CHAPTER THIRTY-THREE

Writing Programs for "The Book"

Brian Hayes

The mathematician Paul Erdös often spoke of The Book, a legendary volume (not to be found on the shelves of any earthly library) in which are inscribed the best possible proofs of all mathematical theorems. Perhaps there is also a *Book* for programs and algorithms, listing the best solution to every computational problem. To earn a place in those pages, a program must be more than just correct; it must also be lucid, elegant, concise, even witty.

We all strive to create such gems of algorithmic artistry. And we all struggle, now and then, with a stubborn bit of code that just won't shine, no matter how hard we polish it. Even if the program produces correct results, there's something strained and awkward about it. The logic is a tangle of special cases and exceptions to exceptions; the whole structure seems brittle and fragile. Then, unexpectedly, inspiration strikes, or else a friend from down the hall shows you a new trick, and suddenly you've got one for *The Book*.

In this chapter I tell the story of one such struggle. It's a story with a happy ending, although I'll leave it to readers to decide whether the final program deserves a place in *The Book*. I wouldn't be brash enough even to suggest the possibility except that this is one of those cases where the crucial insight came not from me but from a friend down the hall—or, rather, from a friend across the continent.

The Nonroyal Road

The program I'll be talking about comes from the field of computational geometry, which seems to be peculiarly rich in problems that look simple on first acquaintance but turn out to be real stinkers when you get into the details. What do I mean by computational geometry? It's not the same as computer graphics, although there are close connections. Algorithms in computational geometry live not in the world of pixels but in that idealized ruler-and-compass realm where points are dimensionless, lines have zero thickness, and circles are perfectly round. Getting exact results is often essential in these algorithms, because even the slightest inaccuracy can utterly transform the outcome of a computation. Changing a digit far to the right of the decimal point might turn the world inside out.

Euclid supposedly told a princely student, "There is no royal road to geometry." Among the nonroyal roads, the computational pathway is notably muddy, rutty, and potholed. The difficulties met along the way sometimes have to do with computational efficiency: keeping a program's running time and memory consumption within reasonable bounds. But efficiency is not the main issue with the geometric algorithms that concern me here; instead, the challenges are conceptual and aesthetic. Can we get it right? Can we make it beautiful?

The program presented below in several versions is meant to answer a very elementary question: given three points in the plane, do all of the points lie along the same line? This is such a simple-sounding problem, it ought to have a simple solution as well. A few months ago, when I needed a routine to answer the collinearity question (as a component of a larger program), the task looked so straightforward that I didn't even pause to consult the literature and see how others might have solved it. I don't exactly regret that haste—wrestling with the problem on my own must have taught me something, or at least built some character—but I admit it was not the royal road to the right answer. I wound up repeating the steps of many who went before me. (Maybe that's why the road is so rutted!)

Warning to Parenthophobes

I present the code in Lisp. I'm not going to apologize for my choice of programming language, but neither do I want to turn this chapter into a tract for recruiting Lisp converts. I'll just say that I believe multilingualism is a good thing. If reading the code snippets below teaches you something about an unfamiliar language, the experience will do you no harm. All of the procedures are very short—half a dozen lines or so. Figure 33-1 offers a thumbnail guide to the structure of a Lisp procedure.

Incidentally, the algorithm implemented by the program in the figure is surely in *The Book*. It is Euclid's algorithm for calculating the greatest common divisor of two numbers.

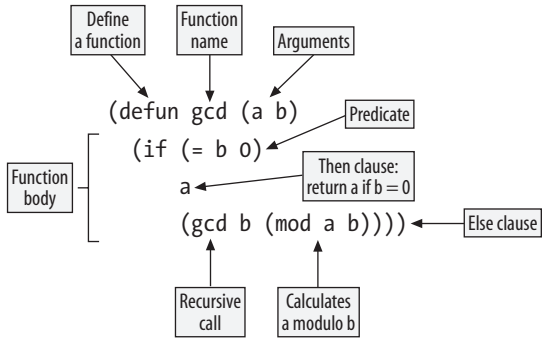

FIGURE 33-1. Bits and pieces of a Lisp procedure definition

Three in a Row

If you were working out a collinearity problem with pencil and paper, how would you go about it? One natural approach is to plot the positions of the three points on graph paper, and then, if the answer isn't obvious by inspection, draw a line through two of the points and see whether the line passes through the third point (see Figure 33-2). If it's a close call, accuracy in placing the points and drawing the line becomes critical.

