# programming pearls

*by Jon Bentley*

## BIRTH OF A CRUNCHER

Its practitioners give it the glorified name of "numerical analysis," but for most programmers the field of number crunching is a lot like plumbing. We use it often, but we don't think about how it works until something goes wrong.

I once held that Neanderthal view. I was cured by a fine course in numerical analysis, which showed me the elegance of the field. My appraisal of the subject changed from "ugly and useless" to "beautiful and useless." I have good numerical routines available in libraries; why would I ever have to make my own?

I was recently delighted to discover that even for a layman like me, numerical analysis can be very useful. This column tells how I used some elementary techniques to write a simple numerical routine. I replaced a library function with a version specialized to the problem at hand; the code grew from five lines to a dozen, but the routine was three times faster and it made a big program run twice as fast.

### The Problem

I was working on a program to compute traveling salesman tours through point sets. Profiling the thousand-line program showed that about eighty percent of the time was spent in a five-line routine to compute distances. The specification called for the Euclidean distance between points in $K$-dimensional space. For instance, the distance between the three-dimensional points $(a_1, a_2, a_3)$ and $(b_1, b_2, b_3)$ is

$$\sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + (a_3 - b_3)^2}$$

Program 1 computes the distance between the points represented by the vectors $A[1..K]$ and $B[1..K]$.

```
Sum := 0.0
for J := 1 to K do
    T := A[J] - B[J]
    Sum := Sum + T*T
return sqrt(Sum)
```

**PROGRAM 1. A Simple Distance Routine**

Program 1 has the advantage of simplicity: it is easy to understand. Unfortunately, it has several disadvantages as well. It may, for instance, generate an arithmetic overflow even if all inputs, intermediate differences, and the output are in a valid range. Suppose the machine can represent floating point numbers up to $10^{30}$ and consider computing the distance between (0, 0) and $(3 \times 10^{20}, 4 \times 10^{20})$, which is $5 \times 10^{20}$. Squaring the difference $0–3 \times 10^{20}$ yields $9 \times 10^{40}$ and an overflow. This problem (and a similar problem with underflow) were not important in the program at hand; the context ensured that differences were neither extremely large nor extremely small.

A second problem with Program 1 is that it is very expensive, at least as implemented in C on a VAX 11/750®. The March 1986 column sketched the performance of that hardware/software system: arithmetic operations range in cost from 3.3 microseconds for integer addition to 15.7 microseconds for floating-point division. When $K = 2$, Program 1 requires a whopping 1140 microseconds to compute the Euclidean distance between a pair of points in the plane. Straightforward experiments showed that the lion's share of that time goes to computing the square root, which requires about 940 microseconds.

My goal for the program was to provide a faster distance routine. A method that works in many applications is simply to remove the `sqrt` from Program 1; if distances are only compared, then the monotonicity of `sqrt` makes it superfluous (the February 1984 column describes such an application). That wouldn't work on this job; I therefore sought a $K$-dimensional Euclidean distance routine with the following attributes.

Domain: $K$ is in the range 1..16 (but typically 2, 3, or 4); the coordinates of points are in single-precision.

Accuracy: The single-precision output should be accurate to the last decimal digit, or a relative accuracy of about 1.0e-7.



VAX is a trademark of Digital Equipment Corporation.

Robustness: The inputs may be assumed to be well behaved; overflow and underflow are not major concerns.

Performance: The routine should be as fast as possible.

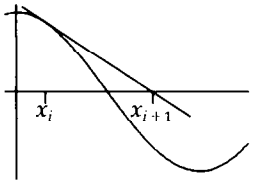
The rest of this column will focus on a routine with these characteristics; Problem 17 describes an accurate and robust method that is somewhat slower.

### Newton Iteration

Numerical analysts have developed many techniques for finding a *zero* (or a *root*) of a function. Given a function $f(x)$, a zero is a real number $r$ such that $f(r) = 0$. To compute $\sqrt{a}$ we can find a zero of $f(x) = x^2 - a$; if $r^2 - a = 0$, then $r = \sqrt{a}$. Thus if we can find zeroes we can compute square roots.

So how do we find the zero of a function? We could use our old friend, binary search. If $a \geq 1$, then $\sqrt{a}$ is in the range $[1, a]$. We can successively halve that range until we get a good approximation to $\sqrt{a}$. If $a = 4$, for instance, we will examine the ranges [1, 4], [1, 2.5], [1, 1.75], [1.375, 1.75], . . . . Numerical analysts call this the *bisection* method; each step yields one additional bit of accuracy in the answer.

