3 Solution of Nonlinear Equations

The problem of solving nonlinear equations arises frequently and naturally from the study of a wide range of practical problems. The problem may involve a system of nonlinear equations in many variables or one equation in one unknown. We shall initially confine ourselves to considering the solution of one equation in one unknown. The general form of the problem may be simply stated as finding a value of the variable x such that

$$f(x) = 0$$

where f is any nonlinear function of x. The value of x is then called a solution or root of this equation and may be one of many values satisfying the equation.

3.1 Introduction

To illustrate our discussion and provide a practical insight into the solution of nonlinear equations we shall consider an equation described by Armstrong and Kulesza (1981). These authors report a problem that arises from the study of resistive mixer circuits. Given an applied current and voltage, it is necessary to find the current flowing in part of the circuit. This leads to a simple nonlinear equation, which after some manipulation may be expressed in the form

$$x - \exp(-x/c) = 0 \text{ or equivalently } x = \exp(-x/c) \tag{3.1}$$

Here c is a given constant and x is the variable we wish to determine. The solution of such equations is not obvious, but Armstrong and Kulesza provide an approximate solution based on a series expansion that gives a reasonably accurate solution of this equation for a large range of values of c. This approximation is given in terms of c by

$$x = cu[1 - \log_e\{(1+c)u\}/(1+u)] \tag{3.2}$$

where $u = \log_e(1 + 1/c)$. This is an interesting and useful result since it is reasonably accurate for values of c in the five-decade range $[10^{-3}, 100]$ and gives a relatively easy way of finding the solutions of a whole family of equations generated by varying c. Although this result is useful for this particular equation, when we attempt to use this type of *ad hoc*

Numerical Methods Using MATLAB®. DOI: 10.1016/B978-0-12-386942-5.00003-5

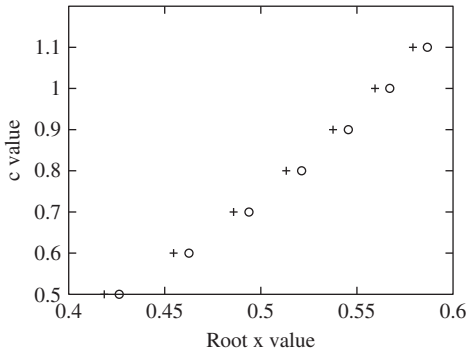

FIGURE 3.1 Solution of $x = \exp(-x/c)$. Results from the function `fzero` are indicated by $\circ$ and those from the Armstrong and Kulesza formula by $+$.

approach for the general solution of nonlinear equations, there are significant drawbacks. These are

1. *Ad hoc* approaches to the solutions of equations are rarely as successful as this example in finding a formula for the solution of a given equation; usually it is impossible to obtain such formulae.
2. Even when they exist, such formulae require considerable time and ingenuity to develop.
3. We may require greater accuracy than any ad hoc formula can provide.

To illustrate point 3 consider Figure 3.1, which is generated by the MATLAB script that follows. This figure shows the results obtained using the formula (3.2) together with the results using the MATLAB function `fzero` to solve the nonlinear equation (3.1).

```
% e3s301.m
ro = [ ]; ve = [ ]; x = [ ];
c = 0.5:0.1:1.1; u = log(1+1./c);
x = c.*u.*(1-log((1+c).*u)./(1+u));
% solve equation using MATLAB function fzero
i = 0;
for c1 = 0.5:0.1:1.1
    i = i+1;
    ro(i) = fzero(@(x) x-exp(-x/c1),1,0.00005);
end
plot(x,c,'+')
axis([0.4 0.6 0.5 1.2])
hold on
plot(ro,c,'o')
xlabel('Root x value'),  ylabel('c value')
hold off
```

The function `fzero` is discussed in detail in Section 3.10. Note that the call `fzero` takes the form `fzero(@(x) x-exp(-x/c1),1,0.00005)`. This gives an accuracy of 0.00005 for the roots and uses an initial approximation of 1. The function `fzero` provides the root with up to 16-digit accuracy, if required, whereas the formula (3.2) of Armstrong and Kulesza, although faster, gives the result to one or two decimal places only. In fact, the method of Armstrong and Kulesza becomes more accurate for large values of c.

