System Support for Scientific Computation

Notes by David Bindel from a lecture of W. Kahan

Notes from May 07, 2001

1 Testing

Notes on pseudo-multiply/pseudo-divide algorithms and on dealing with branch cuts will be put online. What is not in the notes, and what will take some time to write up, is a description of how to test these algorithms.

We have already talked about a testing philosophy: look at and near singularities, test modules independently, and test interconnections. To test the connections, we can normally only use formulas and then employ a theorem prover. This route is applicable only if the routines are designed with testing in mind. For example, it is a significant challenge to get any symbolic algebra program to verify the proposed stable formulas for the volume of a tetrahedron without a lot of hand-holding. It’s even hard to get a verifier to prove formulas to test integer edge-length volume generators! At this point, theorem provers only really help for corroborating short calculations.

Though there is a canonical form for rational functions, finding that form for a particular function can sometimes be complicated. For instance, expanding

\[ \sum \frac{a_j}{z - b_j} \]

as a ratio of polynomials with no common factors quickly becomes messy.

Expressing a rational function using partial fractions requires factoring the denominator. But even for cubics, the explicit formula for finding the roots invokes transcendental functions; and for polynomials of degree five and higher, Abel’s famous result tells us there is no closed formula for finding the factors.

For most algebraic functions, there is no convenient canonical form. It is possible to compare functions involving finitely many adds, subtractions, multiplies, divides, and square roots, but the cost is exponential in the length of the formula.

The point is that it’s a great deal of work to get some of these proofs, if you can get them at all. Is it worth it?

It is possible to get formal proofs of correctness in some cases, and it is often much less difficult than it used to be. The cost, though, is a great deal of time. And even when you’ve done such a proof, you may have made a mistake in the statement of the problem, or in the implementation. Or you may have given the prover slightly incorrect hypotheses.

An amusing example comes from Oxford in the 80s. In order to prove the correctness of the Inmos T800 implementation of IEEE-compliant floating point, a graduate student translated the IEEE standard into a specification language called Z, and the Inmos circuit design into an implementation language called Occam. With the aid of these formal specifications and a theorem prover, the student came up with a mechanically verifiable proof of correctness of the implementation. This was an impressive accomplishment, particularly at a time when some people criticized the IEEE standard as “unimplementable and untestable.”
Unfortunately, the proof was not bullet-proof. Recall that the IEEE standard specifies five classes of floating point exceptions: invalid operation, overflow, exact \( \infty \) from finite inputs (divide by zero), underflow, and inexact. The treatment of the inexact flag and underflows occasionally irks hardware designers. The Inmos designers went to a lot of effort to make context switching fast, and in their zeal they combined the overflow, underflow, and divide by zero exception bits. Their reasoning was that any of these bits indicates that an error occurred, so why be so picky about figuring out what type of error? The formal specification explicitly described this modified version of exception bit handling instead of the IEEE mandated handling. But division by zero is usually harmless on its own, since if it occurred by mistake and the result is not simply discarded, it will likely be used in an invalid operation soon. And overflow and invalid operation also really are not the same thing.

Another mistake was in the treatment of the sign of zero. In IEEE standard arithmetic, zeros are signed. Multiplication and division work as you’d expect; the sign bit of a product or quotient is the exclusive or of the sign bits of the operands. The expression \( x - x \) gives \(+0\) for all finite \( x \) in the default rounding mode, and \( \pm 0 \) in the directed rounding modes. The expressions \((-0) + (-0)\) and \((-0) - (+0)\) yield \(-0\), and other combinations yield \(+0\). The basic rules that lead to this behavior are

- \( x - y = x + (-y) \)
- positive + positive = positive
- negative + negative = negative

It turns out that these choices are good for dealing with complex branch cuts. A programmer can choose to handle jump discontinuities in a variety of ways: if \( f_l \) and \( f_r \) are the limiting values from the left and right at a jump discontinuity point \( x \), the programmer might choose to return \( f_r \), or \( f_l \), or even \((f_r + f_l)/2\). The last choice is natural in the context of Fourier series, for instance. To a physicist though, a jump discontinuity is like a cut in a piece of paper: it has two edges. An extra bit of information is needed to choose between the edges, and that extra bit of information is in the sign of zero.

