Generating a Minimal Interval Arithmetic Based on GNU MPFR

(in the context of the search for hard-to-round cases)

Vincent LEFÈVRE

Arénaire, INRIA Grenoble – Rhône-Alpes / LIP, ENS-Lyon

Dagstuhl Seminar 11371

Uncertainty modeling and analysis with intervals: Foundations, tools, applications

11-16 September 2011

Outline

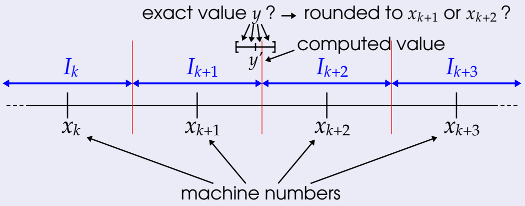
Searching for the Hardest-to-Round Cases

The goal: obtain guaranteed results for the search for the hardest-to-round cases for correct rounding in a fixed precision.

Rounding the elementary functions:

- We want to evaluate and round $y = f(x)$ correctly, i.e. to return $\circ(y)$.
- We know how to compute an approximation y' to y with error bound ε.
- We know how to round y', but do we obtain the same result $\circ(y') = \circ(y)$?

Example in rounding to nearest



Problem known as the *Table Maker's Dilemma* (TMD). Cases needing $\varepsilon < \varepsilon_0$?

Searching for the Hardest-to-Round Cases [2]

Breakpoint numbers: discontinuity points of the rounding functions.

- Directed rounding: the machine numbers.
- Round-to-nearest: the midpoint numbers.

The form of hard-to-round cases in radix 2, n denoting the precision:

- in directed rounding:

$$\overbrace{\underbrace{1.xx\ldots xx}_{n\text{ bits}}0000\ldots00}^{m\text{ bits}}xx\ldots \quad \text{or} \quad \overbrace{\underbrace{1.xx\ldots xx}_{n\text{ bits}}1111\ldots11}^{m\text{ bits}}xx\ldots$$

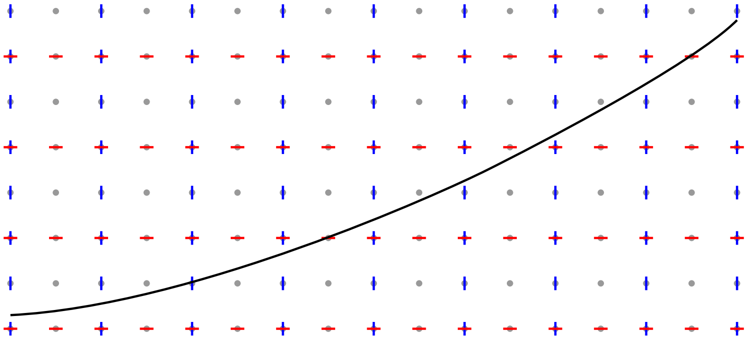
- in round-to-nearest:

$$\overbrace{\underbrace{1.xx\ldots xx}_{n\text{ bits}}1000\ldots00}^{m\text{ bits}}xx\ldots \quad \text{or} \quad \overbrace{\underbrace{1.xx\ldots xx}_{n\text{ bits}}0111\ldots11}^{m\text{ bits}}xx\ldots$$

Searching for the Hardest-to-Round Cases [3]

The problem: find all the machine (or breakpoint to get the results for f^{-1} too) numbers x such that $f(x)$ is very close to a breakpoint number.

Small segment cut: hard-to-round case for f and/or the inverse function f^{-1}.



In binary64, up to 2^{53} inputs per exponent!

Searching for the Hardest-to-Round Cases [4]

Input: a floating-point system (radix, precision), a function f, and an interval I.

Output: hardest-to-round cases of f on I (actually, a superset).

Several steps:

1. Determine an enclosure of the range $f(I)$, more precisely the exponent range (in general, consisting of 1 or 2 consecutive integers), so that fixed point can be used in the last step.
2. Approximate f by a polynomial in a large precision (several words).
3. Fast algorithms applied on the polynomial (hierarchical approximations by polynomial of smaller degrees on subintervals, algorithms in sublinear time complexity) for the search itself. **The slowest step.**
4. Check and filter the potential hard-to-round cases x by computing $f(x)$ in a large precision (2 million cases per exponent $\rightarrow$ a few hundreds).

Here we focus on steps 1-2 (C code generator written in Perl) and 4.
Performance is not crucial, but proving the results is.

The Current (Historical) Solution

Steps 1-2 (“initialization”) and 4 are done in interval arithmetic, by using Maple and the intpakX package. Steps 1-2 take a fraction of second per interval I.

