An experimental stochastic lazy floating-point arithmetic

Keith Briggs

Keith.Briggs@bt.com

http://keithbriggs.info

RNC7 LORIA Nancy 2006 July 10-12

mpfs_poster.tex typeset 2006 June 30 11:09 in pdfLaTeX on a linux system

Introduction and aim

- We are concerned here *only* with problems of the type "determine rigorously whether $x < y$ or not", where x and y are computed floating-point numbers. Such a truth value can in principle be decided by a computation which terminates in finite time, assuming x and y are not equal. Problems of this type occur in computational number theory and computational geometry.
- Simple example: straight line through $(x_1, y_1), (x_2, y_2)$; is the point (x_0, y_0) to the left or right of the line? This is determined by $\mathrm{sign}\,((y_1-y_0)(x_2-x_1)-(x_1-x_0)(y_2-y_1))$.
- Related problems include computing the floor of a computed real, and the continued fraction of an irrational (which needs the floor function).

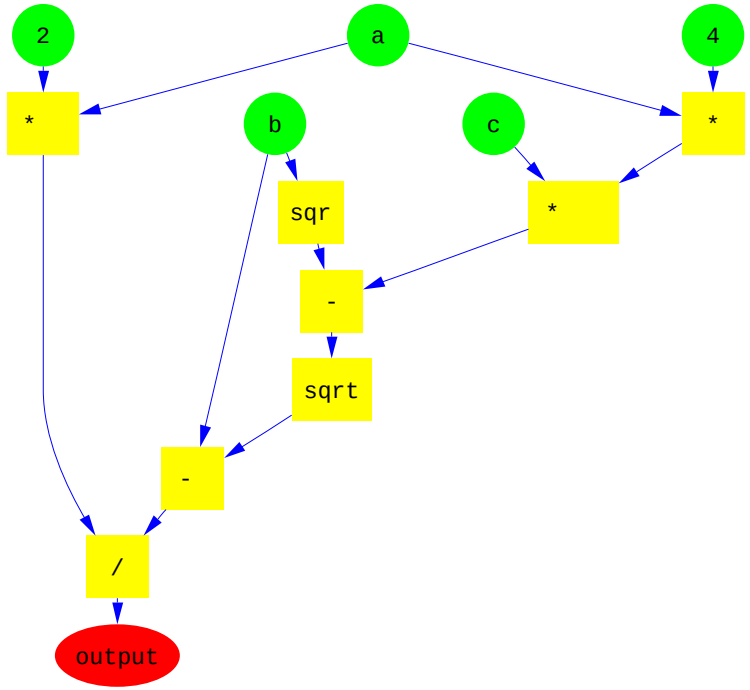
Aim: to produce a software library in C which is easy-to-use and is as efficient as possible

Previous work

- Heuristic (avoiding the real problem): set a precision level for the whole computation. Record the result. Repeat the whole computation at a higher precision. If the result doesn't change, it's probably correct. (maple, mathematica?)
- Exact real arithmetic: there have been many attempts at this:
  - *Essentially, these methods use lazy evaluation where the whole directed acyclic graph (DAG) representing the computation is stored, and intermediate results are re-evaluated as needed. Each step is performed with a bounded error, so the final error is bounded. The main problems in practice concern the storage required to store the DAG, and the fact that the error bounds are typically too pessimistic. In many cases, the minimum possible "graininess" of 1 bit (the smallest possible increase in precision per step) is larger than is actually needed to satisfy error bounds. In large calculations, this can result in unacceptable inefficiencies.*
  - *XR - in C++ (with functional library) and python*
  - *xrc - in C, using scaled-integer representation*
  - *other authors - Boehm, Ménissier-Morain, Harrison, Potts etc.*

Data flow example

★ find the sign of one root $x = \frac{-b+\sqrt{b^2-4ac}}{2a}$ of the quadratic $ax^2+bx+c=0$



- *input nodes don't know how much precision to send*
- *all input nodes send data, even if it eventually may not be needed*
- *to recalculate requires the whole tree to be re-evaluated*

The idea

- The present method is an experimental stochastic variation of exact real arithmetic. The novelty is a way to avoid the 1-bit graininess mentioned above. It is assumed that arbitrary precision floating-point arithmetic with correct rounding is available (such as provided by mpfr).
- The complete DAG is stored.
- The intermediate value at each node is a floating-point approximation x and absolute error bound δ. The interval $[x-\delta, x+\delta]$ always strictly bounds the correct result, and will be refined automatically as necessary.
- General idea: recompute to higher precision - on demand, but transparently.
- Nodes cache the most precise value so far computed.
- The emphasis everywhere is on *practicality*, not theoretical elegance.

