Chapter 8

Nonlinear Equations and Optimization

§8.1 Finding Roots

§8.2 Minimizing a Function of a Single Variable

§8.3 Minimizing Multivariate Functions

§8.4 Solving Systems of Nonlinear Equations

In this chapter we consider several types of nonlinear problems. They differ in whether or not the solution sought is a vector or a scalar and whether or not the goal is to produce a root or a minimizer of some given function. The presentation is organized around a family of "orbit" problems. Suppose the vector-valued functions

$$\mathcal{P}_1(t) = \begin{bmatrix} x_1(t) \\ y_1(t) \end{bmatrix} \qquad \text{and} \qquad \mathcal{P}_2(t) = \begin{bmatrix} x_2(t) \\ y_2(t) \end{bmatrix}$$

specify the location at time t of a pair of planets that go around the Sun. Assume that the orbits are elliptical and that the Sun is situated at $(0, 0)$.

Question 1. At what times t are the planets and the Sun collinear? If $f(t)$ is the sine of the angle between $\mathcal{P}_1$ and $\mathcal{P}_2$, then this problem is equivalent to finding a zero of $f(t)$. Root-finding problems with a single unknown are covered in §8.1. We focus on the bisection and Newton methods, and the MATLAB zero-finder `fzero`.

Question 2. How close do the two *planets* get for $t \in [0, t_{max}]$? If $f(t)$ is the distance from $\mathcal{P}_1$ to $\mathcal{P}_2$, then this is a single-variable minimization problem. In §8.2 we develop the method of golden section search and discuss the MATLAB minimizer `fmin`. The role of graphics in building intuition about a search-for-a-min problem is highlighted.

Question 3. How close do the two *orbits* get? This is a minimization problem in two variables. For example, working with the 2-norm we could solve

$$\min_{t_1, t_2} \|\mathcal{P}_1(t_1) - \mathcal{P}_2(t_2)\|_2.$$

The method of steepest descent and the MATLAB multivariable minimizer `fmins` are designed to solve problems of this variety. They are discussed in §8.3 along with the idea of a line search.

Question 4. Where (if at all) do the two orbits intersect? This is an example of a multivariable root-finding problem:

$$F(t_1, t_2) = \mathcal{P}_2(t_2) \; - \; \mathcal{P}_1(t_1) \; = \; \begin{bmatrix} x_2(t_2) - x_1(t_1) \\ y_2(t_2) - y_1(t_1) \end{bmatrix} \; = \; 0.$$

The Newton framework for systems of nonlinear equations is discussed in §8.4. Related topics include the use of finite differences to approximate the Jacobian and the Gauss-Newton method for the nonlinear least squares problem.

8.1 Finding Roots

Suppose a chemical reaction is taking place and the concentration of a particular ion at time t is given by $10e^{-3t} + 2e^{-5t}$. We want to know when this concentration is one-half of its value at $t = 0$. Since $f(0) = 12$, this is equivalent to finding a zero of the function

$$f(t) = 10e^{-3t} + 2e^{-5t} - 6.$$

This particular problem has a unique root. In other zero-finding applications, the function involved may have several roots, some of which are required. For example, we may want to find the largest and smallest zeros (in absolute value) of the quintic polynomial $x^5 - 2x^4 + x^2 - x + 7$. In this section we discuss the bisection and Newton frameworks that can be applied to problems of this kind.

Algorithms in this area are *iterative* and proceed by producing a sequence of numbers that (it is hoped) converge to a root of interest. The implementation of any iterative technique requires that we deal with three major issues:

- Where do we start the iteration?
- Does the iteration converge, and how fast?
- How do we know when to terminate the iteration?

To build an appreciation for these issues, we start with a discussion about computing square roots.

8.1.1 The Square Root Problem

Suppose A is a positive real number and we want to compute its square root. Geometrically, this task is equivalent to constructing a square whose area is A. Having an approximate square root x_c is equivalent to having an approximating rectangle with sides x_c and A/x_c. To make the approximating rectangle "more square," it is reasonable to replace x_c with

$$x_+ = \frac{1}{2}\left(x_c + \frac{A}{x_c}\right),$$

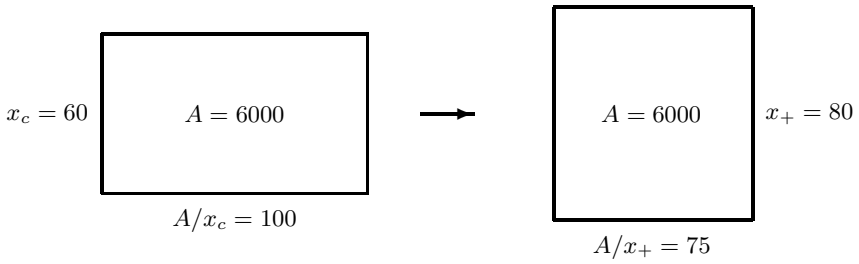

FIGURE 8.1. A better square root

since the value sought is clearly in between x_c and A/x_c. (See Figure 8.1 from which we conclude that $\sqrt{6000}$ is in between $x_c = 60$ and $A/x_c = 100$.) The process can obviously be repeated and it appears to converge:

```
                xc                          6000/xc
-------------------------------------------------------
 60.0000000000000000     100.0000000000000000
 80.0000000000000000      75.0000000000000000
 77.5000000000000000      77.4193548387096797
 77.4596774193548470      77.4596564289432479
 77.4596669241490474      77.4596669241476263
 77.4596669241483369      77.4596669241483369
 77.4596669241483369      77.4596669241483369
          :                        :
```

So much for how to improve an approximate square root. How do we produce an initial guess? The "search space" for the square root problem can be significantly reduced by writing A in the form

$$A = m \times 4^e, \qquad \frac{1}{4} \leq m < 1,$$

where e is an integer, because then

$$\sqrt{A} = \sqrt{m} \times 2^e.$$

Thus, an arbitrary square root problem reduces to the problem of computing the square root of a number in the range $[.25, 1]$.

A good initial guess for the reduced range problem can be obtained by linear interpolation with

$$L(m) = (1 + 2m)/3.$$

This function interpolates $f(m) = \sqrt{m}$ at $m = .25$ and $m = 1$. (See Figure 8.2 on the next page.)

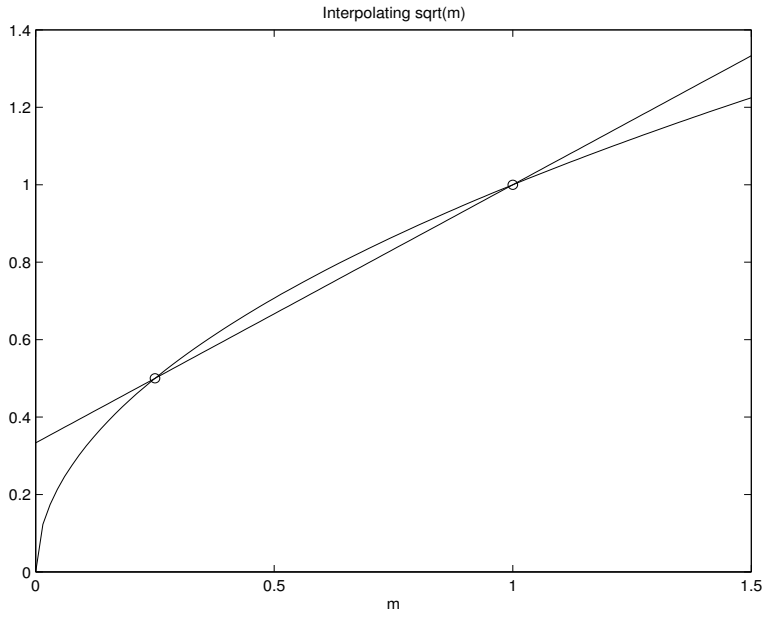

FIGURE 8.2 Approximating $\sqrt{m}$ with $L(m) = (1 + 2m)/3$

Moreover, it can be shown that

$$|L(m) - \sqrt{m}| \leq .05 \tag{8.1}$$

for all m in the interval. It follows that $L(m)$ can serve as a reasonable starting value.

This brings us to the analysis of the iteration itself. We can use the equation

$$x_+ - \sqrt{m} = \frac{1}{2}\left(x_c + \frac{m}{x_c}\right) - \sqrt{m} \;=\; \frac{1}{2}\left(\frac{x_c - \sqrt{m}}{\sqrt{x_c}}\right)^2 \tag{8.2}$$

to bound the error after k steps. Switching to subscripts, let $x_0 = L(m)$ be the initial guess and let x_k be the kth iterate. If $e_k = x_k - \sqrt{m}$, then (8.2) says that

$$e_{k+1} = \frac{1}{2x_k} e_k^2.$$

It can be shown that the iterates x_k are always in the interval $[.5, 1]$, and so $e_{k+1} \leq e_k^2$. Thus,

$$e_4 \leq e_3^2 \leq e_2^4 \leq e_1^8 \leq e_0^{16} \leq (.05)^{16},$$

implying that four steps are adequate if the unit roundoff is in the vicinity of 10^{-16}. Combining the range reduction with this analysis, we obtain the following implementation:

```
  function x = MySqrt(A)
% x = MySqrt(A)
% A is a non-negative real and x is its square root.

if A==0
   x = 0;
else
   TwoPower = 1;
   m = A;
   while m>=1,    m = m/4; TwoPower = 2*TwoPower; end
   while m < .25, m = m*4; TwoPower = TwoPower/2; end
   % sqrt(A) = sqrt(m)*TwoPower
   x = (1+2*m)/3;
   for k=1:4
      x = (x + (m/x))/2;
   end
   x = x*TwoPower;
end
```

Notice the special handling of the $A = 0$ case. The script `ShowMySqrt` plots the relative error of `MySqrt` across any specified interval. (See Figure 8.3.)

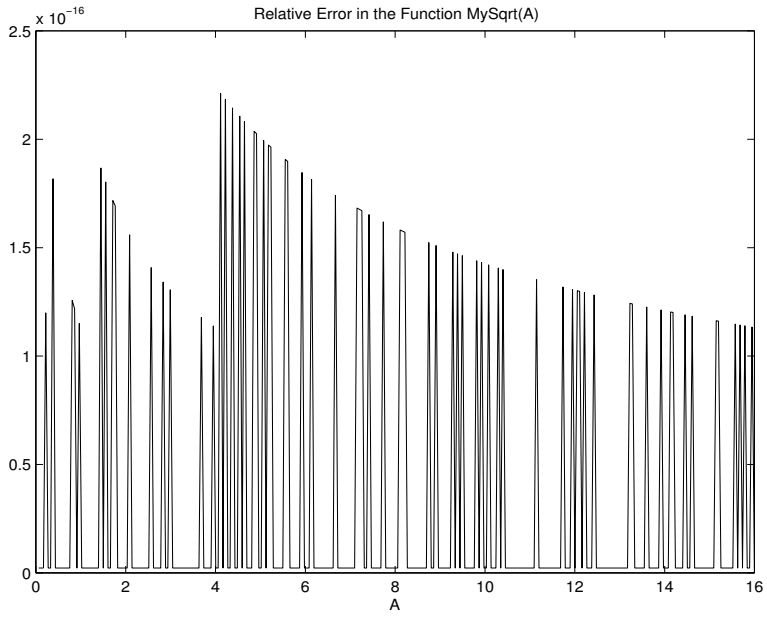

FIGURE 8.3 Relative error in `MySqrt`

Problems

P8.1.1 Another idea for a starting value in `MySqrt` is to choose the parameters s and t so that the maximum value of $|s + t\alpha - \sqrt{\alpha}|$ is minimized on $[.25, 1]$. Implement this change in `MySqrt` and examine the error.

P8.1.2 Repeat the previous problem with s and t chosen to minimize

$$e(s,t) = \int_{.25}^{1} [s + tm - \sqrt{m}]^2 dm.$$

P8.1.3 Prove (8.1).

P8.1.4 Vectorize `MySqrt` so that it can handle the case when A is a matrix of nonnegative numbers.

8.1.2 Bisection

If a continuous function $f(x)$ changes sign on an interval, then it must have at least one root in the interval. This fact can be used to produce a sequence of ever-smaller intervals that each bracket a root of $f(x)$. To see this, assume that $f(a)f(b) \leq 0$ and let $m = (a+b)/2$, the midpoint of $[a, b]$. It follows that either $f(a)f(m) \leq 0$ or $f(m)f(b) \leq 0$. In the former case we know that there is a root in $[a, m]$, while the latter situation implies that there is a root in $[m, b]$. In either case, we are left with a half-sized bracketing interval. The halving process can continue until the current interval is shorter than a designated positive tolerance `delta`:

```
while abs(a-b) > delta
   if f(a)*f((a+b)/2) <= 0
      b = (a+b)/2;
   else
      a = (a+b)/2;
   end
end
root = (a+b)/2;
```

This is "version zero" of the method of *bisection.* It needs to be cleaned up in two ways. First, the `while`-loop may never terminate if `delta` is smaller than the spacing of the floating point numbers between a and b. To rectify this, we change the `while`-loop control to

```
while abs(a-b) > delta+eps*max(abs(a),abs(b))
```

This guarantees termination even if `delta` is too small. The second flaw in the preceding implementation is that it requires two function evaluations per iteration. However, with a little manipulation, it is possible to reduce this overhead to just one f-evaluation per step. Overall we obtain

```
  function root = Bisection(fname,a,b,delta)
% root = Bisection(fname,a,b,delta)
% fname is a string that names a continuous function  f(x) of a single variable.
% a and b define an interval [a,b] and f(a)f(b) < 0. delta is a non-negative real.
% root is the midpoint of an interval [alpha,beta] with the property that
% f(alpha)f(beta)<=0 and |beta-alpha| <= delta+eps*max(|alpha|,|beta|)
```

```
fa = feval(fname,a);
fb = feval(fname,b);
if fa*fb > 0
   disp('Initial interval is not bracketing.')
   return
end
if nargin==3
   delta = 0;
end
while abs(a-b) > delta+eps*max(abs(a),abs(b))
   mid = (a+b)/2;
   fmid = feval(fname,mid);
   if fa*fmid<=0
      % There is a root in [a,mid].
      b  = mid;
      fb = fmid;
   else
      % There is a root in [mid,b].
      a  = mid;
      fa = fmid;
   end
end
root = (a+b)/2;
```

The script `ShowBisect` can be used to trace the progress of the iteration. It should be stressed that the time required by a root-finder like `Bisection` is proportional to the *number of function-evaluations*. The arithmetic that takes place outside of the f-evaluations is typically insignificant.

If $[a_k, b_k]$ is the k-th bracketing interval, then $|a_k - b_k| \leq |a_0 - b_0|/2^k$ showing that convergence is guaranteed. Thinking of $x_k = (a_k + b_k)/2$ as the kth iterate, we see that there exists a root x_* for $f(x)$, with the property that

$$|x_k - x_*| \;\leq\; \frac{|a_0 - b_0|}{2^{k+1}}.$$

The error bound is halved at each step. This is an example of *linear convergence*. In general, a sequence $\{x_k\}$ converges to x_* linearly if there is a constant c in the interval $[0, 1)$ and an integer k_0 such that

$$|x_{k+1} - x_*| \;\leq\; c\,|x_k - x_*|$$

for all $k \geq k_0$. To illustrate what linear convergence is like, we apply `Bisection` to $f(x) = \tan(x/4) - 1$ with initial interval $[a_0, b_0] = [2, 4]$:

k	a_k	b_k	$b_k - a_k$
0	2.00000000000000	4.00000000000000	2.00000000000000
1	3.00000000000000	4.00000000000000	1.00000000000000
2	3.00000000000000	3.50000000000000	0.50000000000000
3	3.00000000000000	3.25000000000000	0.25000000000000
4	3.12500000000000	3.25000000000000	0.12500000000000
5	3.12500000000000	3.18750000000000	0.06250000000000
⋮	⋮	⋮	⋮
43	3.14159265358967	3.14159265358990	0.00000000000023
44	3.14159265358978	3.14159265358990	0.00000000000011
45	3.14159265358978	3.14159265358984	0.00000000000006
46	3.14159265358978	3.14159265358981	0.00000000000003
47	3.14159265358978	3.14159265358980	0.00000000000001

Notice that a new digit of π is acquired every three or so iterations. This kind of uniform acquisition of significant digits is the hallmark of methods that converge linearly.

Problems

P8.1.5 What can go wrong in `Bisection` if the comparison `fa*fmid<=0` is changed to `fa*fmid<0`? Give an example.

