System Support for Scientific Computation

Notes by David Bindel from a lecture of W. Kahan and talk by Joe Darcy

Notes from April 9, 2001

The next 754R meeting is this Wednesday, April 11. Outstanding action items related to the committee include re-drafting a brief statement of purpose for 754R; drawing up rules of thumb for programmers using floating point and compiler writers implementing floating point support; and appropriate bindings of 754R to a language like Java or C99.

1 Guest talk

We opened the lecture with a guest talk on floating point in Java by Joe Darcy to be presented at the Java One conference. The slides will be on the class web page.

During the discussion following Joe’s talk, we touched on a few language issues, including appropriate means of supporting operator overloading, as well as scoping issues. Many of these topics are discussed in some detail in Joe’s thesis, which discusses the design of Borneo, an upwardly compatible extension of the Java language with enhanced support for floating point.

One aspect of this discussion, which lead neatly into Kahan’s lecture plans for the rest of the class, was a discussion of how to support sticky flags and rounding modes. The Standard Apple Numerics Environment (SANE) supported programmer access to flags and modes via the getenv and setenv commands, which get and save the flag and mode bits. The typical flow for a function foo would be

1. Get the current environment at the beginning of the function
2. Set the current environment parameters to those desired in foo
3. Execute the main text of foo
4. Merge the current state into the environment at the beginning of the function and restore that appropriately modified initial environment. Typically this involves restoring the initial modes and merging the status flags.

Unfortunately, this rather unstructured approach to modes and status bits can lead to cluttered code and overly defensive programming practices. Since most functions use the default environment, function calls to set and get the environment on every function entry and exit seem wasteful.

Access to modes and flags in Borneo is more structured. The constructs are similar to the synchronize construct. A typical segment of program text might read

```
routing ... rounding mode desired ... {
... code to be executed in stated rounding mode ...
}
```
When one function calls another, it is the responsibility of the caller to set the modes to what the caller expects. Modes are set dynamically, but scoped statically. Modes can also be set for debugging purposes; for details, see Joe’s thesis.

Bormeio also adds to the function declaration additional specifications in the same spirit as throws. The two extensions are admits and yields; a function using these specifiers might read

```c
double foo()
{
    throws ... exception list ...
    admits ... what flags and modes should be taken from the outside environment ...
    yields ... what flags / modes should be merged into the environment on return ...
}
```

For compatibility, function declarations which do not explicitly include admits and yields clauses default to admits all and yields all.

## 2 Rounding modes and debugging

Joe mentioned that the Alpha does not have both the round up and round down modes, but simulates one of them. This trick goes back to Fred Ris, who observed that while changing rounding modes was often expensive, the same effect could be attained cheaply by strategically negating operands and results.

Rounding on the Alphas is a little strange in general. The rounding mode is determined by two bits in the op codes for floating point instructions. Three options are hardwired; the fourth bit pattern tells the machine to use the rounding mode indicated in a global control register. Unfortunately, the default round-to-nearest mode is one of the three hardwired options. It’s unclear why they did it this way, but we will shortly see why you might want to use the global control register whenever you are using the default behavior.

What are the uses of directed rounding? Some functions are monotonic with respect to rounding errors; a somewhat esoteric trick is to run such functions twice in the round-up and round-down modes in order to get error bounds. This is far from the most common use, though.

Error analysis is horrible. To illustrate the point, we return to the tetrahedron case study. There are two questions posed in that study: how wrong is the answer, and how wrong does it deserve to be? We know digress to treat these questions, particularly the second one.

In general, we want the function \( f(x) \), but what we have is \( F(X) \). We can say that \( F \) is “about as good as the data deserves?” There must be some uncertainty in the input, or what your data deserves is complete accuracy. We could say “about as good as your equipment deserves,” but that way lies even greater sloppiness, since we have already seen that properly implemented machine arithmetic is not limited by the hardware precision, but by the over/underflow thresholds.

By “\( F \) is about as accurate as the data deserves”, we mean that \( F(X) \) is as close to \( f(X) \) as some of the points in the image of the region of uncertainty under \( f \) are. That is, \( F(X) \) is not appreciably worse than we would believe it to be due to uncertainty in the data. The use of the phrase “not appreciably worse” here means that a notion of distance must enter into the picture. Note that this definition does not require that \( F(X) \) actually lie in the region of uncertainty. Often a backward error analysis shows that, in fact, \( F(X) \) does lie in the region of uncertainty. This is not always the case, however. For example, for the problem of matrix inversion, Gauss-Jordan elimination has excellent forward error bounds, but is not backward stable (see Higham chapter 13).

Theorems about whether \( F \) is as good as the data deserves are usually accomplished by backward error analysis. Even for those who know how to do it, backward error analysis is expensive. To see
how expensive, try figuring out how accurately the data determine the volume of a tetrahedron. Don’t expect a snappy answer; but it is possible to get a relative condition number

\[
\frac{|df(x)/dx|}{|f(x)/x|} \bigg|_{x=X}
\]

There is a way to compute the number, which is related to the following theorem

**Theorem 1** Suppose every vertex of the tetrahedron projects perpendicularly onto the opposite face. Then the condition number for the volume is 3. This is true regardless of how needle-shaped the tetrahedron may be.

The proof is left as an exercise to the student.

Even when this theorem holds, the obvious formula can potentially yield garbage. The point is that it takes some thought to do such an analysis, and we should not blame the programmer for failing to do it.

Furthermore, the whole definition of “how accurately the data deserves” is lousy. Who are we to say what the data deserves? This notion is an explanation, not an excuse, for getting a bad answer. In exact arithmetic, functions computed from the same data satisfy certain consistency relationships. But when we compute different functions which are individually “as good as the data deserves,” the overall result may be senseless due to inconsistency. The quality of the answer does not come from a single function, but is a gestalt property. What the programmer actually needs is not functions “as good as the data deserves” but functions “as good as you need for what you want.”

There are alternatives to error analysis perhaps; for example, the programmer could turn to interval arithmetic. But interval arithmetic is overly pessimistic, and not all that useful without variable precision and polymorphic libraries. Another thing you can do is shake up the rounding errors in an attempt to uncover problems in a module which you suspect. Since you may not even have the source code to such a module, about the most you can do is twitch a rounding mode. Contrary to popular belief, computations usually go wrong due to a small number of critical rounding errors, not by the conspiracy of hordes of individually innocuous errors. Consequently, we would like to be able to test code by running it in several rounding modes, in the hope that in one of those modes a critical roundoff will go the other way and hence reveal the bug. In this way the dynamic availability of several rounding modes enormously eases debugging.