From the preceding discussion we conclude that, although occasionally ingenious alternatives may be available, in the vast majority of cases we must use algorithms which provide, with reasonable computational effort, the solutions of general problems to any specified accuracy. Before describing the nature of these algorithms in detail, we consider different types of equations and the general nature of their solutions.

3.2 The Nature of Solutions to Nonlinear Equations

We illustrate the nature of the solutions to nonlinear equations by considering two equations that we wish to solve for the variable x.

(a) $(x-1)^3(x+2)^2(x-3)=0$ – that is,
$x^6-2x^5-8x^4+14x^3+11x^2-28x+12=0$

(b) $\exp(-x/10)\sin(10x)=0$

The first equation is a special type of nonlinear equation known as a polynomial equation since it involves only integer powers of the variable x and no other functions. Such polynomial equations have the important characteristic that they have n roots where n is the degree of the polynomial. In this example the highest power of x, and hence the degree of the polynomial, is six. The solutions of a polynomial may be complex or real, separate or coincident. Figure 3.2 illustrates the nature of the solutions of this equation. Although there must be six roots, three are coincident at $x=1$ and two are coincident at $x=-2$. There is also a single root at $x=3$. Coincident roots may present difficulties for

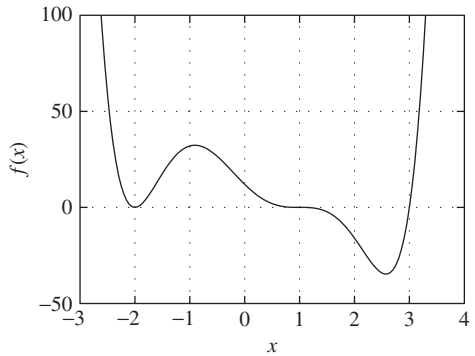

FIGURE 3.2 Plot of the function $f(x)=(x-1)^3(x+2)^2(x-3)$.

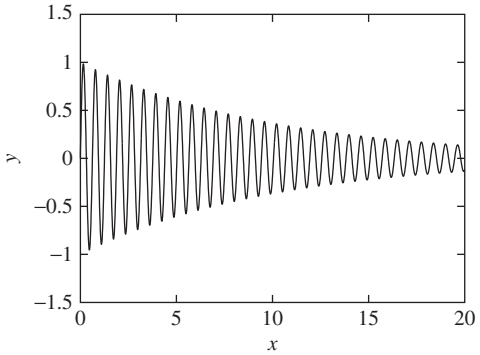

FIGURE 3.3 Plot of $f(x) = \exp(-x/10)\sin(10x)$.

some algorithms, as do roots which are very close together, so it is important to appreciate their existence. The user may require a particular root of the equation or all the roots. In the case of polynomial equations special algorithms exist to find all the roots.

The preceding second equation is a nonlinear equation involving transcendental functions. The task of finding all the roots of this class of nonlinear equation is a daunting one, since the number of roots may not be known or there may be an infinity of roots. This situation is illustrated by Figure 3.3, which shows the graph of the second equation for x in the range $[0, 20]$. If we extended the range of x, more roots would be revealed.

We now consider some simple algorithms to find a specific root of a given nonlinear equation.

3.3 The Bisection Algorithm

This simple algorithm assumes that an initial interval is known in which a root of the equation $f(x) = 0$ lies and then proceeds to reduce this interval until the required accuracy is achieved for the root. This algorithm is mentioned only briefly since it is not in practice used by itself but in conjunction with other algorithms to improve their reliability. The algorithm may be described by

input interval in which the root lies

while interval too large

1. Bisect the current interval in which the root lies.
2. Determine in which half of the interval the root lies.

end

display root

The principles on which this algorithm based is simple. Given an initial interval in which a specific root lies, the algorithm will provide an improved approximation for the root.

However, the requirement that an interval be known is sometimes difficult to achieve, and although the algorithm is reliable it is extremely slow.

Alternative algorithms have been developed that converge more rapidly; this chapter is concerned with describing some of the most important of these. All the algorithms we consider are iterative in character—that is, they proceed by repeating the same sequence of steps until the root approximation is accurate enough to satisfy the user. We now consider the general form of an iterative method, the nature of the convergence of such methods, and the problems they encounter.