P8.1.6 Consider the following recursive formulation of bisection:

```
function x = rBisection(fname,a,b,delta)
if abs(a-b) <= delta
   x = (a+b)/2;
else
   mid = (a+b)/2;
   if eval([ fname '(a)*' fname '(c) <=0])
      x = rBisection(fname,a,c,delta);
   else
      x = rBisection(fname,c,b,delta);
   end
end
```

Modify this procedure so that it does not involve redundant function evaluations.

P8.1.7 Complete the following MATLAB function:

```
  function r = MiddleRoot(a,b,c,d)
% The cubic equation ax^3 + bx^2 + cx + d = 0 has three real distinct roots.
% r is within 100*EPS of the middle root.
```

Write a script that prints the middle root of all cubics of the form $x^3 + bx^2 + cx + d$ where b, c, and d are integers from the interval $[-2, 2]$ so that the resulting cubic has three distinct real roots. Note that a cubic $c(x)$ has three real distinct roots if $c'(x)$ has two real distinct roots r_1 and r_2 and $c(r_1)c(r_2) < 0$. In this case, the interval defined by r_1 and r_2 brackets the middle root. Check your results with the MATLAB polynomial root-finder `roots`.

P8.1.8 Assume that the function $f(x)$ is positive on $[0, 1]$ and that

$$\int_0^1 f(x)dx = 1.$$

Write a complete, efficient MATLAB fragment that uses bisection to compute z so that

$$\left| \int_0^z f(x)dx - \frac{1}{2} \right| \leq 10^{-14}.$$

Assume `f` is an available MATLAB function. Compute all integrals via the MATLAB function `Quad8(fname,a,b,tol)` with `tol = 10*EPS`.

P8.1.9 Let A be the 8-by-8 pentadiagonal matrix

$$A = \begin{bmatrix} 6 & -4 & 1 & 0 & 0 & 0 & 0 & 0 \\ -4 & 6 & -4 & 1 & 0 & 0 & 0 & 0 \\ 1 & -4 & 6 & -4 & 1 & 0 & 0 & 0 \\ 0 & 1 & -4 & 6 & -4 & 1 & 0 & 0 \\ 0 & 0 & 1 & -4 & 6 & -4 & 1 & 0 \\ 0 & 0 & 0 & 1 & -4 & 6 & -4 & 1 \\ 0 & 0 & 0 & 0 & 1 & -4 & 6 & -4 \\ 0 & 0 & 0 & 0 & 0 & 1 & -4 & 6 \end{bmatrix}.$$

To within 12 significant digits, determine $\alpha > 0$ so that the solution to $(A + \alpha I)x = ones(8,1)$ satisfies $x^T x = 1$. (Here, I is the 8×8 identity.)

P8.1.10 Assume that

- the function `F(t)` implements a continuous function F:$\mathbb{R} \rightarrow \mathbb{R}^n$;
- `A` is a given n-by-n nonsingular matrix;
- t_0 and and t_1 are given scalars that satisfy $\| A^{-1}F(t_0) \|_2 < 1$ and $\| A^{-1}F(t_0) \|_2 > 1$.

Write a script that assigns to `root` a value t with the property that $|t - t_*| \leq 10^{-6}$, where t_* is in the interval $[t0, t1]$ and satisfies $\| A^{-1}F(t_*) \|_2 = 1$. Make effective use of the above functions. Show the complete function that is passed to `Bisection`. Ignore the effect of rounding errors and make effective use of `LTriSol`, `UTriSol`, `GEpiv`, and `Bisection`. Refer to the function that you pass to `Bisection` as `MyF` and assume that rounding errors are made whenever it is evaluated. Suppose $\text{cond}(A) = 10^5$ and that `EPS` $= 10^{-17}$. What can you say about the accuracy of a `MyF` evaluation and about the claim that the value assigned to `root` is within 10^{-6} of a true `MyF` zero?

8.1.3 The Newton Method Idea

Suppose we have the value of a function $f(x)$ and its derivative $f'(x)$ at $x = x_c$. The tangent line

$$L_c(x) = f(x_c) + (x - x_c)f'(x_c)$$

can be thought of as a linear model of the function at this point. (See Figure 8.4 on the next page.) The zero x_+ of $L_c(x)$ is given by

$$x_+ = x_c - \frac{f(x_c)}{f'(x_c)}.$$

If $L_c(x)$ is a good model over a sufficiently wide range of values, then x_+ should be a good approximation to a root of f. The repeated use of the x_+ update formula defines the *Newton framework*:

```
xc = input('Enter starting value:');
fc = feval(fname,xc);
fpc = feval(fpname,xc);
while input('Newton step? (0=no, 1=yes)')
   xnew = xc - fc/fpc;
   xc = xnew;
   fc = feval(fname,xc);
   fpc = feval(fpname,xc);
end
```

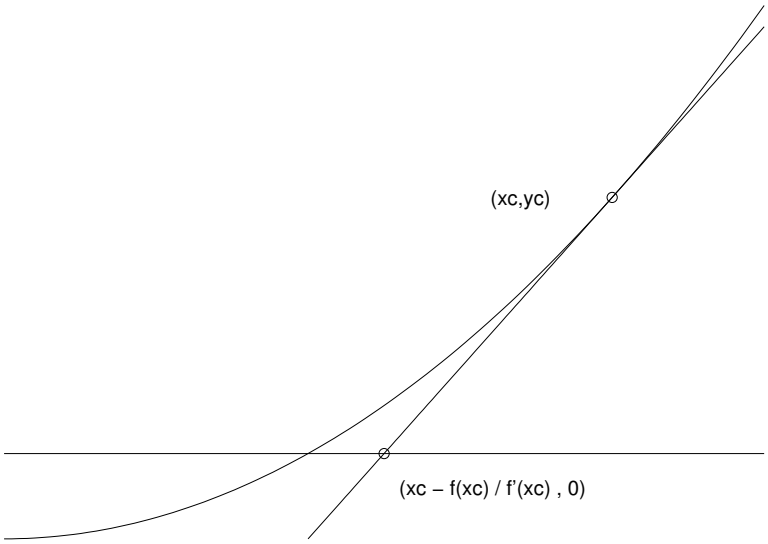

FIGURE 8.4 Modeling a nonlinear function

This assumes that `fname` and `fpname` are strings that house respectively the name of a function and the name of its derivative (e.g., `fname = 'sin'; fpname = 'cos'`). As an illustration, we apply the Newton framework to the problem of finding a zero of the function $f(x) = \tan(x/4) - 1$:

k	x_k	$\lvert x_k - \pi \rvert$
0	1.00000000000000	2.14159265358979
1	3.79631404657234	0.65472139298255
2	3.25943543617547	0.11784278258568
3	3.14513155420752	0.00353890061772
4	3.14159578639006	0.00000313280027
5	3.14159265359225	0.00000000000245
6	3.14159265358979	0.00000000000000

This should be compared to the 47 bisection steps required to produce the same accuracy. Apparently, after a few iterations we reach a stage at which the error is approximately squared at each iteration. This characterizes methods that converge *quadratically*. For a sequence with this property, there exists an integer k_0 and a positive constant c such that

$$|x_{k+1} - x_*| \leq c|x_k - x_*|^2$$

for all $k \geq k_0$. Quadratic convergence is a much-sought-after commodity in the root-finding business.

However, unlike the plodding but guaranteed-to-converge method of bisection, big things can go wrong during the Newton iteration. There is no guarantee that x_+ is closer to a root than x_c. After all, it is produced by a linear model, and f may be very nonlinear. Moreover, a small value for $f'(x_*)$ works against rapid convergence. If the first derivative is small at the solution, then the tangent line on which the Newton step is based is nearly horizontal and it has trouble predicting x_*. Here is a table showing the number of iterations that are required to zero the function $f(x) = x^2 - a^2$ with starting value $x = 2$:

a	Iterations
10^0	5
10^{-4}	18
10^{-8}	31
10^{-12}	43
0	50

Note that $f'(x_*) = 2a$ and that the rate of convergence drifts from strongly quadratic to essentially linear as a decreases from 1 to 0.

But even worse things can happen because of small derivatives. If $f'(x_c)$ is small relative to $f(x_c)$, then the Newton step can transport us to a region far away from a root. The classic example of this is the function $f(x) = \arctan(x)$. Since $f'(x) = 1/(1 + x^2)$, the Newton update has the form $x_+ = x_c - (1+x_c^2)\arctan(x_c)$. It can be shown that if $|x_c| > 1.3917$, then $|x_+| > |x_c|$. This implies that the iteration *diverges* for starting values outside $[-1.3917, 1.3917]$.

Explore the reliability of the Newton iteration by experimenting with the script `ShowNewton`. To guarantee convergence for any starting value in the vicinity of a root, you will discover that f cannot be too nonlinear and that the f' must be bounded away from zero. The following theorem makes this precise.

Theorem 8 *Suppose $f(x)$ and $f'(x)$ are defined on an interval $I = [x_* - \mu, x_* + \mu]$, where $f(x_*) = 0$, $\mu > 0$, and positive constants ρ and δ exist such that*

(1) $|f'(x)| \geq \rho$ *for all* $x \in I$.
(2) $|f'(x) - f'(y)| \leq \delta|x - y|$ *for all* $x, y \in I$.
(3) $\mu \leq \rho/\delta$.

If $x_c \in I$, then $x_+ = x_c - f(x_c)/f'(x_c) \in I$ and

$$|x_+ - x_*| \;\leq\; \frac{\delta}{2\rho}\,|x_c - x_*|^2 \;\leq\; \frac{1}{2}\,|x_c - x_*|\,.$$

Thus, for any $x_c \in I$, a Newton step more than halves the distance to x_*, thereby guaranteeing convergence for any starting value in I. The rate of convergence is quadratic.

Proof From the fundamental theorem of calculus

$$-f(x_c) = f(x_*) - f(x_c) = \int_{x_c}^{x_*} f'(z)dz$$

and so

$$-f(x_c) - (x_* - x_c)f'(x_c) = \int_{x_c}^{x_*} (f'(z) - f'(x_c))\,dz.$$

Dividing both sides by $f'(x_c)$ (which we know is nonzero), we get

$$\left(x_c - \frac{f(x_c)}{f'(x_c)}\right) - x_* \;=\; \frac{1}{f'(x_c)} \int_{x_c}^{x_*} (f'(z) - f'(x_c))dz.$$

Taking absolute values, using the definition of x_+, and invoking (1) and (2) we obtain

$$|x_+ - x_*| \;\leq\; \frac{1}{\rho} \int_{x_c}^{x_*} |f'(z) - f'(x_c)|\, dz \;\leq\; \frac{\delta}{\rho} \int_{x_c}^{x_*} |z - x_c|\, dz \;=\; \frac{\delta}{2\rho} |x_c - x_*|^2 .$$

Since $x_c \in I$, it follows that $|x_c - x_*| \leq \mu$, and so by (3) we have

$$\frac{\delta}{2\rho} |x_c - x_*|^2 \;\leq\; \frac{1}{2} \frac{\delta\mu}{\rho} |x_c - x_*| \;\leq\; \frac{1}{2} |x_c - x_*|$$

This completes the proof since it shows that $x_+ \in I$. □

The theorem shows that if (1) f' does not change sign in a neighborhood of x_*, (2) f is not too nonlinear, and (3) the Newton iteration starts close enough to x_*, then convergence is guaranteed and at a quadratic rate.

For all but the simplest functions, it may be impossible to verify that the conditions of the theorem apply. A general-purpose, Newton-based nonlinear equation solver requires careful packaging in order for it to be useful in practice. This includes the intelligent handling of termination. Unlike bisection, where the current interval length bounds the error, we have to rely on heuristics in order to conclude that an iterate is acceptable as a computed root. One possibility is to quit as soon as $|x_+ - x_c|$ is small enough. After all, if $\lim x_k = x_*$, then $\lim |x_{k+1} - x_k| = 0$. But the converse does not necessarily follow. If $f(x) = \tan(x)$ and $x_c = \pi/2 - 10^{-5}$, then $|x_+ - x_c| \approx 10^{-15}$ even though the nearest root is at $x = 0$. Alternatively, we could terminate as soon as $|f(x_c)|$ is small. But this does *not* mean that x_c is close to a root. For example, if $f(x) = (x-1)^{10}$, then $x_c = 1.1$ makes f very small even though it is relatively far away from the root $x_* = 1$.

8.1.4 Globalizing the Newton Iteration

The difficulties just discussed can be handled with success by combining the Newton and bisection ideas in a way that captures the best features of each framework. At the beginning of a step we assume that we have a bracketing interval $[a, b]$ and that x_c equals one of the endpoints. If

$$x_+ = x_c - \frac{f(x_c)}{f'(x_c)} \;\in\; [a, b],$$

then we proceed with either $[a, x_+]$ or $[x_+, b]$, whichever is bracketing. The new x_c equals x_+. If the Newton step carries us out of $[a, b]$, then we take a bisection step setting the new x_c to $(a + b)/2$. To check whether the Newton step stays in $[a, b]$, we assume the availability of

```
  function ok = StepIsIn(x,fx,fpx,a,b)
% ok = StepIsIn(x,fx,fpx,a,b)
% Yields 1 if the next Newton iterate is in [a,b] and 0 otherwise.
% x is the current iterate, fx is the value of f at x, and fpx is
% the value of f' at x.
```

```
if fpx > 0,     ok = ((a-x)*fpx <= -fx) & (-fx <= (b-x)*fpx);
elseif fpx < 0, ok = ((a-x)*fpx >= -fx) & (-fx >= (b-x)*fpx);
else            ok = 0;
end
```

To guarantee termination, we bring the iteration to a halt if any of the following three conditions hold:

- The length of the current bracketing interval is less than `tolx`, a specified nonnegative tolerance. This ensures that the computed root is within `tolx` of a true root. It does *not* guarantee that f is small. For example, if $f(x) = (x-1)^{.1}$, then $x_c = 1.0000000001$ is very close to the root $x_* = 1$, but $f(x_c)$ is not particularly small.
- The absolute value of $f(x_c)$ is less than or equal to `tolf`, a specified nonnegative tolerance. This does *not* mean that x_c is close to a true root. For example, if $f(x) = (x-1)^{10}$, then $x_c = 1.1$ makes f very small even though it is relatively far away from the root $x_* = 1$.
- The number of f-evaluations exceeds a specified positive integer `nEvalsMax`. This means that neither of the preceding two conditions is satisfied.

Putting it all together, we obtain

```
  function [x,fx,nEvals,aF,bF] = GlobalNewton(fName,fpName,a,b,tolx,tolf,nEvalsMax)
% [ x,fx,nEvals,aF,bF] = GlobalNewton(fName,fpName,a,b,tolx,tolf,nEvalsMax)
%
%   fName       string that names a function f(x).
%   fpName      string that names the derivative function f'(x).
%   a,b         A root of f(x) is sought in the interval [a,b]
%               and f(a)*f(b)<=0.
%   tolx,tolf   Nonnegative termination criteria.
%   nEvalsMax   Maximum number of derivative evaluations.
%
%   x          An approximate zero of f.
%   fx         The value of f at x.
%   nEvals     The number of derivative evaluations required.
%   aF,bF      The final bracketing interval is [aF,bF].
%
%   The iteration terminates as soon as x is with tolx of a true zero or
%   if |f(x)|<= tolf or after nEvalMax f-evaluations

fa  = feval(fName,a);
fb  = feval(fName,b);
if fa*fb>0
   disp('Initial interval not bracketing.')
   return
end
```

```
x   = a;
fx  = feval(fName,x);
fpx = feval(fpName,x);
disp(sprintf('%20.15f  %20.15f    %20.15f',a,x,b))

nEvals = 1;
while (abs(a-b) > tolx ) & (abs(fx) > tolf) & ((nEvals<nEvalsMax) | (nEvals==1))
   %[a,b] brackets a root and x = a or x = b.
   if StepIsIn(x,fx,fpx,a,b)
      %Take Newton Step
      disp('Newton')
      x   = x-fx/fpx;
   else
      %Take a Bisection Step:
      disp('Bisection')
      x = (a+b)/2;
   end
   fx  = feval(fName,x);
   fpx = feval(fpName,x);
   nEvals = nEvals+1;
   if fa*fx<=0
      % There is a root in [a,x]. Bring in right endpoint.
      b  = x;
      fb = fx;
   else
      % There is a root in [x,b]. Bring in left endpoint.
      a  = x;
      fa = fx;
   end
   disp(sprintf('%20.15f  %20.15f    %20.15f',a,x,b))
end
aF = a;
bF = b;
```

In a typical situation, a number of bisection steps are taken before the Newton iteration takes over. Here is what happens when we try to find a zero of $f(x) = \sin(x)$ with initial bracketing interval $[-7\pi/2, 15\pi + .1]$:

```
  Step                a                    x                    b
-------------------------------------------------------------------------
<Start>     -10.995574287564276  -10.995574287564276   47.223889803846895
Bisection   -10.995574287564276   18.114157758141310   18.114157758141310
Bisection   -10.995574287564276    3.559291735288517    3.559291735288517
Newton        3.115476144648328    3.115476144648328    3.559291735288517
Newton        3.115476144648328    3.141598592990409    3.141598592990409
Newton        3.141592653589793    3.141592653589793    3.141598592990409
```

Problems

P8.1.11 The function $f(z) = z^4 - 1$ has four roots: $r_1 = 1$, $r_2 = i$, $r_3 = -1$, and $r_4 = -i$. The Newton iteration is defined for complex numbers, so if we repeatedly apply the update

$$z_+ = z_c - \frac{z_c^4 - 1}{4z_c^3} = \frac{1}{4}\left(3z_c + \frac{1}{z_c^3}\right),$$

the iterates converge to one of the four roots. **(a)** Determine the largest value of ρ so that if z_0 is within ρ of a root, then Newton iteration converges to the same root. **(b)** Write a function `[steps,i] = WhichRoot(z0)` that returns the index i of the root obtained by the Newton iteration when `z0` is the starting value. The number of Newton steps required to get with ρ of r_i should be returned in `steps`. **(c)** We say that the line segment that connects z_1 and z_2 has property *two root* if `WhichRoot(z1)` and `WhichRoot(z2)` indicate convergence to two different roots. Write a function

```
  function [w1,w2] = Close(z1,z2,delta)
% z1 and z2 are complex scalars that define a two root segment and delta
% is a positive real scalar.
% w1 and w2 are complex scalars that define a two-root segment and satisfy
% |w1-w2| <= delta*|z1-z2|
```

P8.1.12 In Newton's method, x_{new} is a zero of a linear approximation $L_c(x) = a_c + b_c(x - x_c)$ that "models" the function f at the current iterate x_c. Develop a Newton-like method that is based on a rational model $R_c(x) = a_c + 1/(x - x_c$ of f at x_c. Determine the model parameters a_c and b_c so that $R_c(x_c = f(x_c)$ and $R_c'(x_c) = f'(x_c)$. Determine x_+ from $R_c(x_+) = 0$.

