The Boost interval arithmetic library

Hervé Brönnimann, Guillaume Melquiond, Sylvain Pion

Polytechnic University

ECOLE NORMALE SUPERIEURE DE LYON

Introduction

- Interval computations are a way to extend the usual arithmetic on numbers to intervals on these numbers. Common uses of interval arithmetic are, for example, guaranteed equation solving and error bounding.
- Some libraries already exist but they are specialized in floating-point intervals. We propose a library that accepts any kind of bounds and a wide collection of interval types.
- The computation of determinant signs will serve as an application of the library.

BOOST library

- BOOST is
  - a repository of free peer-reviewed portable C++ libraries (52 libraries, two thousands subscribers),
  - a test bed for future inclusion of libraries in the C++ standard (STL).
- The interval arithmetic library provides a single C++ class template `interval` and supporting functions:
  - $+$, $-$, $\times$, $\div$, $\sqrt{x}$, $|x|$, x^y,
  - trigonometric, hyperbolic, inverse functions,
  - set functions, comparison operators, and so on.

Interval arithmetic

- Intervals:
  - closed and convex sets of a totally ordered field $\mathbb{F}$.
- Representation by a pair of numbers:
  - bounded intervals:
    $[a, b]$ with $a \in \mathbb{F}, b \in \mathbb{F}, a \leq b$,
  - unbounded intervals:
    $[-\infty, b]$, $[a, +\infty]$ and $[-\infty, +\infty]$,
  - empty intervals:
    $[a, b]$ without $a \leq b$.

Operations

- Based on canonical set extensions:
  - the extension G of $g : (\mathbb{F} \times \mathbb{F}) \to \mathbb{F}$ verifies $\forall X \subset \mathbb{F}, \forall Y \subset \mathbb{F}, G(X,Y) = \{g(x,y) \mid x \in X, y \in Y\}$,
  - this approach offers the inclusion property and smallest enclosing intervals.
- Example: basic operations for bounded intervals
  - $[a,b] + [c,d] = [a+c, b+d]$,
  - $[a,b] - [c,d] = [a-d, b-c]$,
  - $[a,b] \times [c,d] = [\min(ac,bc,ad,bd), \max(ac,bc,ad,bd)]$,
  - $[a,b] \div [c,d] = [a,b] \times [1 \div d, 1 \div c]$ if $0 \notin [c,d]$ (division extended by enclosure).

Rounding

- The set of bounds does not have to be $\mathbb{F}$.
  - Example: for real intervals, bounds can be restricted to rational or floating-point numbers.
  - It involves rounded operations on bounds.
- Basic operations:
  - $[a, b] + [c, d] = [a \underline{+} c, b \overline{+} d]$,
  - $[a, b] - [c, d] = [a \underline{-} d, b \overline{-} c]$,
  - etc.

Boost C++ `interval` class

- `interval<T, Policies>`:
  - T is the type of the bounds:
    - the library can handle any type of bounds (for which interval arithmetic is meaningful): integer, rational, floating-point numbers, etc.
  - `Policies` is an optional argument describing properties of $\mathbb{F}$ and of the intervals.
- The policy-based design of the library allows it to handle all the common applications of interval arithmetic.

Policies

- Rounding policy:
  - it provides the arithmetic kernel on bounds:
    - $\overline{+}$, $\underline{\times}$, $\underline{\sqrt{x}}$, $\overline{\cos}$, etc,
    - all the interval functions rely on this kernel,
  - the default kernel of the library handles:
    - any exact type (integers, rationals, etc)
    - floating-point numbers

      (elementary functions of the standard library are not adapted),
  - the policy-based design allows the user to provide its own custom kernel:
    - for example, a MPFR-based policy can be used to compute elementary functions.

Policies

- Checking policy:
  - this policy decides how the `interval` template deals with:
    - empty intervals (handled, forbidden, ignored, etc),
    - unbounded intervals, invalid numbers, etc,
  - the default policy forbids empty intervals, allows unbounded intervals, and ignores invalid numbers.
- Comparison policy:
  - there is no obvious total order on intervals: what is the result of $[1,3] < [2,4]$?
  - this external policy allows to locally compare them,
  - the default policy extends the total order of $\mathbb{F}$ in a partial order.

Example

- Evaluation of the sign of a floating-point polynomial:
  - interval library additional parts,
  - = * + < >: interval operators.

```
int sign_polynomial(double x, double P[], int sz) {
  // Horner's scheme; no empty intervals and standard rounding
  interval<double> y = P[sz - 1];
  for(int i = sz - 2; i >= 0; i--)
    y = y * x + P[i];

  // sign evaluation; comparison operators now follow the "certain" policy
  using namespace compare::certain;
  if (y > 0.)  return 1;
  if (y < 0.)  return -1;
  return 0;
}
```

Arithmetic properties of the library

- Unbounded intervals are correctly handled: $[-1, 0] \times [5, +\infty] = [-\infty, 0]$.
- Division returns the smallest enclosing interval: $[1, 2] \div [0, 1] = [1, +\infty]$.
  Functions can also be used to compute a pair of intervals: $[1, 2] \div [-1, 1] = [-\infty, -1] \cup [1, +\infty]$.
- Empty intervals are handled if allowed by the policy: $([1, 2] \cap [3, 4]) + [5, 6] = \emptyset$.

