Chapter 4

Zeros and Roots

This chapter describes several basic methods for computing zeros of functions and then combines three of the basic methods into a fast, reliable algorithm known as "zeroin".

4.1 Bisection

Let's compute $\sqrt{2}$. We will use *interval bisection*, which is a kind of systematic trial and error. We know that $\sqrt{2}$ is between 1 and 2. Try $x = 1\frac{1}{2}$. Because x^2 is greater than 2, this x is too big. Try $x = 1\frac{1}{4}$. Because x^2 is less than 2, this x is too small. Continuing in this way, our approximations to $\sqrt{2}$ are

$$1\frac{1}{2},\ 1\frac{1}{4},\ 1\frac{3}{8},\ 1\frac{5}{16},\ 1\frac{13}{32},\ 1\frac{27}{64},\ \ldots$$

Here is a MATLAB program, including a step counter.

```
M = 2
a = 1
b = 2
k = 0;
while b-a > eps
   x = (a + b)/2;
   if x^2 > M
      b = x
   else
      a = x
   end
   k = k + 1;
end
```

We are sure that $\sqrt{2}$ is in the initial interval `[a,b]`. This interval is repeatedly cut in half and always brackets the answer. The entire process requires 52 steps. Here are the first few and the last few values:

```
b = 1.50000000000000
a = 1.25000000000000
a = 1.37500000000000
b = 1.43750000000000
a = 1.40625000000000
b = 1.42187500000000
a = 1.41406250000000
b = 1.41796875000000
b = 1.41601562500000
b = 1.41503906250000
b = 1.41455078125000
.....
b = 1.41421356237311
a = 1.41421356237299
a = 1.41421356237305
a = 1.41421356237308
a = 1.41421356237309
b = 1.41421356237310
b = 1.41421356237310
```

Using `format hex`, here are the final values of `a` and `b`.

```
a = 3ff6a09e667f3bcc
b = 3ff6a09e667f3bcd
```

They agree up to the last bit. We haven't actually computed $\sqrt{2}$, which is irrational and cannot be represented in floating point. But we have found two *successive* floating-point numbers, one on either side of the theoretical result. We've come as close as we can using floating-point arithmetic. The process takes 52 steps because there are 52 bits in the fraction of an IEEE double-precision number. Each step decreases the interval length by about one bit.

Interval bisection is a slow but sure algorithm for finding a zero of $f(x)$, a real-valued function of a real variable. All we assume about the function $f(x)$ is that we can write a MATLAB program that evaluates it for any x. We also assume that we know an interval $[a, b]$ on which $f(x)$ changes sign. If $f(x)$ is actually a *continuous* mathematical function, then there must be a point x_* somewhere in the interval where $f(x_*) = 0$. But the notion of continuity does not strictly apply to floating-point computation. We might not be able to actually find a point where $f(x)$ is exactly zero. Our goal is

> *Find a very small interval, perhaps two successive floating-point numbers, on which the function changes sign.*

The MATLAB code for bisection is

```
k = 0;
while abs(b-a) > eps*abs(b)
   x = (a + b)/2;
```

```
    if sign(f(x)) == sign(f(b))
       b = x;
    else
       a = x;
    end
    k = k + 1;
end
```

Bisection is slow. With the termination condition in the above code, it always takes 52 steps for any function. But it is completely reliable. If we can find a starting interval with a change of sign, then bisection cannot fail to reduce that interval to two successive floating-point numbers that bracket the desired result.

4.2 Newton's Method

Newton's method for solving $f(x) = 0$ draws the tangent to the graph of $f(x)$ at any point and determines where the tangent intersects the x-axis. The method requires one starting value, x_0. The iteration is

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

The MATLAB code is

```
k = 0;
while abs(x - xprev) > eps*abs(x)
   xprev = x;
   x = x - f(x)/fprime(x)
   k = k + 1;
end
```

As a method for computing square roots, Newton's method is particularly elegant and effective. To compute $\sqrt{M}$, find a zero of

$$f(x) = x^2 - M$$

In this case, $f'(x) = 2x$ and

$$\begin{aligned} x_{n+1} &= x_n - \frac{x_n^2 - M}{2x_n} \\ &= \frac{1}{2}\left(x_n + \frac{M}{x_n}\right) \end{aligned}$$

The algorithm repeatedly averages x and M/x. The MATLAB code is

```
while abs(x - xprev) > eps*abs(x)
   xprev = x;
   x = 0.5*(x + M/x);
end
```

Here are the results for $\sqrt{2}$, starting at $x = 1$.

```
1.50000000000000
1.41666666666667
1.41421568627451
1.41421356237469
1.41421356237309
1.41421356237309
```

Newton's method takes only six iterations. In fact, it was done in five, but the sixth iteration was needed to meet the termination condition.

When Newton's method works as it does for square roots, it is very effective. It is the basis for many powerful numerical methods. But, as a general-purpose algorithm for finding zeros of functions, it has three serious drawbacks.

- The function $f(x)$ must be smooth.
- It might not be convenient to compute the derivative $f'(x)$.
- The starting guess must be close to the final result.

In principle, the computation of the derivative $f'(x)$ can be done using a technique known as *tomatic differentiation*. A MATLAB function, `f(x)`, or a suitable code in any other programming language, defines a mathematical function of its arguments. By combining modern computer science parsing techniques with the rules of calculus, especially the chain rule, it is theoretically possible to generate the code for another function, `fprime(x)`, that computes $f'(x)$. However, the actual implementation of such techniques is quite complicated and has not yet been fully realized.

The local convergence properties of Newton's method are very attractive. Let x_* be a zero of $f(x)$ and let $e_n = x_n - x_*$ be the error in the nth iterate. Assume

- $f'(x)$ and $f''(x)$ exist and are continuous.
- x_0 is close to x_*.