8.1.5 Avoiding Derivatives

A problem with the Newton framework is that it requires software for both f and its derivative f'. In many instances, the latter is difficult or impossible to encode. A way around this is to approximate the required derivative with an appropriate divided difference:

$$f'(x_c) \approx \frac{f(x_c + \delta_c) - f(x_c)}{\delta_c}.$$

This leads to the *finite difference Newton framework*:

```
fval = feval(fname,x);
Choose delta.
fpval = (feval(fname,x+delta) - fval)/delta']);
x = x - fval/fpval;
```

The choice of δ_c requires some care if quadratic-like convergence is to be ensured. A good heuristic would involve $|x_c|$, $|f(x_c)|$, and the machine precision. Note that a step requires two f-evaluations.

In the *secant method*, $f'(x_c)$ is approximated with the divided difference

$$f'(x_c) \approx \frac{f(x_c) - f(x_-)}{x_c - x_-},$$

where x_- is the iterate before x_c. A step in the secant iteration requires a single function evaluation:

```
fpc = (fc - f_)/(xc - x_);
xnew = xc - fc/fpc;
x_ = xc;
f_ = fc;
xc = xnew;
fc = feval(fname,xc);
```

The method requires two starting values. Typically, an initial guess $\tilde{x}$ and a slight perturbation $\tilde{x} + \delta$ are used.

If the iteration converges, then it is usually the case that

$$|x_{k+1} - x_*| \ \leq \ c|x_k - x_*|^r,$$

where

$$r = \frac{1+\sqrt{5}}{2} \approx 1.6$$

for all sufficiently large k. Applying the secant method to the function $f(x) = \tan(x/4) - 1$ with starting values $x_0 = 1$ and $x_1 = 2$, we get

k	x_k	$\lvert x_k - \pi \rvert$
0	1.00000000000000	2.14159265358979
1	2.00000000000000	1.14159265358979
2	3.55930926415136	0.41771661056157
3	3.02848476491863	0.11310788867116
4	3.12946888739926	0.01212376619053
5	3.14193188940880	0.00033923581901
6	3.14159162639551	0.00000102719428
7	3.14159265350268	0.00000000008712
8	3.14159265358979	0.00000000000000

Thus, there is only a small increase in the number of iterations compared to the Newton method. Note that two starting values are required. A finite difference Newton step could be used to get the extra function value that is necessary to initiate the iteration.

It is important to recognize that the total number of function evaluations required by a root-finding framework depends on the rate of convergence of the underlying method *and* on how long it takes for the iterates to get close enough for the local convergence theory to "take hold." This makes it impossible, for example, to assert that the Newton approach is quicker than the secant approach. There are lots of application-specific factors that enter the efficiency equation: quality of initial guess, the second derivative behavior of f, the relative cost of f-evaluations and f'-evaluations, etc.

Problems

P8.1.13 Write a script file that illustrates the finite difference Newton method framework. Experiment with the choice of the difference parameter δ_c.

P8.1.14 Write a `ShowSecant` script file that illustrates the secant method framework.

8.1.6 The MATLAB `fzero` Function

The MATLAB function `fzero` is a general-purpose root finder that does not require derivatives. A simple call involves only the name of the function and a starting value. For example, if

```
function y = decay(t)
y = 10*exp(-3*t) +2*exp(-2*t) -6;
```

is available, then

```
r = fzero('decay',1)
```

assigns the value of the root .24620829278302 to `r`.

If all goes well, then `fzero` returns a computed root r that is within $|r|$`EPS` of a true root. A less stringent error tolerance can be used by specifying a third parameter. The call

```
r = fzero('decay',1,.001)
```

returns .24666930469558, which is correct to three significant digits. A nonzero fourth input parameter generates a trace of the `fzero iteration`, e.g.,

```
r = fzero('decay',1,.001,1)
```

The function passed to `fzero` must be a function of a single scalar variable. However, the function can have parameters. For example, if

```
function y = cubic(x,a,b,c,d)
y = ((a*x + b)*x + c)*x + d;
```

then

```
xStart = 1; aVal = 2; bVal = -3; cVal = 4; dVal = 2;
root = fzero('cubic',xStart,[],[],aVal,bVal,cVal,dVal)
```

will try to find a zero of the cubic $2x^3 - 3x^2 + 4x + 2$ near $x = 1$, using the default values for the tolerance and trace.

As a more involved application of `fzero`, suppose the locations of planets M and E at time t are given by

$$x_M(t) = -11.9084 + 57.9117\cos(2\pi t/87.97) \tag{8.3}$$
$$y_M(t) = 56.6741\sin(2\pi t/87.97) \tag{8.4}$$

and

$$x_E(t) = -2.4987 + 149.6041\cos(2\pi t/365.25) \tag{8.5}$$
$$y_E(t) = 149.5832\sin(2\pi t/365.25) \tag{8.6}$$

These are crude impersonations of Mercury and Earth. Both orbits are elliptical with one focus, the Sun, situated at $(0, 0)$. To an observer on E, M is in *conjunction* if it is located on the Sun-to-Earth line segment. Clearly there is a conjunction at $t = 0$. Our goal is to compute the time of the next 10 conjunctions and the spacing between them. To that end, we define the function

```
  function s = SineMercEarth(t)
% s = SineMercEarth(t)
% The sine of the Mercury-Sun-Earth angle at time t

% Mercury location:
xm = -11.9084 +  57.9117*cos(2*pi*t/87.97);
ym =             56.6741*sin(2*pi*t/87.97);

% Earth location:
xe =  -2.4987 + 149.6041*cos(2*pi*t/365.25);
ye =            149.5832*sin(2*pi*t/365.25);

s = (xm.*ye - xe.*ym)./(sqrt(xm.^2 + ym.^2).*sqrt(xe.^2 + ye.^2));
```

and note that at the time of conjunction, this function is zero. Before we go after the next 10 conjunction times, we plot `SineMercEarth` to get a rough idea about the spacing of the conjunctions. (See Figure 8.5.) At first glance it looks like one occurs every 60 or so days. However, `SineMercEarth` is also zero whenever the two planets are aligned with the sun in the middle. (M is then said to be in *opposition*.) It follows that the spacing between conjunctions is about 120 days and so it is reasonable to call `fzero` with starting value $120k$ in order to compute the time of the kth conjunction. Here is a script file that permits the input of an arbitrary spacing factor and then computes the conjunctions accordingly:

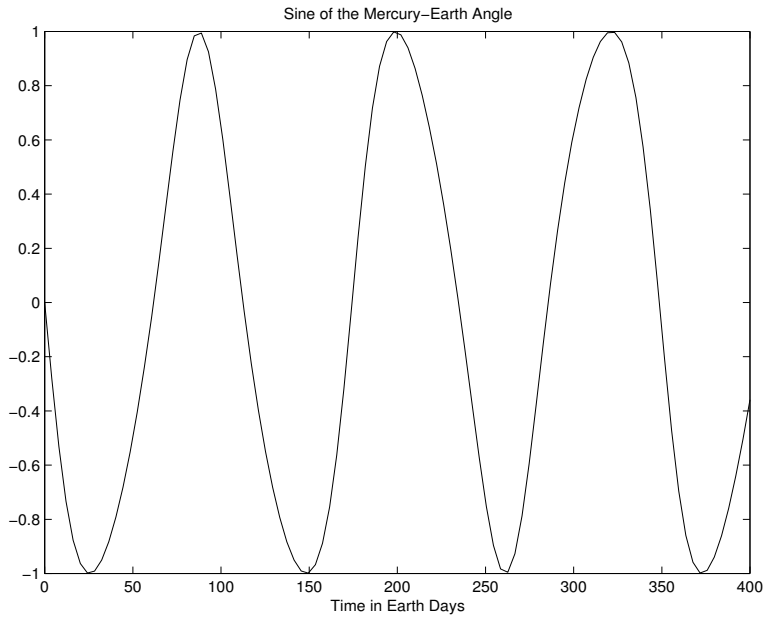

FIGURE 8.5 Sine of the Mercury-Sun-Earth angle

```
% Script File: FindConj
% Estimates spacing between Mercury-Earth conjunctions

clc
GapEst = input('Spacing Estimate = ');
disp(sprintf('Next ten conjunctions, Spacing Estimate = %8.3f',GapEst))
disp(' ')
t = zeros(11,1);
disp('Conjunction    Time       Spacing')
disp('-----------------------------------')
for k=1:10;
   t(k+1) = fzero('SineMercEarth',k*GapEst);
   disp(sprintf('    %2.0f       %8.3f    %8.3f',k,t(k+1),t(k+1)-t(k)))
end
```

If we respond with a spacing factor equal to 115, then here is the output:

```
Next ten conjunctions, Spacing Estimate =  115.000

Conjunction    Time       Spacing
-----------------------------------
     1        112.476     112.476
     2        234.682     122.206
     3        348.554     113.872
     4        459.986     111.433
     5        581.491     121.505
     6        697.052     115.561
     7        807.815     110.762
     8        928.020     120.206
     9       1045.440     117.420
    10       1155.908     110.467
```

Problems

P8.1.15 Determine the longest possible interval $[L, R]$ so that if `GapEst` is in this interval, then the 10 required conjunctions are computed in `FindConj`.

P8.1.16 Suppose $f(x, y)$ is a continuous function of two variables. Let L be the line segment that connects the given points (a, b) and (c, d). Give a solution procedure that uses `fzero` to compute a point (x_*, y_*) on L with the property that $f(x_*, y_*) = (f(a, b) + f(c, d))/2$.

8.2 Minimizing a Function of a Single Variable

Suppose we want to compute the minimum Mercury-Earth separation over the next 1000 days, given that their respective locations $[(x_M(t), y_M(t))$ and $(x_E(t), y_E(t))]$ are specified by (8.3)

through (8.6). Thus we want to solve the problem

$$\begin{array}{l} \min \quad f(t), \\ t \in S \end{array}$$

where

$$f(t) \;=\; \sqrt{(x_E(t) - x_M(t))^2 \;+\; (y_E(t) - y_M(t))^2}$$

and S is the set $\{t : 0 \leq t \leq 1000\}$. This is an example of an *optimization* problem in one dimension. The function f that we are trying to optimize is called the *objective function*. The set S may be the entire set of real numbers or a subset thereof because of constraints on the independent variable.

The optimization problems considered in this book can be cast in find-the-max or find-the-min terms. We work with the latter formulation, noting that any max problem can be turned into a min problem simply by negating the objective function.

8.2.1 Graphical Search

If function evaluations are cheap, then plotting can be used to narrow the search for a minimizer. We have designed a simple interactive search environment for this purpose, and its specification is as follows:

```
  function [L,R] = GraphSearch(fname,a,b,Save,nfevals)
% [L,R] = GraphSearch(fname,a,b,Save,nfevals)
%
% Graphical search. Produces sequence of plots of the function f(x). The user
% specifies the x-ranges by mouseclicks.
%
% name is a string that names a function f(x) that is defined
% on the interval [a,b].
% nfevals>=2
%
% Save is used to determine how the plots are saved. If Save is
% nonzero, then each plot is saved in a separate figure window.
% If Save is zero or if GraphSearch is called with just three
% arguments, then only the final plot is saved.
%
% [L,R] is the x-range of the final plot. The plots are based on nfevals
% function evaluations.
%
% If GraphSearch is called with less
% than five arguments, then nfevals is set to 100.
```

Let's apply `GraphSearch` to the problem of minimizing the following Mercury-to-Earth separation function:

```
  function s = DistMercEarth(t)
% s = DistMercEarth(t)
% The distance between Mercury and Earth at time t.

% Mercury location:
xm = -11.9084 +  57.9117*cos(2*pi*t/87.97);
ym =             56.6741*sin(2*pi*t/87.97);

% Earth location:
xe =  -2.4987 + 149.6041*cos(2*pi*t/365.25);
ye =            149.5832*sin(2*pi*t/365.25);

s = sqrt((xe-xm).^2 + (ye-ym).^2);
```

The initial plot is displayed in Figure 8.6. Whenever `GraphSearch` displays a plot, it prompts for a new, presumably smaller "interval of interest." Left and right endpoints are obtained by mouseclicks. From Figure 8.6 we see that the minimum Mercury-Earth separation is somewhere in the interval $[900, 950]$. Clicking in the endpoints renders the refined plot shown in Figure 8.7 (on the next page). By repeating the process, we can "zoom" in to relatively short intervals of interest.

Note that our example has many *local minima* across $[0, 1000]$, and this brings up an important aspect of the optimization problem. In some problems, such as the one under consideration, a *global minimum* is sought. In other settings some or all of the *local minima* are of interest.

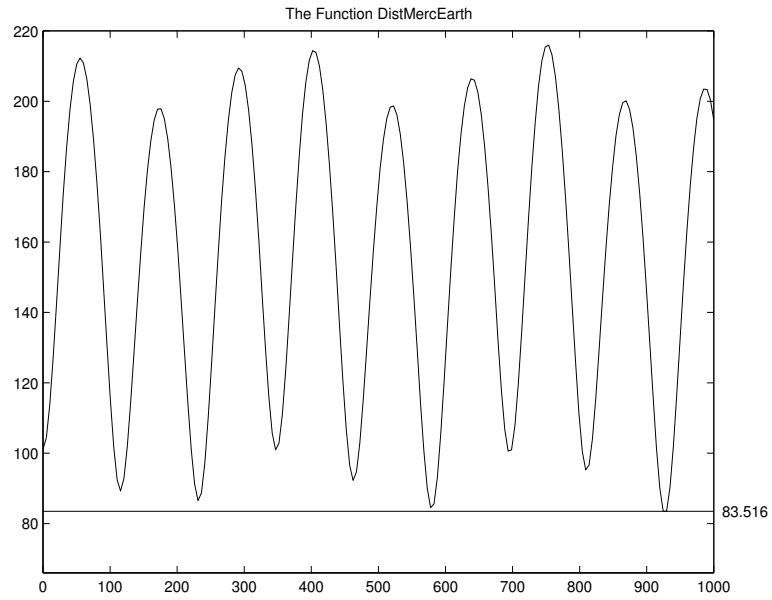

FIGURE 8.6 The `GraphSearch` environment

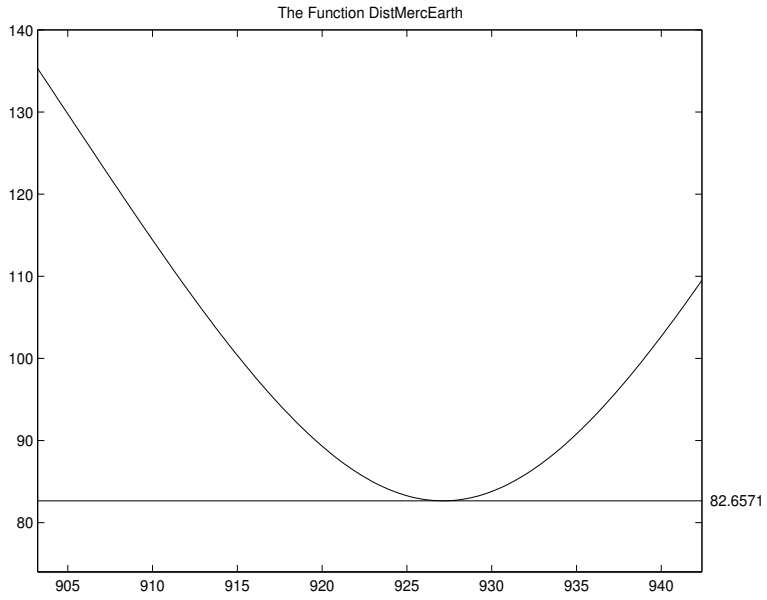

FIGURE 8.7 A refined plot

Detailed knowledge of the objective function is usually required to tackle a global min problem. This may come from plotting or from taking into consideration various aspects of the underlying application. The search for a local minimum is easier to automate and is the next item on our agenda.

