Oops! I Did It Again

If you’re wondering where I was last month, I was taking a much-needed vacation. I used the time to go out to San Jose for the Embedded Systems Conference. Among other things, I did a book signing for my new book (shameless plug), Math Toolkit for Real-Time Programming. I’ll bet you can never guess where the material for it came from. A few readers dropped by to say hello and listened to Jean Labrosse and me try to talk them into buying our books. It was good to see you all.

I have a lot to say about the ESC, mainly that it was huge! If you’ve never attended one of these conferences, you really owe it to yourself to do so. Beg, plead, and cajole your boss to send you, or just pony up the money on your own. But get there! If you’re like me, you’ll find its atmosphere of excitement and change a cure for every caught-in-a-rut feeling you’ve ever had. Exciting things are going on.

I’ll have a lot more to say about the conference and other things, such as my occasional Good Guys/Bad Guys list. But this month I want to dive right into the minimization problem. So I’ll defer the rest, except for these two comments.

First, I had speculated in an earlier column that large companies nowadays practice “planned disgruntlement,” meaning that they optimize their tech support to save costs by intentionally alienating a certain percentage of customers. A reader (who I’ll keep anonymous, along with the company he names, to avoid any repercussions) sent me the following e-mail, under the heading, “Suspicions confirmed”:

“Jack, I want to confirm your belief that companies do exactly what you think about QA on their products. During the ’80s, I learned that none less than [Company X, a very large, very prestigious company, noted for its quality] formally used a way to reduce testing on new products even if it meant losing some customers. This was factored into their cost of doing business. This was documented in a paper in an obscure statistics proceedings conference paper in [deleted] during the mid ’80s.

I didn’t believe this when a [Company X] salesman told me. It took our tech library [Company Y] some time to find it.”

To this I will add an observation from a colleague, who noted that he seemed to be getting a lot of dead-on-arrival items from a large department store that has a reputation for replacing broken items, no questions asked. After some sleuthing, he concluded that the company had made this policy possible by skipping acceptance testing on their products. In effect, they’re using their customers as their QA department. Interesting concept.

Second, a reader, Mike Mullen, was kind enough to send me his personal copy of Brent’s book, Algorithms for Minimization Without Derivatives. Was that generous or what? A cursory glance goes a long ways towards explaining why nobody ever seems to explain how Brent’s method works. It’s the same reason why everyone quotes, but very few people ever actually read, Kalman’s paper on Kalman filtering, or Cooley’s papers on Fast Fourier Transforms. Brent is a mathematician, and mathematicians, bless ’em, think in terms of theorems and lemmas. Such words tend to make my eyes cross in the same way that some of my stuff on matrices or quaternions does for you readers out there.

Even more to the point, and most astonishingly, even Brent doesn’t explain Brent’s algorithm! That’s right, you read correctly. After going through all sorts of proofs of guaranteed superlinear performance and the effects of round-off, Brent just sort of plops his algorithm out there, with no explanation whatsoever. No wonder Press et al. didn’t explain it! They had nothing from which to work.

This puts us all in a strange situation, it seems to me. The algorithm seems to have proven itself, and has been adopted by most of the scientific
Before each of the last few columns, I had hoped to wrap up the study in a blaze of glory, and present you with a totally finished, canned algorithm, à la Brent, that could solve any minimization problem we throw at it.

programming world as the algorithm of choice. Yet no one seems to be quite sure how it does what it does. The general notion of mixing golden section and quadratic fit is straightforward enough, but the details of such features as the halt criteria, or the logic to decide when to inject a golden section step, are retained as just so much witchcraft. I suppose it goes without saying that I don’t consider such a situation acceptable.

Back to the grindstone

This problem seems to have no end to it. The deeper I get into it, the deeper I get into it, so to speak. I feel a little bit like Brer Fox, in the Uncle Remus story, fighting the Tarbaby. (If you have never heard of Brer Fox, the Tarbaby, or Uncle Remus, don’t worry about it. The story comes from another generation, and it’s no longer politically correct. It’s still a great story, and those who missed it have been deprived of some precious and insightful lessons.)