Then it is possible to prove [2] that

$$e_{n+1} = \frac{1}{2}\frac{f''(\xi)}{f'(x_n)}e_n^2$$

where ξ is some point between x_n and x_*. In other words,

$$e_{n+1} = O(e_n^2)$$

This is called *quadratic* convergence. For nice, smooth functions, once you are close enough to the zero, the error is roughly squared with each iteration. The number of correct digits approximately doubles with each iteration. The results we saw for $\sqrt{2}$ are typical.

When the assumptions underlying the local convergence theory are not satisfied, Newton's method might be unreliable. If $f(x)$ does not have continuous,

bounded first and second derivatives, or if the starting point is not close enough to the zero, then the local theory does not apply and we might get slow convergence, or even no convergence at all. The next section provides one example of what might happen.

4.3 A Perverse Example

Let's see if we can get Newton's method to iterate forever. The iteration

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

cycles back and forth around a point a if

$$x_{n+1} - a = -(x_n - a)$$

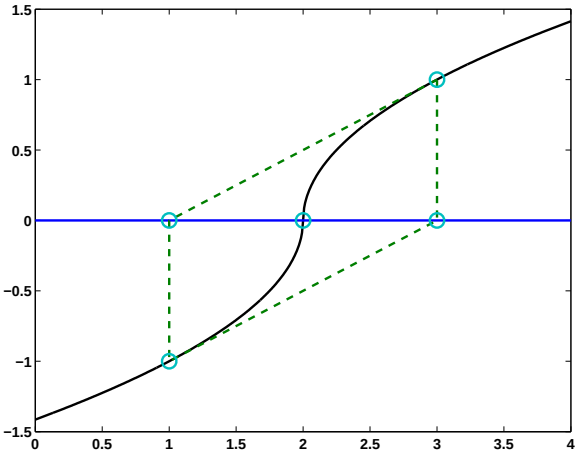

Figure 4.1. *Newton's method in an infinite loop*

This happens if $f(x)$ satisfies

$$x - a - \frac{f(x)}{f'(x)} = -(x - a)$$

This is a separable ordinary differential equation.

$$\frac{f'(x)}{f(x)} = \frac{1}{2(x-a)}$$

A solution is

$$f(x) = \text{sign}(x-a)\sqrt{|x-a|}$$

The zero of $f(x)$ is, of course, at $x_* = a$. A plot of $f(x)$, with $a = 2$, is obtained with

```
ezplot('sign(x-2)*sqrt(abs(x-2))',0,4)
```

If we draw the tangent to the graph at any point, it intersects the x-axis on the opposite side of $x = a$. Newton's method cycles forever, neither converging nor diverging.

The convergence theory for Newton's method fails in this case because $f'(x)$ is unbounded as $x \to a$. It is also interesting to apply the algorithms discussed in the next sections to this function.

4.4 Secant Method

The secant method replaces the derivative evaluation in Newton's method with a finite difference approximation based on the two most recent iterates. Instead of drawing a tangent to the graph of $f(x)$ at one point, you draw a secant through two points. The next iterate is the intersection of this secant with the x-axis.

The iteration requires two starting values, x_0 and x_1. The subsequent iterates are given by

$$s_n = \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}$$

$$x_{n+1} = x_n - \frac{f(x_n)}{s_n}$$

This formulation makes it clear how Newton's $f'(x_n)$ is being replaced by the slope of the secant, s_n. The formulation in the following MATLAB code is a little more compact.

```
while abs(b-a) > eps*abs(b)
   c = a;
   a = b;
   b = b + (b - c)/(f(c)/f(b)-1);
   k = k + 1;
end
```

For $\sqrt{2}$, starting with `a = 1` and `b = 2`, the secant method requires seven iterations, compared with Newton's six.

```
1.33333333333333
1.40000000000000
1.41463414634146
1.41421143847487
1.41421356205732
1.41421356237310
1.41421356237310
```

The secant method's primary advantage over Newton's method is that it does not require code to compute $f'(x)$. Its convergence properties are similar. Again, assuming $f'(x)$ and $f''(x)$ are continuous, it is possible to prove [2] that

$$e_{n+1} = \frac{1}{2}\frac{f''(\xi)f'(\xi_n)f'(\xi_{n-1})}{f'(\xi)^3}\, e_n e_{n-1}$$

where ξ is some point between x_n and x_*. In other words,

$$e_{n+1} = O(e_n e_{n-1})$$

This is not quadratic convergence, but it is *superlinear* convergence. It turns out that

$$e_{n+1} = O(e_n^{\phi})$$

where ϕ is the golden ratio, $(1+\sqrt{5})/2$. Once you get close, the number of correct digits is roughly multiplied by 1.6 with each iteration. That's almost as fast as Newton's method and a whole lot faster than the one bit per step produced by bisection.

We leave it as an exercise to investigate the behavior of the secant method on the perverse function from the previous section,

$$f(x) = \text{sign}(x-a)\sqrt{|x-a|}$$

4.5 Inverse Quadratic Interpolation

The secant method uses two previous points to get the next one, so why not use three?

Suppose we have three values, a, b, and c, and corresponding function values, $f(a)$, $f(b)$, and $f(c)$. We could interpolate these values by a parabola, a quadratic function of x, and take the next iterate to be the point where the parabola intersects the x-axis. The difficulty is that the parabola might not intersect the x-axis; a quadratic function does not necessarily have real roots. This could be regarded as an advantage. An algorithm known as Muller's method uses the complex roots of the quadratic to produce approximations to complex zeros of $f(x)$. But, for now, we want to avoid complex arithmetic.

