Foreword

About Standard Numerics

Part I of this book is mainly for people who perform scientific, statistical, or engineering computations on Apple® computers. The rest is mainly for producers of software, especially of language processors, that people will use on Apple computers to perform computations in those fields and in finance and business too. Moreover, if the first edition was any indication, people who have nothing to do with Apple computers may well buy this book just to learn a little about an arcane subject, floating-point arithmetic on computers, and will wish they had an Apple.

Computer arithmetic has two properties that add to its mystery:

- What you see is often not what you get, and
- What you get is sometimes not what you wanted.

Floating-point arithmetic, the kind computers use for protracted work with approximate data, is intrinsically approximate because the alternative, exact arithmetic, could take longer than most people are willing to wait—perhaps forever. Approximate results are customarily displayed or printed to show only as many of their leading digits as matter instead of all digits; what you see need not be exactly what you've got. To complicate matters, whatever digits you see are *decimal* digits, the kind you saw first in school and the kind used in hand-held calculators. Nowadays almost no computers perform their arithmetic with decimal digits; most of them use *binary*, which is mathematically better than decimal where they differ, but different nonetheless. So, unless you have a small integer, what you see is rarely just what you have.

In the mid-1960's, computer architects discovered shortcuts that made arithmetic run faster at the cost of what they reckoned to be a slight increase in the level of rounding-error; they thought you could not object to slight alterations in the rightmost digits of numbers since you could not see those digits anyway. They had the best intentions, but they accomplished the opposite of what they intended. Computer throughputs were not improved perceptibly by those shortcuts, but a few programs that had previously been trusted unreservedly turned treacherous, failing in mysterious ways on extremely rare occasions.

For instance, a very Important Bunch of Machines launched in 1964 were found to have two anomalies in their double-precision arithmetic (though not in single): First, multiplying a number Z by 1.0 would lop off Z's last digit. Second, the difference between two nearly equal numbers, whose digits mostly canceled, could be computed wrong by a factor almost as big as 16 instead of being computed exactly as is normal. The anomalies introduced a kind of noise in the feedback loops by which some programs had compensated for their own rounding errors, so those programs lost their high accuracies. These anomalies were not "bugs"; they were "features" designed into the arithmetic by designers who thought nobody would care. Customers did care; the arithmetic was redesigned and repairs were retrofitted in 1967.

Not all Capriciously Designed Computer arithmetics have been repaired. One family of computers has enjoyed notoriety for two decades by allowing programs to generate tiny "partially underflowed" numbers. When one of these creatures turns up as the value of T in an otherwise innocuous statement like

```
if T = 0.0 then Q := 0.0 else Q := 702345.6/(T + 0.00189/T);
```

it causes the computer to stop execution and emit a message alleging "Division by Zero." The machine's schizophrenic attitude toward zero comes about because the test for `T = 0.0` is carried out by the adder, which examines at least 13 of T's leading bits, whereas the divider and multiplier examine only 12 to recognize zero. Doing so saved less than a dollar's worth of transistors and maybe a picosecond of time, but at the cost of some disagreement about whether a very tiny number T is zero or not. Fortunately, the divider agrees with the multiplier about whether T is zero, so programmers could prevent spurious divisions by zero by slightly altering the foregoing statement as follows:

```
if 1.0 * T = 0.0 then Q := 0.0 else Q := 702345.6/(T + 0.00189/T);
```

Unfortunately, the Same Computer designer responsible for "partial underflow" designed another machine that can generate "partially overflowed" numbers T for which this statement malfunctions. On that machine, Q would be computed unexceptionably except that the product `1.0 * T` causes the machine to stop and emit a message alleging "Overflow." How should a programmer rewrite that innocuous statement so that it will work correctly on both machines? We should be thankful that such a task is not encountered every day.

Anomalies related to roundoff are extremely difficult to diagnose. For instance, the machine on which `1.0 * T` can overflow also divides in a peculiar way that causes quotients like `240.0/80.0`, which ought to produce small integers, sometimes to produce nonintegers instead, sometimes slightly too big, sometimes slightly too small. The same machine multiplies in a peculiar way, and it subtracts in a peculiar way that can get the difference wrong by almost a factor of 2 when it ought to be exact because of cancellation.