Before each of the last few columns, I had hoped to wrap up the study in a blaze of glory, and present you with a totally finished, canned algorithm, à la Brent, that could solve any minimization problem we throw at it. This series must surely already be the longest running in a magazine like ESP. It’s become more of a daytime soap than a collection of articles; it’s still since Brent’s day. Think about it. The story comes from another generation, and it’s no longer politically correct. It’s still a great story, and those who missed it have been deprived of some precious and insightful lessons.)

In my last column, Moving Right Along (October 2000, p. 17), I mentioned that software like Brent’s is so difficult to follow because it’s written with efficiency, rather than readability, uppermost in mind. These algorithms tend to get used in other algorithms, such as multivariate minimization problems, so they must be very, very fast. But at this point we don’t want fast so much as understandable. So in that column and this one, I’m concentrating on writing an algorithm that’s easily understandable. We can always optimize the implementation later, but the algorithm will not change.

Another point occurred to me as I was preparing for this article. I can hardly claim it is profound, but it’s definitely worth mentioning: the art of software engineering has not stood still since Brent’s day. Think about it. All the programs I’ve been able to find on minimization, or almost any other canned method for that matter, are written in the style of ‘70s Fortran programs. Some of them actually are old Fortran programs, translated into C or C++. One popular text, whose title I’ll leave out this month (but you all know who I mean), even goes so far as to override the C indexing convention, so the authors can write their software with indices starting at 1 instead of 0. That approach must surely be close to the ultimate in lazy translation.

When I was writing the software for the last column, I collected the various parts of the algorithm into subroutines, with arguments passed in and out to control modularity. I also decided to let each subroutine inform me how successful it was, by returning a flag indicating success or failure. If, for example, a quadratic fit fails to give a reliable value, the function that performs the fit is in the best position to tell us whether it succeeded, and thereby trigger a golden ratio bisection. I ended up with functions that returned a Boolean variable to indicate success or failure.

In the midst of this process, however, I found myself constrained a bit by having the function return only a Boolean. It would be a lot better if it could suggest the next step, so the driver program doesn’t have to guess.

In preparation for this column, I asked myself, can one get a called function to suggest which way to go next? The answer is yes, using a state machine. This happens to be one of my favorite approaches for embedded software involving complex logic. The implementation of a state machine is the kind of thing that rarely gets done in Fortran, because the language—at least, early versions of it—didn’t support the enumerated type that makes
In general, another thing that makes most algorithms so inscrutable is the desire to minimize storage, especially global variables, and data motion. In this month’s column, you’ll see some radical departures in style.

it all practical. But it’s perfect for our uses. So, this month, in addition to seeing yet another attempt to develop a good, reliable, general-purpose function minimizer, you’re also going to see a program architecture that solves a lot of problems in a lot of different areas. I’ve often used nested state machines in real-time systems; the switch statement that implements them doesn’t cost much overhead, and neither does the single integer value returned. My experience has been that nested state machines can turn even the most hairy and convoluted exercise in logic and decision making into a clean, fast, and perfectly understandable solution. Here’s hoping it will do the same in this case.

In general, another thing that makes most algorithms so inscrutable is the desire to minimize storage, especially global variables, and data motion. In this month’s column, you’ll see some radical departures in style. At this point, I’m interested in keeping the count of function evaluations low—that’s the part that is potentially expensive. But I don’t really care if I’m using one more memory location than I need to, or if I’m moving data around too much. The performance impact of such issues is typically small compared to the cost of a function evaluation, and I can save making the algorithm optimally efficient (and therefore suitably inscrutable) for another day.

