System Support for Scientific Computation

Notes by David Bindel from a lecture of W. Kahan

Notes from June 04, 2001

1 Rules of thumb revisited

The rules of thumb presented in previous lectures are deceptively simple to summarize. The implications of these rules for language designers, implementors, and users are far from trivial. Jason and Neil have been workingon summarizing these rules for language designers and for the community at Sun, respectively. Class began with a discussion of some issues which must be considered in a complete document on these rules of thumb.

As an example of some of the issues involved in implementing the rules of thumb, consider the following questions:

- In an ideal world, how would a dynamically typed language like Matlab handle multiple precisions?
- If you were able, would you want to use higher precision for *all* intermediate values?
- What counts as an intermediate value?

The proposed rules state that intermediate calculations should run in the highest precision that is not too slow. But what about objections like those raised by Cleve Moler, who at one time objected to the use of higher precision in routines like Matlab's eigensolver on the basis of loss of reproducability? The newest version of Matlab uses LAPACK with BLAS tuned in a platform-dependent way by ATLAS, so in this particular case the objection has become moot. However, there is still a tradeoff between reproducability of results and the benefits of using an extended precision which is not universally supported in fast hardware.

The rules of thumb explicitly state that intermediate values should be carried with high precision, while input data and final results should be rounded. There must be places where values are rounded back to a narrow precision in order for the user to enjoy the full benefits of extended precision computation of intermediates. This is not an intuitive notion, and it is hard for people to accept that they should throw away figures that they paid to compute.

When should we round? In most computations, there are phases where the values involved describe measurable attributes of objects. In physical situations, those values may correspond to temperatures, chemical concentrations, lengths, and so forth. These are the types of values that should be rounded. But why?

As we have discussed in previous lectures, the benefits of higher precision tend to be most pronounced when a computation comes near to a "pejorative surface", a manifold where there is some singularity. Increasing the precision tends to decrease the volume of data about pejorative surfaces where the computation will go astray. But if the precision of the input data is increased at the same time the intermediate precision is increased, then the decrease in the region where bad behavior occurs will be offset by an increase in the concentration of points in that region.

A surprisingly high proportion of data found in computations in practice tends to lie near some singularity. One reason is that optimal solutions often occur at singularity points. If an optimal solution is determined to double precision and not rounded, the very fact that the solution is optimal (and hence near some singularity) will cause problems for future computations.

Sometimes, computations are performed in higher precision in order to preserve certain relationships. For instance, consider the following formula for interest computations:

$$A = \frac{(1+i)^N - 1}{i} P$$

where A is the accumulation, P is the invested principal, N is the number of payments, and i is the interest rate. Given the accumulation, principal, and number of payments, we would like to compute the interest rate. By inspection, we see that if A increases, i should also increase. But i typically is computed by an iteration, and the result of the iteration depends not only on A, P, and n, but also upon an initial guess, a stopping criterion, and a sequence of rounding errors committed during the iteration. If the data is too close together with respect to the computing precision, the computed i may not be a monotonic function of A. For instance, what if the convergence test kicks you out a bit too soon? Places where the iteration transitions from taking four steps to taking five steps also represent a type of singularity in our algorithm, and must be treated with care.

The iteration used to compute i will be based on some sort of Newton update formula:

$$x_{\text{new}} = x_{\text{old}} - f'(x_{\text{old}})^{-1} f(x_{\text{old}})$$

One possible stopping criterion is that the iteration should run until the iterates are within a few units in the last place of each other, or are even equal in finite precision arithmetic. But then the results tend to be sensitive to random-looking end-figure perturbations. This happens frequently when the number of digits to which x is carried equal the number of digits in which the updates are computed. The end result is that the iteration dithers, and either goes on too long or perhaps stops too soon. The solution: round!