A superior scheme was invented by Isaac Newton, the famous English computer scientist who also dabbled in mathematics and physics. His method does not compute a range explicitly, but rather starts with an initial guess $x_0$ and generates a sequence of approximations $x_1, x_2, x_3, \ldots$. To generate $x_{i+1}$ we must know both $f(x_i)$ and its derivative $f'(x_i)$. We then proceed down the tangent line until it crosses the $x$-axis:




Intuitively, we are approximating the function locally by a straight line with equal $y$-value and slope. Mathematically, we compute the next iterate by

$$x_{i+1} = x_i - f(x_i)/f'(x_i)$$

To use Newton iteration we must be able to compute both the function and its derivative.

To find $\sqrt{a}$ we will find the zero of $f(x) = x^2 - a$, so $f'(x) = 2x$. Newton's iteration formula is then

$$\begin{aligned} x_{i+1} &= x_i - (x_i^2 - a)/2x_i \\ &= x_i - x_i/2 + a/2x_i \\ &= (x_i + a/x_i)/2 \end{aligned}$$

For an intuitive appreciation of why the formula works, observe that if $x_i$ is too small then $a/x_i$ is too big; the average of the two is a better estimate. (Schoolchildren call this the "divide and average" technique.) Thus once we reach the final answer, we don't move away: if $x_i = \sqrt{a}$, then

$$x_{i+1} = (\sqrt{a} + a/\sqrt{a})/2 = \sqrt{a}$$

Figure 1 shows one step of Newton iteration for finding $\sqrt{2}$, in which $a = 2$, $x_0 = 2$, and $x_1 = (2 + 2/2)/2 = 1.5$. That figure hints at the rapid rate at which this method converges, but the story can't be told graphically. Here are the next few elements in the sequence $x_i$:

```
2.0000000000000000
1.5000000000000000
1.4166666666666667
1.4142156862745098
1.4142135623746899
1.4142135623730951
```

The values were computed by a simple "scaffolding" program; see Problem 6. The final answer is correct to 16 decimal places.

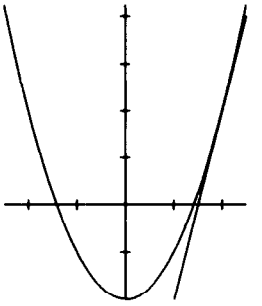


**FIGURE 1. One Step in Finding Sqrt(2)**

### A Great Place to Start

There's the basic idea of Newton iteration; two problems stand between us and a program:

What is a good initial value $x_0$?

How many iterations should be made until $x_i$ is declared to be the final answer?

We will explore the second question in the next section; this section concentrates on the first.

The example in the previous section showed Newton's method converging very quickly. Each iteration roughly doubled the number of accurate digits; because the error at the $i + 1^{st}$ step is proportional to the square of the error at the $i^{th}$ step, numerical analysts refer to this as "quadratic convergence." That behavior is typical of the method, so long as two conditions hold. The first requirement is

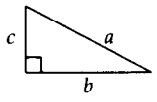
that the derivative is not near zero; that is always true for square roots (so long as we compute $\sqrt{0}$ as a special case), but it can be difficult for other functions.

The second requirement for quadratic convergence is that the initial guess must be near the final root. Table I shows that when the current value is far from the square root, Newton's method gives only one bit of accuracy per iteration. (Beware, though, that for functions less well behaved, Newton's method will not even converge if it starts far from the root.)

```
1000.0000000000000000
 500.0010000000000000
 250.0024999960000100
 125.0052499580004700
  62.5106246430170320
  31.2713096020621940
  15.6676329948683660
   7.8976423478563581
   4.0754412405194990
   2.2830928243925538
   1.5795487524060154
   1.4228665795786682
   1.4142398735915306
   1.4142135626178485
   1.4142135623730951
```

**TABLE I. Convergence to Sqrt(2)**

Most general-purpose square root routines get an initial guess by black magic of some sort, such as extracting the bit field that is the exponent of a floating point number and halving it to approximate the square root. (Using the last square root computed is very effective in some applications; see Problem 9.) In the context of a distance function, we can use other information to get the initial guess. When $K$ is 2, for instance, we wish to compute $a = \sqrt{b^2 + c^2}$:




We can use the maximum of $b$ and $c$ ($b$ in the above figure) as the initial guess $x_0$. Thus we have the inequalities

$$c \le b \le a = \sqrt{b^2 + c^2} \le \sqrt{2 \times b^2} = \sqrt{2} \times b$$

so we know that $a$ is in the range $[b, \sqrt{2} \times b]$.