Random thoughts

This month, I’m also going to concentrate on some nagging thoughts that have been haunting me for some time. My general feelings are:

• We’re not using the data from previous steps as effectively as we could

• We need more robust criteria for deciding when to quit

Relative to both points, I’ve noted before that Brent’s method compares only previous values of the predicted x-value of the minimum, not the y-value. We’ve also seen that two applications of the quadratic fit can sometimes predict exactly the same x-value, even though the y-values are radically different (and both values are nowhere close to the minimum). It seems to me that we are not squeezing enough information out of the available data. One way to do so would be to track the y-values as well as x-values.

Also, consider the process involved in two golden-section bisections. We start with three points, which define two intervals, and bisect (evenly or based on the golden section) one of the intervals. Then, based upon the results of the first bisection, we throw away one of those points, and bisect again. After all is said and done, we have two new points, giving us a total of five. Yet only three—the lowest point, and its two neighbors—are passed on to the quadratic fit.

Can we get more information by using all five points? I think so. In past issues, I’ve suggested that if we consider four points at a time instead of three, we can perform two parabolic fits, each using the two endpoints plus one of the two interior points. The golden section generates that fourth point by bisecting one of the two intervals. But if we retain the last five points close to the minimum, we can fit two parabolas, each of which passes through the current best point. That approach must surely be superior to the one previously suggested.

I also noted last time that Brent’s method uses a relative, rather than absolute, error criterion in x. On this point, I’m absolutely adamant. I don’t care how popular the method is, this choice is simply No Good, because x may be zero, and dividing by zero is not a good idea. Implementations of Brent’s algorithm all use additive deltas to prevent divide-by-zero exceptions. In my not-so-humble opinion, this approach is a band-aid that hides a fundamental problem. I’ll have more to say about this in a moment.

If you were paying careful attention, you may have noticed that I previously wrote that we begin the process using two golden section steps, bisecting once on the left and once on the right. But then the thought occurred to me: why do we do this first bisection when we begin the solution sequence with three points chosen to fit the condition:

\[f(x_0) < f(x_1) < f(x_2)\]  \hspace{1cm} (1)

These points are perfectly good points for at least attempting a quadratic fit. So why bother bisecting? In retrospect, we shouldn’t have to do so unless the solution given by the fit is somehow unsatisfactory (for example, the value \(x_{\text{min}}\) is identical to \(x_1\)). I must admit to you that I’m still in the process of adjusting my attitude for this one. I’ve been assuming for so long that we begin with the golden section that I’m going to have to adapt to this new notion. Most of what I present later in this article still assumes the golden section as a starter. Nevertheless, I believe that in the long term we don’t need to do it this way.

One last thought: in my last column, I said that I’d forego Brent’s algorithm for a time, to give you guys time to try out the method I presented. But y’all let me down. Though I’ve gotten a lot of e-mail concerning my column (like Mike Mullen’s incredible offer of a copy of Brent), not one of you seems to have actually tried the
algorithm and tried to break it. Sigh. I guess I’m going it alone here, so I might as well get on with it.

Some parabolic math

It’s been a while since we specifically discussed the math of parabolic fits. What’s more, it can be quite tricky, and you will get slightly different results depending on the form you assume for \( f(x) \). To make matters worse, I’ve used different forms in past articles, so I think it’s best if we summarize the process of fitting a parabola to three data points here. We start by assuming that we have three data points:

\[
P_0 = [x_0, y_0] \\
P_1 = [x_1, y_1] \\
P_2 = [x_2, y_2]
\]

Next, we assume a quadratic equation using Newton’s form:

\[
f(x) = a + \left( x - x_0 \right) \left[ b + c \left( x - x_2 \right) \right]
\]

Note carefully the choice of \( x \) indices used. This is very important.