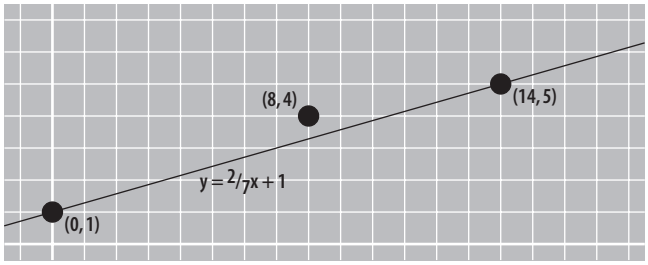

FIGURE 33-2. Three noncollinear points

A computer program can do the same thing, although for the computer nothing is ever "obvious by inspection." To draw a line through two points, the program derives the equation of that line. To see whether the third point lies on the line, the program tests whether or not the coordinates of the point satisfy the equation. (Exercise: For any set of three given points, there are three pairs of points you could choose to connect, in each case leaving a different third point to be tested for collinearity. Some choices may make the task easier than others, in the sense that less precision is needed. Is there some simple criterion for making this decision?)

The equation of a line takes the form $y = mx + b$, where m is the slope and b is the y-intercept, the point (if there is one) where the line crosses the y-axis. So, given three points p, q, and r, you want to find the values of m and b for the line that passes through two of them, and then test the x- and y-coordinates of the third point to see if the same equation holds.

Here's the code:

```
(defun naive-collinear (px py qx qy rx ry)
  (let ((m (slope px py qx qy))
        (b (y-intercept px py qx qy)))
    (= ry (+ (* m rx) b))))
```

The procedure is a predicate: It returns a boolean value of true or false (in Lisp argot, `t` or `nil`). The six arguments are the *x*- and *y*-coordinates of the points *p*, *q*, and *r*. The `let` form introduces local variables named `m` and `b`, binding them to values returned by the procedures `slope` and `y-intercept`. I'll return shortly to the definitions of those procedures, but their functions should be apparent from their names. Finally, the last line of the procedure does all the work, posing the question: is the *y*-coordinate of point *r* equal to *m* times the *x*-coordinate of *r*, plus *b*? The answer is returned as the value of the `naive-collinear` function.

Could it be simpler? Well, we'll see. Does it work? Often. If you were to set the procedure loose on a large collection of points generated at random, it would probably run without error for a very long time. Nevertheless, it's easy to break it. Just try applying it to points with (*x y*) coordinates (0 0), (0 1), and (0 2). These points are surely collinear—they all lie on the *y*-axis—and yet the `naive-collinear` procedure can't be expected to return a sensible value when given them as arguments.

The root cause of this failure is lurking inside the definition of `slope`. Mathematically, the slope *m* is $\Delta y/\Delta x$, which the program calculates as follows:

```
(defun slope (px py qx qy)
  (/ (- qy py) (- qx px)))
```

If *p* and *q* happen to have the same *x*-coordinate, then Δx is zero, and $\Delta y/\Delta x$ is undefined. If you insist on trying to calculate the slope, you'll get no further than a divide-by-zero error. There are lots of ways of coping with this annoyance. The method I chose as I first assembled the pieces of this little program was to have `slope` return a special signal value if `px` is equal to `qx`. The Lisp custom is to use the value `nil` for this purpose:

```
(defun slope (px py qx qy)
  (if (= px qx)
      nil
      (/ (- qy py) (- qx px))))
```

Like the slope, the *y*-intercept of a vertical line is also undefined because the line crosses the *y*-axis either nowhere or (if $x=0$) everywhere. The same `nil` trick applies:

```
(defun y-intercept (px py qx qy)
  (let ((m (slope px py qx qy)))
    (if (not m)
        nil
        (- py (* m px)))))
```

Now I also had to re-rig the calling procedure to handle the possibility that the slope `m` is not a number but a bogus value:

```
(defun less-naive-collinearp (px py qx qy rx ry)
  (let ((m (slope px py qx qy))
        (b (y-intercept px py qx qy)))
    (if (numberp m)
        (= ry (+ (* m rx) b))
        (= px rx))))
```

If *m* is numeric—if the predicate (`numberp m`) returns `t`—then I proceed as before. Otherwise, I know that *p* and *q* share the same *x*-coordinate. It follows that the three points are collinear if *r* also has this same *x* value.