In higher dimensions we will use as an initial estimate the maximum of the differences in all dimensions, which we'll call $D$. The distance is at least $D$ and the sum of the squares of the $K$ differences is at most $K \times D^2$, so the distance is in the range $[D, D\sqrt{K}]$.

## The Code

We can now write a program for computing Euclidean distances. Program 2 uses as its initial value the maximum difference. It iterates until two subsequent values are reasonably close: until $|x_{i+1} - x_i|/x_{i+1}$ is at most the one part in ten million that corresponds to single-precision accuracy on my machine.

```
T := abs(A[1] - B[1])
Max := T
Sum := T*T
for J := 2 to K do
    T := abs(A[J] - B[J])
    if T > Max then Max := T
    Sum := Sum + T*T
if Sum = 0.0 then return 0.0
/* find sqrt(Sum), starting at Max */
Eps = 1.0e-7
Z := Max
loop
    NewZ := 0.5 * (Z + Sum/Z)
    if abs(NewZ-Z)<=Eps*NewZ then break
    Z := NewZ
return NewZ
```

**PROGRAM 2. A General Distance Routine**

Table II (on p. 1159) gives the run time of all programs discussed in this column. It shows that Program 2 is about 35 percent faster than Program 1 when $K = 2$: the new square root code is indeed faster than the system routine. When $K = 16$, though, Program 2 is only about 1.5 percent faster than Program 1: the bottleneck in this case is not the square root, and finding the maximum difference chews up most of the time saved by the faster root-finder. Fortunately, the specifications stated that $K$ tends to be small.

There are two ways to improve Program 2: we'll start by speeding up the root-finder, and then shortly work on computing the maximum difference. The current version iterates until it is close enough; the next version will iterate a fixed number of times guaranteed to produce convergence. That will remove the cost of loop overhead, of testing for convergence, and of computing the final iteration that is so very close to its predecessor.

So how many iterations do we need? The specifications state that $K \le 16$ and that we must compute to single-precision accuracy. Because $K \le 16$, we know that the distance is at most $\sqrt{16} \times D$ (where $D$ is the maximum difference, `max`), and therefore in the range $[D, 4D]$. It seemed that the geometric

### The Best Possible Square Root Routine

This column describes a few hours' work by an amateur, using techniques ranging in age from a few decades to a few centuries. We'll now briefly glimpse how a world-class numerical analyst attacked the problem of constructing a numerical routine. Professor W. Kahan lectured on "Implementation of Algorithms" in the early 1970s; lecture notes taken by Haugeland and Hough appeared as Berkeley Computer Science Technical Report #20 and are now available as National Technical Information Service Report AD-769 124. Of the 339 pages in the report, 53 are devoted to the construction and error analysis of an unbeatable square root routine for the IBM 7094.

Kahan starts by specifying the properties that the routine should have. Monotonicity implies that if $x \geq y$, then $sqrt(x) \geq sqrt(y)$. He demands that $sqrt(x*x) = x$, but observes that one cannot ensure that $sqrt(x)*sqrt(x) = x$. He gives particularly stringent requirements on the accuracy of the routine: his code gives incorrectly rounded answers for only 29 distinct mantissas.

Kahan's routine uses a few loop-unrolled Newton iterations (which he calls Heron's rule) after getting a good starting value. He observes that the best method for finding a starting value is quite machine-dependent (on such issues as the relative cost of table lookup versus multiplication). Kahan therefore built his routine by considering a tree of all possible sequences of IBM 7094 instructions:

> You begin by doing necessary things, like loading the arguments. . . . I used to work on [the tree] in the evenings. It took several—3 or 4 or 5. It did cover a big table. I would connect one branch to another, indicating that they computed the same

mean of that range, $2D$, would make a good initial value. I used my scaffolding program to examine the convergence from that midpoint to the bounds of the range. I first computed $\sqrt{1}$ starting from 2:

| x | abs(x-1.0)/1.0 |
|---|---|
| 2.0000000000000000 | 1.0000000000000000 |
| 1.2500000000000000 | 0.2500000000000000 |
| 1.0250000000000000 | 0.0250000000000000 |
| 1.0003048780487805 | 0.0003048780487805 |
| 1.0000000464611473 | 0.0000000464611473 |
| 1.0000000000000011 | 0.0000000000000011 |
| 1.0000000000000000 | 0.0000000000000000 |