Next, we force the function to fit at these three points. This allows us to solve for the three coefficients \( a, b, \) and \( c \). The equations are:

\[
y_0 = a + \left( x_0 - x_0 \right) \left[ b + c \left( x_0 - x_2 \right) \right]
\]

\[
y_1 = a + \left( x_1 - x_0 \right) \left[ b + c \left( x_1 - x_2 \right) \right]
\]

\[
y_2 = a + \left( x_2 - x_0 \right) \left[ b + c \left( x_2 - x_2 \right) \right]
\]

Simplifying gives:

\[
y_0 = a
\]

\[
y_1 = a + \left( x_1 - x_0 \right) \left[ b + c \left( x_1 - x_2 \right) \right]
\]

\[
y_1 = a + \left( x_2 - x_0 \right) b
\]

The first equation clearly gives the value of \( a \), directly. The third one gives us \( b \):

\[
b = \frac{y_2 - y_0}{x_2 - x_0} = m
\]

Note that this parameter has the form of the slope, \( m \), of a straight line. In this case, the line connects points \( P_0 \) and \( P_2 \), the two end points of the range.

We are now left to solve the second equation for \( c \). We have:

\[
y_1 - y_0 = \frac{m}{x_1 - x_0} \left[ x_1 - x_2 \right]
\]

Again, the left side has the form of a slope, this time of the line between \( P_0 \) and \( P_1 \). We can properly define this as \( m_0 \). Then we get:

\[
c = \frac{m - m_0}{x_2 - x_1}
\]

It is not at all obvious from the form, but this is the same thing as the “slope of slopes”:

\[
c = \frac{m_1 - m_0}{x_2 - x_0}
\]

where \( m_1 \) is a slope just like \( m_0 \), but from \( P_1 \) to \( P_2 \). The easiest way to prove the equivalence is to expand both expressions out and compare terms.

Using the slopes gives us an easy way to compute \( b \) and \( c \). Define:

\[
m_0 = \frac{y_1 - y_0}{x_1 - x_0}
\]

\[
m_1 = \frac{y_2 - y_1}{x_2 - x_1}
\]

\[
m = \frac{y_2 - y_0}{x_2 - x_0}
\]

Then \( b = m \), and \( c \) is the slope of slopes given in Equation 7. Our functional form can then be written:

\[
f(x) = y_0 + \left( x - x_0 \right) \left[ m + c \left( x - x_2 \right) \right]
\]

Recall that at the minimum the slope of this function must be zero. Differentiating Equation 9 gives:

\[
0 = m + c \left( x_{\text{min}} - x_2 \right) + \left( x_{\text{min}} - x_0 \right) c
\]

or:

\[
x_{\text{min}} = \frac{1}{2} \left( x_2 + x_0 - \frac{m}{c} \right)
\]

It’s worth noting that \( c \) cannot be zero, since it’s the difference of two slopes that are necessarily non-zero and of opposite signs. That condition is a necessary consequence of Equation 1. The slope \( m \), on the other hand, can be zero, and will be if \( y_0 = y_2 \). This is the case when we start out with a balanced geometry. In this case, \( x_{\text{min}} \) is simply the average of \( x_0 \) and \( x_2 \), and is independent of the values of \( y = f(x) \).

Substituting the value for \( x_{\text{min}} \) into our function gives the predicted minimum value of \( f(x) \). It is:

\[
y_{\text{min}} = y_0 + \left( x_{\text{min}} - x_0 \right) \left[ m + c \left( x_{\text{min}} - x_2 \right) \right]
\]

Simplifying gives:

\[
y_{\text{min}} = y_0 + \left( \frac{1}{2} \left( x_2 + x_0 - \frac{m}{c} \right) - x_0 \right)
\]
Remember, because the shape of the function near its minimum is almost always parabolic when looked at with enough "zoom" applied, the trough of the minimum gets more and more shallow as we get closer to it.

