Title: **When Good Computers Make Bad Calculations: A Cautionary Tale**

Summary: Basic mathematical operations can test the limits of today's powerful computer systems. Insufficiently robust code can crank out nonsensical answers and lack of uniform standards across platforms can produce differing results even with the same code. With financial and other decisions riding on the results of today's analytical applications, it pays more than ever to use good practices in development.

There was a time when computers filled air-conditioned rooms, big programs contained a thousand lines of Fortran code, and big datasets contained a few thousand observations. Researchers and developers mostly wrote their own analytical routines and could often validate the results of their work with a calculator and paper. In just the past few years, the computers on our desks (and laps) have become many times more powerful than those early behemoths. Many of us write high-level programs in desktop environments like MATLAB, Maple, and Mathematica. Many also write source code programs in C/C++, Fortran, Java, and Visual Basic, calling numerical and statistical software components written by others.

These changes have brought tremendous increases in productivity for scientists, engineers, financial analysts and others. Conventional wisdom would suggest that we are finally seeing the fruits of many years of labor in hardware and software design. So it would probably surprise some that there are dark clouds in this otherwise sunny sky.

The root of concern is three-fold: Basic mathematical operations can test the limits of even today's powerful systems; insufficiently robust code can crank out nonsensical answers; and lack of uniform standards across computer platforms can produce differing results even with the same code.

Round-off errors. Consider the problem, for example, of round-off errors. A computer, any computer, can retain only a finite number of significant digits to represent the result of an operation. If a result cannot be expressed exactly, a round-off error is introduced. For a simple example, think of 2 divided by 3. Each time we produce a result that cannot be exactly represented and combine it with other results, we accumulate further errors. For another example, suppose that a part of your application sums a sequence of positive numbers. The numbers may be presented in ascending order of magnitude, descending order, or in arbitrary order. Because finite machine arithmetic is not associative, $(1 + x) + x$ is not necessarily equal to $1 + (x + x)$, if x is sufficiently small. You may get 1 as the answer, or you may not, and the order in which you add things can determine the outcome. With small datasets or simple applications, the effect of the errors induced would hardly be noticeable, but they can lead to suboptimal decisions and significant financial consequences in other cases.

(Un)graceful failures. Another area of potential problems can crop up even when round-off errors are not an issue. Many of the component routines used by developers to add sophisticated functions to applications can be tripped up by data that is outside the conditions anticipated by the writer of the component routine. When this occurs, there are at least three things that can happen: The component routine fails gracefully, providing the user with a message to help diagnose the problem; the routine can fail ungracefully with system-error codes that obscure the real problem; and finally, and perhaps worst, the routine can produce a computed result that is nonsensical and the application proceeds as if the results were correct. If the application is simple and the dataset is modest, the results can be validated by other methods. Increasingly, however, today's sophisticated analytical applications are valuable because they handle complex models with massive datasets.

Low-level trouble. Increasingly, an application is developed on one hardware/operating system platform with the intent that it is run on two or more other platforms in order to reach all of the users/customers.

Analytical routines called by the developer may execute with seemingly minor variations on different platforms due to lower-level code provided by the platform manufacturer's software engineers. This can produce results similar to the first example of rounding errors above. In practical user terms, the same application, using the same data, can produce different results on different platforms.

What should you do?

Be skeptical. The first defense is to adopt a skeptical attitude toward numerical results until you can verify them by independent methods. In our development, we build in error-handling methods to detect and report errors; we develop stringent test strings to find the operational boundaries of our code; we do peer review of new code to gain an independent review for robustness and accuracy; and we frequently talk with developers who use the code in their particular applications on a variety of platforms. You may wish to do the same.

Research the method. Whether you choose to write your own numerical routines, use non commercial ones or incorporate commercial library routines within your application, it is worth your while to do some research on the algorithm used in the code. Some code may be based upon older, out-dated methods that are susceptible to computational errors or methods that may not scale well to larger datasets.

Write the documentation. As basic as it may seem, good documentation is critical to a long-lived analytical application. When problems ultimately arise, good documentation of the individual algorithms and the application are critical to helping the original developer (or one who inherits the application) diagnose and fix problems. The absence of documentation can lengthen the time to repair a problem and result in fixes that ultimately create more problems.

Plan for (and test) multiple platforms. Few applications today only run under a single hardware/operating system platform. The lack of standards for how basic computations are carried out on different platforms means that the same application can produce somewhat different results on different platforms. Good practice requires the development of rigorous test data and testing to validate results on a platform different from the one on which the application was developed. This testing is particularly important even as the application is migrated to newer versions of the operating system or to a newer generation chip.

Summary

Much has changed in computing over the past twenty or more years – dramatic improvements in computing power, evolution of languages, tremendous growth in data and an increase in the complexity of application requirements. The introduction of integrated development environments and other software tools has improved developer productivity but the development of a sophisticated analytical application still requires a major investment of time and money.

To protect this investment against the effects of loss of precision, non-robust behavior, and cross-platform inconsistencies, we advocate healthy skepticism on the part of developers. Whether using internally developed algorithms or those from others, we advocate having a good understanding of the algorithm and its implementation; developing independent code validation methods; carefully documenting code so that it can be correctly fixed when future data exposes weaknesses; and, thoroughly testing cross-platform version for consistency of results.

With the financial and other decisions riding on the results of today's analytical applications, it pays more than ever to use good practices in development.

By Rob Meyer, D.Sc.

Rob Meyer is President of the Numerical Algorithms Group (NAG), a worldwide organization dedicated to developing quality mathematical, statistical and 3D visualization software components. Reach him at rob@nag.com.

Originally published by SD Times, July 2001 (www.sdtimes.com).

Numerical Algorithms Group
www.nag.com
infodesk@nag.com