Instead of a quadratic in x, we can interpolate the three points with a quadratic function in y. That's a "sideways" parabola, $P(y)$, determined by the interpolation conditions

$$a = P(f(a)),\ b = P(f(b)),\ c = P(f(c))$$

This parabola always intersects the x-axis, which is $y = 0$. So, $x = P(0)$ is the next iterate.

This method is known as *inverse quadratic interpolation.* We will abbreviate it with IQI. Here is MATLAB code that illustrates the idea.

```
k = 0;
while abs(c-b) > eps*abs(c)
   x = polyinterp([f(a),f(b),f(c)],[a,b,c],0)
   a = b;
   b = c;
   c = x;
   k = k + 1;
end
```

The trouble with this "pure" IQI algorithm is that polynomial interpolation requires the abscissae, which in this case are $f(a)$, $f(b)$, and $f(c)$, to be distinct. We have no guarantee that they are. For example, if we try to compute $\sqrt{2}$ using $f(x) = x^2 - 2$ and start with $a = -2, b = 0, c = 2$, we are starting with $f(a) = f(c)$ and the first step is undefined. If we start nearby this singular situation, say with $a = -2.001, b = 0, c = 1.999$, the next iterate is near $x = 500$.

So, IQI is like an immature race horse. It moves very quickly when it is near the finish line, but its global behavior can be erratic. It needs a good trainer to keep it under control.

4.6 Zeroin

The idea behind the *zeroin* algorithm is to combine the reliability of bisection with the convergence speed of secant and inverse quadratic interpolation. T. J. Dekker and colleagues at the Mathematical Center in Amsterdam developed the first version of the algorithm in the 1960s [3]. Our implementation is based on a version by Richard Brent [1]. Here is the outline:

- Start with a and b so that $f(a)$ and $f(b)$ have opposite signs.
- Use a secant step to give c between a and b.
- Repeat the following steps until $|b - a| < \epsilon|b|$ or $f(b) = 0$.
- Arrange a, b, and c so that
  - $f(a)$ and $f(b)$ have opposite signs.
  - $|f(b)| \leq |f(a)|$
  - c is the previous value of b.
- If $c \neq a$, consider an IQI step.
- If $c = a$, consider a secant step.
- If the IQI or secant step is in the interval [a,b], take it.
- If the step is not in the interval, use bisection.

This algorithm is foolproof. It never loses track of the zero trapped in a shrinking interval. It uses rapidly convergent methods when they are reliable. It uses a slow, but sure, method when it is necessary.

4.7 `fzerotx, feval`

The MATLAB implementation of the *zeroin* algorithm is called `fzero`. It has several features beyond the basic algorithm. A preamble takes a single starting guess and searches for an interval with a sign change. The values returned by the function `f(x)` are checked for infinities, `NaN`s, and complex numbers. Default tolerances can

be changed. Additional output, including a count of function evaluations, can be requested. Our textbook version of *zeroin* is `fzerotx`. We have simplified `fzero` by removing most of its additional features, while retaining the essential features of *zeroin*.

We can illustrate the use of `fzerotx` with the zeroth order Bessel function of the first kind, $J_0(x)$. This function is available in MATLAB as `besselj(0,x)`. Its first zero is computed, starting with the interval $[0, \pi]$, by the statement

```
fzerotx('besselj(0,x)',[0 pi])
```

The result is

```
ans =
   2.4048
```

You can see from figure 4.2 that the graph of $J_0(x)$ is like an amplitude and frequency modulated version of $\cos x$. The distance between successive zeros is close to π. The following code fragment produces figure 4.2 (except for the `'x'`, which we will add later).

```
for n = 1:10
   z(n) = fzerotx('besselj(0,x)',[(n-1) n]*pi);
end
x = 0:pi/50:10*pi;
y = besselj(0,x);
plot(z,zeros(1,10),'o',x,y,'-')
line([0 10*pi],[0 0],'color','black')
axis([0 10*pi -0.5 1.0])
```

The function `fzerotx` takes two arguments. The first specifies the function $F(x)$ whose zero is being sought and the second specifies the interval $[a, b]$ to search. `fzerotx` is an example of a MATLAB *function function*, which is a function that takes another function as an argument. `ezplot` is another example. Other chapters of this book — quadrature, ordinary differential equations, and even random numbers — also describe "tx" and "gui" M-files that are function functions.

A function can be passed as an argument to another function in five different ways:

- Function handle
- Inline object
- Name of an M-file
- Expression string
- Symbolic expression

The last two of these are available only with the function functions in our NCM package.

An expression string is the easiest to use for simple cases, but the least flexible for more complicated situations. Examples include

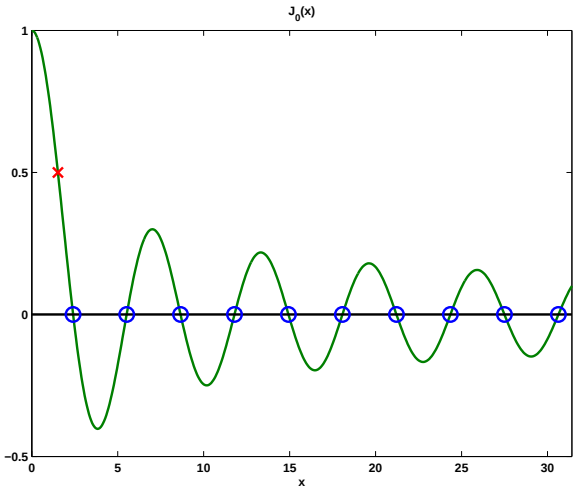

Figure 4.2. *Zeros of* $J_0(x)$

```
'cos(pi*t)'
'z^3-2*z-5'
'besselj(0,x)'
```

Note the single quotation marks that turn the expressions into strings.

