL_Interval_nt I(double d = 0);

introduces an interval which bounds are d.

CGAL_Interval_nt I(double d, double e);

introduces the interval [d;e].

Several functions should provide a cast from any numerical type used by CGAL, to an interval CGAL_Interval_nt.

1.2.2 Class CGAL_Interval_nt_advanced

CGAL_Interval_nt derives from *CGAL_Interval_nt_advanced*, which 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 below for details, and the tests programs.

1.3 Implementation details

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 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]. It is easy to see that this can be done nearly the same way for 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 work with this correctly set. The class *CGAL_Interval_nt* takes care of this, but is a bit slower.

Assertions are going to be implemented to help you using the fastest class with less problems.

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

1.4 FPU rounding modes

We provide C functions to change the rounding mode, these are :

- _FPU_set_rounding_to_zero()
- _FPU_set_rounding_to_nearest()
- _FPU_set_rounding_to_infinity()
- _FPU_set_rounding_to_minus_infinity()

They are assembly based whenever possible.

Those functions also alter the FPU exceptions flags, so we think this should be common for all platforms, and initialized at the beginning of each CGAL program (via the initialization of a static object in the library, for example).

1.5 Platform support state

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 the supported platforms :

- Sparc
  - g++ uses the assembly versions, independant of the OS (Solaris, SunOS, Linux).
  - CC uses the C library functions to change the rounding modes. Works under Solaris (SunOS doesn't use the same files, sorry).
- Intel
  - g++ uses the assembly versions, independant of the OS (Linux tested).
  - I don't know about Windows (NT, 95), nor Solaris or others.

- Alpha
  - g++ 2.7.2 doesn't support (yet, egcs does) the necessary command line option (to set the dynamic rounding mode). There are several possibilities to make it work, but all of them are painful.
  - CC: I don't know, but cc works when you specify the command line option "-fprm d".
- Mips
  - g++ and CC use the C library functions.

1.6 Example

I give here a simple example about how to make safe and fast predicates. This kind of problem is going to be resolved automatically by the GPC, but before it is available, you can do it by hand. Let's consider an easy predicate : *which_side*.

```
int which_side(Point<double> P, Point<double> Q, Point<double> R)
{
  double a = P.x-R.x;
  double b = Q.x-R.x;
  double c = P.y-R.y;
  double d = Q.y-R.y;
  double det = a*d-b*c;

  return (det>0) ? 1 : ( (det<0) ? -1 : 0 );
}
```

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

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

int which_side(Point<double> P, Point<double> Q, Point<double> R)
{
  Dis a = (Dis) P.x - (Dis) R.x;
  Dis b = (Dis) Q.x - (Dis) R.x;
  Dis c = (Dis) P.y - (Dis) R.y;
  Dis d = (Dis) Q.y - (Dis) R.y;
  Dis det = a*d-b*c;

  if (det == 0)
  {
    // Note: if I remember well, this doesn't work, but the idea is here.
   return which_side((Point<LEDA_Real>) P, (Point<LEDA_Real>) Q, (Point<LEDA_Real>) R);
  }
  else return (det>0) ? 1 : -1;
}
```

1.7 Files

The files involved are :

- *CGAL/Interval_arithmetic.h* contains the definition of the two classes.
- *CGAL/_FPU.h* contains the C and assembly definitions for the rounding mode selection.

Note that symbols in _FPU.h are all prefixed by *_FPU_*. Maybe this should be corrected to avoid name clashes.