Problems

P8.2.1 Modify `GraphSearch` so that on the first plot, an arbitrary number of intervals of interest can be solicited (e.g., $[a_i, b_i]$, $i = 1{:}n$). Each of these should then be graphically searched, with the final plot being saved. Thus, upon completion figures 1 through n should contain the final plots for each of the n searches.

8.2.2 Golden Section Search

A function $f(t)$ is *unimodal* on $[a, b]$ if there is a point $t_* \in [a, b]$ with the property that f is strictly monotone decreasing on $[a, t_*]$ and strictly monotone increasing on $[t_*, b]$. This is just a derivative-free way of saying that f has a unique global minimum at t_*. The function `DistMercEarth` is unimodal on $[900, 950]$.

The method of *golden section search* can be used to find t_*. In this method, an interval $[a, b]$ that brackets t_* is maintained from step to step in addition to a pair of "interior" points c and d that satisfy

$$\begin{aligned} c &= a + r(b - a) \\ d &= a + (1 - r)(b - a). \end{aligned}$$

Here, $0 < r < 1/2$ is a real number to be determined in a special way. If $f(c) \leq f(d)$, then from unimodality it follows that f is increasing in $[d, b]$ and so $t_* \in [a, d]$. On the other hand, if $f(c) > f(d)$, then a similar argument tells us that $t_* \in [c, b]$. This suggests the following maneuver to reduce the search space:

```
%  Assume a < c < d < b where c = a+r*(b-a) and d = a+(1-r)(b-a).
%  fa = f(a), fc = f(c), fd = f(d), and fb = f(b)

if fc<=fd
   b = d; fb = fd;
else
   a = c; fa = fc;
end
c = a+r*(b-a);     fc = f(c);
d = a+(1-r)*(b-a); fd = f(d);
```

After this step the length of the search interval is reduced by a factor of $(1 - r)$. Notice that two function evaluations per step are required, and this is where the careful choice of r comes in. If we can choose r so that (1) in the `fc<=fd` case the new d equals the old c and (2) in the `fc>fd` case the new c equals the old d, then the update step can be written as follows:

```
if fc<=fd
   b = d;             fb = fd;
   d = c;             fd = fc;
   c = a+r*(b-a);     fc = f(c);
else
   a = c;             fa = fc;
   c = d;             fc = fd;
   d = a+(1-r)*(b-a); fd = f(d);
end
```

In this setting only one new function evaluation is required. The value of r sought must satisfy two conditions:

$$\begin{aligned} a + (1 - r)(d - a) &= a + r(b - a) \\ c + r(b - c) &= a + (1 - r)(b - a). \end{aligned}$$

Manipulation of either of these equations shows that

$$\frac{r}{1 - r} = \frac{d - a}{b - a}.$$

From the definition of d, we have $d - a = (1 - r)(b - a)$, and so r must satisfy

$$\frac{r}{1 - r} = 1 - r.$$

This leads to the quadratic equation $1 - 3r + r^2 = 0$. The root we want is

$$r = \frac{3 - \sqrt{5}}{2},$$

and with this choice the interval length reduction factor is

$$1 - r = 1 - \frac{3-\sqrt{5}}{2} = \frac{\sqrt{5}-1}{2} = \frac{1}{r_*} \approx .618,$$

where $r_* = (1+\sqrt{5})/2$ is the well-known *golden ratio*. Packaging these ideas, we obtain

```
  function tmin = Golden(fname,a,b)
% tmin = Golden(fname,a,b)
% Golden Section Search
%
% fname   string that names function  f(t) of a single variable.
% a,b   define an interval [a,b] upon which f is unimodal.
%
% tmin   approximate global minimizer of f on [a,b].

r = (3 - sqrt(5))/2;
c = a + r*(b-a);     fc = feval(fname,c);
d = a + (1-r)*(b-a); fd = feval(fname,d);
while (d-c) > sqrt(eps)*max(abs(c),abs(d))
   if fc >= fd
      z = c + (1-r)*(b-c);
      % [a c d b ] <--- [c d z b]
      a = c;
      c = d; fc = fd;
      d = z; fd = feval(fname,z);
   else
      z = a + r*(d-a);
      % [a c d b ] <--- [a z c d]
      b = d;
      d = c; fd = fc;
      c = z; fc = feval(fname,z);
   end
end
tmin = (c+d)/2;
```

A trace of the (a, c, d, b) when this min-finder is applied to the function `DistMercEarth` is given in Figure 8.8 (on the next page). The figure shows the location of a, b, c, and d at the beginning of the first five steps. Notice how only one new function evaluation is required per step.

An important observation to make about the termination criteria in `Golden` is that the square root of the unit roundoff is used. The rationale for this is as follows: If f' and f'' exist, then

$$f(x_* + \delta) \approx f(x_*) + \frac{\delta^2}{2} f''(x_*),$$

since $f'(x_*) = 0$. If there are $O(\texttt{eps})$ errors in an f-evaluation, then there will be no significant difference between the computed value of f at x_* and the computed value of f at $x_* + \delta$ if

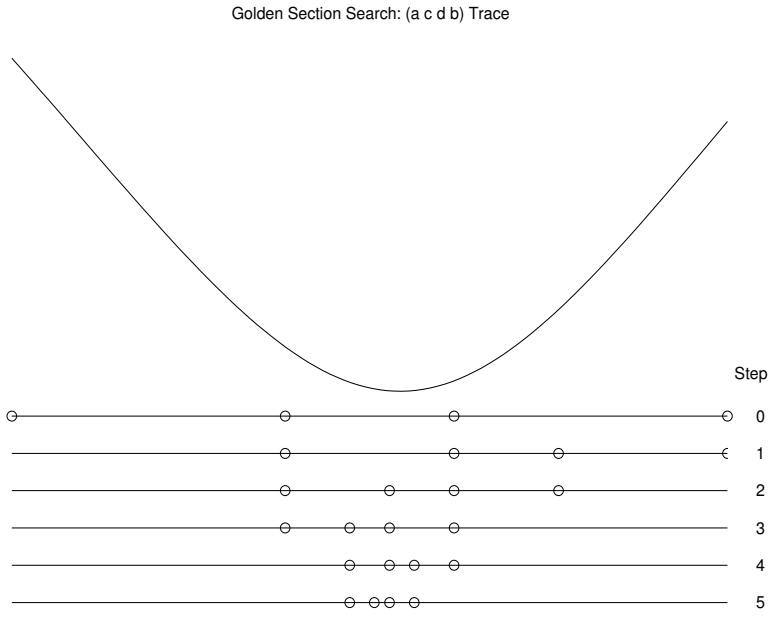

FIGURE 8.8 Golden section search

$\delta^2 \leq$ `eps`. Thus, there is no point iterating further if the relative spacing of the floating numbers is less than $\sqrt{\texttt{eps}}$.

If we use `Golden` to minimize the Mercury-Earth separation function with initial interval $[900, 950]$, then 29 iterations are required, rendering the solution $x_* = 927.1243$

`Golden` always converges, but if f is not unimodal across the initial interval, then the limit point need not have any bearing on the search for even a local minimizer. Unimodality is to `Golden` as the requirement $f(a)f(b) \leq 0$ is to `Bisection`: Without it we cannot guarantee that anything of interest emerges from the iteration.

8.2.3 The MATLAB `fmin` Function

The method of golden section search converges linearly, like the method of bisection. To obtain a more rapidly convergent method, we could use the Newton framework to find zeros of the derivative f', assuming that this function and f'' exist. However, this approach requires the encoding of both the first and second derivative functions, and this can be a time-consuming endeavor.

The MATLAB minimizer `fmin` circumvents this problem but still exhibits quadratic-like convergence. It uses a combination of golden section search and a parabolic fitting method that relies only upon f evaluations. The call

```
tmin = fmin('DistMercEarth',900,950,)
```

applies the method to the Mercury-Earth distance function on the interval $[900, 950]$ and returns the minimizer $t_* = 927.1243$.

At the user's discretion, an "options vector" can be passed to `fmin` specifying various options. For example,

```
options = [1 .00000001];
tmin = fmin('DistMercEarth',900,950,options)
```

generates a trace because `options(1)` is nonzero and sets the x-tolerance to 10^{-8}, overriding the default value 10^{-4}. The essence of the output is reported in the following table:

k	t_k	$f(t_k)$	Step
1	919.0983	909.95980919060	(initialize)
2	930.9016	846.11648380405	golden
3	938.1966	979.73638285186	golden
4	927.1769	826.56580710972	parabolic
5	927.0499	826.56961589590	parabolic
6	927.1244	826.56196213246	parabolic
7	927.1243	826.56196211235	parabolic
8	927.1243	826.56196211254	parabolic
9	927.1242	826.56196211268	parabolic

A call of the form `[tmin options] = fmin(fname,a,b,...)` returns the minimizer in `tmin` and iteration information in the vector `options`. In particular, `options(8)` returns $f(t_*)$ and `options(10)` equals the number of f evaluations required. Here is a script that shows how `fmin` can be used to compute the eight local minima of the Mercury-Earth separation function in the interval $[0, 1000]$:

```
% Script File: ShowFmin
% Illustrates fmin.
clc
tol = .000001;
disp('Local minima of the Mercury-Earth separation function.')
disp(sprintf('\ntol = %8.3e\n\n',tol))
disp(' Initial Interval     tmin        f(tmin)    f evals')
disp('-----------------------------------------------------')
options = zeros(18,1);

for k=1:8
   L = 100+(k-1)*112;
   R = L+112;
   options(2) = tol;
   [tmin options] = fmin('DistMercEarth',L,R,options);
   minfeval = options(8);
   nfevals = options(10);
   disp(sprintf('[%3.0f,%3.0f]%10.5f %10.5f,%6.0f',L,R,tmin,minfeval,nfevals))
end
```

Here is the output obtained by running this script file:

```
Local minima of the Mercury-Earth separation function.
tol = 1.000e-06
      Initial Interval     tmin        f(tmin)    f evals
      ---------------------------------------------------
         [100,212]      115.42354     89.27527      12
         [212,324]      232.09209     86.45270      11
         [324,436]      347.86308    100.80500,     10
         [436,548]      462.96252     92.21594,     10
         [548,660]      579.60462     84.12374,     10
         [660,772]      695.69309     99.91281,      9
         [772,884]      810.54878     94.96463,     10
         [884,996]      927.12431     82.65620,     10
```

8.2.4 A Note on Objective Functions

We close this section with some remarks that touch on symmetry, dimension, and choice of objective function. We peg the discussion to a nice geometric problem: What is the largest tetrahedron whose vertices P_0, P_1, P_2, and P_3 are on the unit sphere $x^2 + y^2 + z^2 = 1$? (A tetrahedron is a polyhedron with four triangular faces.)

The problem statement is ambiguous because it assumes that we have some way of measuring the size of a tetrahedron. Let's work with surface area and see what develops. If

$$\left\{ \begin{array}{c} A_{012} \\ A_{023} \\ A_{031} \\ A_{123} \end{array} \right\} \text{ is the area of the face with vertices } \left\{ \begin{array}{c} P_0, P_1, P_2 \\ P_0, P_2, P_3 \\ P_0, P_3, P_1 \\ P_1, P_2, P_3 \end{array} \right\},$$

then our goal is to maximize

$$A = A_{012} + A_{023} + A_{031} + A_{123}.$$

Since the four vertices each have a "latitude" θ (assume $-\pi/2 \leq \theta \leq \pi/2$) and a "longitude" ϕ (assume $-\pi < \phi \leq \pi$), it looks like this is a problem with eight unknowns since $P_0 = (\theta_0, \phi_0)$, $P_1 = (\theta_1, \phi_1)$, $P_2 = (\theta_2, \phi_2)$, and $P_3 = (\theta_3, \phi_3)$. Moreover, with this formulation there are an infinite number of solutions because any tetrahedron, including the optimal one that we seek, can "slide around" in the sphere. To make the situation less fluid without losing generality, we fix P_0 at the "north pole" and confine P_1 to the zero meridian. With this maneuver, the number of unknowns is reduced to five:

$$\begin{aligned} P_0 &= (\pi/2, 0) \\ P_1 &= (\theta_1, 0) \\ P_2 &= (\theta_2, \phi_2) \\ P_3 &= (\theta_3, \phi_3) \end{aligned}$$

A symmetry argument further reduces the number of free parameters if we think of P_0 as the "apex" and the triangle defined by the other three vertices as the "base." It is clear that for the

optimum tetrahedron, the base vertices P_1, P_2, and P_3 will all have the same latitude and define an equilateral triangle. (This may be a bit of a stretch if you are not familiar enough with solid geometry, but it can be formally verified.) With these observations, the number of unknowns is reduced to one:

$$\begin{aligned} P_0 &= (\pi/2, 0) \\ P_1 &= (\theta, 0) \\ P_2 &= (\theta, 2\pi/3) \\ P_3 &= (\theta, -2\pi/3) \end{aligned}$$

Let $T(\theta)$ be the tetrahedron with these vertices. The function that we must optimize is just a function of a single variable:

$$\begin{aligned} A(\theta) &= A_{012}(\theta) + A_{134}(\theta) + A_{142}(\theta) + A_{234}(\theta) \\ &= \frac{3\sqrt{3}}{4}\cos(\theta)\left(\sqrt{(3\sin(\theta)-5)(\sin(\theta)-1)} \; + \; \cos(\theta)\right), \end{aligned}$$

which is displayed in Figure 8.9. It is clear that there is a unique maximum somewhere in the interval $[-0.5, 0.0]$.

The collapse of the original eight-parameter problem to one with just a single unknown is dramatic and by no means typical of what happens in practice. However, it is a reminder that the exploitation of structure is usually rewarding.

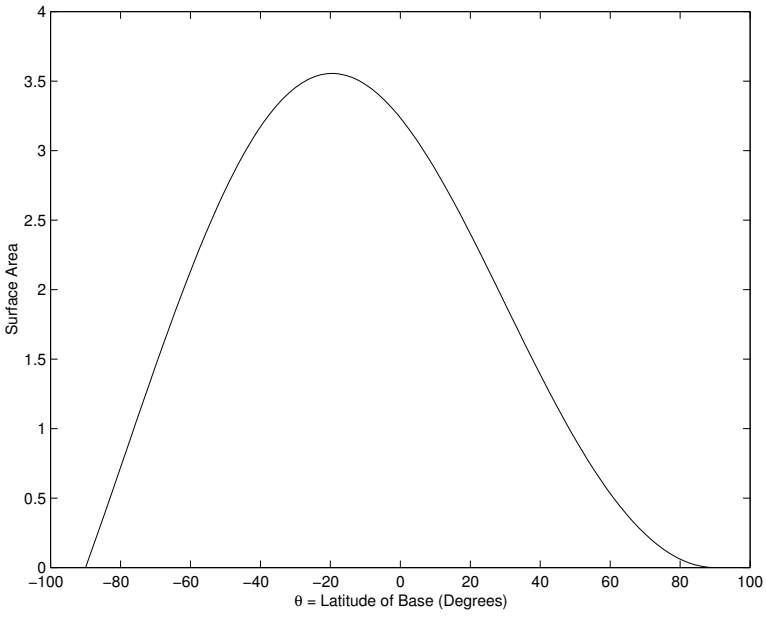

FIGURE 8.9 The surface area of $T(\theta)$

As for finding a solution, we note that maximizing $A(\theta)$ is equivalent to minimizing

$$T_A(\theta) = -\cos(\theta)\left(\sqrt{(3\sin(\theta)-5)(\sin(\theta)-1)}+\cos(\theta)\right),$$

which can be handed over to `fmin` for minimization.

