Software and Programming for Scientific Computing

Jonathan Goodman

January 25, 1997

Numerical algorithms must be turned into software. To get an accurate, reliable answer quickly is not automatic, even given an excellent basic numerical algorithm. This issue, "software engineering", is more difficult than numerical mathematics because: (i) it relies on judgements, feelings, and an understanding of human strengths and weaknesses, and (ii) it depends on changing software and hardware facilities. It is the subject of a whole course at NYU. These notes contain a few remarks of an experienced amateur (me) intended specifically for someone creating numerical software.

1 Robustness

A program is robust if it behaves well (does the right thing) in difficult conditions. A consequence of Murphy's law is that such difficult conditions will arise in the lifetime of any piece of software. To write robust software, the programmer must constantly ask: "what could possibly go wrong here?" Could the parameters be out of range? Could the algorithm fail to converge? If something has gone irreparably wrong, the program should report this, by printing an error message or returning an error flag, rather than continuing along as though things were fine.

1.1 Numerical Robustness

A code should be designed with inexact computer arithmetic in mind. For example, the code in Figure 1 is almost certainly an infinite loop. Rounding errors make it unlikely that `t` will ever exactly (to the last bit) be equal to `t_final`.

Changing the `for` statement to

```
for( t = 0; t <= t_final; t = t + delta_t) { ...  }
```

will make the loop terminate but, because of rounding errors, it is uncertain whether the loop will take `n` trips or `n+1` trips. The intended loop is coded more reliably in Figure 2.

```
float delta_t, t, t_final;
int n;                    // The desired number of time steps.

delta_t = t_final/n;      // Each time step has size delta_t.

for ( t=0; t != t_final; t + t + delta_t ) { // Step in increments of
      ... ;                                  // delta_t until you reach
      ... ;                                  // t_final.
}
```

Figure 1: A program fragment. The loop would terminate in exact arithmetic but most likely not in floating point arithmetic.

```
float delta_t, t, t_final;
int n, i;
delta_t = t_final/n;
for ( i=0; i < n; i++) {
   t = i*delta_t;
   ...
   ...      }
```

Figure 2: A program fragment. Using exact integer arithmetic to insure the loop is executed exactly `n` times.

Another danger with finite precision arithmetic is error amplification through cancellation. This arises because we usually are interested in relative rather than absolute error. If x is some number that we approximate by $x + \epsilon_x$ (e.g. $x = \sqrt{2}$, $x + \epsilon_x = 1.414$), then the absolute error is ϵ_x, while the relative error is

$$\frac{\epsilon_x}{|x|} \ .$$

Relative error has the advantage of being dimensionless and of corresponding more closely to intuition; a relative error of .02 corresponds to a 2% error.

Unlike absolute error, relative can be greatly amplified by subtracting nearly equal quantities (or adding nearly opposite quantities). Suppose x is approximated by $x + \epsilon_x$ and y by $y + \epsilon_y$ and that $x - y$ is computed from these approximations without adding additional rounding error. The relative error would then be

$$\frac{x + \epsilon_x - (y + \epsilon_y)}{|x - y|} = \frac{\epsilon_x - \epsilon_y}{|x - y|} \ .$$

This could be orders of magnitude larger than $\epsilon_x / |x|$ or $\epsilon_y / |y|$ if x and y are nearly equal. The situation could be expressed as follows (not absolutely correctly): absolute error is scientific and therefore linear and stable; relative error is psychological and therefore nonlinear and subject to unpredictable amplification.

As an example of this "catastrophic cancellation" consider computation of the function $f(h) = (1 - \cos(h))/h^2$ when h is very small. When I did this in double precision with $h = 10^{-6}$, I got an error on the order of 10^{-6}, which is many orders of magnitude larger than double precision rounding error level. When I recomputed $f(h = 10^{-6})$ using the equivalent (in exact arithmetic) formula $f(h) = 2\sin(h/2)^2$, I got an error on the order of 10^{-17}. A code for playing with this is on my web page. Another good example illustrating cancellation and its avoidance is example 2.7 in Kahaner, Moler, and Nash (p. 29).

Catastrophic cancellation may occur whenever we use finite difference approximations to derivatives, especially higher derivatives. It can amplify truncation error as well as roundoff error, but this is much more subtle since often the truncation errors in related quantities are related. That is, $\epsilon_x - \epsilon_y$ could be much less than ϵ_x or ϵ_y (very unlikely for rounding error).