An inline object is a way of defining simple functions without creating new files. Examples include

```
F = inline('cos(pi*t)')
F = inline('z^3-2*z-5')
F = inline('besselj(0,x)')
```

An inline object can be used as an argument to a function function, as in

```
z = fzerotx(F,[0,pi])
```

An inline object can also be evaluated directly, as in

```
residual = F(z)
```

A function handle uses the `'@'` character preceding the name of a built-in function or a function defined in an M-file. Examples include

```
@cos
@humps
@bessj0
```

where `bessj0.m` is the two-line M-file

```
function y = bessj0(x)
y = besselj(0,x)
```

These handles can then be used as arguments to function functions.

```
z = fzerotx(@bessj0,[0,pi])
```

Note that `@besselj` is also a valid function handle, but for a function of two arguments.

Older versions of MATLAB allowed the name of an M-file in quotes to specify a function argument, e.g.

```
z = fzerotx('bessj0',[0,pi])
```

It is still possible to use this mechanism in MATLAB 6.x, but we recommend that you use function handles instead.

The function functions in the NCM collection also accept a symbolic expression involving one free variable as their first argument.

```
syms x
F = besselj(0,x)
z = fzerotx(F,[0,pi])
```

Inline objects and functions referenced by function handles can define functions of more than one argument. In this case, the values of the extra arguments can be passed through `fzerotx` to the objective function. These values remain constant during the zero finding iteration. This allows us to find where a particular function takes on a specified value y, instead of just finding a zero. For example, consider the equation

$$J_0(\xi) = 0.5$$

Define an inline object of two or even three arguments:

```
F = inline('besselj(0,x)-y','x','y')
```

or

```
B = inline('besselj(n,x)-y','x','n','y')
```

Then either

```
xi = fzerotx(F,[0,z],.5)
```

or

```
xi = fzerotx(B,[0,z],0,.5)
```

produces

```
xi =
   1.5211
```

The point $(\xi, J_0(\xi))$ is marked with an `'x'` in figure 4.2.

These functional arguments are evaluated using `feval`. The expression

```
feval(F,x,...)
```

is the same as

```
F(x,...)
```

except that `feval` allows `F` to be passed as an argument.

The preamble for `fzerotx` is:

```
function b = fzerotx(F,ab,varargin);
%FZEROTX  Textbook version of FZERO.
%   x = fzerotx(F,[a,b]) tries to find a zero of F(x) between
%   a and b.  F(a) and F(b) must have opposite signs.
%   fzerotx returns one end point of a small subinterval of
%   [a,b] where F changes sign.
%   Additional arguments, fzerotx(F,[a,b],p1,p2,...),
%   are passed on, F(x,p1,p2,..).
```

The first section of code in `fzerotx` manipulates the argument `F` to make it acceptable to `feval`.

```
if ischar(F) & exist(F)~=2
   F = inline(F);
elseif isa(F,'sym')
   F = inline(char(F));
end
```

The next section of code initializes the variables `a`, `b`, and `c` that characterize the search interval. The function `F` is evaluated at the end points of the initial interval.

```
a = ab(1);
b = ab(2);
fa = feval(F,a,varargin{:});
fb = feval(F,b,varargin{:});
if sign(fa) == sign(fb)
   error('Function must change sign on the interval')
end
c = a;
fc = fa;
d = b - c;
e = d;
```

Here is the beginning of the main loop. At the start of each pass through the loop `a`, `b`, and `c` are rearranged to satisfy the conditions of the *zeroin* algorithm.

```
while fb ~= 0

   % The three current points, a, b, and c, satisfy:
   %    f(x) changes sign between a and b.
   %    abs(f(b)) <= abs(f(a)).
```

```
%    c = previous b, so c might = a.
% The next point is chosen from
%    Bisection point, (a+b)/2.
%    Secant point determined by b and c.
%    Inverse quadratic interpolation point determined
%    by a, b, and c if they are distinct.

if sign(fa) == sign(fb)
   a = c;  fa = fc;
   d = b - c;  e = d;
end
if abs(fa) < abs(fb)
   c = b;    b = a;    a = c;
   fc = fb;  fb = fa;  fa = fc;
end
```

This section is the convergence test and possible exit from the loop.

```
m = 0.5*(a - b);
tol = 2.0*eps*max(abs(b),1.0);
if (abs(m) <= tol) | (fb == 0.0),
   break
end
```

The next section of code makes the choice between bisection and the two flavors of interpolation.

```
% Choose bisection or interpolation
if (abs(e) < tol) | (abs(fc) <= abs(fb))
   % Bisection
   d = m;
   e = m;
else
   % Interpolation
   s = fb/fc;
   if (a == c)
      % Linear interpolation (secant)
      p = 2.0*m*s;
      q = 1.0 - s;
   else
      % Inverse quadratic interpolation
      q = fc/fa;
      r = fb/fa;
      p = s*(2.0*m*q*(q - r) - (b - c)*(r - 1.0));
      q = (q - 1.0)*(r - 1.0)*(s - 1.0);
   end;
   if p > 0, q = -q; else p = -p; end;
   % Is interpolated point acceptable
```

```
         if (2.0*p < 3.0*m*q - abs(tol*q)) & (p < abs(0.5*e*q))
            e = d;
            d = p/q;
         else
            d = m;
            e = m;
         end;
      end
```

The final section evaluates `F` at the next iterate.

```
      % Next point
      c = b;
      fc = fb;
      if abs(d) > tol
         b = b + d;
      else
         b = b - sign(b-a)*tol;
      end
      fb = feval(F,b,varargin{:});
   end
```

4.8 fzerogui

The M-file `fzerogui` demonstrates the behavior of *zeroin* and `fzerotx`. At each step of the iteration, you are offered a chance to choose the next point. The choice always includes the bisection point, shown in red on the computer screen. When there are three distinct points active, a, b, and c, the IQI point is shown in blue. When $a = c$, so there are only two distinct points, the secant point is shown in green. A plot of $f(x)$ is also provided as a dotted line, but the algorithm does not "know" these other function values. You can choose any point you like as the next iterate. You do not have to follow the *zeroin* algorithm and choose the bisection or interpolant point. You can even cheat by trying to pick the point where the dotted line crosses the axis.