Now let's change the problem so that instead of maximizing surface area, we maximize volume. The same symmetry arguments prevail and it can be shown that

$$V(\theta) = \frac{\sqrt{3}}{4}(1-\sin(\theta)^2)(1-\sin(\theta))$$

is the volume of $T(\theta)$. The optimizing θ can be obtained by applying `fmin` to

$$T_V(\theta) = (1-\sin(\theta)^2)(\sin(\theta)-1).$$

Finally, we could also maximize total edge length:

$$E(\theta) = 3\left(\sqrt{2(1-\sin(\theta))}\ +\ \sqrt{3}\cos(\theta)\right)$$

by minimizing

$$T_E(\theta) = -(\sqrt{(1-\sin(\theta))}+\sqrt{3/2}\cos(\theta)).$$

The script `FindTet` applies `fmin` to these three objective functions, and here are the results:

Objective Function	Time	Theta
Area	1.000	-0.3398369144798928
Volume	1.111	-0.3398369181995862
Edge	1.213	-0.3398369274385691

It turns out that the same tetrahedron solves each problem, and this tetrahedron is *regular*. This means that the faces are identical in area and the edges are equal in length. Notice that accuracy is only good to the square root of the machine precision, confirming the remarks made at the end of §8.2.2 on pages 298-9.

Because the optimum tetrahedron is regular, we can compute θ a different way. The function

$$\tilde{T}_E(\theta) = \sqrt{1-\sin(\theta)}-\sqrt{3/2}\cos(\theta)$$

is the difference between the length of edge P_0P_1 and the length of edge P_1P_2. This function is zero at θ_* if $T(\theta_*)$ is the regular, solution tetrahedron. By applying `fzero` to $\tilde{T}_E(\theta)$ we get θ_* to full machine precision: -0.3398369094541218.

The exact solution is given by

$$\theta_* = \arcsin(-1/3).$$

To derive this result, we observe that the function $T_V(\theta)$ is cubic in $s = \sin(\theta)$:

$$p(s) = (1 - s^2)(1 - s).$$

From the equation $p'(s) = 0$, we conclude that $s =$ -1/3 is a minimizer and so $\theta_* = \arcsin(-1/3)$.

The point of all this is that the same problem can sometimes be formulated in different ways. The mathematical equivalence of these formulations does *not* necessarily lead to implementations that are equally efficient and accurate.

Problems

P8.2.2 What is the area of the largest triangle whose vertices are situated on the following ellipse

$$\frac{x^2}{4} + \frac{y^2}{9} = 1.$$

Solve this problem by applying `Golden`.

P8.2.3 Write a function `HexA = MaxHex(A,P)` that returns the area $H(A, P)$ of the largest hexagon that has its vertices on the ellipse

$$\begin{aligned} x(t) &= \frac{P - A}{2} + \frac{P + A}{2}\cos(t) \\ y(t) &= \sqrt{PA}\sin(t). \end{aligned}$$

Assume that $P \leq A$. Think about symmetry and see if you can't turn this into a problem that involves the minimization of a function of a single scalar variable.

The area of the above ellipse is given by

$$E(A, P) = \pi\sqrt{\frac{P + A}{2}}\sqrt{P \cdot A}.$$

Print a table that reports the value of $H(A, P)/E(A, P)$ for $P/A =$ [.00001 .0001 .001 .01 .1 .2 .3 .4 .5 .6 .7 .8 .9 1]. Results should be correct to six significant digits.

P8.2.4 A scientist gathers n data points $(x_1, y_1), \ldots, (x_n, y_n)$ and observes that they are arranged in an approximate circle about the origin. How could `fmin` be used to find a circle $x^2 + y^2 = r^2$ that approximately fits the data? Clearly specify and justify your choice for the function that is to be passed to `fmin`.

P8.2.5 A point on the unit sphere can be specified by a latitude θ ($-\pi/2 \leq \theta \leq \pi/2$) and a longitude ϕ ($-\pi \leq \phi \leq \pi$). The "great circle" distance between the points (θ_1, ϕ_1) and (θ_2, ϕ_2) is given by

$$\delta = 2\arcsin(d/2),$$

where $d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$ and

$$\begin{aligned} x_i &= \cos(\phi_i)\cos(\theta_i) \\ y_i &= \cos(\phi_i)\sin(\theta_i) \\ z_i &= \sin(\phi_i) \end{aligned} \quad , \qquad i = 1{:}2.$$

For points P_1, P_2, P_3 and P_4 on the sphere, let $f(P_1, P_2, P_3, P_4)$ be the shortest round trip path that visits all the points. Thus, $f(P_1, P_2, P_3, P_4)$ is the shortest of the 4-leg journeys $P_1P_2P_3P_4P_1$, $P_1P_2P_4P_3P_1$, and $P_1P_3P_2P_4P_1$ where distance along each leg is great circle distance. How big can $f(P_1, P_2, P_3, P_4)$ be? Exploit structure as much as possible. Include a brief discussion of your chosen objective function and make it efficient.

P8.2.6 Let $d(u, v, x_c, y_c, r)$ be the distance from the point (u, v) to the circle $(x - x_c)^2 + (y - y_c)^2 = r^2$. Assume that `x` and `y` are given column n-vectors. Given that x_c and y_c are stored in `xc` and `yc`, write a script that assigns to `r` the minimizer of

$$\phi(r) = \sum_{i=1}^{n} d(x_i, y_i, x_c, y_c, r)^2.$$

P8.2.7 Let E be the ellipse defined by

$$\begin{aligned} x(t) &= a\cos(t) \\ y(t) &= b\sin(t)\,, \end{aligned}$$

where $0 \leq t \leq 2\pi$. Let (x_0, y_0) be a given point. Let $(x(t_*), y(t_*))$ be a point on E that is nearest to (x_0, y_0). One way to compute t_* is to use `fmin` to minimize

$$f(t) = \sqrt{(x(t)-x_0)^2 + (y(t)-y_0)^2}.$$

Another way is to use `fzero` to find an appropriate zero of $g(t)$ where $g(t)$ is the sine of the angle between

$$\begin{bmatrix} x(t)-x_0 \\ y(t)-y_0 \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} x(t)/a \\ y(t)/b \end{bmatrix}.$$

(For your information, if u and v are unit 2-vectors, then $u_1v_2 - u_2v_1$ is the sine of the angle between them.)

Implement a function `d = PtoE1(x0,y0,a,b)` that computes the distance from (x_0, y_0) to E using the `fmin` method. Implement a function `d = PtoE2(x0,y0,a,b)` that does the same thing using the `fzero` method. You'll have to implement the functions f and g above to do this. Use the default tolerances when using `fmin` and `fzero` and choose the initial bracketing interval for `fmin` and the initial guess for `fzero` intelligently.

P8.2.8 Suppose `f1.m` and `f2.m` are given implementations of the real-valued functions $f_1(x)$ and $f_2(x)$ that are defined everywhere. Write a Matlab fragment that computes the maximum value of the function $s(x) = s_1(x) * s_2(x)$ on [0,1], where s_1 is the not-a-knot spline interpolant of f_1 on `linspace(0,1,20)` and s_2 is the not-a-knot spline interpolant of f_2 on `linspace(0,1,20)`. Make effective use of `spline`, `ppval`, and `fmin`. You may assume that `fmin(fname,L,R)` returns a global minimizer.

P8.2.9 Determine unknown parameters c_1, c_2, c_3, c_4 and λ so that if $y(t) = c_1 + c_2t + c_3t^2 + c_4e^{\lambda t}$, then

$$\phi(c_1, c_2, c_3, c_4, \lambda) \;=\; \sum_{i=1}^{9} (y(t_i) - y_i)^2$$

is as small as possible where

i	1	2	3	4	5	6	7	8	9
t(i)	0.00	0.25	0.50	0.75	1.00	1.25	1.50	1.75	2.00
y(i)	20.00	51.58	68.73	75.46	74.36	67.09	54.73	37.98	17.28

Note that

$$\phi(c_1, c_2, c_3, c_4, \lambda) \;=\; \| A(\lambda)c - y \|_2^2,$$

where $c^T = (c_1\ c_2\ c_3\ c_4)$, $y^T = (y_1, \ldots, y_9)$, and $A(\lambda)$ is a 9-by-4 matrix whose entries depend on λ. Note that if we "freeze" λ, then the "best" c is given by $c(\lambda) = A(\lambda)\backslash y$. Thus, we can solve the problem by applying `fmin` to the function

$$\tilde{y}(\lambda) = \| A(\lambda)c(\lambda) - y \|_2^2.$$

This renders λ_{opt}, from which we obtain $c_{opt} = A(\lambda_{opt})\backslash y)$. Write a script that computes and prints the optimal c and λ. The script should also plot in a single window the optimal fitting function $y(t)$ and the data (t_i, y_i), $i = 1{:}9$. Choose tolerances that are consistent with data that is accurate through the second decimal place. Use "\" to solve all linear LS problems. In the call to `fmin`, use the starting value $\lambda = 0$.

8.3 Minimizing Multivariate Functions

The vector-valued function

$$\begin{bmatrix} x_1(t) \\ y_1(t) \end{bmatrix} = \begin{bmatrix} \cos(\phi_1) & \sin(\phi_1) \\ -\sin(\phi_1) & \cos(\phi_1) \end{bmatrix} \begin{bmatrix} \dfrac{P_1 - A_1}{2} + \dfrac{P_1 + A_1}{2}\cos(t) \\ \sqrt{P_1A_1}\sin(t) \end{bmatrix}$$

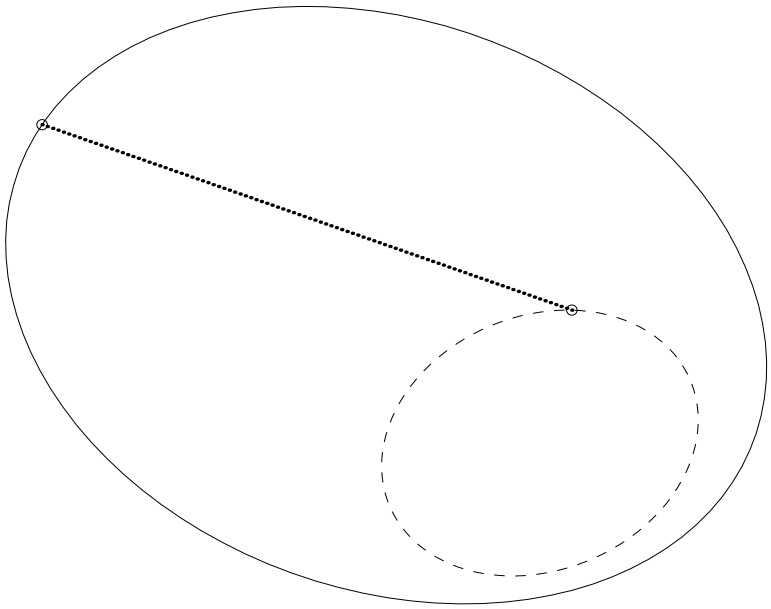

FIGURE 8.10 Orbits $(A_1, P_1, \phi_1) = (10, 2, \pi/8)$ and $(A_2, P_2, \phi_2) = (4, 1, -\pi/7)$

describes a "tilted," elliptical orbit having one focus at $(0, 0)$. Think of the Sun as being situated at this point. The parameter ϕ_1 is the tilt angle. If $A_1 \geq P_1$, then A_1 and P_1 are the maximum and minimum Sun-to-orbit distances. Let

$$\begin{bmatrix} x_2(t) \\ y_2(t) \end{bmatrix} = \begin{bmatrix} \cos(\phi_2) & \sin(\phi_2) \\ -\sin(\phi_2) & \cos(\phi_2) \end{bmatrix} \begin{bmatrix} \dfrac{P_2 - A_2}{2} + \dfrac{P_2 + A_2}{2}\cos(t) \\ \sqrt{P_2 A_2}\sin(t) \end{bmatrix}$$

be a second such orbit and consider the the display in Figure 8.10. Our goal is to find the minimum distance from a point on the first orbit (A_1, P_1, ϕ_1) to a point on the second orbit (A_2, P_2, ϕ_2). For a distance measure, we use

$$\text{sep}(t_1, t_2) = \frac{1}{2}\left[(x_1(t_1) - x_2(t_2))^2 \ + \ (y_1(t_1) - y_2(t_2))^2\right]. \tag{8.7}$$

This is a function of two variables, and in accordance with terminology established in §8.2, sep is the objective function. (See page 294.) Note that t_1 selects a point on the outer orbit and t_2 selects a point on the inner orbit. This section is about minimizing multivariate functions such as $\text{sep}(t)$.

8.3.1 Setting Up the Objective Function

Throughout we use a 3-field structure to represent the A, P, and ϕ values of an orbit, e.g., `planet = struct('A',10,'P',2,'phi',pi/8)`. Our implementation of sep(t) depends on the following function that can be used to generate orbit points:

```
  function pLoc = Orbit(t,planet,lineStyle)
% pLoc = Orbit(t,planet,lineStyle)
%
% t is a row vector and for k=1:length(t), pLoc.x(k) = x(t(k)) and
% pLoc.y(k) = y(t(k)) where
%
%  x(tau)   cos(phi) sin(phi)  (planet.A-planet.P)/2+((planet.A+planet.P)/2)cos(tau)
%        =                    *
%  y(tau)  -sin(phi) cos(phi)           sqrt(planet.A*planet.P)sin(tau)
%
% If nargin==3 then the points are plotted with line style defined by the
% string lineStyle.

c = cos(t); s = sin(t);
x0 = ((planet.P-planet.A)/2) + ((planet.P+planet.A)/2)*c;
y0 = sqrt(planet.A*planet.P)*s;
cphi = cos(planet.phi);
sphi = sin(planet.phi);
pLoc = struct('x',cphi*x0 + sphi*y0,'y',-sphi*x0 + cphi*y0);
if nargin==3
   plot(pLoc.x,pLoc.y,lineStyle)
end
```

Note the use of a two-field structure to represent sets of orbit points. With this function we can easily generate and plot orbit points. The script

```
figure
axis equal off
hold on
t = linspace(0,2*pi);
planet1 = struct('A',10,'P',2,'phi', pi/8);
planet2 = struct('A', 4,'P',1,'phi',-pi/7);
pLoc1 = Orbit(t,planet1,'-');  pt1 = orbit(3,planet1);
pLoc2 = Orbit(t,planet2,'--'); pt2 = Orbit(1,planet2);
plot(linspace(pt1.x,pt2.x),linspace(pt1.y,pt2.y),'.')
plot([pt1.x pt2.x],[pt1.y pt2.y],'o')
hold off
```

plots the two orbits in Figure 8.10 and draws a line that connects the "$t_1 = 3$" point on the first orbit with the "$t_2 = 4$" point on the second orbit. The separation between these two points is a function of a 2-vector of independent variables and the parameters A_1, P_1, ϕ_1, A_2, P_2, and ϕ_2:

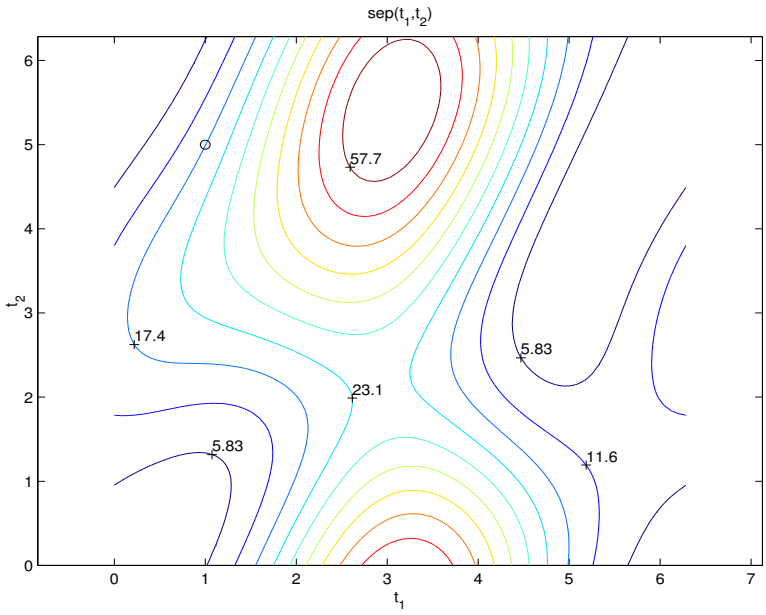

FIGURE 8.11 A contour plot of $sep(t_1, t_2)$

```
  function d = Sep(t,planet1,planet2)
% d = Sep(t,planet1,planet2)
%
% t is a 2-vector and planet1 and planet2 are orbit structures.
% t(1) defines a planet1 orbit point, t(2) defines a planet2 orbit point,
% and d is the distance between them.
pLoc1 = Orbit(t(1),planet1);
pLoc2 = Orbit(t(2),planet2);
d = ((pLoc1.x-pLoc2.x)^2 + (pLoc1.y-pLoc2.y)^2)/2;
```

As a rule, when solving nonlinear problems in the MATLAB environment, it is good practice not to "hardwire" a function's parameters but to pass them as a second argument. The functions `fmin` and `fmins` are designed to handle this flexibility.