As the program evolved, the need to make special provisions for vertical lines was a continual irritation. It began to look like every procedure I wrote would have some ugly patch bolted on to deal with the possibility that a line is parallel to the *y*-axis. Admittedly, the patch was just an `if` expression, an extra line or two of code, not a major issue of software engineering. Conceptually, though, it seemed a needless complication, and perhaps a sign that I was doing something wrong or making life harder than it had to be. Vertical lines are not fundamentally any different from horizontal ones, or from lines that wander across the plane at any other angle. It's an arbitrary convention to measure slope with respect to the *y*-axis; the universe would look no different if we all adopted a different reference direction.

This observation suggests a way around the problem: rotate the whole coordinate frame. If a set of points are collinear in one frame, they must be collinear in all other frames as well. Tilt the axes by a few degrees one way or the other, and the divide-by-zero impasse disappears. The rotation is not difficult or computationally expensive; it's just a matrix multiplication. On the other hand, taking this approach means I still have to write that `if` expression somewhere, testing to see whether `px` is equal to `qx`. What I'd really prefer is to streamline the logic and get rid of the branch point altogether. Shouldn't it be possible to test for collinearity by means of some simple calculation on the coordinates of the points, without any kind of case analysis?

Here's a solution recommended (in a slightly different context) by one web site, which I shall allow to remain anonymous: when Δx is 0, just set $\Delta y/\Delta x$ to 10^{10}, a value "close enough to infinity." As a practical matter, I suspect that this policy might actually work quite well, most of the time. After all, if the input to the program derives in any way from measurements made in the real world, there will be errors far larger than 1 part in 10^{10}. All the same, this is a strategy I did not consider seriously. I may not know what the *Book* version of `collinear` looks like, but I refuse to believe it has a constant defined as "close enough to infinity."

The Slippery Slope

Instead of drawing a line through two points and seeing whether the third point is on the line, suppose I drew all three lines and checked to see whether they are really the same line. Actually, I would need to draw only two of the lines, because if line $\overline{pq}$ is identical to line $\overline{qr}$, it must also be equal to $\overline{pr}$. Furthermore, it turns out I need to compare only the slopes, not the *y*-intercepts. (Do you see why?) Judging by eye whether two lines are really coincident or form a narrow scissors angle might not be the most reliable procedure, but in the computational world it comes down to a simple comparison of two numbers, the *m* values (see Figure 33-3).

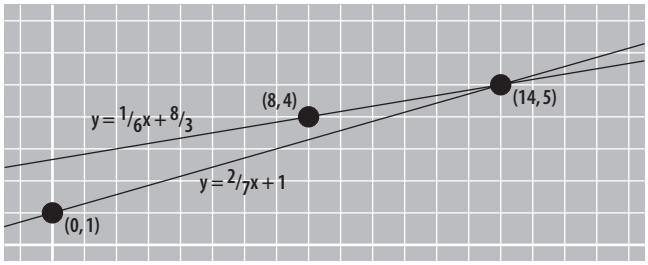

FIGURE 33-3. Testing collinearity by comparing slopes

I wrote a new version of `collinear` as follows:

```
(defun mm-collinear (px py qx qy rx ry)
  (equalp (slope px py qx qy)
          (slope qx qy rx ry)))
```

What an improvement! This looks much simpler. There's no `if` expression calling attention to the distinguished status of vertical lines; all sets of points are treated the same way.

I must confess, however, that the simplicity and the apparent uniformity are an illusion, based on some Lisp trickery going on behind the scenes. Note that I compare the slopes not with `=` but with the generic equality predicate `equalp`. The procedure works correctly only because `equalp` happens to do the right thing whether `slope` returns a number or `nil`. (That is, the two slopes are considered equal if they are both the same number or if they are both `nil`.) In a language with a fussier type system, the definition would not be so sweetly concise. It would have to look something like this:

```
(defun typed-mm-collinear (px py qx qy rx ry)
  (let ((pq-slope (slope px py qx qy))
        (qr-slope (slope qx qy rx ry)))
    (or (and (numberp pq-slope)
             (numberp qr-slope)
             (= pq-slope qr-slope))
        (and (not pq-slope)
             (not qr-slope)))))
```

This is not nearly as pretty, although even in this more-explicit form, the logic seems to me less tortured than the "naïve" version. The reasoning is that $\overline{pq}$ and $\overline{qr}$ are the same line if the slopes are both numbers and those numbers are equal, or if both slopes are `nil`. And, anyway, should one penalize a clever Lisp program just because other languages can't do the same trick?

