Can we verify/maintain a program if we can't do the maths?

James Davenport (Bath)

Thanks to Russell Bradford (Bath CS), David Wilson (Bath CS), Scott McCallum (Macquarie), Jessica Jones (Bath/Southampton) and Matthew England (Coventry)

SSI workshop March 2016

Traditional Classification of Problems

blunder (of the coding variety) This is the sort of error traditionally addressed in "program verification". Typically independent of the arithmetic.

parallelism Issues of deadlocks or races occurring due to the parallelism of an otherwise correct sequential program. Again, arithmetic-independent.

numerical Do truncation and round-off errors, individually or combined, mean that the program computes approximations to the "true" answers which are out of tolerance.

N.B. Binary program as compiled, not necessarily high-level program as specified.

Traditional Classification of Problems

How often are they considered? Statistics from [CE05]

blunder (83%) (of the coding variety) This is the sort of error traditionally addressed in "program verification". Typically independent of the arithmetic.

parallelism (13%) Issues of deadlocks or races occurring due to the parallelism of an otherwise correct sequential program. Again, arithmetic-independent.

numerical (3%) Do truncation and round-off errors, individually or combined, mean that the program computes approximations to the "true" answers which are out of tolerance.

compilers Typically folklore (ask NAG!): often crop up in convergence tests

Compilers should

produce a program that executes precisely in line with the semantics of the programming language, bearing in mind that the "reals" are floating-point numbers, generally with IEEE semantics (or a variant therof, as in 80-bit internal format).

The semantics of the programming language might or might not be precise: what is `a+b+c` when $a = 1$, $b = 10^{20}$, $c = -10^{20}$?

Many languages specify that `(a+b)+c`=0, but `a+(b+c)`=1.

Compilers do

of course, attempt to produce the most efficient code they can, especially when instructed (−O, −O2, special flags etc.) to do so.

These aims may be mutually incompatible, so what should a good compiler do?

Clearly Only break associativity (etc.) when *explicitly* instructed to do so

But Intel's C compiler regards −O3 as an explicit instruction, GCC's −O3 does not!

Beware of compilers bearing speed-ups!

Traditional Classification of Problems

How often are they considered? Statistics from [CE05]

- **blunder (83%)** (of the coding variety) This is the sort of error traditionally addressed in "program verification". Typically independent of the arithmetic.
- **parallelism (13%)** Issues of deadlocks or races occurring due to the parallelism of an otherwise correct sequential program. Again, arithmetic-independent.
- **numerical (3%)** Do truncation and round-off errors, individually or combined, mean that the program computes approximations to the "true" answers which are out of tolerance.
- **compilers** Typically folklore

To this, I wish to add a fourth kind

What about manual "optimisations"

Or "The bug that dares not speak its name"

manipulation A piece of algebra, which is "obviously correct",

(0%!) turns out not to be correct when interpreted, not as abstract algebra, but as the manipulation of functions $\mathbf{R} \rightarrow \mathbf{R}$ or $\mathbf{C} \rightarrow \mathbf{C}$.

Good $\sqrt{1-z}\sqrt{1+z} \Rightarrow \sqrt{1-z^2}$

Bad $\sqrt{z-1}\sqrt{z+1} \Rightarrow \sqrt{z^2-1}$

Consider $z = -2$: $\sqrt{-3}\sqrt{-1} \not\Rightarrow \sqrt{3}$

Well, of course we all knew that ...

A note on complex numbers

Most of our examples involve complex numbers, and people say

real programs don't use complex numbers

However

- `COMPLEX` in Fortran II (1958–61) was the first programming language data type not corresponding to a machine one
- Even C99 introduced `_Complex`
- Many examples, notably in fluid mechanics.

Kahan's example [Kah87]

Flow in a slotted strip, transformed by

$$w = g(z) := 2\,\mathrm{arccosh}\left(1 + \frac{2z}{3}\right) - \mathrm{arccosh}\left(\frac{5z+12}{3(z+4)}\right) \qquad (1)$$

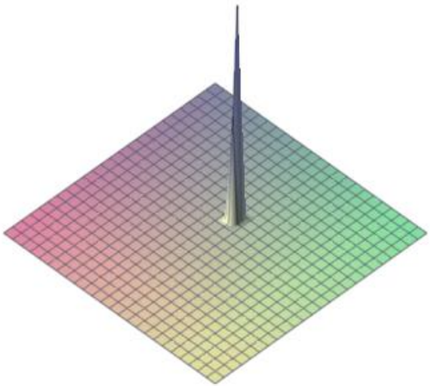
into a more tractable region.
Is this the same transformation as

$$w \stackrel{?}{=} q(z) := 2\,\mathrm{arccosh}\left(2(z+3)\sqrt{\frac{z+3}{27(z+4)}}\right)? \qquad (2)$$

Or possibly

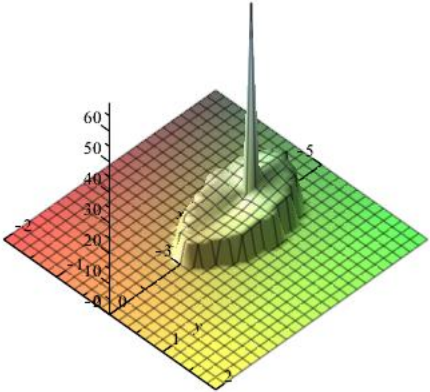
$$w \stackrel{?}{=} h(z) := 2\ \ln\left(\frac{1}{3}\,\frac{\sqrt{3\,z+12}\,\left(\sqrt{z+3}+\sqrt{z}\right)^2}{2\,\sqrt{z+3}+\sqrt{z}}\right)? \qquad (3)$$

$g - q$ might look OK



"OK apart from a slight glitch."