1.2 Conditioning of Computational Problems and Stability of Algorithms

The $f(h)$ example above and the e^x example in Kahaner, Moler, and Nash are examples of *numerically unstable* algorithms for *well conditioned* problems. A problem (such as the problem of computing the value of $f(h)$ for a given h value) is well conditioned if small relative changes in the input lead to small relative changes in the exact answer. Slightly more formally (but still not completely precisely) we could define the *condition number* of a problem to be

$$\kappa = \max \frac{\frac{|\Delta A|}{|A|}}{\frac{|\Delta I|}{|I|}} \ , \tag{1}$$

Where A is the (exact) answer and I is the input(s). The perturbed answer $A + \Delta A$ results from the perturbed inputs $I + \Delta I$. The max is taken over sufficiently but otherwise arbitrary perturbations, ΔI. In other words, the condition number of a problem, for a given input, I, is the worst case relative error amplification factor. Remember that the definition of condition number is strictly mathematical and has nothing to do with computers or computer arithmetic.

The condition number of a problem predicts the greatest possible accuracy in solving it on a computer using floating point arithmetic. The first step in computing A is to represent I in floating point. The rounding errors in doing this will be on the order of $|\epsilon_I| \sim \epsilon_{\text{mach}} |I|$. If we were then to do the rest of the computation exactly, the error predicted by (1) would be on the order of

$$\frac{\epsilon_A}{|A|} >\sim \kappa \epsilon_{\text{mach}} \quad . \tag{2}$$

This is a consequence of Murphy's law, which in this tells us that the ϵ_I produced by rounding errors will be produce errors of the order of the worst possible in (1). For example, a problem with condition number 10^7 should lead to no digits of accuracy if computed in single precision and about 7 correct digits if computed in double precision. Problems with such a large condition number may seem artificial in small toy examples, but they are annoyingly common in large applications.

In the case where the problem is to evaluate a differentiable function, $f(h)$, we can suppose that ΔI (which is Δh in this case) and ΔA (which is Δf) are infinitesimals and use calculus to compute $\kappa = f'(h)/(hf(h))$. For the particular function given above, the condition number at $h = 0$ is $1/6$. The problem of computing $f(h)$ is perfectly well conditioned at $h = 0$.

A numerical algorithm is called *numerically stable* if it produces an error not much larger than (2). This refers to roundoff error only; we would not call the trapezoid rule with 10 points unstable because it fails to give double precision accuracy. The formula $f(h) = 2\sin(h/2)^2$ leads to a numerically stable algorithm for evaluating f near $h = 0$ while the naive direct formula $f(h) = (1 - \cos(h))/h^2$ does not.

When Einstein discovered that gravity was governed by a tensor field instead of a scalar field, he is supposed to have said: "Everything should be as simple as possible, but no simpler." This section can be summarized as follows: "We should make a computation as accurate as possible, but no more accurate."

1.3 Logical Robustness

A program should not fail silently. Instead, it should be filled with error check points. When a procedure returns, it should not just return a value, but it should also return an error code, or at least print an error message if something went wrong. That something could be a parameter out of range, an iterative process that failed to converge, an error tolerance bound that could not be met, a file for output that didn't open, etc.. To write a program like this calls for vigilance bordering on paranoia. It can be taken too far, so that it interferes with the running time or readability of the code, but this is possibility is less likely than the other possibility of too little checking.

1.4 Sermon

Experienced programmers know that 90% of programming time is spent getting the last bug out. The time spent in the coding is far less than the time spent debugging. Most of us have been embarrassed when computed results turned out to be wrong because of a bug or faulty numerics. A practiced and thoughtful programmer can get the right answer with much more ease and confidence than a novice or sloppy programmer. The principles of good programming practice, discussed here and elsewhere, can keep the last bug from being so bad or make it easier to find. Good programming practice may make the initial coding seem more painful, but it ultimately saves the programmer lots of time and embarrassment. Do it!

Here is a list of good programming hints:

- Learn the system. Take the time to read the manuals on the operating system, the programming language, and the compiler and its environment.

- Use a window based debugger. Almost all PC compilers come with window based debuggers, as do gnu compilers. If you have to pay extra for it, do so. Take a few hours learning how to use the debugger. Run through a new code line by line using the debugger just to check it out as a first stage of debugging.

- Program from the top down and from the outside in. This means, decide on the overall program structure (what procedures do what) and on the parameters of procedures before coding the procedures. Decide which procedures know what information. Write the logical structure before putting the numerics in. This is like building a house, first the frame, then the walls, and the kitchen sink last. The hints for the first homework have examples of this philosophy.

- Use graphics in debugging.

- Put in lots of comments and organize the code visually (indentation, blank space) so it is easy to follow. Don't When coding, imagine somebody else trying to figure out what you're doing. That person will be you in half a year.