Next I computed $\sqrt{16}$ from the same start:

| x | abs(x-4.0)/4.0 |
|---|---|
| 2.0000000000000000 | 0.5000000000000000 |
| 5.0000000000000000 | 0.2500000000000000 |
| 4.1000000000000000 | 0.0250000000000000 |
| 4.0012195121951220 | 0.0003048780487805 |
| 4.0000001858445894 | 0.0000000464611473 |
| 4.0000000000000043 | 0.0000000000000011 |
| 4.0000000000000000 | 0.0000000000000000 |

Because Newton iteration scales linearly, these two cases model computing $\sqrt{D^2}$ and $\sqrt{16D^2}$ from any starting value $2D$. Problem 15 proves that these two extremes are indeed the two that are slowest to converge; the right columns show that after the first step the two inputs give the same relative error. The process yields the required seven-digit accuracy after four steps. The unrolled loop in Program 3 thus computes an accurate answer when $K \leq 16$.

```
    ... same as Program 2 ...
/* compute sqrt(Sum),
   starting at 2.0*Max */
Max := Max * 2.0
Max := 0.5 * (Max + Sum/Max)
Max := 0.5 * (Max + Sum/Max)
Max := 0.5 * (Max + Sum/Max)
return 0.5 * (Max + Sum/Max)
```

**PROGRAM 3. A Fast Distance Routine for $K \leq 16$**

Problem 11 suggests a further speedup to computing square roots: using table lookup to obtain a better initial guess. The numerical examples above show that if we can get the relative error down to 2.5 percent, then two further iterations suffice for single-precision accuracy.

The final improvements leave the lofty planes of numerical analysis to employ a couple of old coding tricks. The first one is specialized to the C language. Both the real program and the test program implemented a vector of points as a two-dimensional array of floating point numbers. The final program introduced two new variables to point to the two Euclidean points being compared, and thus replaced $K$ two-dimensional array accesses with $K$ references to a one-dimensional vector. The second trick is described in Problem 10; it exploits an algebraic identity. Because these speedups are quite particular to

function, using leftover telephone wire.

His routine was about 20 percent faster than the previous system routine, and much more accurate. Because he considered the tree of possible programs, Kahan could ensure that his program could never be beaten.

Kahan knew that his methods weren't easy:

> [This analysis] may have frightened you into thinking that to write a square root routine you have to have spent years studying abstruse theories. I guess if you want to write the best possible square root routine, maybe you do. There is a limit to how near perfection it is worthwhile to come, and it is not my intention to suggest that you should write a program in this way, since only a simple program could be optimized by examining a tree structure in this way.

And he kept perspective on his work:

> You're trying to ask me, was it worth the money spent. Of course it wasn't worth the money spent if you want to figure it in terms of the number of happier users. I probably tested more numbers than will be run through the SQRT in a year on the 7094.
>
> But we are trying to see how well we can do. For the practical question, I hope most people would have stopped [at an earlier routine]. We can't afford too many guys like me. But we can't afford to do without them, either.

I enthusiastically recommend this document. If you're fascinated by a "best possible" program for a substantial task, you can't afford to do without reading Kahan's report.

the implementation language, Program 4 was timed but is not shown in pseudocode.

The four distance routines are summarized in Table II. The speedup from Program 1 to Program 4 is a factor of 3.5 for $K = 2$, 2.8 for $K = 4$, and 1.9 for $K = 16$.

**TABLE II. Summary of the Programs**

| Routine number | Microseconds | | |
|---|---|---|---|
| | $K = 2$ | $K = 4$ | $K = 16$ |
| 1 | 1140 | 1270 | 2030 |
| 2 | 730 | 990 | 2000 |
| 3 | 350 | 500 | 1340 |
| 4 | 330 | 450 | 1070 |

## Principles

Distance computations are the workhorse in many programs. The new distance routine doubled the speed of my 1000-line traveling salesman program, and similar speedups are common for other geometric programs. In addition to producing a useful routine, this exercise has illustrated several general principles.