But if we look closer



Definitely not OK

But, in fact $g = h$

Most computer algebra systems (these days!) will refuse to "simplify" g to q

But will also refuse to simplify g to h.

Indeed Maple's `coulditbe(g<>h);` returns `true`, which *ought* to indicate that there is a counter-example.

If $g = h$ then $g - h$ is zero:

$$\begin{aligned}\frac{d(g-h)}{dz} = 2\Bigg(&\sqrt{\frac{z}{z+4}}\sqrt{\frac{z+3}{z+4}}z^{3/2} - 2\,z^{3/2} + 2\,\sqrt{z+3}\sqrt{\frac{z}{z+4}}\cdot\\ &\sqrt{\frac{z+3}{z+4}}z - z\sqrt{z+3} + 4\,\sqrt{\frac{z}{z+4}}\sqrt{\frac{z+3}{z+4}}\sqrt{z} + 8\,\sqrt{z+3}\sqrt{\frac{z}{z+4}}\cdot\\ &\sqrt{\frac{z+3}{z+4}} - 6\,\sqrt{z}\Bigg)\frac{1}{\sqrt{z+3}}\frac{1}{\sqrt{z}}\frac{1}{\sqrt{\frac{z}{z+4}}}\frac{1}{\sqrt{\frac{z+3}{z+4}}}(z+4)^{-2}\left(2\,\sqrt{z+3}+\sqrt{z}\right)^{-1}\end{aligned}$$

and it's a bold person who would say "$= 0$"

Challenges

Challenge (1)

Demonstrate automatically that g and q are not equal, by producing a z at which they give different results.

The technology described in [BBDP07] will isolate the curve $y = \pm\sqrt{\frac{(x+3)^2(-2x-9)}{2x+5}}$ as a potential obstacle (it is the branch cut of q), but the geometry questions are too hard for a fully-automated solution at the moment.

Challenge (2)

Demonstrate automatically that g and h are equal.

Again, the technology in [BBDP07], implemented in a mixture of Maple and QEPCAD, could in principle do this

Why is this so hard? (1) — CAD

The first truly algorithmic approach is over ten years old ([BCD$^+$02], refined in [BBDP07]), and has various difficulties.
At its core is the use of Cylindrical Algebraic Decomposition of $\mathbf{R}^N$ to find the connected components of $\mathbf{C}^{N/2} \setminus \{\mathrm{branch\ cuts}\}$. The complexity of this is doubly exponential in N: upper bound of $d^{O(2^N)}$ and lower bounds of $2^{2^{(N-1)/3}}$.
While better algorithms are in principle known ($d^{O(N\sqrt{N})}$), we do not know of any accessible implementations.
Furthermore, we are clearly limited to small values of N, at which point looking at $O(\ldots)$ complexity is of limited use. We note that the cross-over point between $2^{(N-1)/3}$ and $N\sqrt{N}$ is at $N = 21$.
A more detailed comparison is given in [Hon91]. Hence there is a need for practical research on low-N Cylindrical Algebraic Decomposition.

Why is this so hard? (1) — CAD continued

While the fundamental branch cut of log is simple enough, being $\{z = x + iy | y = 0 \wedge x < 0\}$, actual branch cuts are messier. Part of the branch cut of (2) is

$$2x^3 + 21x^2 + 72x + 2xy^2 + 5y^2 + 81 = 0 \wedge \text{other conditions}, \quad (4)$$

whose solution accounts for the curious boundary of the bad region. While there has been some progress in manipulating such images of half-lines (described in Phisanbut's Bath PhD), there is almost certainly more to be done.

Conclusions/Recommendations

Beware of "Optimisations" (manual or automatic).

- If your code is sensitive to algebraic effects (distributivity, associativity) document the fact!

But how do you know?

Try running with "unsafe" optimisations.

- Document in the Makefile as well as the source

e.g. A separate compilation line for sensitive routines

These days if an algebra system says that an algebraic optimisation is safe $\mathbf{C}^{N/2} \to \mathbf{C}$, it probably is

But currently no good production tools to verify other optimisations, or correctness over $\mathbf{R}$ even when not over $\mathbf{C}$

- J.C. Beaumont, RJB, JHD, and N. Phisanbut.
  Testing Elementary Function Identities Using CAD.
  AAECC, 18:513–543, 2007.
- RJB, R.M. Corless, JHD, D.J. Jeffrey, and S.M. Watt.
  Reasoning about the Elementary Functions of Complex Analysis.
  Ann. Math. Artificial Intelligence, 36:303–318, 2002.
- R. Cousot (Ed.).
  Verification, Model Checking, and Abstract Interpretation.
  Springer Lecture Notes in Computer Science 3385, 2005.
- P. Henrici.
  Applied and Computational Complex Analysis I. Wiley, 1974.
- H. Hong. Comparison of several decision algorithms for the existential theory of the reals. RISC Technical Report 91-41, 1991.
- W. Kahan. Branch Cuts for Complex Elementary Functions.
  Proc. The State of Art in Numerical Analysis, pages 165–211, 1987.
- G.O. Passmore and P.B. Jackson.
  Combined Decision Techniques for the Existential Theory of the Reals. *Proc. Intelligent Computer Mathematics, pages 122–137,*