It is useful to think of sep(t) as the "elevation" at point $t = (t_1, t_2)$. With this topographical point of view, our goal is to reach the deepest valley point. The terrain in the search area is depicted by the contour plot in Figure 8.11. This map tells us that the minimizing t is somewhere in the vicinity of $(5.5, 4.5)$.

8.3.2 The Gradient

It is foggy, and you are standing on a hillside without a map. To take a step consistent with the goal of reaching the valley bottom, it is reasonable to take that step in the "most downhill"

direction. Mathematically, this is in the direction of the negative gradient. Recall that the gradient of a function $f{:}\mathbb{R}^n \to \mathbb{R}$ at $t = t_c$ is a vector of partial derivatives:

$$\nabla f(t_c) = \begin{bmatrix} \dfrac{\partial f(t_c)}{\partial t_1} \\ \vdots \\ \dfrac{\partial f(t_c)}{\partial t_n} \end{bmatrix}.$$

The gradient always points in the direction of greatest increase and its negation points in the direction of greatest decrease. Taking the t_1 and t_2 partials in (8.7), we see that

$$\nabla \text{sep}(t) = \begin{bmatrix} \dfrac{\partial \text{sep}(t_1,t_2)}{\partial t_1} \\ \dfrac{\partial \text{sep}(t_1,t_2)}{\partial t_1} \end{bmatrix} = \begin{bmatrix} [x_1(t_1) - x_2(t_2)]\dot{x}_1(t_1) & + & [y_1(t_1) - y_2(t_2)]\dot{y}_1(t_1) \\ -[x_1(t_1) - x_2(t_2)]\dot{x}_1(t_2) & - & [y_1(t_1) - y_2(t_2)]\dot{y}_1(t_2) \end{bmatrix}.$$

Substituting the definitions for the component functions, we carefully arrive at the following implementation of the gradient:

```
  function g = gSep(t,planet1,planet2)
% g = gSep(t,planet1,planet2)
% The gradient of sep(t,planet1,planet2) with respect to 2-vector t.

A1 = planet1.A; P1 = planet1.P; phi1 = planet1.phi;
A2 = planet2.A; P2 = planet2.P; phi2 = planet2.phi;

alfa1  = (P1-A1)/2; beta1  = (P1+A1)/2; gamma1 = sqrt(P1*A1);
alfa2  = (P2-A2)/2; beta2  = (P2+A2)/2; gamma2 = sqrt(P2*A2);
s1     = sin(t(1)); c1     = cos(t(1));
s2     = sin(t(2)); c2     = cos(t(2));
cphi1  = cos(phi1); sphi1  = sin(phi1);
cphi2  = cos(phi2); sphi2  = sin(phi2);
Rot1   = [cphi1 sphi1; -sphi1 cphi1];
Rot2   = [cphi2 sphi2; -sphi2 cphi2];
P1     = Rot1*[alfa1+beta1*c1;gamma1*s1];
P2     = Rot2*[alfa2+beta2*c2;gamma2*s2];
dP1    = Rot1*[-beta1*s1;gamma1*c1];
dP2    = Rot2*[-beta2*s2;gamma2*c2];
g = [-dP1';dP2']*(P2-P1);
```

The derivation not important. But what *is* important is to appreciate that gradient function implementation can be very involved and time-consuming. Our problem is small ($n = 2$) and simple. For high-dimension applications with complicated objective functions, it is often necessary to enlist the services of a symbolic differentiation package.

8.3.3 The Method of Steepest Descent

From multivariable Taylor approximation theory, we know that if λ_c is real and s_c is an n-vector, then

$$f(t_c + \lambda_c s_c) = f(t_c) + \lambda_c \nabla f(t_c)^T s_c \ + \ O(\lambda_c^2).$$

Thus, we can expect a decrease in the value of f if we set the *step direction* s_c to be

$$s_c \ = \ -\nabla f(t_c)$$

and the *step length* parameter λ_c to be small enough. If the gradient is zero, then this argument breaks down. In that case t_c is a critical point, but may not be a local minimum. Our search for the valley bottom can "stall" at such a location.

The practical determination of λ_c is the *line search* problem. Ideally, we would like to minimize $f(t_c + \lambda s_c)$ over all λ. However, an approximate solution to this one-dimensional minimization problem is usually good enough and often essential if f is expensive to evaluate. This brings us to the *steepest descent* framework for computing an improvement t_+ to the current minimizer:

- Compute the gradient $g_c = \nabla f(t_c)$ and set $s_c = -g_c$.
- Inexpensively choose λ_c so $f(t_c + \lambda_c s_c)$ is sufficiently less than $f(t_c)$.
- Set $t_+ = t_c + \lambda_c s_c$.

The design of a reasonable line search strategy is a nontrivial task. To build an appreciation for this point and for the method of steepest descent, we have built a simple environment `ShowSD` that can be used to minimize the *sep* function. At each step it plots $\text{sep}(t_c + \lambda s_c)$ and obtains λ_c by mouseclick. (See Figure 8.12.) After two such graphical line search steps, the `ShowSD`

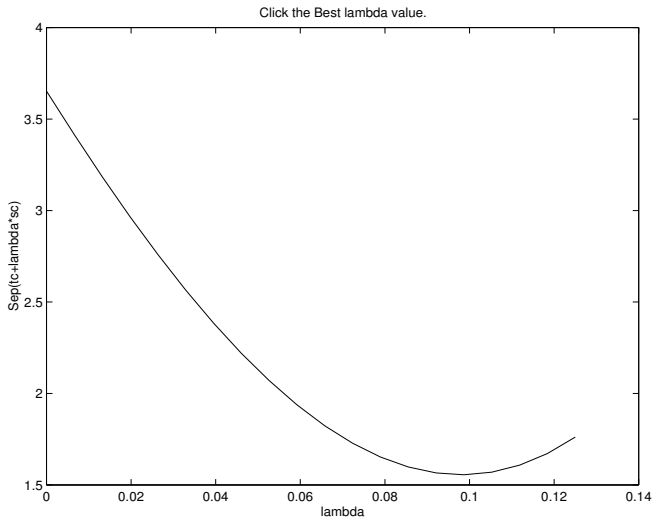

FIGURE 8.12 Graphical line search

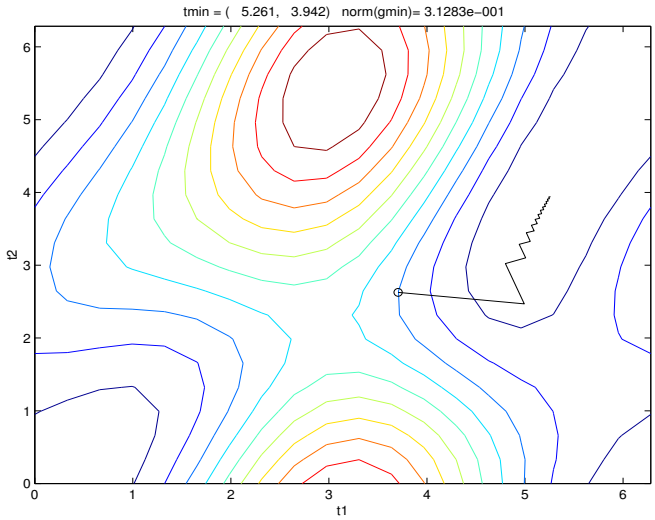

FIGURE 8.13 The steepest descent path

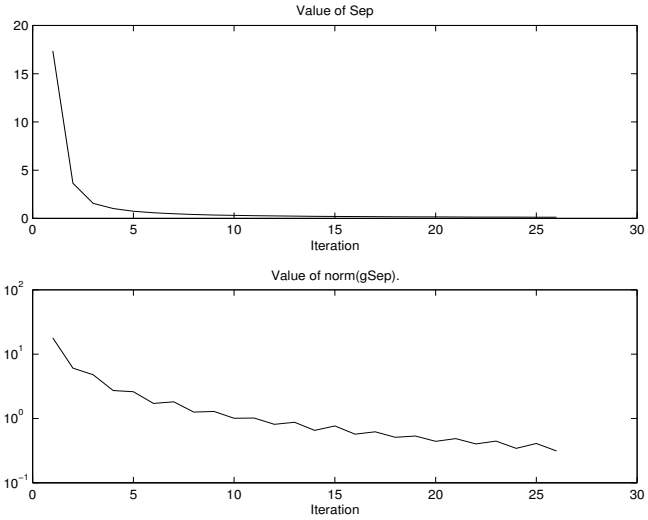

FIGURE 8.14 The descent of $\text{sep}(t_1, t_2)$ and $\|\nabla \text{sep}(t_1, t_2)\|_2$

environment slips into an automated line search mode and performs additional steepest descent steps. On the previous page, Figure 8.13 shows the search path on the contour plot and Figure 8.14 shows how `sep` and the norm of its gradient behave during the iteration. The exhibited behavior is typical for the method. The first few iterations are very successful in bringing about a reduction in the value of the objective function. However, the convergence rate for steepest descent (with reasonable line search) is linear and the method invariably starts plodding along with short, ineffective steps.

8.3.4 The MATLAB `fmins` Function

The `fmins` function can be used to minimize a multivariate function. It uses a simplex search method and does not require gradient function implementation. The following script shows how `fmins` is typically invoked:

```
plist = [10  2  pi/8  4  1  -pi/7];
StartingValue = [5;4];
trace = 1;
steptol = .000001;
ftol = .000001;
options = [trace steptol ftol];
[t,options] = fmins('Sep',StartingValue,options,[],plist);
disp(sprintf('Iterations = %3.0f',options(10)))
disp(sprintf('t(1) = %10.6f,  t(2) = %10.6f',t))
```

The call requires the name of the objective function, a starting value, various option specifications, and a list of parameters required by the objective function. Standard input options are used to indicate whether or not a trace of the iteration is desired and how to bring about termination. Setting `trace = 1` in the script means that the results of the `fmins` iterates are displayed. Setting `trace = 0` turns off this feature. If the iterates are close with respect to steptol and the difference between function evaluations is small with respect to ftol, then the iteration is brought to a halt. An iteration maximum brings about termination if these criteria are too stringent.

Upon return, `t` is assigned the approximate solution and `options(10)` contains the number of required steps.

The function `ShowFmins` can be used to experiment with this minimizer when it is applied to the function `sep`. The starting value is obtained by mouseclicks. The orbit points corresponding to the solution are graphically displayed. (See Figure 8.15.)

Problems

P8.3.1 Let $f_c(\lambda) = f(t_c + \lambda s_c)$. Assume that $\lambda_2 > \lambda_1$ and that $f_c(\lambda_2)$ or $f_c(\lambda_1)$ is less than $f_c(0)$. Let λ_c be the minimizer on $[0,\lambda_2]$ of the quadratic that interpolates $(0, f_c(0))$, $(\lambda_1, f_c(\lambda_1))$, and $(\lambda_2, f_c(\lambda_2))$. Modify the line search strategy in the function `SDStep` so that it incorporates this method for choosing λ_c.

P8.3.2 The function *sep* is periodic, and there are an infinite number of minimizers. Use `ShowSD` to find a different solution than the one reported in the text.

P8.3.3 What happens in `ShowSD` if the two orbits intersect? How is the rate of convergence affected if the inner ellipse is very "cigar-like"?

P8.3.4 Modify `ShowFmins` so that for a given starting value, it reports the number of iterations required when `fmins` is run with `steptol` = `ftol` = 10^{-d} for $d = 0{:}6$.

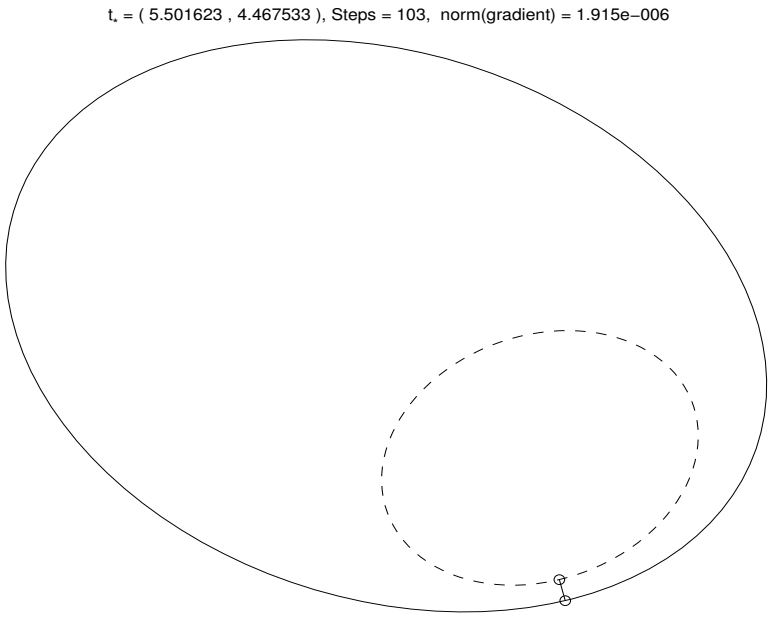

FIGURE 8.15 Solution by `fmins`

8.4 Systems of Nonlinear Equations

Suppose we have a pair of orbits whose intersection points are sought. (See Figure 8.16 on the next page.) In this problem, the unknown is a 2-vector $t = (t_1, t_2)$ having the property that

$$\text{sepV}(t) = \begin{bmatrix} x_2(t_2) - x_1(t_1) \\ y_2(t_2) - y_1(t_1) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix},$$

where

$$\begin{bmatrix} x_i(t_i) \\ y_i(t_i) \end{bmatrix} = \begin{bmatrix} \cos(\phi_i) & \sin(\phi_i) \\ -\sin(\phi_i) & \cos(\phi_i) \end{bmatrix} \begin{bmatrix} \dfrac{P_i - A_i}{2} + \dfrac{P_i + A_i}{2}\cos(t_i) \\ \sqrt{P_i A_i}\sin(t_i) \end{bmatrix} \qquad i = 1, 2.$$

It is easy to implement this using `Orbit`:

```
  function z = SepV(t,planet1,planet2)
% z = SepV(t,planet1,planet2)
% The vector from the t(1) point on the planet1 orbit
% to the t(2) point on the planet2 orbit.
pt1 = Orbit(t(1),planet1);
pt2 = Orbit(t(2),planet2);
z = [pt2.x-pt1.x;pt2.y-pt1.y];
```

Our goal is to develop a systems version of Newton's method that can zero functions like this.