*The Importance of Context.* The process of producing a fast distance routine changes dramatically with many factors. For instance, most of the work described in this column would have been counterproductive on a system with a hardware square root instruction. For very large values of $K$ (say, 1000), the cost of the square root is relatively minor; for $K = 2$ (that is, for planar points), the method sketched in Program 17 is often faster than Program 4 and always much more robust. One must know a great deal of context before starting to code.

*Newton Iteration.* This powerful technique is of everyday utility for numerical analysts, and also of occasional use to mere mortal programmers (see Problem 1).

*Coding Tricks.* Though the big improvements are usually due to algorithmic changes, little improvements to code can reduce run time. In this case study, unrolling the iteration loop was very effective: it removes loop overhead, convergence testing, and an extra iteration. Other tricks included exploiting algebraic identities, optimizing array references, and storing precomputed answers in tables (see problems 10, 11, and 12).

*The Role of Libraries.* An excellent library is a delight to use; most numerical libraries provide accurate and numerically robust code. It is wise to remember, though, that few routines can be all things to all users. In this case study, special-purpose code was tailored to the context in which it was used to be much more efficient than the general routine. Reusability and numerical accuracy were sacrificed for raw speed; in this case, that was a sound engineering trade-off.

## Problems

1. Your library square root routine provides only single-precision accuracy, yet your application

requires double precision. What do you do?

2. On a hand-held calculator, repeatedly take the square root of a number then square it back again. What does this exercise tell you about the calculator's internal structure?
3. Newton's method does not work when $f'(x) = 0$; this happens for square roots only when computing $\sqrt{0}$. What happens when Newton's method attempts to compute $\sqrt{0}$ from a starting value of $x_0 = 1$? Does the algorithm have similar problems for computing the roots of positive numbers near zero?
4. Study the square root routine provided by your system. If it uses Newton's method, what is its initial value and how many iterations does it make?
5. Some computers have very fast hardware multipliers and no hardware dividers; they implement division by multiplying by an inverse. Show how to compute $1/a$ by using Newton's method to find a zero of $f(x) = a - 1/x$. Try using Newton's method to compute cube roots, or to find roots of arbitrary polynomials.
6. Implement a "scaffolding" program for Newton iteration. Its input is a number whose square root is to be taken, a starting value, and the number of iterations to be performed (provide defaults); its output is a trace of the values.
7. Implement Programs 1, 2, 3, and 4 on your system. How do you test their correctness? Build a testbed for timing them; how do your results compare to Table II?
8. [J. L. Blue] This column explicitly ignored the problems of overflow and underflow in summing the squares of differences. Write a program that is sensitive to those problems.
9. A common heuristic uses the last square root computed as the starting value for the next Newton iteration. Measure this in an application. How many iterations does it make on the average? How does it compare to other starting values?
10. The second optimization to Program 4 observes that Program 3 doubles `Max` only to halve it in the next statement:

```
Max := Max * 2.0
Max := 0.5 * (Max + Sum/Max)
```

Use a little algebra to speed up those statements.

11. Table lookup can speed up a program by trading space for run time. How can this technique be used in computing a good starting value? How could you use table lookup to compute Euclidean distances if the planar point set has both $x$ and $y$ coordinates in the range 0..9999?
12. [A. Appel] Show how the $K$ absolute values used by Program 2 to compute `Max` can be replaced with a single absolute value. (Hint: keep track of the largest square seen so far.)
13. Hardware designers have observed that a division and a square root box of comparable efficiency require comparable amounts of hardware. Show that square root is about as hard as division in software, too, by sketching a routine to compute $\sqrt{2}$ accurate to one million decimal digits.
14. [S. Crocker] Consideration of finite-precision arithmetic complicates many programs, but makes this square root routine particularly simple:

```
X := 1
loop
    NewX := 0.5 * (X + A/X)
    if NewX = X then return NewX
    X := NewX
```

Does it converge on your machine for all nonzero inputs $A$? On all machines? (For a better starting value, see Problem 9.)

15. [M. D. McIlroy] What is the best starting value for Newton's method for square roots in a bounded range? Let $n$ be a natural number and let $a$, $b$, and $r$ be reals satisfying $0 < a \le r \le b$; let $R = r^2$. Given $n$, $a$, and $b$, we desire to choose a starting value $x_0 = x$ for the Newton iteration $x_{i+1} = (x_i + R/x_i)/2$ to minimize the worst-case relative error

$$max_{a \le r \le b}(x_n - r)/r$$

Show that the optimal choice is $x = \sqrt{ab}$, independent of the value of $n$.