This example is drawn from actual experience. A program Prof Kahan wrote to do interest computations on an HP financial calculator was carefully coded to round at certain points in the computation. Later, a microprogrammer porting the microcode to a later model financial calculator specified that those rounded computations should be done in higher precision. He thought his changes were an improvement, but the modified code ran into convergence problems. It took most of a week for Kahan to diagnose the problem and convince the microprogrammer that, in this case, indiscriminate use of higher precision was more harmful than helpful.

As an exercise for the student: find a continuous function for which computing in one precision will give a same wrong answer, no matter how many digits are carried. A related function gives the *same* wrong answer for almost all data and almost all precisions.

2 Exception handling

Some of this material is in the notes "On the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic."

We already discussed a "menu" of exception handling options. These included default return values and branches to change the flow of control. We also talked about pausing for the debugger, possibly at an approximate stopping point with the aid of compiler-placed milestones. We have not discussed pre-emption: should Ada or similar languages pre-empt IEEE or vice-versa if both

are available? We also have not discussed what to do if certain features are absent; coming up with locutions to handle this situation is a problem.

Many of the schemes in the menu do not (visibly) change the flow of control. This does not mean that the implementation may not use trapping, only that the flow control does not change from the programmer's perspective. For example, suppose we wanted a flag, when raised, to point to the location where it was raised, possibly with an associated stack trace. Implementing that seems expensive: it may require a user-defined trap, for instance. But once the flag is raised, the trap will typically be disabled. There will only be many (potentially expensive) flag raisings if there are many flag lowerings. If the flag was lowered inside a tight loop, the expense could be problematic. But even without the matter of cost, generating log entries for every iteration of a tight loop would likely more irk than help the programmer faced with analyzing the resulting long error log.

Many of the schemes we discussed fall under the rubrik of "pre-substitution." This type of scheme already exists in the 754 standard: for instance, by default dividing a finite number by zero results in an infinity. Invalid exceptions like 0/0 often can be handled using pre-substitution by keeping in mind the *removal of singularities*. For instance, we might pre-substitute 13 for 0/0 when computing

$$\frac{\sin(13x)}{x}.$$

At zero, this function has what a mathematician would call a removable singularity and a programmer might call a nuisance.

An example in the notes on the status of IEEE 754 involves a continued fraction. Typically the inner loop in evaluating a continued fraction has the form

$$f := a_j + x + b_j/f$$

Here putting in ∞ on division by zero works wonderfully. If we augment the code to simultaneously compute the derivative, however, things get more interesting:

$$\begin{aligned} f' &:= 1 + b_j(f')/f^2 \\ f &:= a_j + x + b_j/f \end{aligned}$$

The algorithm in the notes handles this instruction sequence by adding a presubstitution statement:

$$\text{PRESUBST}(\infty/\infty \to \ldots)$$

This is advantageous only if it costs less than a test and branch. Of course, the compiler is allowed to generate a test and branch if that proves advantageous.

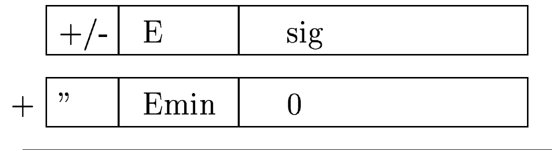
3 Gradual underflow

We now turn to the fast implementation of gradual underflow. At the time the IEEE 754 standard was drafted, many people involved knew how good ways to implement gradual underflow. At the time, though, these people were keeping silent about implementation strategies, bound by nondisclosure agreements. Later, the people who knew how to implement gradual underflow moved away from floating-point work, or retired, or moved. But gradual underflow can be implemented quickly and economically using mechanisms similar to those used in cache-miss handling.

In this scheme, registers will require a few extra bits:

- ?? extra exponent bits
- ?? extra trailing significant bits
- 2 tag bits

To handle underflow, we just make clever use of the floating point adder by adding a funny representation of zero



and then skipping the ordinary post-normalization step. This denormalization does not have to be done immediately; we could wait until performing a store or an add.

I'm unfortunately confused again on my notes about how multiply is handled.