3.4 Iterative or Fixed Point Methods

We are required to solve the general equation $f(x) = 0$; however, to illustrate iterative methods clearly we consider a simple example. Suppose we wish to solve the quadratic

$$x^2 - x - 1 = 0 \tag{3.3}$$

This equation can be solved by using the standard formula for solving quadratics but we take a different approach. Rearrange (3.3) as follows:

$$x = 1 + 1/x$$

Then rewrite it in iterative form using subscripts as follows:

$$x_{r+1} = 1 + 1/x_r \quad \text{for} \quad r = 0, 1, 2, \ldots \tag{3.4}$$

Assuming we have an initial approximation x_0 to the root we are seeking, we can proceed from one approximation to another using this formula. The iterates we obtain in this way may or may not converge to the solution of the original equation. This is not the only iterative procedure for attempting to solve (3.3); we can generate two others from (3.3) as follows:

$$x_{r+1} = x_r^2 - 1 \quad \text{for} \quad r = 0, 1, 2, \ldots \tag{3.5}$$

and

$$x_{r+1} = \sqrt{x_r + 1} \quad \text{for} \quad r = 0, 1, 2, \ldots \tag{3.6}$$

Starting from the same initial approximation, these iterative procedures may or may not converge to the same root. Table 3.1 shows what happens when we use the initial approximation $x_0 = 2$ with the iterative procedures (3.4), (3.5), and (3.6). It shows that iterations (3.4) and (3.6) converge but (3.5) does not.

Note that when the root is reached no further improvement is possible and the point remains fixed. Hence the roots of the equation are the *fixed points* of the iteration. To remove the unpredictability of this approach we must be able to find general conditions

Table 3.1 Difference between Exact Root and Iterate for $x^2 - x - 1 = 0$

Iteration (3.4)	Iteration (3.5)	Iteration (3.6)
–0.1180	1.3820	0.1140
0.0486	6.3820	0.0349
–0.0180	61.3820	0.0107
0.0070	3966.3820	0.0033
–0.0026	15745021.3820	0.0010

that determine when such iterative schemes converge, when they do not, and the nature of this convergence.

3.5 The Convergence of Iterative Methods

The procedure described in Section 3.4 can be applied to any equation $f(x) = 0$ and has the general form

$$x_{r+1} = g(x_r) \quad \text{for} \quad r = 0, 1, 2, \ldots \tag{3.7}$$

It is not our purpose to give the details of the derivation of convergence conditions for this form of iteration, but to point out some of the difficulties that may arise in using them even when this condition is satisfied. The detailed derivation is given in many textbooks; see, for example, Lindfield and Penny (1989). It can be shown that the approximate relation between the current error ε_{r+1} at the $(r+1)$th iteration and the previous error ε_r is given by

$$\varepsilon_{r+1} = \varepsilon_r g'(t_r)$$

where t_r is a point lying between the exact root and the current approximation to the root. Thus the error will be decreasing if the absolute value of the derivative at these points is less than 1. However, this does not guarantee convergence from all starting points and the initial approximation must be sufficiently close to the root for convergence to occur.

In the case of the specific iterative procedures (3.4) and (3.5), Table 3.2 shows how the values of the derivatives of the corresponding $g(x)$ vary with the values of the approximations to x_r. This table provides numerical evidence for the theoretical assertion in the case of iterations (3.4) and (3.5).

However, the concept of convergence is more complex than this. We need to give some answer to the crucial question: If an iterative procedure converges, how can we classify the rate of convergence? We do not derive this result but refer the reader to Lindfield and Penny (1989) and state the answer to the question. Suppose all derivatives of the function $g(x)$ of order 1 to $p-1$ are zero at the exact root a. Then the relation between the current

Table 3.2 Values of the Derivatives for Iterations Given by (3.4) and (3.5)

Iteration (3.4)	Derivative	Iteration (3.5)	Derivative
–0.1180	–0.44	1.3820	6.00
0.0486	–0.36	6.3820	16.00
–0.0180	–0.39	61.3820	126.00
0.0070	–0.38	3966.3820	7936.00

error ε_{r+1} at the $(r+1)$th iteration and the previous error ε_r is given by

$$\varepsilon_{r+1} = (\varepsilon_r)^p g^{(p)}(t_r)/p! \tag{3.8}$$

where t_r lies between the exact root and the current approximation to the root and $g^{(p)}$ denotes the pth derivative of g. The importance of this result is that it means the current error is proportional to the pth power of the previous error and clearly, on the basis of the reasonable assumption that the errors are much smaller than 1, the higher the value of p, the faster the convergence. Such methods are said to have pth-order convergence. In general it is cumbersome to derive iterative methods of order higher than two or three and second-order methods have proved very satisfactory in practice for solving a wide range of nonlinear equations. In this case, the current error is proportional to the square of the previous error. This is often called quadratic convergence; if the error is proportional to the previous error it is called linear convergence. This provides a convenient classification for the convergence of iterative methods but avoids the difficult questions: For what range of starting values will the process converge and how sensitive is convergence to changes in the starting values?