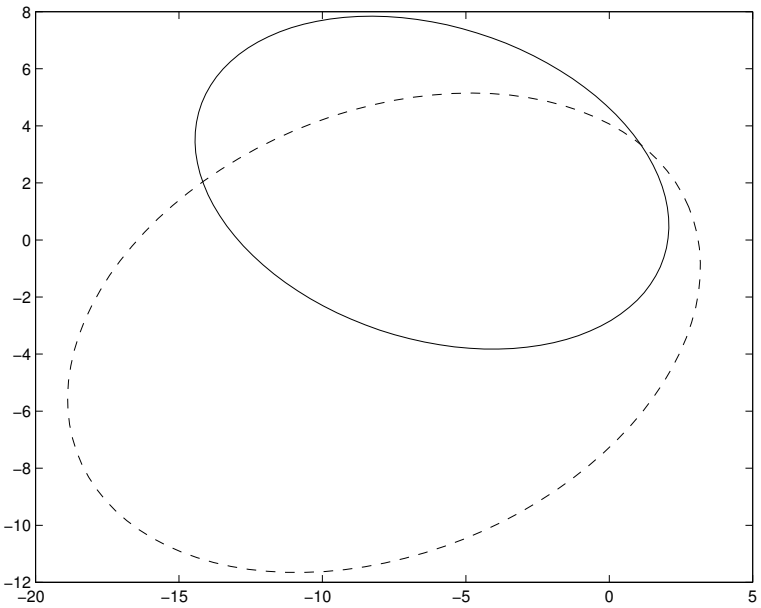

FIGURE 8.16 The orbits $(15, 2, \pi/10)$ and $(20, 3, -\pi/8)$

8.4.1 The Jacobian

In §8.1 we derived Newton's method through the idea of a linear model, and we take the same approach here. Suppose $t_c \in \mathbb{R}^n$ is an approximate zero of the vector-valued function

$$F(t) = \begin{bmatrix} F_1(t) \\ \vdots \\ F_n(t) \end{bmatrix}.$$

To compute an improvement $t_c + s_c$, we build a linear model of F at $t = t_c$. For each component function we have $F_i(t_c + s_c) \approx F_i(t_c) + \nabla F_i(t_c)^T s_c$, and so

$$F(t_c + s_c) \approx F(t_c) + J(t_c)s_c,$$

where $J(t_c) \in \mathbb{R}^{n \times n}$ is a matrix whose ith row equals $\nabla F_i(t_c)^T$. $J(t_c)$ is the *Jacobian* of F at t_c. The Jacobian of the sepV(t) function is given by

$$J(t) = \begin{bmatrix} -\dot{x}_1(t_1) & \dot{x}_2(t_2) \\ -\dot{y}_1(t_1) & \dot{y}_2(t_2) \end{bmatrix},$$

and after using the component function definitions we get

```
  function J = JSepV(t,planet1,planet2)
% J = JSepV(t,planet1,planet2)
% J is the Jacobian of sepV(t,planet1,planet2).

A1 = planet1.A; P1 = planet1.P; phi1 = planet1.phi;
A2 = planet2.A; P2 = planet2.P; phi2 = planet2.phi;

s1 = sin(t(1)); c1 = cos(t(1));
s2 = sin(t(2)); c2 = cos(t(2));
beta1 = (P1+A1)/2; gamma1 = sqrt(P1*A1);
beta2 = (P2+A2)/2; gamma2 = sqrt(P2*A2);
cphi1 = cos(phi1); sphi1  = sin(phi1);
cphi2 = cos(phi2); sphi2  = sin(phi2);
Rot1  = [cphi1 sphi1; - sphi1 cphi1];
Rot2  = [cphi2 sphi2; - sphi2 cphi2];
```

Do not conclude from this that Jacobian implementation is easy. Our problem is small and simple. Remember that $O(n^2)$ partial derivatives are involved. It does not take a very large n or a very complex F to make Jacobian evaluation a formidable task. An approach using finite differences is discussed in §8.4.3.

8.4.2 The Newton Idea

The Newton idea for systems involves choosing s_c so that the local model of F is zero:

$$F(t_c + s_c) \;\approx\; F(t_c) + J(t_c)s_c \;=\; 0.$$

This involves a function evaluation, a Jacobian evaluation, *and* the solution of a linear system $J(t_c)s = -F(t_c)$, as the following implementation shows:

```
  function [tnew,Fnew,Jnew] = NStep(tc,Fc,Jc,planet1,planet2)
% [tnew,Fnew,Jnew] = NStep(tc,Fc,Jc,planet1,planet2)
%
% tc is a column 2-vector, Fc is the value of SepV(t,planet1,planet2) at t=tc
% and Jc is the value of JSepV(t,planet1,planet2) at t=tc. Does one Newton step
% applied to SepV rendering a new approximate root tnew. Fnew and Jnew are the
% values of SepV(tnew,planet1,planet2) and jSepV(tnew,planet1,planet2) respectively.

sc = -(Jc\Fc);
tnew = tc + sc;
Fnew = SepV(tnew,planet1,planet2);
Jnew = JSepV(tnew,planet1,planet2);
```

Repetition of this process produces the *Newton iteration*. Even if we disregard the function and Jacobian evaluation, there are still $O(n^3)$ flops per step, so Newton for systems is a decidedly more intense computation than when the method is applied in the single-variable case.

The function `ShowN` can be used to explore the Newton iteration behavior when it is applied to the sepV$(t) = 0$ problem. The starting value is obtained by mouseclicking a point on each orbit.

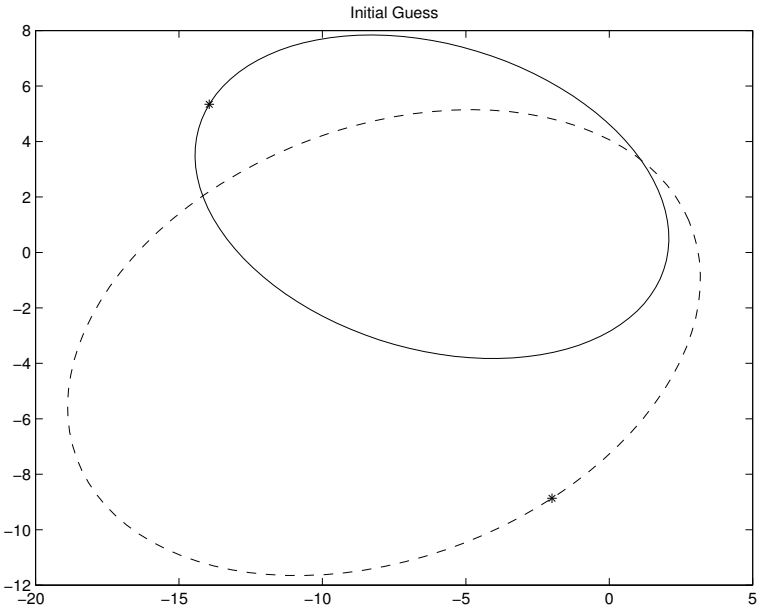

FIGURE 8.17 Initial guess with orbits $(15, 2, \pi/10)$ and $(20, 3, -\pi/8)$

The corresponding t-values are computed and form the "initial guess." The iterates are displayed graphically, see Figure 8.17 and Figure 8.18 on the next page. For this particular starting value, an intersection point is found in eight iterations:

```
Iteration        tc(1)                  tc(2)              norm(Fc)   cond(Jc)
-------------------------------------------------------------------------------
     0     3.0000000000000000     5.0000000000000000     1.857e+001   5.618e+000
     1    10.6471515673799160     1.4174812578245817     6.626e+000   2.443e+000
     2     9.6882379275920627     2.1574580549081377     3.415e+000   2.842e+000
     3     9.8554124695134444     1.9001757004462649     1.880e-001   1.995e+000
     4     9.8869591366045153     1.9091841669218315     3.745e-003   1.934e+000
     5     9.8866102770856905     1.9088507362936675     3.219e-007   1.935e+000
     6     9.8866103036526045     1.9088507152479945     1.776e-015   1.935e+000

 Intersection = ( -14.1731346193976350 , 2.0389395028622435 )
```

Observe the quadratic convergence that typifies the Newton approach. The condition number of the Jacobian is displayed because if this number is large, then the accuracy of the correction and its possibly large norm could have an adverse effect on convergence. For the problem under consideration, this is not an issue. But if we attempt to find the intersection points of the nearly tangential orbits $(15, 2, \pi/10)$ and $(15, 2.000001, \pi/10)$, then it is a different matter:

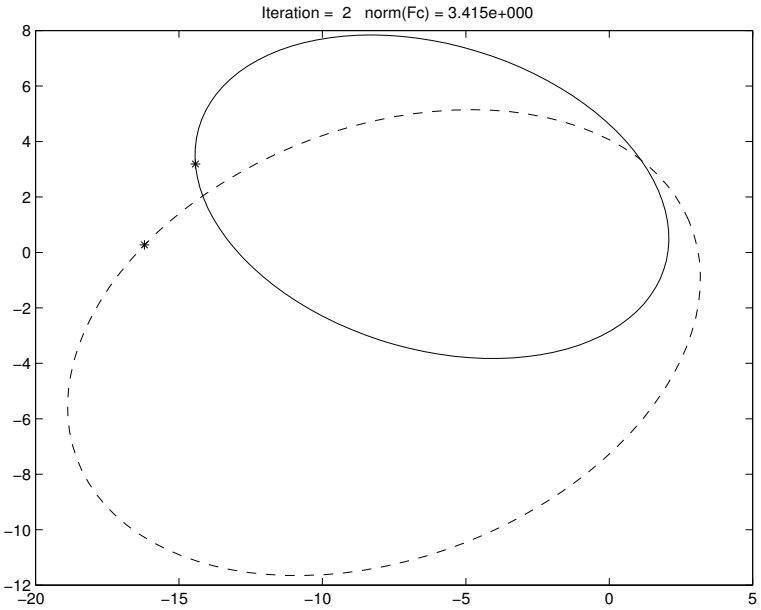

FIGURE 8.18 First step

```
Iteration        tc(1)                 tc(2)              norm(Fc)   cond(Jc)
-------------------------------------------------------------------------------
    0     0.2129893324467656     0.2129893324467656     1.030e-06   3.468e+07
    1    -2.9481695167573863    -2.9481687805353460     4.391e-06   1.936e+06
    2    -2.9365137654751829    -2.9365138156763226     7.703e-08   1.544e+07
    3    -3.0402412755186443    -3.0402413005044142     1.907e-08   2.956e+07
    :            :                      :                  :          :
   12    -3.1413980211918338    -3.1413980212404917     7.248e-14   1.501e+10
   13    -3.1414973191506728    -3.1414973191745066     2.079e-14   3.064e+10
   14    -3.1415554417361111    -3.1415554417454143     3.202e-15   7.850e+10

Intersection = ( -14.2659107219684653 , 4.6350610716380816 )
```

Notice that in this case the Jacobian is ill conditioned near the solution and that linear-like convergence is displayed. This corresponds to what we discovered in §8.1 for functions that have a very small derivative near a root.

8.4.3 Finite Difference Jacobians

A fundamental problem with Newton's method is the requirement that the user supply a procedure for Jacobian evaluation. One way around this difficulty is to approximate the partial derivatives with sufficiently accurate divided differences. In this regard, it is helpful to look at

the Jacobian from the column point of view. The qth column is given by

$$\begin{bmatrix} \dfrac{\partial F_1(t_c)}{\partial t_q} \\ \vdots \\ \dfrac{\partial F_n(t_c)}{\partial t_q} \end{bmatrix} = \frac{\partial F(t_c)}{\partial t_q}.$$

Thus,

$$\frac{\partial F(t_c)}{\partial t_q} \approx \frac{F(t_1,\ldots,t_q+\delta_k,\ldots,t_n) - F(t_1,\ldots,t_n)}{\delta_k},$$

where δ_k is a small, carefully chosen real number. An approximate Jacobian can be built up in this fashion column by column. Instead of n^2 partials to encode and evaluate, we have $n+1$ evaluations of the function F, making the finite difference Newton method very attractive. If the difference parameters δ_k are computed properly, then the quadratic convergence of the exact Newton approach remains in force. The function `ShowFDN` supports experimentation with the method.

Problems

P8.4.1 Write a function that returns all the intersection points of two given orbits. Use `ShowN`.

P8.4.2 How is the convergence rate of the finite difference Newton method affected by the choice of the δ parameters?

P8.4.3 Let C be the unit cube "cornered" in the positive orthant. Assume that the density of the cube at (u,v,w) is given by $d(u,v,w) = e^{-(u-x_1)^2(v-x_2)^2(w-x_3)^2}$, where $x = [x_1,x_2,x_3]^T$ is a vector of parameters to be determined. We wish to choose x so that C's center of gravity is at $(.49,.495,.51)$. This will be the case if each of the nonlinear equations

$$\begin{aligned}
F_1(x_1,x_2,x_3) &= \int_0^1\int_0^1\int_0^1 u\cdot d(u,v,w)dudvdw - .2\int_0^1\int_0^1\int_0^1 d(u,v,w)dudvdw \\
F_2(x_1,x_2,x_3) &= \int_0^1\int_0^1\int_0^1 v\cdot d(u,v,w)dudvdw - .3\int_0^1\int_0^1\int_0^1 d(u,v,w)dudvdw \\
F_3(x_1,x_2,x_3) &= \int_0^1\int_0^1\int_0^1 w\cdot d(u,v,w)dudvdw - .6\int_0^1\int_0^1\int_0^1 d(u,v,w)dudvdw
\end{aligned}$$

is zeroed. Solve this problem using Newton's method. Handle the integrals with the trapezoidal rule.

8.4.4 Zeroing the Gradient

If Newton's method is applied to the problem of zeroing the gradient of a function $f{:}\mathbb{R}^n \to \mathbb{R}^1$, then the Jacobian is special and called the *Hessian*:

$$J(t_c) = \nabla^2 f(t_c) = H_c = (h_{ij}) \qquad h_{ij} = \frac{\partial^2 f}{\partial t_i \partial t_j}.$$

It is symmetric, and at a local minimum it is also positive definite. In this case a Newton step looks like this:

- Evaluate the gradient $g_c = \nabla f(t_c)$ and the Hessian $H_c = \nabla^2 f(t_c)$.
- Solve the linear system $H_c s_c = -g_c$, possibly using Cholesky.
- Compute the new approximate solution $t_+ = t_c + s_c$.

Hessian computation is more challenging than ordinary Jacobian computation because second derivatives are involved. For the sep(t) function (8.7), we find

$$\nabla^2 \text{sep}(t) = \begin{bmatrix} (x_1 - x_2)\ddot{x}_1 + (y_1 - y_2)\ddot{y}_1 & -\dot{x}_1\dot{x}_2 - \dot{y}_1\dot{y}_2 \\ -\dot{x}_1\dot{x}_2 - \dot{y}_1\dot{y}_2 & (x_2 - x_1)\ddot{x}_2 + (y_2 - y_1)\ddot{y}_2 \end{bmatrix},$$

whereupon

```
  function [g,H] = gHSep(t,planet1,planet2)
% [g,H] = gHSep(t,planet1,planet2)
%
% t is a 2-vector and planet1 and planet2 are orbits.
% g is the gradient of Sep(t,planet1,planet2) and H is the Hessian.

A1 = planet1.A; P1 = planet1.P; phi1 = planet1.phi;
A2 = planet2.A; P2 = planet2.P; phi2 = planet2.phi;

alpha1  = (P1-A1)/2; beta1  = (P1+A1)/2; gamma1 = sqrt(P1*A1);
alpha2  = (P2-A2)/2; beta2  = (P2+A2)/2; gamma2 = sqrt(P2*A2);
s1     = sin(t(1)); c1     = cos(t(1));
s2     = sin(t(2)); c2     = cos(t(2));
cphi1  = cos(phi1); sphi1  = sin(phi1);
cphi2  = cos(phi2); sphi2  = sin(phi2);
Rot1   = [cphi1 sphi1; -sphi1 cphi1];
Rot2   = [cphi2 sphi2; -sphi2 cphi2];
P1     = Rot1*[alpha1+beta1*c1;gamma1*s1];
P2     = Rot2*[alpha2+beta2*c2;gamma2*s2];
dP1    = Rot1*[-beta1*s1;gamma1*c1];
dP2    = Rot2*[-beta2*s2;gamma2*c2];
g = [-dP1';dP2']*(P2-P1);
ddP1   = Rot1*[-beta1*c1;-gamma1*s1];
ddP2   = Rot2*[-beta2*c2;-gamma2*s2];
H = zeros(2,2);
H(1,1) = (P1(1)-P2(1))*ddP1(1) + (P1(2)-P2(2))*ddP1(2) + ...
   dP1(1)^2 + dP1(2)^2;
H(2,2) = (P2(1)-P1(1))*ddP2(1) + (P2(2)-P1(2))*ddP2(2) + ...
   dP2(1)^2 + dP2(2)^2;
H(1,2) = -dP1(1)*dP2(1) - dP1(2)*dP2(2);
H(2,1) = H(1,2);
```

Because gradient and Hessian computation often involve the evaluation of similar quantities, it usually makes sense to write a single function to evaluate them both.

If we apply Newton's method to the problem of zeroing the gradient of sep, then the good properties that we expect of the iteration only apply if the current solution is "close enough." Complementing this behavior is the steepest descent iteration, which tends to perform well in the early stages. Just as we combined bisection and Newton methods for one-dimensional problems in §8.2, we can do the same thing here with steepest descent and Newton applied to the gradient. The function `ShowMixed` supports experimentation with this idea applied to the $sep(t)$ minimization problem. Here is the result from an experiment in which two steepest descent steps were performed followed by the Newton iteration:

```
Step     sep            t(1)                t(2)          norm(grad)
-------------------------------------------------------------------------
  0   19.447493      3.575914218602207   2.314857744750374    1.4e+001
  1    3.096900      4.969531464925577   2.561747838986417    5.6e+000
  2    1.400192      4.823924341486837   3.056459223394815    4.4e+000
  3    0.376080      5.073231173612004   3.499141731419454    9.4e-001
  4    0.143489      5.217591520162172   3.880268402697888    4.6e-001
  5    0.080276      5.352122519062924   4.172044821701761    2.1e-001
  6    0.065675      5.455054653917450   4.380466709519947    9.5e-002
  7    0.064355      5.497074329824716   4.459437020615108    1.4e-002
  8    0.064341      5.501575688398053   4.467449009283879    1.5e-004
  9    0.064341      5.501623047668216   4.467532519901488    1.7e-008
 10    0.064341      5.501623052934421   4.467532529194002    7.7e-015
```

The path to the solution is displayed in Figure 8.19 (on the next page). The development of a general-purpose optimizer that combines Newton and steepest descent ideas is very challenging. All we have done here is give a snapshot of the potential.