I would have been willing to call it quits at this point and accept `mm-collinear` as the final version of the program, but for another anomaly that turned up in testing. Both `mm-collinear` and `less-naive-collinear` could successfully discriminate between collinear points and near misses; a case like p=(0 0), q=(1 0), r=(1000000 1) was not a challenge. But both procedures failed on this set of points: p=(0 0), q=(0 0), r=(1 1).

A first question is what *should* happen in this instance. The program is supposed to be testing the collinearity of three points, but here p and q are actually the same point. My own view is that such points are indeed collinear because a single line can be drawn through all of them. I suppose the opposite position is also defensible, on the grounds that a line of *any* slope could be drawn through two coincident points. Unfortunately, the two procedures, as written, do not conform to *either* of these rules. They return `nil` for the example given above but `t` for the points p=(0 0), q=(0 0), and r=(0 1). Surely this is pathological behavior by anyone's standards.

I could solve this problem by edict, declaring that the three arguments to the procedure must be distinct points. But then I'd have to write code to catch violations of the rule, raise exceptions, return error values, scold criminals, etc. It's not worth the bother.

The Triangle Inequality

Here's yet another way of rethinking the problem. Observe that if p, q, and r are *not* collinear, they define a triangle (Figure 33-4). It's a property of any triangle that the longest side is shorter than the sum of the smaller sides. If the three points are collinear, however, the triangle collapses on itself, and the longest "side" is exactly equal to the sum of the smaller "sides."

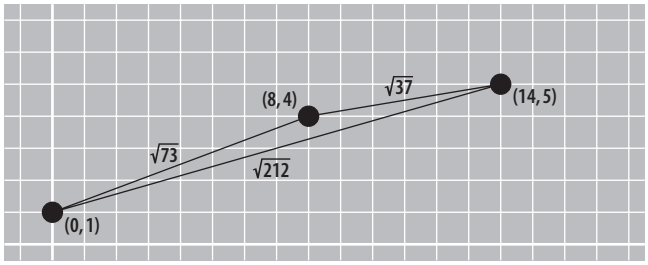

FIGURE 33-4. Testing collinearity by the triangle inequality

(For the example shown in the figure, the long side is shorter than the sum of the other two sides by about 0.067.)

The code for this version of the function is not quite so compact as the others, but what's going on inside is simple enough:

```
(defun triangle-collinear (px py qx qy rx ry)
  (let ((pq (distance px py qx qy))
        (pr (distance px py rx ry))
        (qr (distance qx qy rx ry)))
    (let ((sidelist (sort (list pq pr qr) #'>)))
      (= (first sidelist)
         (+ (second sidelist) (third sidelist))))))
```

The idea is to calculate the three side lengths, put them in a list, sort them in descending order of magnitude, and then compare the first (longest) side with the sum of the other two. If and only if these lengths are equal are the points collinear. This approach has a lot to recommend it. The calculation depends only on the geometric relations among the points themselves; it's independent of their position and orientation on the plane. Slopes and intercepts are not even mentioned. As a bonus, this version of the procedure also gives consistent and sensible answers when two or three of the points are coincident: all such point sets are considered collinear.

Unfortunately, there is a heavy price to be paid for this simplicity. Up to this point, all computations have been done with exact arithmetic. If the original coordinates are specified by means of integers or rational numbers, then the slopes and intercepts are calculated without round-off or other error. For example, the line passing through (1 1) and (4 2) has slope m=1/3 and y-intercept b=2/3 (not decimal approximations such as 0.33 and 0.67). With numbers represented in this way, comparisons are guaranteed to give the mathematically correct answer. But exactness is unattainable in measuring distances. The procedure `distance` invoked by `triangle-collinear` is defined like this:

```
(defun distance (px py qx qy)
  (sqrt (+ (square (- qx px))
           (square (- qy py)))))
```

The square root is the culprit, of course. If `sqrt` returns an irrational result, there's no hope of finding an exact, finite, numeric representation. When distances are calculated with double-precision IEEE floating-point arithmetic, `triangle-collinear` gives trustworthy answers for points whose coordinates are no larger than about 10^5. Go much beyond that threshold, and the procedure inevitably starts to mistake very skinny triangles for degenerate ones, incorrectly reporting that the vertices are collinear.