We can demonstrate how `fzerogui` behaves by seeking the first zero of the Bessel function. It turns out that the first local minimum of $J_0(x)$ is located near $x = 3.83$. So here are the first few steps of

```
fzerogui('besselj(0,x)',[0 3.83])
```

Initially, $c = b$, so the two choices are the bisection point and the secant point.

If you choose the secant point, then b moves there and $J_0(x)$ is evaluated at $x = b$. We then have three distinct points, so the choice is between the bisection point and the IQI point.

If you choose the IQI point, the interval shrinks, the GUI zooms in on the reduced interval, and the choice is again between the bisection and secant points, which now happen to be close together.

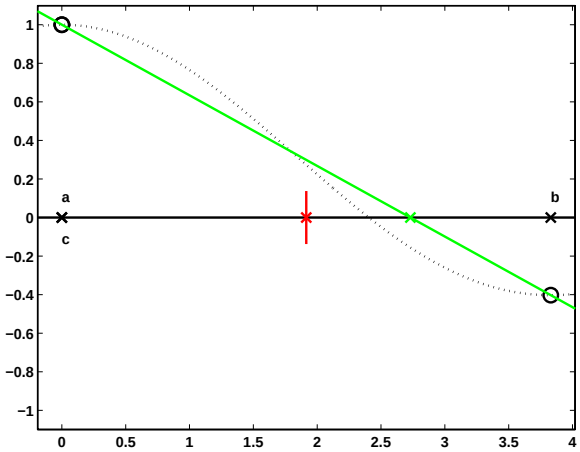

Figure 4.3. *Initially, choose secant or bisection*

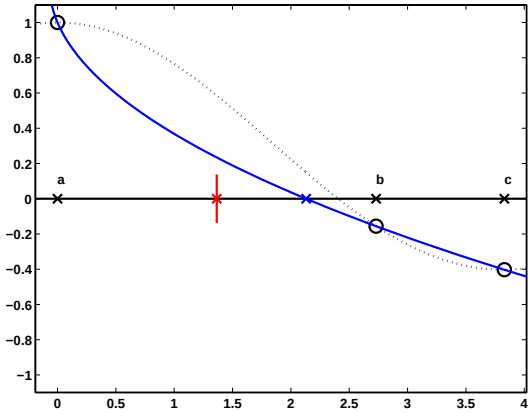

Figure 4.4. *Choose IQI or bisection*

You can choose either point, or any other point close to them. After two more steps, the interval shrinks again and the following situation is reached. This is the typical configuration as we approach convergence. The graph of the function looks nearly like a straight line and the secant or IQI point is much closer to the desired zero than the bisection point. It now becomes clear that choosing secant or IQI will lead to much faster convergence than bisection.

After several more steps, the length of the interval containing a change of sign is reduced to a tiny fraction of the original length and the algorithm terminates, returning the final b as its result.

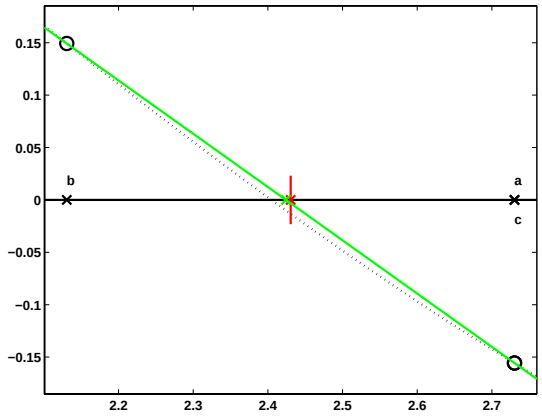

Figure 4.5. *Secant and bisection points nearly coincide*

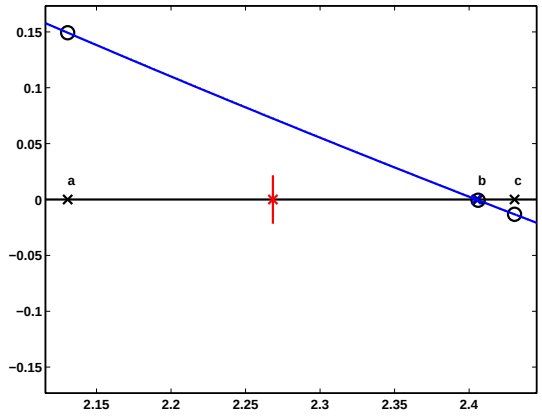

Figure 4.6. *Nearing convergence*

4.9 Value Finding and Reverse Interpolation

These two problems look very similar.

- Given a function $F(x)$ and a value η, find ξ so that $F(\xi) = \eta$.
- Given data (x_k, y_k) that samples an unknown function $F(x)$, and a value η, find ξ so that $F(\xi) = \eta$.

For the first problem, we are able to evaluate $F(x)$ at any x, so we can use a zero finder on the translated function $f(x) = F(x) - \eta$. This gives us the desired ξ so that $f(\xi) = 0$, and hence $F(\xi) = \eta$.