(Hint from Doug McIlroy: Define the $i^{th}$ relative error to be $e_i(x, r) = (x_i - r)/r$ and find a recursion for it. Iterate that recurrence relation a few times to guess a closed form, then prove it by induction.)

16. Problem 15 identifies the best starting value for Newton iteration. How many iterations are required as a function of the number of dimensions ($K$) and desired accuracy?
17. Moler and Morrison have described a fast, robust, and portable algorithm for computing $\sqrt{P^2 + Q^2}$ (see "Replacing Square Roots by Pythagorean Sums" in the *IBM Journal of Re-*

*search and Development* 27, 6, November 1983, pp. 577–581). Their algorithm can be sketched as

```
P := abs(P); Q := abs(Q)
if P < Q then Swap(P, Q)
if P = 0.0 then return Q
repeat IterCount times
    R := Q / P
    R := R * R
    R := R / (4 + R)
    P := P + 2*R*P
    Q := Q * R
return P
```

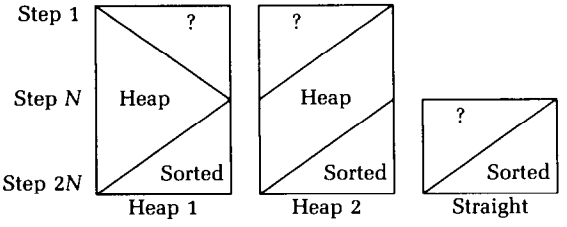
Its cubic convergence means that the result is accurate to 6.5 decimal digits after two iterations, to 20 digits after three iterations, and to 62 digits after four iterations; its intermediate results avoid overflow and underflow.

a. Use this code in a subroutine to compute planar Euclidean distances. How does its run time compare to Program 3 when $K = 2$?
b. How can you use this routine to compute Euclidean distances in $K$ space? How long would your code take when $K = 1000$, and how does that compare to Program 3?

**18.** How would you write a Euclidean distance routine to run on a parallel processor that can perform $P$ arithmetic operations at once?

### Solutions to September's Problems

**2.** Three variations of selection sort are shown in this figure, in which the array is represented horizontally and time proceeds down the vertical axis.




The left diagram shows a simple heapsort, which builds the heap by sifting each element up the partially built heap. The middle heapsort has the same second phase, but builds the heap right-to-left by sifting elements down. The right diagram shows a straight insertion sort; it does not build a heap, which avoids the construction cost but greatly increases the cost of each selection.

### Further Reading

There are dozens of excellent textbooks on numerical analysis. Which one is best for you depends on your desires for breadth and depth and your interest in mathematics and code.

**3.** The problem asked how programs should be typeset to achieve Willamson's three primary goals of correctness, consistency, and clarity.

*Correctness.* The best way to get a correct program in a document is to start with a correct program on a computer. Life is easiest when one can test and typeset the program from the same source file; I do that whenever possible. In this column, however, I presented the algorithms in pseudocode but I implemented and tested them in C. I therefore wrote the C programs in a form as close as possible to the final pseudocode (for instance, I was quite aware of the width of *Communications*' columns), and then used a text editor to make the remaining changes (I know—I should write a program to do the job).

*Consistency.* Programmers should be consistent about little details such as capitalization and indentation. Even better than adhering to your own standard, follow one that already exists in the field. For instance, if I did present C programs, I would use the format employed by Kernighan and Ritchie in their *C Programming Language.*

*Clarity.* The May and June columns showed how Don Knuth's WEB system produces clear programs by varying fonts: **bold** for keywords, *italic* for variables, `typewriter` for text strings, roman for explanation, etc. The programs in this column are typeset in `typewriter` font: that fixed-size font reflects what most programmers (myself included) see on their terminals, and it is still readable even when shrunk to a fairly small size.

**CR Categories and Subject Descriptors:** D.1.1 [**Programming Techniques**]; G.1 [**Numerical Analysis**]: General
**General Terms:** Performance
**Additional Key Words and Phrases:** Kahan, W. (Professor)

For Correspondence: Jon Bentley, AT&T Bell Laboratories, Room 2C-317, 600 Mountain Ave., Murray Hill, NJ 07974.