\[
\left[ m + \epsilon \left( \frac{1}{2} \left( x_2 + x_0 - \frac{m}{c} \right) - x_2 \right) \right] \\
= y_0 + \epsilon \left( x_2 - x_0 - \frac{m}{c} \right) \left[ m + \frac{c}{2} \left( x_2 - x_0 - \frac{m}{c} \right) \right] \\
= y_0 + \frac{\epsilon}{4} \left( x_2 - x_0 - \frac{m}{c} \right)^2 \\
\]

or:

\[
y_{\min} = y_0 - \frac{\epsilon}{4} \left( x_2 - x_0 - \frac{m}{c} \right)^2
\]

About error criteria

A moment ago I said that I felt Brent’s choice of a relative error criterion in \( x \) was unfortunate, because it may lead to division by zero. Only the addition of an arbitrary constant prevents numerical disaster.

A far better approach is to use an absolute criterion in the first place. Fortunately, we are in a perfect position to take this approach because the original search range is given going in. If we can reduce the error in \( x \) to, say, one part in a million of the original range, I’d have to say we’ve pinned down the minimum pretty doggone well. Remember, because the shape of the function near its minimum is almost always parabolic when looked at with enough “zoom” applied, the trough of the minimum gets more and more shallow as we get closer to it. Therefore, we can’t expect this algorithm (or any other) to give the kind of precision we’re used to seeing in double-precision computations. The best we can hope for is roughly the square root of the machine precision.

On the other hand, we can expect the \( y \)-value to be accurate to machine precision, and I’m intending to use both. Again, I intend to use an absolute measure (or, to be more precise, a measure that is relative, as measured by the full range as opposed to the change from step to step). The only problem is, the full range of values of \( x \) is known going in, but those of \( y = f(x) \) are not. Nevertheless, we can determine the full range simply by recording the largest and smallest values of \( y \) found so far.

Based on these notions, I’m going to define the following error criteria for convergence. Let \( \epsilon \) be some input value giving the precision desired, as a precision relative to the maximum range. Then define:

\[
x_{\text{range}} = x_2 - x_0
\]

where the \( x_2 \) and \( x_0 \) referred to here are the original values, that is, they define the overall range within which we search for a minimum. Likewise, let:

\[
y_{\text{range}} = \text{big}_y - \text{little}_y
\]

where the values of \( \text{big}_y \) and \( \text{little}_y \) are dynamic. The parameter \( \text{big}_y \) will, of necessity, be either the original \( y_0 \) or \( y_2 \), depending on which is larger. The starting value for \( \text{little}_y \) will be the first value of \( y_1 \), the result after we get into the configuration where Equation 1 holds. As the search proceeds, we will hopefully find points whose heights are lower than the current minimum; that is, we are sliding downhill to the true minimum. As this happens, \( \text{little}_y \) can only get smaller, and as a result, \( y_{\text{range}} \) will increase.

Given these two ranges, I can now define what I consider to be a robust criterion for convergence:

\[
\Delta x < \sqrt{\epsilon} (x_{\text{range}}) \tag{15}
\]

\[
\Delta y < \epsilon (y_{\text{range}})
\]

Here, I’ve been necessarily vague as to what \( \Delta x \) and \( \Delta y \) are. In general, they’re the differences between two estimates of the minimum values, but I can’t be more specific at this time. We might define \( \Delta x \), for example, to be the difference between the values given by the two parabolic fits of a five-point fit. We also might define it to be the difference between the results of two successive passes or, as Brent does, we might define it to be the difference between estimates two passes apart.

Those are details that remain to be worked out. Either way, Equation 15 defines the condition that we are ready to accept the estimate as a converged solution. Also, either way, both of the conditions of Equation 15 must be satisfied.

Recall that in the golden section method we stop when the points bracketing the minimum (\( x_0 \) and \( x_2 \), in our current notation) are shrunk together to the necessary degree of accuracy. I am not sure yet that this is a reasonable requirement on the parabolic solution. I have the feeling that we cannot reasonably achieve this condition without using golden sections. That is why both Brent and I lean towards comparing successive estimates, rather than the bracketing values.