Error propagation

- Definitions: exact number $\check{x}$, computed approximation x, error bound $\delta_x \geqslant |x-\check{x}|$, mantissa precision p_x, exponent e_x, unit roundoff $u_x = 2^{e_x-p_x}$ (for $x \neq 0$), $\mathcal{N}$=round-to-nearest.
- $z = \mathcal{N}(x \pm y)$: $\delta_z \leqslant u_z/2 + \delta_x + \delta_y$
- $z = \mathcal{N}(x * y)$: $\delta_z \leqslant u_z/2 + (1+\delta_y 2^{-e_y})|y|\delta_x + (1+\delta_x 2^{-e_x})|x|\delta_y$
- $z = \mathcal{N}(x^{1/2})$: $\delta_z \leqslant u_z/2 + \delta_x/(x^{1/2}(1+(1-2^{1-p_x})^{1/2}))$
- $z = \mathcal{N}(\log(x))$: $\delta_z \leqslant u_z/2 + \delta_x/x$
- and so on - difficult cases include asin near ± 1.
- implemented in an `mpfre` layer: floating point real + error bound.
- typically 32 bits are enough for the error bound.
- tradeoffs are possible here - tighter bounds need more computing.

Evaluation

- When evaluation is initiated, the increase in precision as we move away from the output node is probabilistic, controlled by a parameter α.
  - *The optimum value (i.e. that minimizing computation time) for α will depend on the problem, but the final results are independent of α.*
  - *If $\alpha < 1$, then 1 bit of precision is added with probability α.*
  - *If $\alpha > 1$, then $\lfloor \alpha \rfloor$ bits are added.*
  - *This allows tuning - for example, for the logistic map ($4x(1-x)$) problem, we know that $\alpha = 1$ should be optimal, because the Liapunov exponent is such that one bit is lost on average per iteration.*
- The results of evaluation-initiating functions (i.e. comparison and floor) are guaranteed correct.
- In essence, the software implements an interpreter with evaluate-on-demand semantics. This is hidden from the user.

Heuristics

- The stochastic component is a heuristic. Why might this be reasonable?
- One answer is already given above - error bounds are always pessimistic. Another answer is that computing error bounds itself takes time.
- My mpfs software has a two-level design - the lower level (`mpfre`) computes error bounds, but cannot guarantee to meet a demand for an error bound to be less than a specified value; the upper layer (the only layer with which the user interacts) recomputes on demand to ensure that the truth value of comparisons is computed correctly.
- It is *not* considered a reasonable aim to provide a function to compute a decimal representation of an output to a specified number n of decimal places. However (especially for debugging) a function `mpfs_out_str` is provided, which recomputes until a relative error of about 10^{-n} is achieved.

Example

- Function names follow `mpfr` conventions, except that all outputs are return values. Thus, for $z = x+y$ we write `mpfr_add(z,x,y,GMP_RNDN);` but `z=mpfs_add(x,y);`.

- ```
  /* mpfs "which side of the line?" example. sign((y_1-y_0)(x_2-x_1)-(x_1-x_0)(y_2-y_1)) */

  #include "mpfs.h"

  int main() {
    mpfs_t d,
      x0=mpfs_set_si(...), y0=mpfs_set_si(...), /* test point */
      x1=mpfs_set_si(...), x2=mpfs_set_si(...), /* 1st point on line */
      y1=mpfs_set_si(...), y2=mpfs_set_si(...); /* 2nd point on line */
    d=mpfs_sub(
      mpfs_mul(mpfs_sub(y1,y0),mpfs_sub(x2,x1)),
      mpfs_mul(mpfs_sub(x1,x0),mpfs_sub(y2,y1))
    );
    printf("sign=%d\n",mpfs_cmp_ui(d,0));
    return 0;
  }
  ```

- Tests (`mpfs-0.9`) on a range of specific problems indicate a performance comparable to hand-tuned codes.
  ```