For the second problem, we need to do some kind of interpolation. The most obvious approach is to use a zero finder on $f(x) = P(x) - \eta$, where $P(x)$ is some interpolant, such as `pchiptx(xk,yk,x)` or `splinetx(xk,yk,x)`. This often works well, but it can be expensive. The zero finder calls for repeated evaluation of the interpolant. With the implementations we have in this book, that involves repeated calculation of the interpolant's parameters, and repeated determination of the appropriate interval index.

A sometimes preferable alternative, known as *reverse interpolation*, uses `pchip` or `spline` with the roles of x_k and y_k reversed. This requires monotonicity in the y_k, or at least in a subset of the y_k around the target value η. A different piecewise polynomial, say $Q(y)$, is created with the property that $Q(y_k) = x_k$. Now it is not necessary to use a zero finder. We simply evaluate $\xi = Q(y)$ at $y = \eta$.

The choice between these two alternatives depends on how the data is best approximated by piecewise polynomials. Is it better to use x or y as the independent variable?

4.10 Optimization and `fmintx`

The task of finding maxima and minima of functions is closely related to zero finding. In this section we describe an algorithm similar to *zeroin* that finds a local minimum of a function of one variable. The problem specification involves a function $f(x)$ and an interval $[a, b]$. The objective is to find a value of x that gives a local minimum of $f(x)$ on the given interval. If the function is *unimodular*, that is, has only one local minimum on the interval, then that minimum will be found. But if there is more than one local minimum, only one of them will be found, and that one will not necessarily be minimum for the entire interval. It is also possible that one of the end points is a minimizer.

Interval bisection cannot be used. Even if we know the values of $f(a)$, $f(b)$, and $f((a + b)/2)$, we cannot decide which half of the interval to discard and still keep the minimum enclosed.

Interval trisection is feasible, but inefficient. Let $h = (b - a)/3$, so $u = a + h$ and $v = b - h$ divide the interval into three equal parts. Assume we find that $f(u) < f(v)$. Then we could replace b by v, thereby cutting the interval length by a factor of two-thirds, and still be sure that the minimum is in the reduced interval. However, u would be in the center of the new interval and would not be useful in the next step. We would need to evaluate the function twice each step.

The natural minimization analog of bisection is *golden section* search. The idea is illustrated for $a = 0$ and $b = 1$ in figure 4.7. Let $h = \rho(b - a)$ where ρ is a quantity a little bit larger than $1/3$ that we have yet to determine. Then the points $u = a + h$ and $v = b - h$ divide the interval into three unequal parts. For the first step, evaluate both $f(u)$ and $f(v)$. Assume we find that $f(u) < f(v)$. Then we know the minimum is between a and v. We can replace b by v and repeat the process. If we choose the right value for ρ, the point u is in the proper position to be used in the next step. After the first step, the function has to be evaluated only once each step.

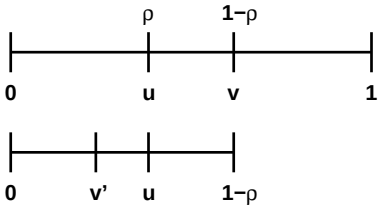

Figure 4.7. *Golden section search*

The defining equation for ρ is

$$\frac{\rho}{1-\rho} = \frac{1-\rho}{1}$$

or

$$\rho^2 - 3\rho + 1 = 0$$

The solution is

$$\rho = 2 - \phi = (3 - \sqrt{5})/2 \approx .382$$

Here ϕ is the golden ratio that we used to introduce MATLAB in the first chapter of this book.

With golden section search, the length of the interval is reduced by a factor of $\phi - 1 \approx .618$ each step. It would take

$$\frac{-52}{\log_2(\phi - 1)} \approx 75$$

steps to reduce the interval length to roughly `eps`, the size of IEEE double-precision roundoff error, times its original value.

After the first few steps, there is often enough history to give three distinct points and corresponding function values in the active interval. If the minimum of the parabola interpolating these three points is in the interval, then it, rather than the golden section point, is usually a better choice for the next point. This combination of golden section search and parabolic interpolation provides a reliable and efficient method for one-dimensional optimization.

The proper stopping criteria for optimization searches can be tricky. At a minimum of $f(x)$, the first derivative $f'(x)$ is zero. Consequently, near a minimum, $f(x)$ acts like a quadratic with no linear term,

$$f(x) \approx a + b(x - c)^2 + \ldots$$

The minimum occurs at $x = c$ and has the value $f(c) = a$. If x is close to c, say $x \approx c + \delta$ for small δ, then

$$f(x) \approx a + b\delta^2$$

Small changes in x are squared when computing function values. If a and b are comparable in size, and nonzero, then the stopping criterion should involve `sqrt(eps)` because any smaller changes in x will not affect $f(x)$. But if a and b have different orders of magnitude, or if either a or c is nearly zero, then interval lengths of size `eps`, rather than `sqrt(eps)`, are appropriate.

MATLAB includes a function function `fminbnd` that uses golden section search and parabolic interpolation to find a local minimum of a real-valued function of a real variable. The function is based upon a Fortran subroutine by Richard Brent [1]. MATLAB also includes a function function, `fminsearch`, that uses a technique known as the Nelder Mead simplex algorithm to search for a local minimum of a real-valued function of several real variables. The MATLAB Optimization Toolbox is a collection of programs for other kinds of optimization problems, including constrained optimization, linear programming, and large-scale, sparse optimization.

Our NCM collection includes a function `fmintx` that is a simplified textbook version of `fminbnd`. One of the simplifications involves the stopping criterion. The search is terminated when the length of the interval becomes less than a specified parameter `tol`. The default value of `tol` is 10^{-6}. More sophisticated stopping criteria involving relative and absolute tolerances in both x and $f(x)$ are used in the full codes.