The test functions

Until now, I’ve been mostly using a single function for my testing. This is the now familiar one:

\[
f(x) = \cos(2\pi x^3)
\]
The loosest requirement on the function to be minimized is that it be continuous. If it is not—if its value jumps around between values of \( x \) that are arbitrarily close—all bets are off, and we have nothing left to work with but random probings.

On occasion, I’ve also used a variation:

\[
g(x) = \sin(2\pi x^3)
\]  

(17)

I use this variation because \( g(x) \) has a maximum before its minimum. As we will see in a moment, it’s important that our algorithm work even when the function increases for a time before decreasing—that’s why \( g(x) \) has value for us.

It should go without saying that one or two functions are not enough to prove the algorithm. As I get ever closer to the completed algorithm, I want to be quite sure that it works for functions that might seem, in some way, to be “pathological.”

The loosest requirement on the function to be minimized is that it be continuous. If it is not—if its value jumps around between values of \( x \) that are arbitrarily close—all bets are off, and we have nothing left to work with but random probings. If we are to have any hope of searching for a solution in the usual sense of the term, we must require that the function be continuous.

The functions \( f(x) \) and \( g(x) \), however, also have continuous first derivatives (in fact, all their derivatives are continuous). That is why their shape in the vicinity of the minimum must look like a parabola.

It should go without saying that an algorithm that assumes a function shaped like a parabola works very well when the function is a parabola. In fact, such a method should converge right away, and if it doesn’t, we need to fall back and regroup. On the other hand, many functions in nature don’t have continuous first derivatives, and therefore don’t look like parabolas, no matter what power of zoom lens we apply. An obvious example is:

\[
f(x) = |x|
\]  

(18)

Any decent general-purpose algorithm should find the minimum at \( x = 0 \). Not surprisingly, Brent’s method, being based on successive quadratic fits, works well when the function looks like a parabola, and pretty badly with a function such as the one given in Equation 18. It ends up falling back on successive bisections, not the parabolic fit.

For the remainder of this series, I’m adopting an even more stringent function:

\[
h(x) = \sqrt{\pi x - 1} - 1
\]  

(19)

The square root function causes the shape to be the opposite of a parabola, in a sense. The closer one gets to the minimum, the steeper the slope of the function, and the more a sharp edge forms at the minimum.

Does this mean that we don’t need the simple absolute value? Not at all. This function is unique in that it has segments that are straight lines on either side of the minimum. A smart minimizer should recognize this fact, and exploit it. Brent’s method doesn’t. Fortunately, we are in position to achieve fast convergence for this case.
Suppose, as I suggested earlier, we retain (at least) five successive data points, with the middle one being the best minimum to date. This means that we have two points on either side of the minimum. And that means we can fit straight lines through the two points on either side of the middle one. If the function is indeed an absolute value, we should find the solution immediately. Consider the two straight line segments:

\[ f_1(x) = y_0 + m_1(x - x_0) \]
\[ f_2(x) = y_3 + m_2(x - x_3) \]  \hspace{1cm} (20)

where \( m_1 \) and \( m_2 \) are the usual slopes:

\[ m_1 = \frac{y_1 - y_0}{x_1 - x_0} \]
\[ m_2 = \frac{y_4 - y_3}{x_4 - x_3} \]  \hspace{1cm} (21)

Where these two lines cross, their values are equal. Hence, at this point:

\[ y_0 + m_1(x - x_0) = y_3 + m_2(x - x_3) \]  \hspace{1cm} (22)

Solving for \( x \) gives:

\[ m_1(x - x_0) - m_2(x - x_3) = y_3 - y_0 \]
\[ (m_1 - m_2)x + (m_2x_3 - m_1x_0) = y_3 - y_0 \]
\[ x = \frac{m_2x_3 - m_1x_0 - (y_3 - y_0)}{m_2 - m_1} \]  \hspace{1cm} (23)

This approach gives us yet another estimate of the minimum. That estimate is clearly exact for the case of the absolute value function. It is also exact for the true parabola when the points are evenly spaced, left and right of the actual minimum. Thus, the formula can serve as an additional way of locating the minimum, and it has a reasonable chance of giving us a value as good as the parabolic fit. It can’t hurt to try, and it definitely needs to be there for functions of the absolute value kind. For the record, I’ve been experimenting with this approach, and I find that the values it suggests are not at all unreasonable, and sometimes quite good.