Efficient floating-point intervals

- Interval arithmetic with hardware floating-point bounds is usually done by using roundings toward $-\infty$ and $+\infty$ as provided by the *IEEE-754* standard.
- However, on many processors, the rounding mode is a global flag, its change breaks the execution flow and slows down interval computations.
- Solution:
  - only use one global rounding mode:
    - $a \underline{+} b$ can be replaced by $-((-a) \overline{-} b)$,
    - $a \underline{\times} b$ by $-(a \overline{\times} (-b))$, etc.

Comparison with other libraries

- Comparison with Profil/BIAS [1], Sun library [2], CGAL interval kernel [3] and MPFI [4].
  Some drawbacks of these libraries:
  - they only deal with floating-point formats,
  - they have a fixed behavior with respect to empty intervals,
  - infix comparison operators, if available, are not usable for any order.

[1] `http://www.ti3.tu-harburg.de/Software/PROFIL.html`

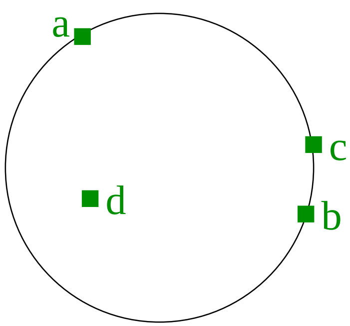
[2] `http://wwws.sun.com/software/sundev/previous/cplusplus/intervals/`

[3] `http://www.cgal.org/`

[4] `http://www.ens-lyon.fr/~nrevol/mpfi.html`

Application of the library

- Sign of a determinant:
  - useful in computational geometry:
    "is a point inside or outside a sphere?"



$$\operatorname{sign}\begin{vmatrix} a_x & a_y & a_x^2 + a_y^2 & 1 \\ b_x & b_y & b_x^2 + b_y^2 & 1 \\ c_x & c_y & c_x^2 + c_y^2 & 1 \\ d_x & d_y & d_x^2 + d_y^2 & 1 \end{vmatrix}$$

- *Interval arithmetic yields efficient dynamic filters for computational geometry*
  by Brönnimann, Burnikel and Pion, 2001.
- A filter will give the exact sign of the determinant or answer it cannot compute it.

Sign of a determinant

- Theoretical method to compute the sign of $\det A$:
  - a LU-decomposition with partial pivoting: $P \cdot A = L \cdot U$,
  - $\mathrm{sign}(\det A) = \det P \times \prod_i \mathrm{sign}(u_{ii})$,

Sign of a determinant

- Theoretical method to compute the sign of $\det A$:
  - a LU-decomposition with partial pivoting: $P \cdot A = L \cdot U$,
  - $\mathrm{sign}(\det A) = \det P \times \prod_i \mathrm{sign}(u_{ii})$,
- Floating-point method:
  - $P \cdot A \approx L' \cdot U'$,
  - the result is not guaranteed since the decomposition is only an approximation.

Sign of a determinant

- Theoretical method to compute the sign of $\det A$:
  - a LU-decomposition with partial pivoting: $P \cdot A = L \cdot U$,
  - $\operatorname{sign}(\det A) = \det P \times \prod_i \operatorname{sign}(u_{ii})$,
- Naïve method with intervals:
  - interval decomposition: $P \cdot A \in [L''] \cdot [U'']$,
  - if no interval $[u''_{ii}]$ contains 0, then the sign of each u_{ii} is known. The sign of $\det A$ is guaranteed.

Sign of a determinant

- Theoretical method to compute the sign of $\det A$:
  - a LU-decomposition with partial pivoting: $P \cdot A = L \cdot U$,
  - $\operatorname{sign}(\det A) = \det P \times \prod_i \operatorname{sign}(u_{ii})$,
- *A posteriori* method:
  - floating-point decomposition: $P \cdot A \approx L' \cdot U'$,
  - compute floating-point matrices $U_{inv} \approx U'^{-1}$ and $L_{inv} \approx L'^{-1}$,
  - evaluation with intervals of $||U_{inv} L_{inv} P A - I||$,
  - if the norm is < 1, the result of the floating-point algorithm is guaranteed.

Complexity and overhead

- In theory, interval computations are two times slower than floating-point computations.
  In practice, the overhead for multiplication and division is rather a factor 3 or 4.
- Time complexity, slowdown and overhead:

	operations		algorithm	multiplication
	number	interval	slowdown	overhead
fbating-point	$n^3/3$	0		
naïve	0	$n^3/3$	$2.6 - 3$	≈ 3.5
a posteriori	n^3	n^3	$8 - 10$	≈ 2

- By careful design, the library can reach the optimal overhead for the *a posteriori* filter.

Some more thoughts

- Spatial complexity:
  - the *a posteriori* method only requires twice the space needed by the floating-point algorithm.
- Dealing with imprecise inputs:
  - the matrix is given by an interval enclosure $[A]$,
  - both methods keep the same overall complexity.
- Small determinants:
  - speed and precision comparison of direct, block, naïve methods for 4×4 determinants.

Conclusion

- We have designed a `C++` interval arithmetic library:
  - its policy-based design allows it to emulate a wide collection of interval types and to handle any kind of bounds,
  - it is as fast as other optimized libraries when it comes to intervals with hardware floating-point bounds,
  - it works on x86, Sparc, PowerPC, and can be easily adapted to other architectures.
- The library is available in the Boost repository: `http://www.boost.org/`

Questions?