The MATLAB `demos` directory includes a function named `humps` that is intended to illustrate the behavior of graphics, quadrature, and zero-finding routines. The function is

$$h(x) = \frac{1}{(x-0.3)^2 + .01} + \frac{1}{(x-0.9)^2 + .04}$$

The statements

```
F = inline('-humps(x)');
fmintx(F,-1,2,1.e-4)
```

takes the steps shown in figure 4.8 and in the following output. We see that golden section search is used for the second, third, and seventh steps, and that parabolic interpolation is used exclusively once the search nears the minimizer.

```
step           x              f(x)
init:      0.1458980337   -25.2748253202
gold:      0.8541019662   -20.9035150009
gold:     -0.2917960675     2.5391843579
para:      0.4492755129   -29.0885282699
para:      0.4333426114   -33.8762343193
para:      0.3033578448   -96.4127439649
gold:      0.2432135488   -71.7375588319
para:      0.3170404333   -93.8108500149
para:      0.2985083078   -96.4666018623
para:      0.3003583547   -96.5014055840
para:      0.3003763623   -96.5014085540
para:      0.3003756221   -96.5014085603
```

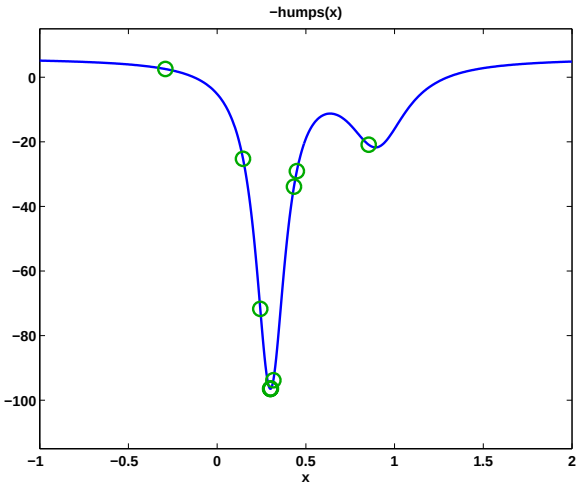

Figure 4.8. *Finding the minimum of* `-humps(x)`

Exercises

4.1. Use `fzerogui` to try to find a zero of each of the following functions in the given interval. Do you see any interesting or unusual behavior?

$x^3 - 2x - 5$	$[0, 3]$
$\sin x$	$[1, 4]$
$x^3 - .001$	$[-1, 1]$
$\log(x + 2/3)$	$[0, 1]$
$\text{sign}(x-2)\sqrt{\lvert x-2\rvert}$	$[1, 4]$
$\text{atan}(x) - \pi/3$	$[0, 5]$
$1/(x - \pi)$	$[0, 5]$

4.2. Here is a little footnote to the history of numerical methods. The polynomial

$$x^3 - 2x - 5$$

was used by Wallis when he first presented Newton's method to the French Academy. It has one real root, between $x = 2$ and $x = 3$, and a pair of complex conjugate roots.
(a) Use the Symbolic Toolbox to find symbolic expressions for the three roots. Warning: the results are not pretty. Convert the expressions to numerical values.
(b) Use the `roots` function in MATLAB to find numerical values for all three roots.
(c) Use `fzerotx` to find the real root.

(d) Use Newton's method starting with a complex initial value to find a complex root.
(e) Can bisection be used to find the complex root? Why or why not?

4.3. Here is a cubic polynomial with three closely spaced real roots.

$$p(x) = 816x^3 - 3835x^2 + 6000x - 3125$$

(a) What are the exact roots of p?
(b) Plot $p(x)$ for $1.43 \le x \le 1.71$. Show the location of the three roots.
(c) Starting with $x_0 = 1.5$, what does Newton's method do?
(d) Starting with $x_0 = 1$ and $x_1 = 2$, what does the secant method do?
(e) Starting with the interval [1,2], what does bisection do?
(f) What is `fzerotx(p,[1,2])`? Why?

4.4. What causes `fzerotx` to terminate?

4.5. (a) How does `fzerotx` choose between the bisection point and the interpolant point for its next iterate?
(b) Why is the quantity `tol` involved in the choice?

4.6. Derive the formula that `fzerotx` uses for inverse quadratic interpolation.

4.7. Hoping to find the zero of $J_0(x)$ in the interval $0 \le x \le \pi$, we might try the statement

```
z = fzerotx(@besselj,[0 pi],0)
```

This is legal usage of a function handle, and of `fzerotx`, but it produces `z = 3.1416`. Why?

4.8. Investigate the behavior of the secant method on the function

$$f(x) = \text{sign}(x-a)\sqrt{|x-a|}$$

4.9. Find the first ten positive values of x for which $x = \tan x$.

4.10. (a) Compute the first ten zeros of $J_0(x)$. You can use our graph of $J_0(x)$ to estimate their location.
(b) Compute the first ten zeros of $Y_0(x)$, the zeroth-order Bessel function of the second kind.
(c) Compute all the values of x between 0 and 10π for which $J_0(x) = Y_0(x)$.
(d) Make a composite plot showing $J_0(x)$ and $Y_0(x)$ for $0 \le x \le 10\pi$, the first ten zeros of both functions, and the points of intersection.

4.11. The *gamma* function is defined by an integral,

$$\Gamma(x+1) = \int_0^\infty t^x e^{-t} dt$$

Integration by parts shows that when evaluated at the integers, $\Gamma(x)$ interpolates the factorial function

$$\Gamma(n+1) = n!$$

$\Gamma(x)$ and $n!$ grow so rapidly that they generate floating point overflow for relatively small values of x and n. It is often more convenient to work with the logarithms of these functions.