I should address the issue of efficiency here, because you might be wondering if all these extra computa-

---

LISTING 2  The final driver function

```c
// Local minimizer; Sets up conditions for iteration
double minimize(double (*f)(double), double x0, double x2, long n){
    double dx = (x2 - x0)/n;
    double y0, x1, y1, y2, xmin, x, y;
    y0 = f(x0);
    xmin = x0;
    ymin = y0;
    x1 = x0+dx;
    y1 = f(x1);
    // wait for function to turn downwards
    while((x1 < x2) && (y1 > ymin)){
        x0 = x1;
        y0 = y1;
        x1 += dx;
        y1 = f(x1);
    }
    if(x1 >= x2)
        return xmin;
    // now watch for it to turn up again
    x = x1+dx;
    y = f(x);
    while((x < x2) && (y < y1)){
        x0 = x1;
        y0 = y1;
        x1 = x;
        y1 = y;
        x = min(x + dx, x2);
        y = f(x);
    }
    if(x >= x2)
        return x;
    else
        return findmin(f, x0, y0, x1, y1, x2, y2);
}
```
The implication here is that one should understand the nature of the function one is trying to minimize. A graph is always a good idea. You just can’t use these routines blindly, trusting them to replace common sense.

LISTING 3 The skeleton state machine

```c
/* global variables */
static int bisect_left = TRUE;
typedef enum{INIT, BISECT, PARAFIT, TEST_BI, TEST_PAR, SUCCEED, FAIL} STATE;
static STATE state = INIT;

// parameters for the halt criterion
static double xrange;
static double yrange;
static double big_y;
static double little_y;
static double EPS;
static double xeps;
static double yeps;

// Stub function for second-order search
double findmin(double (*f)(double), double x0, double y0,
            double x1, double y1,
            double x2, double y2, double eps){
    double x;
    double y;
    while(1){
        switch(state){
            case INIT:
                xrange = x2 - x0;
                big_y = max(y0, y2);
                little_y = y1;
                EPS = eps;
                xeps = sqrt(eps) * xrange;
                yeps = eps * (top_y - bottom_y);
                // do first two bisections
                gold_shrink_left(x0, y0, x1, y1, x2, y2);
                save_best(x1, y1);
                gold_shrink_right(x0, y0, x1, y1, x2, y2);
                break;
        }
    }
    return x;  // do a parabolic fit
}
```

One thing we haven’t talked much about lately is how the minimizer gets started in the first place. In our tests, we always assume that we have already found three successive points $x_0$, $x_1$, $x_2$, or $(x_0 + x_2)/2$ to be reasonable.

At this point I’ve defined a fairly extensive set of test programs, all of which our final algorithm should handle. They are summarized in Table 1.

Getting started
As soon as a value greater than the previous one is found, we know that this point and the two preceding it form the triplet satisfying Equation 1. In short, they form the starting values for the main minimizer.

The general idea is that you give the routine, \texttt{minimize()}, a search range \( x_0..x_2 \), and it starts stepping along the function \( f(x) \) with a step size determined by the total number of intervals, \( n \). As long as \( f(x) \) continues to decrease, the stepping continues. As soon as the routine finds a value of \( f(x) \) larger than the previous one, it knows that this point and the two preceding it form the triplet satisfying Equation 1. These are the last three points generated. Note that the number of intervals must be at least two, so we can have at least three evaluations of the function. In the original version, \texttt{minimize()} simply accepted the next-to-last point as the minimum. That is, it stopped there, either accepting the lowest value on the grid as the solution, or, in later versions, iterating with a smaller step size. We can use this same approach as a driver to get us into the condition of Equation 1. Then we can call another function—call it \texttt{findmin()}—to refine the answer.