Normally, if there is a limit as the argument approaches from the right, and if the formula you use gives an accurate answer no matter how tiny \( x \) is, these rules for handling signed zeros will give \( f(+0) = \) right limit. A similar statement holds for the limit from the left. There are exceptions; for instance, if

\[
\frac{\text{stuff}}{x(1 + x)}
\]

were written as

\[
\frac{\text{stuff}}{x + x^2}
\]

you would probably not get the answer you wanted. This trick may be too clever by half, and it sometimes renders code a little mysterious, but it has turned out to be useful. For example, consider the problem of computing flows around a wing, a problem which can be solved using conformal maps. There are two “critical lines” in this flow, branching from the point above which streamlines flow over the wing and below which streamlines flow under the wing. These critical lines trace the shape of the top and bottom of the wing. There are often two loops to map streamlines above the critical point and below the critical point, and it used to be the case that both critical lines would map to the same place. See also "which note on Kahan’s web page?"
Anyhow, the Inmos T800 didn’t quite get the sign of zero right. Again, this reflected an error in the formal specification. Despite these problems, the Inmos T800 was quite something in its time. But that time has passed.

A second case study involves the infamous Pentium divide bug. Up until the Pentium, the 80x86 chip family used a shift/subtract/test algorithm for division. The test engineers had built up a suite of roughly 2-3 billion test operand pairs to check the implementation for these chips, and thought they were in good shape. The Pentium divider, though, was different from the divider in previous implementations. The Pentium designers used an SRT divider with some little tricks: a carry-save adder, an abbreviated subtracter for quotient digit prediciton, etc.

At the center of the algorithm is a table of quotient digits, which are typically base 4 or so. The table rows are indexed by the leading bits of the partial remainder, and the columns by the leading bits of the divisor. In the lower and upper right corners of the table is a don’t-care region. This process is sufficiently complicated that you would like a proof of correctness. In particular, if the leading bits of the divisor and remainder select any valid cell in the table, the new remainder should once again select an entry in the table, so that if a quotient digit is mis-chosen, things will be fixed up in the next digit.

The designers did indeed carry out the proof, which consisted of solving a collection of inequalities. Then the testing people ran their three billion test pairs without incident. But there was a problem: some of the table entries along the border with the don’t-care region were incorrectly classified as don’t-cares. Because “most” division pairs only use the entries in the center of the table, the problem was not caught, though some quotients were incorrect even in the sixth bit.

Intel did some simulations, and decided that the bug would only occur once in every eight or nine billion randomly chosen operand pairs. Since it was so uncommon, they decided they ought to let it die a natural death. This proved to be a disastrous error from the publicity perspective, though not many people accepted Intel’s eventual offer to replace chips with the division bug. Consequently, Intel ended up with a large number of 90 MHz Pentium chips that never left the warehouse. Matlab made a patch to work around the divide bug, and incorporated a bit of code to report how often the patch made a difference. As it turned out, though, hardly anybody turned the patch on. Despite the furor, few people seemed to really care.

Why was the Pentium bug so problematic? After all, disk drives fail, and CMOS is densely enough packed that it can be scrambled by alpha particles from nearby sources or stray gamma rays. The probabilistic reasoning Intel used is based on a common misconception: everything we know nothing about is equally likely. Such probabilistic arguments about non-probabilistic flaws are not reasonable. In the case of the Pentium bug, a program that did a more careful search found the problem after only a few hundred tests. The program worked by testing divisors near the boundary between columns in the table.

When the Pentiums with the FDIV problem were replaced, the new ones had a bug in the floating-integer-store (FIST) instruction. Other models had other hardware bugs. Given the complexity of modern microprocessors, it would be amazing if occasional bugs did not occur. Intel handled public relations for these later incidents better, though.

What lesson should we take away from the Pentium divide bug? The lesson is not that proofs are pointless. A proof is valuable corroborating evidence of correctness. Tests, too, are valuable corroboration. Efficient testing is a challenge in its own right. Besides devising tests, test designers need to test their tests. Techniques like salting programs with bugs to see if the tests find them are valuable, but not sufficient tests of tests. We would also like to winnow out redundant tests, or tests that check for events that can never happen. For every change of algorithms, there must be a corresponding change of test cases to probe the boundary cases for the new algorithm. None of
this is trivial.

Construction of tests is often more difficult and expensive than the design of the artifact to be tested. However, mistakes are also expensive, and they become rapidly more expensive when it takes a long time to detect and fix them.