The MATLAB functions `gamma` and `gammaln` compute $\Gamma(x)$ and $\log\Gamma(x)$ respectively. The quantity $n!$ is easily computed by the expression

```
prod(1:n)
```

but many people expect there to be a function named `factorial`, so MATLAB has such a function.
(a) What is the largest value of n for which $\Gamma(n+1)$ and $n!$ can be exactly represented by a double-precision floating-point number?
(b) What is the largest value of n for which $\Gamma(n+1)$ and $n!$ can be approximately represented by a double-precision floating-point number that does not overflow?

4.12. Stirling's approximation is a classical estimate for $\log\Gamma(x+1)$.

$$\log\Gamma(x+1) \sim x\log(x) - x + \frac{1}{2}\log(2\pi x)$$

Bill Gosper [4] has noted that a better approximation is

$$\log\Gamma(x+1) \sim x\log(x) - x + \frac{1}{2}\log(2\pi x + \pi/3)$$

The accuracy of both approximations improves as x increases.
(a) What is the relative error in Stirling's approximation and in Gosper's approximation when $x = 2$?
(b) How large must x be for Stirling's approximation and for Gosper's approximation to have a relative error less than 10^{-6}?

4.13. The statements

```
y = 2:.01:10;
x = gammaln(y);
plot(x,y)
```

produce a graph of the inverse of the $\log\Gamma$ function.
(a) Write a MATLAB function `gammalninv` that evaluates this function for any x. That is, given `x`,

```
y = gammalninv(x)
```

computes `y` so that `gammaln(y)` is equal to `x`.
(b) What are the appropriate ranges of x and y for this function?

4.14. Here is a table of the distance, d, that a hypothetical vehicle requires to stop if the brakes are applied when it is traveling with velocity v.

v	d
0	0
10	5
20	20
30	46
40	70
50	102
60	153

What is the speed limit for this vehicle if it must be able to stop in at most 60 meters? Compute the speed three different ways.
(a) Piecewise linear interpolation
(b) Piecewise cubic interpolation with `pchiptx`
(c) Reverse piecewise cubic interpolation with `pchiptx`
Because this is well behaved data, the three values are close to each other, but not identical.

4.15. Kepler's model of planetary orbits includes a quantity E, the eccentricity anomaly, that satisfies the equation

$$M = E - e \sin E$$

where M is the mean anomaly, and e is the eccentricity of the orbit. For this exercise, take $M = 24.851090$ and $e = 0.1$.
(a) Use `fzerotx` to solve for E. You should create an inline function with three parameters,

```
F = inline('E - e*sin(E) - M','E','M','e')
```

and provide `M` and `e` as extra arguments to `fzerotx`.
(b) An "exact" formula for E is known.

$$E = M + 2 \sum_{m=1}^{\infty} \frac{1}{m} J_m(me) sin(mM)$$

where $J_m(x)$ is the Bessel function of the first kind of order m. Use this formula, and the `besselj(m,x)` in MATLAB to compute E. How many terms are needed? How does this value of E compare to the value obtained with `fzerotx`?

4.16. Utilities must avoid freezing water mains. If we assume uniform soil conditions, the temperature $T(x,t)$ at a distance x below the surface and time t after the beginning of a cold snap is given approximately by

$$\frac{T(x,t) - T_s}{T_i - T_s} = \text{erf}\left(\frac{x}{2\sqrt{\alpha t}}\right)$$

Here T_s is the constant surface temperature during the cold period, T_i is the initial soil temperature before the cold snap, and α is the thermal conductivity of the soil. If x is measured in meters and t in seconds, then $\alpha = 0.138 \cdot 10^{-6} m^2/s$. Let $T_i = 20°C$, $T_s = -15°C$, and recall that water freezes at $0°C$. Use `fzerotx` to determine how deep a water main should be buried so that it will not freeze until at least 60 days exposure under these conditions.

4.17. Modify `fmintx` to provide printed and graphical output similar to that at the end of section 4.10. Reproduce the results shown in figure 4.8 for `-humps(x)`.

4.18. Let $f(x) = 9x^2 - 6x + 2$. What is the actual minimizer of $f(x)$? How close to the actual minimizer can you get with `fmintx`? Why?

4.19. Theoretically `fmintx(@cos,2,4,eps)` should return `pi`. How close does it get? Why? On the other hand, `fmintx(@cos,0,2*pi)` does return `pi`. Why?

4.20. If you use `tol = 0` with `fmintx(@F,a,b,tol)`, does the iteration run forever? Why or why not?

4.21. Derive the formulas for minimization by parabolic interpolation used in the following portion of `fmintx`.

```
r = (x-w)*(fx-fv);
q = (x-v)*(fx-fw);
p = (x-v)*q-(x-w)*r;
q = 2.0*(q-r);
if q > 0.0,  p = -p; end
q = abs(q);
r = e;
e = d;
% Is the parabola acceptable?
para = ( (abs(p)<abs(0.5*q*r)) & (p>q*(a-x)) & (p<q*(b-x)) );
if para
   d = p/q;
end
```

Bibliography

[1] R. P. Brent, *Algorithms for Minimization Without Derivatives*, Prentice Hall, Englewood Cliffs, 1973.

[2] G. Dahlquest and A. Björck, *Numerical Methods*, Prentice Hall, Englewood Cliffs, 1974.

[3] T. J. Dekker, *Finding a Zero by Means of Successive Linear Interpolation*, Constructive Aspects of the Fundamental Theorem of Algebra, B. Dejon and P. Henrici (editors), Wiley-Interscience, New York, 1969.

[4] Eric Weisstein, *World of Mathematics Stirling's Approximation*, http://mathworld.wolfram.com/StirlingsApproximation.html