However, a slight problem exists with Listing 1: it fails on functions like functions 2 and 5 from Table 1. If \( f(x) \) is increasing at the start, instead of decreasing, this version of \texttt{minimize()} will kick out immediately and claim that \( x_0 \) is the minimum. Strictly speaking, this is a true statement; \( x_0 \) is certainly the local minimum, and nobody ever promised us a global one. However, it’s not a very satisfactory result. It would be better if we allowed for a function that can have a positive slope for a time. Note that if we want to handle Function 5 in Table 1 properly, we should record \( y_0 \) just in case we discover, after having searched the entire range, that it was indeed the true minimum over that range. If the stepping process hasn’t turned up a better solution, we still have \( x_0 \) to fall back on.

Listing 2 shows a function that will step along until \( f(x) \) begins to decrease, then track the decreasing value until it starts increasing again. As soon as a value greater than the previous one is found, we know that this point and the two preceding it form the triplet satisfying Equation 1. In short, they form the starting values for the main minimizer.

This function, here called \texttt{minimize()}, which we can think of as a driver function for the long-sought perfect minimizer, has a couple of nice features. First, it responds correctly to cases where \( f(x) \) either increases monotonically, or decreases monotonically, in the interval \( x_0..x_2 \). It correctly reports \( x_0 \) to be the minimum in the first case, \( x_2 \) in the second. Not even Brent’s method does this.

Second, in the event that \( f(x) \) is monotonic, this routine never calls the three-point function, here called \texttt{findmin()}, which means it will be fast.
The success of this approach depends upon choosing wisely the interval, which is determined by \( n \). If \( f(x) \) has a minimum that occurs between two mesh points that are otherwise monotonic, we would never see that. On the other hand, we don’t want to waste a lot of time stepping in small steps if coarse ones will do. The implication here is that one should understand the nature of the function one is trying to minimize. A graph is always a good idea. You just can’t use these routines blindly, trusting them to replace common sense. Note again that we must have at least three points, so \( n \) must be at least 2.

Should we require at least five points, so that we can be assured of having the number I now think we need to get a good halting criterion? We could do that, but it’s always possible that the minimum was back at the second point.

Matters of state

With the aid of the driver \( \text{minimize}() \), we can build a truly general-purpose minimizer. But we still need to flesh out the three-point function, \( \text{findmin}() \), that it calls. That will be the thrust of next month’s column. For now, I’ll leave you with a skeleton version of the code, which shows the structure implementing the state machine. That code is shown as Listing 3.

I’ve also made one other basic change. Last time I forced every bisection step to bisect both left and right intervals. This was done to urge \( x_0 \) and \( x_2 \) inwards, toward the true minimum. However, it caused the bisections to burn more function evaluations, which makes the algorithm look bad relative to Brent’s method. So here I’ve reverted to doing only a single bisection, but I keep track of which interval, left or right, I bisected last. I do force two bisections at the start, so that I know I have at least five data points before the first quadratic fit. The direction, past values, and so on are maintained, along with the state machine state, in variables that are global to the file (but would be properly invisible to other files).

We’ll continue this process next month. See you then.

Jack W. Crenshaw is a senior principal design engineer at Alliant Tech Systems in Clearwater, FL. He did much early work in the space program and has developed numerous analysis and real-time programs. He is the author of Math Toolkit for Real-Time Programming from CMP Books. He holds a PhD in physics from Auburn University. Crenshaw enjoys contact and can be reached via e-mail at jcrens@earthlink.net.

References