There is no quick and easy fix for this failing. Tricks like rotating or scaling the coordinate frame will not help. It's just a bug (or feature?) of our universe: rational points can give rise to irrational distances. Getting exact and reliable results under these circumstances is not quite impossible, but it takes an industrial-strength effort. Where the three points really are collinear, this fact can be proved algebraically without evaluating the square roots. For example, given the collinear points (0 0), (3 3) and (5 5), the distance equation is $sqrt(50) = sqrt(18) + sqrt(8)$, which reduces to $5 \times sqrt(2) = 3 \times sqrt(2) + 2 \times sqrt(2)$. When the points are *not* collinear, numerical evaluation will eventually reveal an inequality, if

you calculate enough digits of the roots. But I don't relish the idea of implementing a symbolic algebra system and an adaptive multiprecision arithmetic module just to test trios of points for collinearity. There's gotta be an easier way. In the *Book* version of the algorithm, I expect greater economy of means.

Meandering On

To tell the rest of this story, I need to mention the context in which it took place. Some months ago I was playing with a simple model of river meandering—the formation of those giant horseshoe bends you see in the Lower Mississippi. The model decomposed the smooth curve of the river's course into a chain of short, straight segments. I needed to measure curvature along the river in terms of the bending angles between these segments, and in particular I wanted to detect regions of zero curvature—hence the collinearity predicate.

Another part of the program gave me even more trouble. As meanders grow and migrate, one loop sometimes runs into the next one, at which point the river takes a shortcut and leaves behind a stranded "oxbow" lake. (You don't want to be standing in the way when this happens on the Mississippi.) To detect such events in the model, I needed to scan for intersections of segments. Again, I was able to get a routine working, but it seemed needlessly complex, with a decision tree sprouting a dozen branches. As in the case of collinearity, vertical segments and coincident points required special handling, and I also had to worry about parallel segments.

For the intersection problem, I eventually spent some time in the library and checked out what the Net had to offer. I learned a lot. That's where I found the tip that 10^{10} is close enough to infinity. And Bernard Chazelle and Herbert Edelsbrunner suggested a subtler way of finessing the singularities and degeneracies I had run into. In a 1992 review article on line-segment intersection algorithms (see the "Further Reading" section at the end of this chapter), they wrote:

> For the ease of exposition, we shall assume that no two endpoints have the same *x*- or *y*-coordinates. This, in particular, applies to the two endpoints of the same segment, and thus rules out the presence of vertical or horizontal segments...Our rationale is that the key ideas of the algorithm are best explained without having to worry about special cases every step of the way. Relaxing the assumptions is very easy (no new ideas are required) but tedious. That's for the theory. Implementing the algorithm so that the program works in all cases, however, is a daunting task. There are also numerical problems that alone would justify writing another paper. Following a venerable tradition, however, we shall try not to worry too much about it.

Perhaps the most important lesson learned from this foray into the literature was that others have also found meaty challenges in this field. It's not just that I'm a code wimp. This was a reassuring discovery; on the other hand, it did nothing to actually solve my problem.

Later, I wrote an item about line-segment intersection algorithms on my weblog at *bit-player.org*. This was essentially a plea for help, and help soon came pouring in—more than I could absorb at the time. One reader suggested polar coordinates as a remedy for undefined slopes, and another advocated rewriting the linear equations in parametric form, so that x- and y-coordinates are given as functions of a new variable t. Barry Cipra proposed a somewhat different parametric scheme, and then came up with yet another algorithm, based on the idea of applying an affine transformation to shift one of the segments onto the interval (–1 0),(1 0). David Eppstein advocated removing the problem from Euclidean geometry and solving it on the projective plane, where the presence of "a point at infinity" helps in dealing with singularities. Finally, Jonathan Richard Shewchuk gave me a pointer to his lecture notes, papers, and working code; I'll return to Shewchuk's ideas below.

I was impressed—and slightly abashed—by this flood of thoughtful and creative suggestions. There were several viable candidates for a segment-intersection procedure. Furthermore, I also found an answer to the collinearity problem. Indeed, I believe the solution that was handed to me may well be the *Book* algorithm.

"Duh!"—I Mean "Aha!"