Drawbacks of using Maple + intpakX:

- Maple is commercial, thus not available everywhere.
- On networks, a license server is used. Many practical problems with it.
- The intpakX package assumes that the results of the functions provided by Maple are within some error bound (1 ulp?); however no error bounds are specified by Maple.
- The SIGSTOP/SIGCONT signals (sent by Grid Engine for managing jobs on remote machines) make Maple crash (segmentation fault).
- Separate processes are created, making job control more difficult.
- It seems that Maple tends to increase the load average, so that SGE is more likely to send a SIGSTOP to the job processes when Maple is running.

Now there's GNU MPFR…

GNU MPFR in a Few Words

GNU MPFR features:

- Arbitrary-precision floating-point library in radix 2.
- Each MPFR number (type `mpfr_t`) has its own precision $p \geq 2$.
- Good ideas of the IEEE 754 standard:
  - Correct rounding in 5 rounding modes: to nearest (with the even rounding rule, toward $+$Inf, toward $-$Inf, toward zero, away from zero.
  - Special data: NaN, signed infinities, signed zeros.
  - Exceptions (thread-local flags): the same as in IEEE 754.
  - A few differences: a single NaN, no subnormals (but can be emulated).
- 80 mathematical functions, in addition to utility functions.
- Written in C (ISO + optional extensions + GMP calls), with a conventional C API, but third-party interfaces for other languages (e.g. Perl).
- Free: LGPL licence.

MPFR doesn't track the error bounds (it is a floating-point arithmetic, not an interval arithmetic). Interval arithmetic provided by MPFI, based on MPFR.

Interval Arithmetic in Perl?

The only implementation: MPFI bindings Math::MPFI, rather new (1 year old). Probably not tested very much. Risk of bugs?

Another solution: write a simple implementation that should be easier to check manually or to prove formally. Main ideas:

- Language independent / use code generation.
- Avoid hardcoded functions, by using mathematical properties of the functions: monotonicity, periodicity, local extrema, poles...
- Two transformations: XML/MathML (expression) and function database $\rightarrow$ XML/MathML (interval computations) $\rightarrow$ language-dependant code.

Advantages:

- Code generation $\rightarrow$ closer to the proof, less prone to bugs like typos for particular functions.
- One can control everything (e.g. special cases with no current specification) and get more info (e.g. some form of “monotonicity” decoration).
- Easier to add new primitive functions.

In the Search of the Hardest-to-Round Cases

Our application requires that the tested function be numerically regular (even monotonous, in general) on the tested subdomain, so that there should be no dependency problems and other difficulties.

A typical example (current code in Maple/intpakX syntax)

```
File: def-cos
Function: &cos(x)
T<      1
CTcos   &cos(x0)
CTsin   &sin(x0)
!(i&1)  CTcos &* ((-1)^(i/2)) &/ (i!)
i&1     CTsin &* ((-1)^((i+1)/2)) &/ (i!)
E       (T &intpower (d+1)) &* ((d+2)/((d+1)!*(d+1)))
```

What Could the Generated MathML Contain

- $\cos(x_0)$ and $\sin(x_0)$: for such primitive functions on a floating-point value, there would be an interval constuctor IntervFct, here IntervFct(cos,x_0), that would be mapped to, in C:

  ```
  { int inex = mpfr_cos(c.inf, x0, MPFR_RNDD);
    mpfr_set(c.sup, c.inf, MPFR_RNDN);
    if (inex) mpfr_nextabove(c.sup); }
  ```

  or in Perl:

  ```
  { my $inex = Rmpfr_cos($c{inf}, $x0, MPFR_RNDD);
    Rmpfr_set($c{sup}, $c{inf}, MPFR_RNDN);
    $inex and Rmpfr_nextabove($c{sup}); }
  ```

- For the multiplication of an interval by a positive integer, the MathML code would express:

  $$(y_{\mathrm{inf}}, y_{\mathrm{sup}}) = (\nabla(x_{\mathrm{inf}} \cdot i), \triangle(x_{\mathrm{sup}} \cdot i))$$

  which would map to, in C:

  ```
  mpfr_mul_ui(y.inf, x.inf, i, MPFR_RNDD);
  mpfr_mul_ui(y.sup, x.sup, i, MPFR_RNDU);
  ```

On a Different Subject: The MIRAMAR Project

MIRAMAR: *MIdpoint-RAdius for Multiple-precision ARithmetic*.

A 2-year research/development project for GNU MPFR (and GNU MPC) that should start in October 2011.

$\rightarrow$ To implement an interval arithmetic, where each interval (or ball) is represented by an arbitrary-precision midpoint and a radius (error bound) in low precision.

MPFR already uses something similar internally (approximate results and the error term in a native C integer), though the error term is computed statically when possible. But correct rounding is used for each intermediate operation/function, though in general, this is not necessary. This project should allow us to improve the performance.

INRIA is recruiting an engineer who graduated in 2010 or 2011.