Problems

P8.4.4 Automate the switch from steepest descent to Newton in `ShowMixed`. Cross over when the reduction in $sep(t)$ is less than 1% as a result of a steepest descent step.

P8.4.5 Show how `fmins` could be used to fit the circle $(x-x_c)^2+(y-y_c)^2=r^2$ to the data $(x_1,y_1),\ldots,(x_n.y_n)$. Your answer must include a complete implementation of the objective function `MyF` that is passed to `fmins`. For full credit, `MyF` should implement a function of two variables. Use default tolerances and suggest an intelligent starting value.

8.4.5 Nonlinear Least Squares

A planet moves in the orbit

$$\begin{bmatrix} x(t) \\ y(t) \end{bmatrix} = \begin{bmatrix} \frac{P-A}{2} + \frac{A+P}{2}\cos(t) \\ \sqrt{AP}\sin(t) \end{bmatrix}.$$

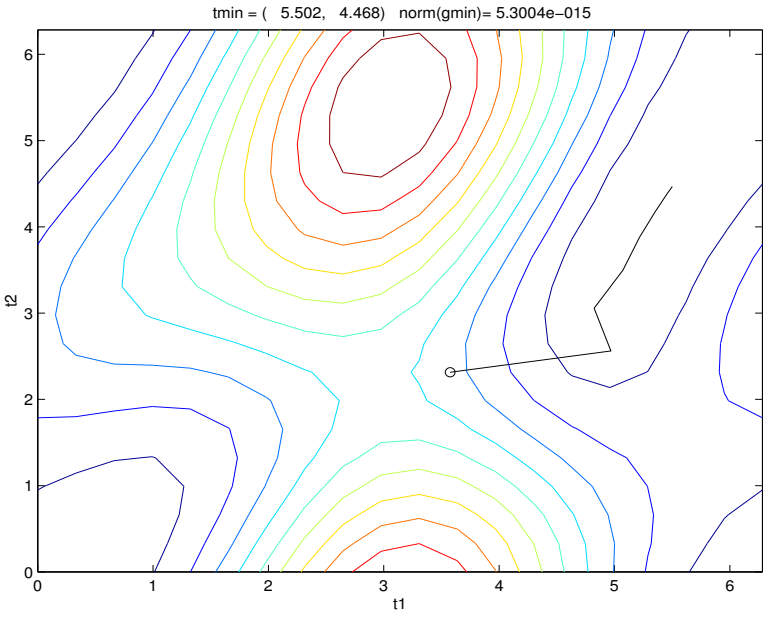

FIGURE 8.19 Steepest descent/Newton path

Our goal is to estimate the orbit parameters A and P, given that we are able to measure $r(\theta)$, the length of the orbit vector where θ is the angle between that vector and the positive real x-axis. It can be shown that

$$r(\theta) = \frac{2AP}{P(1-\cos(\theta)) + A(1+\cos(\theta))}.$$

Let r_i be the measured value of this function at $\theta = \theta_i$. For each measurement (θ_i, r_i), define the function

$$F_i(A,P) = r_i - \frac{2AP}{P(1-\cos(\theta_i)) + A(1+\cos(\theta_i))}.$$

To compute the best choices for A and P from data points $(\theta_1, r_1), \ldots, (\theta_m, r_m)$, we minimize the following function:

$$\mathrm{Orb}(A,P) = \frac{1}{2}\sum_{i=1}^{m} F_i(A,P)^2.$$

This is an example of the nonlinear least squares problem, a very important optimization problem in which Newton ideas have a key role.

In general, we are given a function

$$F(p) = \begin{bmatrix} F_1(p) \\ F_2(p) \\ \vdots \\ F_m(p) \end{bmatrix}$$

and seek a vector of unknown parameters $p \in \mathbb{R}^n$ so that the sum of squares

$$\rho(p) = \frac{1}{2} \sum_{i=1}^{m} F_i(p)^2 \tag{8.8}$$

is as small as possible. We use "p" because nonlinear least squares is often used to resolve *p*arameters in model-fitting problems. A full Newton method involving the step

$$\nabla^2 \rho(p_c) s_c = -\nabla \rho(p_c)$$

entails considerable derivative evaluation since the gradient is given by

$$\nabla \rho(p_c) = J(p_c)^T F(p_c)$$

and

$$\nabla^2 \rho(p_c) = J(p_c)^T J(p_c) + \sum_{i=1}^{m} F_i(p_c) \nabla^2 F_i(p_c)$$

prescribes the Hessian. Here, $J(p_c)$ is the Jacobian of the function $F(p)$ evaluated at $p = p_c$:

$$J(p_c) = \left(\frac{\partial F_i(p_c)}{\partial p_j} \right) \qquad 1 \le i \le m,\ 1 \le j \le n.$$

Note that once this matrix is found, we have "half" the Hessian. The other half is a linear combination of component function Hessians and very awkward to compute because of all the second derivatives. However, if the model we are trying to determine fits the data well, then near the solution we have $F_i(p) \approx 0$, $i = 1{:}m$, and this part of the Hessian goes away. This leads to the *Gauss-Newton* framework:

- Evaluate $F_c = F(p_c)$ and the Jacobian $J_c = J(p_c)$.
- Solve $(J_c^T J_c) s_c = -J_c^T F_c$.
- Update $p_+ = p_c + s_c$.

Note that s_c is the solution to the normal equations for the least squares problem

$$\min_{s \in \mathbb{R}^m} \| J(p_c)s + F(p_c) \|_2.$$

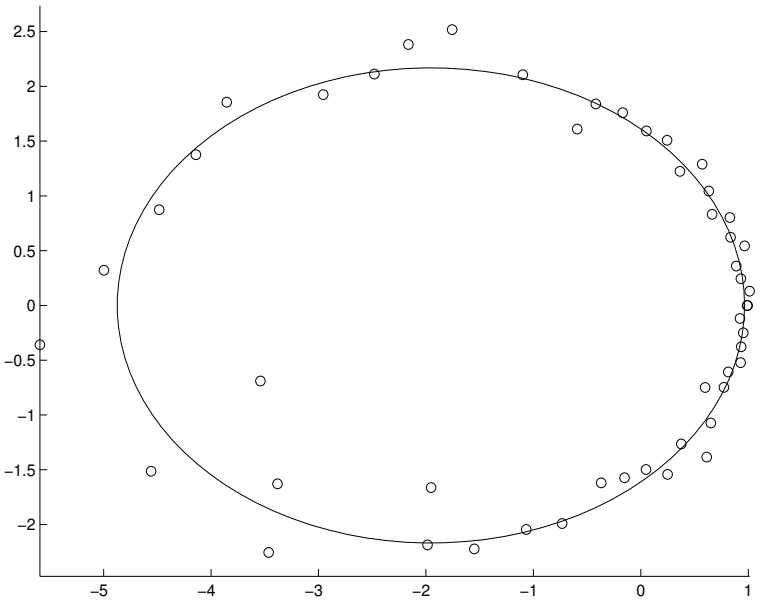

FIGURE 8.20 Fitting an orbit to noisy data

Thus, in a reliable implementation of the Gauss-Newton step we can use a least squares method to compute the step direction.

The function `ShowGN` can be used to experiment with this method applied to the problem of minimizing the function $Orb(A, P)$ above. It solicits an exact pair of parameters A_0 and P_0, computes the exact value of $r(\theta)$ over a range of θ, and then perturbs these values with a prescribed amount of noise. (See Figure 8.20.) A (partially) interactive Gauss-Newton iteration with line search is then initiated. Here are some sample results:

```
A0 = 5.000,  P0 = 1.000,   Relative Noise = 0.100,  Samples =  50

Iteration     wc         A          P          norm(grad)
-------------------------------------------------------------
    0      58.7723   2.946988   2.961369       4.153e+001
    1       7.4936   3.591895   1.157316       9.427e+000
    2       1.9129   4.771246   0.921970       4.077e+000
    3       1.7737   4.870860   0.963656       1.261e-001
    4       1.7736   4.873808   0.965140       6.416e-003
    5       1.7736   4.873641   0.965066       3.288e-004
    6       1.7736   4.873651   0.965070       1.691e-005
    7       1.7736   4.873650   0.965070       9.059e-007
```

In this example, the true values of A_0 and P_0 are 5 and 1, respectively. The relative noise in the

r values is 10%, and the technique does a fairly good job of estimating the parameters from the data.

The Gauss-Newton method works well on small residual problems. On highly nonlinear problems for which the minimum sum of squares is large, the method can be very slow or non-convergent.

Problems

P8.4.6 It is conjectured that the data $(t_1 f_1), \ldots, (t_m, f_m)$ are a sampling of the function

$$F(t) = a_1 e^{\lambda_1 t} + a_2 e^{\lambda_2 t}.$$

To determine the *model parameters* a_1 and a_2 (the coefficients) and λ_1 and λ_2 (the time constants), we minimize

$$\phi(a, \lambda) \;=\; \sum_{i=1}^{m} [f(t_i) - f_i]^2 \;=\; \sum_{i=1}^{m} \left[a_1 e^{\lambda_1 t_i} + a_2 e^{\lambda_2 t_i} - f_i\right]^2, \qquad \lambda = \begin{bmatrix} \lambda_1 \\ \lambda_2 \end{bmatrix}, \quad a = \begin{bmatrix} a_1 \\ a_2 \end{bmatrix}.$$

Defining the m-by-2 matrix E_λ by

$$E_\lambda = \begin{bmatrix} e^{t_1\lambda_1} & e^{t_1\lambda_2} \\ e^{t_2\lambda_1} & e^{t_2\lambda_2} \\ \vdots & \vdots \\ e^{t_m\lambda_1} & e^{t_m\lambda_2} \end{bmatrix},$$

we see that

$$\phi(a, \lambda) \;=\; \| E_\lambda a - F \|^2 \qquad f = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_m \end{bmatrix}.$$

We could apply the MATLAB minimizer `fmins` to this function. However, the four-dimensional search can be reduced to a two-dimensional search by making a simple observation. If $\lambda^{(c)}$ is the current estimate of the time constants and if $E_{\lambda^{(c)}}$ is the corresponding E-matrix, then the best estimate for the coefficients is the vector a that solves

$$\min \| E_{\lambda^{(c)}} a - f \|_2.$$

Let a_λ designate this vector and define

$$\psi(\lambda) = \phi(a_\lambda, \lambda).$$

Apply `fmins` to this function with starting value $\lambda = [0, -1]^T$. To generate data, use

```
function [t,fvals] = GenProb(m,L,R,noise)
rand('normal');
t = linspace(L,R,m)';
fvals = 3*exp(-.2*t) + 5*exp(-4*t) + noise*rand(t);
```

What are the estimated time constants for the problems $(m, L, R, noise) = (10, 0, 2, .001)$, $(10, 1, 3, .001)$, $(10, 2, 4, .001)$, and $(10, 20, 21, .001)$?

P8.4.7 Write a function `MyShowN` that runs both pure Newton and the globalized Newton with the starting value

$$t_c = \begin{bmatrix} 3.6208186515950160 \\ 0.2129893324467656 \end{bmatrix}.$$

`MyShowN` should print the same table as `ShowN` and a second table. The second table should report on what happens when the globalized strategy is used. It should have the same format as the first table, with an additional column that indicates for each iteration whether a Newton step or a steepest descent step was taken.

P8.4.8 Suppose F:$\mathbb{R}^n \rightarrow \mathbb{R}^n$ is a given function and define f:$\mathbb{R}^n \rightarrow \mathbb{R}$ by

$$f(x) = \frac{1}{2}F(x)^T F(x).$$

Notice that the act of zeroing F is the act of minimizing f.

If $F_c \in \mathbb{R}^n$ is F evaluated at $x = x_c$ and $J_c \in \mathbb{R}^{n\times n}$ is the Jacobian of F at $x = x_c$, then it can be shown that the gradient of f at $x = x_c$ is given by

$$g_c = \nabla f(x_c) = J_c^T F_c.$$

The Newton step

$$x_{newt} = x_c - J_c^{-1} F_c$$

generally moves us quadratically to a zero x_* of F if $\| x_c - x_* \|$ is "small enough." On the other hand, if we are not close enough, then x_{Newt} may not represent an improvement. Indeed, we may even have

$$f(x_{newt}) \;=\; \frac{1}{2}\| F(x_{newt}) \|_2^2 \;>\; \frac{1}{2}\| F(x_c) \|_2^2 \;=\; f(x_c).$$

On the other hand, the steepest descent step

$$x_{sd} = x_c - \alpha g_c$$

can be made to bring about a reduction of f if $\alpha > 0$ is small enough. This suggests the following globalizing strategy for computing the next iterate x_+:

- Compute the Newton step $x_{Newt} = x_c - J_c^{-1} F_c$.
- If $f(x_{newt}) < f(x_c)$, then we accept the Newton step and set $x_+ = x_{newt}$.
- If $f(x_{newt}) \geq f(x_c$, then we determine $\alpha > 0$ so that $f(x_c - \alpha g_c) < f(x_c)$ and set $x_+ = x_c - \alpha g_c$.

Thus, we always take a Newton step if it reduces f. This ensures that Newton's method, with its rapid convergence rate, takes over once we get close enough to a zero. On the other hand, by adopting steepest descent as the "fall back position" we are always assured of making progress by guaranteeing a reduction in f.

Incorporate these ideas in `ShowN`, which does a pure Newton iteration to find an intersection point of two orbits. For choosing α, try

$$\alpha = \frac{\| J_c^{-1} F_c \|_2}{\| g_c \|_2}.$$

The philosophy here is that F is sufficiently nonlinear on $\{x$:$\| x - x_c \|_2 \leq \| J_c^{-1} F_c \|\}$ and so it is not worth trying to take a steepest descent step outside this sphere. If $f(x_c - \alpha g_c) \geq f(x_c)$, then repeatedly halve α until $f(x_c - \alpha g_c) < f(x_c)$. If f cannot be decreased after ten halvings of α_{orig}, then use $\alpha = \alpha_{orig}/1024$.

M-Files and References

Script Files

`ShowMySqrt`	Plots relative error associated with `MySqrt`.
`ShowBisect`	Illustrates the method of bisection.
`ShowNewton`	Illustrates the classical Newton iteration.
`FindConj`	Uses `fzero` to compute Mercury-Earth conjunctions.
`FindTet`	Applies `fmin` to three different objective functions.
`ShowGolden`	Illustrates Golden section search.
`ShowSD`	Steepest descent test environment.
`ShowN`	Newton test environment.
`ShowFDN`	Finite difference Newton test environment.
`ShowMixed`	Globalized Newton test environment.
`ShowGN`	Gauss-Newton test environment.
`ShowFmin`	Illustrates `fmin`.
`ShowFmins`	Illustrates `fmins`.

Function Files

`MySqrt`	Canonical square root finder.
`Bisection`	The method of bisection.
`StepIsIn`	Checks next Newton step.
`GlobalNewton`	Newton method with bisection globalizer.
`GraphSearch`	Interactive graphical search environment.
`Golden`	Golden section search.
`SDStep`	Steepest descent step.
`NStep`	Newton step.
`FDNStep`	Finite difference Newton step.
`GNStep`	Gauss-Newton step.
`SineMercEarth`	The sine of the Mercury-Sun-Earth angle.
`DistMercEarth`	Mercury-Earth distance.
`TA`	Surface area of tetrahedron.
`TV`	Volume of tetrahedron.
`TE`	Total edge length of tetrahedron.
`TEDiff`	Difference between tetrahedron edges.
`Orbit`	Generates and plots orbit points.
`Sep`	Separation between points on two orbits.
`gSep`	Gradient of Sep.
`gHSep`	Gradient and Hessian of sep.
`SepV`	Vector from a point on one orbit to a point on another orbit.
`JSepV`	Jacobian of SepV.
`rho`	Residual of orbit fit.
`Jrho`	Jacobian of rho.

References

J.E. Dennis Jr and R.B. Schnabel (1983). *Numerical Methods for Unconstrained Optimization*, Prentice Hall, Englewood Cliffs, NJ.

P. Gill, W. Murray, and M.H. Wright (1981). *Practical Optimization*, Academic Press, New York.

P. Gill, W. Murray, and M.H. Wright (1991). *Numerical Linear Algebra and Optimization, Vol. 1*, Addison-Wesley, Reading, MA.