3.6 Ranges for Convergence and Chaotic Behavior

We illustrate some of the problems of convergence by considering a specific example that highlights some of the difficulties. Short (1992) examined the behavior of the iterative process

$$x_{r+1} = -0.5(x_r^3 - 6x_r^2 + 9x_r - 6) \quad \text{for} \quad r = 0, 1, 2, \ldots$$

for solving the equation $(x-1)(x-2)(x-3) = 0$. This iterative procedure clearly has the form

$$x_{r+1} = g(x_r), \quad r = 0, 1, 2, \ldots$$

and it is easy to verify that it has the following properties:

$$g'(1) = 0 \quad \text{and} \quad g''(1) \neq 0$$
$$g'(2) \neq 0$$
$$g'(3) = 0 \quad \text{and} \quad g''(3) \neq 0$$

Thus by taking $p = 2$ in result (3.8), we can expect, for appropriate starting values, quadratic convergence for the roots at $x = 1$ and $x = 3$ but at best linear convergence for the root at $x = 2$. The major problem is, however, to determine the ranges of initial approximation that will converge to the different roots. This is not an easy task but one simple way of doing this is to draw a graph of $y = x$ and $y = g(x)$. The points of intersection provide the roots. The line $y = x$ has a slope of 1, and points where the slope of $g(x)$ is less than this provide a range of initial approximations that converge to one or other of the roots.

This graphical analysis shows that points within the range 1 to 1.43 (approximately) converge to the root 1 and points in the range 2.57 (approximately) to 3 converge to the root 3. This is the obvious part of the analysis. However, Short demonstrates that there are many other ranges of convergence for this iterative procedure, many of them very narrow indeed, which lead to chaotic behavior in the iterative process. He demonstrates, for example, that taking $x_0 = 4.236067968$ will converge to the root $x = 3$ whereas taking $x_0 = 4.236067970$ converges to the root $x = 1$, a remarkable change for such a small variation in the initial approximation. This should serve as a warning to the reader that the study of convergence properties is in general not an easy task.

Figure 3.4 illustrates this point quite strikingly. It shows the graph of x and the graph of $g(x)$ where

$$g(x) = -0.5(x^3 - 6x^2 + 9x - 6)$$

The x line intersects with $g(x)$ to give the roots of the original equation. The graph also shows iterates starting from $x_0 = 4.236067968$, indicated by "$\circ$," and iterates starting from

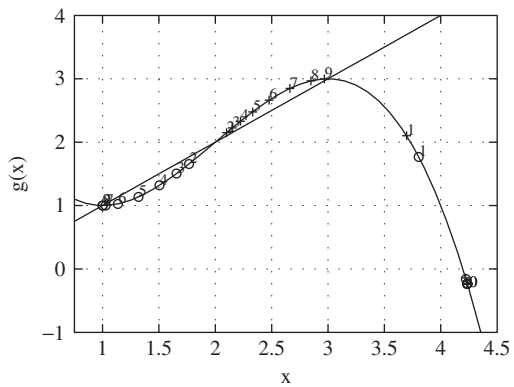

FIGURE 3.4 Iterates in the solution of $(x-1)(x-2)(x-3) = 0$ from close but different starting points.