In cartoons, the moment of discovery is depicted as a light bulb turning on in a thought balloon. In my experience, that sudden flash of understanding feels more like being thumped in the back of the head with a two-by-four. When you wake up afterwards, you've learned something, but by then your new insight is so blindingly obvious that you can't quite believe you didn't know it all along. After a few days more, you begin to suspect that maybe you *did* know it; you *must* have known it; you just needed reminding. And when you pass the discovery along to the next person, you'll begin, "As everyone knows…."

That was my reaction on reading Jonathan Shewchuk's "Lecture Notes on Geometric Robustness." He gives a collinearity algorithm that, once I understood it, seemed so natural and sensible that I'm sure it must have been latent within me somewhere. The key idea is to work with the area of a triangle rather than the perimeter, as in `triangle-collinear`. Clearly, the area of a triangle is zero if and only if the triangle is a degenerate one, with collinear vertices. But measuring a function of the area rather a function of the perimeter has two big advantages. First, it can be done without square roots or other operations that would take us outside the field of rational numbers. Second, it is much less dependent on numerical precision.

Consider a family of isosceles triangles with vertices (0 0), (x 1), and (2x 0). As x increases, the difference between the length of the base and the sum of the lengths of the two legs gets steadily smaller, and so it becomes difficult to distinguish this very shallow triangle from a totally flattened one with vertices (0 0), (x 0), and (2x 0). The area calculation doesn't suffer from this problem. On the contrary, the area grows steadily as the triangle gets longer (see Figure 33-5). Numerically, even without exact arithmetic, the computation is much more robust.

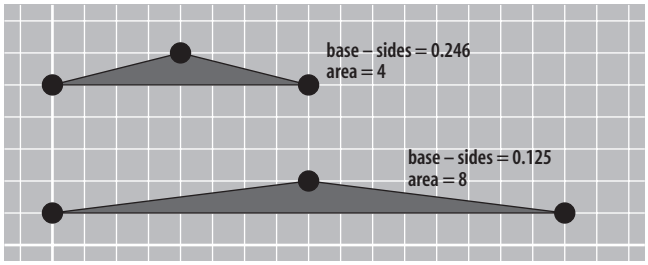

FIGURE 33-5. Testing collinearity by measuring area

How to measure the area? The Euclidean formula 1/2*bh* is not the best answer, and neither is the trigonometric approach. A far better plan is to regard the sides of a triangle as vectors; the two vectors emanating from any one vertex define a parallelogram, whose area is given by the cross product of the vectors. The area of the triangle is just one-half of the area of the parallelogram. Actually, this computation gives the "signed area": the result is positive if the vertices of the triangle are taken in counterclockwise order, and negative if taken in clockwise order. What's most important for present purposes, the signed area is zero if and only if the vertices are collinear.

The vector formula for the area is expressed most succinctly in terms of the determinant of a two-by-two matrix:

$$A = \frac{1}{2}\begin{vmatrix} x_1 - x_3 & y_1 - y_3 \\ x_2 - x_3 & y_2 - y_3 \end{vmatrix} = \frac{1}{2}[(x_1 - x_3)(y_2 - y_3) - (x_2 - x_3)(y_1 - y_3)]$$

Because I'm interested only in the case where the determinant is zero, I can ignore the factor of $^1/_2$ and code the collinearity predicate in this simple form:

```
(defun area-collinear (px py qx qy rx ry)
  (= (* (- px rx) (- qy ry))
     (* (- qx rx) (- py ry))))
```

So here it is: a simple arithmetical function of the *x*- and *y*-coordinates, requiring four subtractions, two multiplications and an equality predicate, but nothing else—no `if`s, no slopes, no intercepts, no square roots, no hazard of divide-by-zero errors. If executed with exact rational arithmetic, the procedure always produces exact and correct results. Characterizing the behavior with floating-point arithmetic is more difficult, but it is far superior to the version based on comparing distances on the perimeter. Shewchuk provides highly tuned C code that uses floating-point when possible and switches to exact arithmetic when necessary.

Conclusion

My adventures and misadventures searching for the ideal collinearity predicate do not make a story with a tidy moral. In the end I believe I stumbled upon the correct solution to my specific problem, but the larger question of how best to find such solutions in general remains unsettled.

One lesson that *might* be drawn from my experience is to seek help without delay: somebody out there knows more than you do. You may as well take advantage of the cumulative wisdom of your peers and predecessors. In other words, Google can probably find the algorithm you want, or even the source code, so why waste time reinventing it?