Another peculiar kind of subtraction, but different, afflicts the machines that are schizophrenic about zero. Sets of three values *X*, *Y*, and *Z* abound for which the statement

```
if (X = Y) and ((X - Z) > (Y - Z)) then writeln('Strange!');
```

will print "Strange!" on those machines. And many machines will print "Strange!" for unlucky values *X* and *Y* in the statement

```
if (X - Y = 0.0) and (X > Y) then writeln('Strange!');
```

because of underflow.

These strange things cannot happen on current Apple computers.

I do not wish to suggest that all but Apple computers have had quirky arithmetics. A few other computer companies, some Highly Prestigious, have Demonstrated Exemplary Concern for arithmetic integrity over many years. Had their concern been shared more widely, numerical computation would now be easier to understand. Instead, because so many computers in the 1960's and 1970's possessed so many different arithmetic anomalies, computational lore has become encumbered with a vast body of superstition purporting to cope with them. One such superstitious rule is "*Never* ask whether floating-point numbers are exactly equal."

Presumably the reasonable thing to do instead is to ask whether the numbers differ by less than some tolerance; and this *is* truly reasonable provided you know what tolerance to choose. But the word *never* is what turns the rule from reasonable into mere superstition. Even if every floating-point comparison in your program involved a tolerance, you would wish to predict which path execution would follow from various input data, and whether the different comparisons were mutually consistent. For instance, the predicates `X < Y - TOL` and `Y - TOL > X` seem equivalent to the naked eye, but computers exist (*not* made by Apple!) on which one can be true and the other false for certain values of the variables. To ask "Which?" violates the superstitious rule.

There have been several attempts to avoid superstition by devising mathematical rules called *axioms* that would be valid for all commercially significant computers and from which a programmer might hope to be able to deduce whether his program will function correctly on all those computers. Unfortunately, such attempts cannot succeed without failing! The paradox arises because any such rules, to be valid universally, have to encompass so wide a range of anomalies as to constitute the specifications for a hypothetical computer far worse arithmetically than any ever actually built. In consequence, many computations provably impossible on that hypothetical computer would be quite feasible on almost every actual computer. For instance, the axioms must imply limits to the accuracy with which differential equations can be solved, integrals evaluated, infinite series summed, and areas of triangles calculated; but these limits are routinely surpassed nowadays by programs that run on most commercially significant computers, although some computers may require programs that are so special that they would be useless on any other machine.

Arithmetic anarchy is where we seemed headed until a decade ago when work began upon IEEE Standard 754 for binary floating-point arithmetic. Apple's mathematicians and engineers helped from the very beginning. The resulting family of coherent designs for computer arithmetic has been adopted more widely, and by more computer manufacturers, than any other single design. Besides the undoubted benefits that flow from any standard, the principal benefit derived from the IEEE standard in particular is this:

Program importability: Almost any application of floating-point arithmetic, designed to work on a few different families of computers in existence before the IEEE Standard and programmed in a higher-level language, will, after recompilation, work at least about as well on an Apple computer or on any other machine that conforms to IEEE Standard 754 as on any nonconforming computer with comparable capacity (memory, speed, and word size).

The Standard Apple Numerics Environment (SANE) is the most thorough implementation of IEEE Standard 754 to date. The fanatical attention to detail that permeates SANE's implementation largely relieves Apple computer users from having to know any more about those details than they like. If you come to an Apple computer from some other computer that you were fond of, you will find the Apple computer's arithmetic at least about as good, and quite likely rather better. An Apple computer can be set up to mimic the worthwhile characteristics of almost any reasonable past computer arithmetic, so existing libraries of numerical software do not have to be discarded if they can be recompiled. SANE also offers features that are unique to the IEEE Standard, new capabilities that previous generations of computer users could only yearn for; but to learn what they are, you will have to read this book.

As one of the designers of IEEE Standard 754, I can only stand in awe of the efforts that Apple has expended to implement that standard faithfully both in hardware and in software, including language processors, so that users of Apple computers will actually reap tangible benefits from the Standard. And I thank Apple for letting me explain in this foreword why we needed that standard.

Professor W. Kahan
Mathematics Department and
Electrical Engineering and
Computer Science Department
University of California at Berkeley
December 16, 1987