I have mixed feelings about this advice. When an engineer is designing a bridge, I expect her to have a thorough knowledge of how others in the profession have solved similar problems in the past. Yet expertise is not merely skill in finding and applying other people's bright ideas; I want my bridge designer to have solved a few problems on her own as well.

Another issue is how long to keep an ailing program on life support. In this chapter I have been discussing the tiniest of programs, so it cost very little to rip it up and start over whenever I encountered the slightest unpleasantness. For larger projects, the decision to throw one away is never so easy. And doing so is not necessarily prudent: you are trading known problems for unknown ones.

Finally there is the question of just how much the quest for "beautiful code" should be allowed to influence the process of programming or software development. The mathematician G. H. Hardy proclaimed, "There is no permanent place in the world for ugly mathematics." Do aesthetic principles carry that much weight in computer science as well? Here's another way of asking the same question: Do we have any guarantee that a *Book*-quality program exists for every well-formulated computational problem? Maybe *The Book* has some blank pages.

Further Reading

Avnaim, F., J.-D. Boissonnat, O. Devillers, F. P. Preparata, and M. Yvinec. "Evaluating signs of determinants using single-precision arithmetic." *Algorithmica*, Vol. 17, pp. 111–132, 1997.

Bentley, Jon L., and Thomas A. Ottmann. "Algorithms for reporting and counting geometric intersections." *IEEE Transactions on Computers*, Vol. C-28, pp. 643–647, 1979.

Braden, Bart. "The surveyor's area formula." *The College Mathematics Journal*, Vol. 17, No. 4, pp. 326–337, 1986.

Chazelle, Bernard, and Herbert Edelsbrunner. "An optimal algorithm for intersecting line segments in the plane." *Journal of the Association for Computing Machinery*, Vol. 39, pp. 1–54, 1992.

Forrest, A. R. "Computational geometry and software engineering: Towards a geometric computing environment." In *Techniques for Computer Graphics* (edited by D. F. Rogers and R. A. Earnshaw), pp. 23–37. New York: Springer-Verlag, 1987.

Forrest, A. R. "Computational geometry and uncertainty." In *Uncertainty in Geometric Computations* (edited by Joab Winkler and Mahesan Niranjan), pp. 69–77. Boston: Kluwer Academic Publishers, 2002.

Fortune, Steven, and Christopher J. Van Wyk. "Efficient exact arithmetic for computational geometry." In *Proceedings of the Ninth Annual Symposium on Computational Geometry*, pp. 163–172. New York: Association for Computing Machinery, 1993.

Guibas, Leonidas, and Jorge Stolfi. "Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams." *ACM Transactions on Graphics*, Vol. 4, No. 2, pp. 74–123, 1985.

Hayes, Brian. "Only connect!" *http://bit-player.org/2006/only-connect*. [Weblog item, posted September 14, 2006.]

Hayes, Brian. "Computing science: Up a lazy river." *American Scientist*, Vol. 94, No. 6, pp. 490–494, 2006. (*http://www.americanscientist.org/AssetDetail/assetid/54078*)

Hoffmann, Christoph M., John E. Hopcroft and Michael S. Karasick. "Towards implementing robust geometric computations." *Proceedings of the Fourth Annual Symposium on Computational Geometry*, pp. 106–117. New York: Association for Computing Machinery, 1988.

O'Rourke, Joseph. *Computational Geometry in C*. Cambridge: Cambridge University Press, 1994.

Preparata, Franco P., and Michael I. Shamos. *Computational Geometry: An Introduction*. New York: Springer-Verlag, 1985.

Qian, Jianbo, and Cao An Wang. "How much precision is needed to compare two sums of square roots of integers?" *Information Processing Letters*, Vol. 100, pp. 194–198, 2006.

Shewchuk, Jonathan Richard. "Adaptive precision floating-point arithmetic and fast robust geometric predicates." *Discrete and Computational Geometry*, Vol. 18, pp. 305–363, 1997. Preprint available at *http://www.cs.cmu.edu/afs/cs/project/quake/public/papers/robust-arithmetic.ps*.

Shewchuk, Jonathan Richard. Lecture notes on geometric robustness. [Version of October 26, 2006.] (*http://www.cs.berkeley.edu/~jrs/meshpapers/robnotes.ps.gz*. See also source code at *http://www.cs.cmu.edu/afs/cs/project/quake/public/code/predicates.c*.)