$x = 4.236067970$, indicated by "+." The starting points are so close they are of course superimposed on the graph. However, the iterates soon take their separate paths to converge on different roots of the equation. The path indicated by "$\circ$" converges to the root $x = 1$ and the path indicated by "+" converges to the root $x = 3$. The sequence of numbers on the graph shows the last nine iterates. The point referenced by zero is in fact all the points that are initially very close together. This is a remarkable example and users should verify these phenomena for themselves by running the following MATLAB script:

```
% e3s302.m
x = 0.75:0.1:4.5;
g = -0.5*(x.^3-6*x.^2+9*x-6);
plot(x,g)
axis([.75,4.5,-1,4])
hold on, plot(x,x)
xlabel('x'), ylabel('g(x)'), grid on
ch = ['o','+'];
num = [ '0','1','2','3','4','5','6','7','8','9'];
ty = 0;
for x1 = [4.236067970 4.236067968]
    ty = ty+1;
    for i = 1:19
        x2 = -0.5*(x1^3-6*x1^2+9*x1-6);
        % First ten points very close, so represent by '0'
        if i==10
            text(4.25,-0.2,'0')
        elseif i>10
            text(x1,x2+0.1,num(i-9))
        end
        plot(x1,x2,ch(ty))
        x1 = x2;
    end
end
hold off
```

It is interesting to note that the iterative form

$$x_{r+1} = x_r^2 + c \quad \text{for} \quad r = 0, 1, 2, \ldots$$

demonstrates strikingly chaotic behavior when the iterates are plotted in the complex plane and for complex ranges of values for c.

We now return to the more mundane task of developing algorithms that work in general for the solution of nonlinear equations. In the next section we shall consider a simple method of order 2.

3.7 Newton's Method

This method for the solution of the equation $f(x) = 0$ is based on the simple geometric properties of the tangent to the curve $f(x)$. The method requires some initial approximation to the root and that the derivative of $f(x)$ exists in the range of interest. Figure 3.5 illustrates the operation of the method. The diagram shows the tangent to the curve at the current approximation x_0. This tangent strikes the x-axis at x_1 and provides us with an improved approximation to the root. Similarly, the tangent at x_1 gives the improved approximation x_2.

The process is repeated until some convergence criterion is satisfied. It is easy to translate this geometrical procedure into a numerical method for finding the root since the tangent of the angle between the x-axis and the tangent equals

$$f(x_0)/(x_1 - x_0)$$

and the slope of this tangent itself equals $f'(x_0)$, the derivative of $f(x)$ at x_0. So we have the equation

$$f'(x_0) = f(x_0)/(x_1 - x_0)$$

Thus the improved approximation, x_1, is given by

$$x_1 = x_0 - f(x_0)/f'(x_0)$$

This may be written in iterative form as

$$x_{r+1} = x_r - f(x_r)/f'(x_r) \quad \text{where} \quad r = 0, 1, 2, \ldots \tag{3.9}$$

We note that this method is of the general iterative form

$$x_{r+1} = g(x_r) \quad \text{where} \quad r = 0, 1, 2, \ldots$$

Consequently, the discussion of Section 3.5 applies to it. On computing $g'(a)$, where a is the exact root, we find it is zero. However, $g''(a)$ is in general nonzero so the method is of order

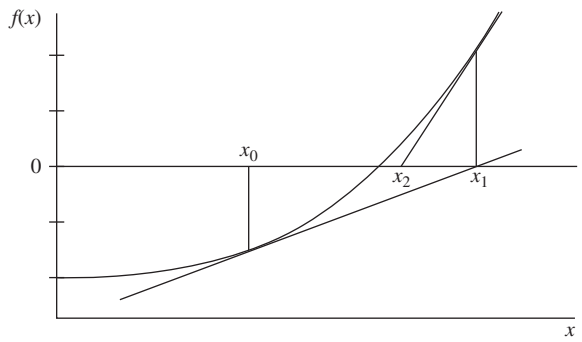

FIGURE 3.5 Geometric interpretation of Newton's method.

2 and we expect convergence to be quadratic. For a sufficiently close initial approximation, convergence to the root will be rapid.

A MATLAB function `fnewton` is supplied for Newton's method. The function that forms the left side of the equation we wish to solve *and* its derivative must be supplied by the user as functions; these become the first and second parameters of the function. The third parameter is an initial approximation to the root. The convergence criterion used is that the difference between successive approximations to the root is less than a small preset value. This value must be supplied by the user and is given as the fourth parameter of the function.

```
function [res, it] = fnewton(func,dfunc,x,tol)
% Finds a root of f(x) = 0 using Newton's method.
% Example call: [res, it] = fnewton(func,dfunc,x,tol)
% The user defined function func is the function f(x).
% The user defined function dfunc is df/dx.
% x is an initial starting value, tol is required accuracy.
it = 0; x0 = x;
d = feval(func,x0)/feval(dfunc,x0);
while abs(d) > tol
    x1 = x0-d;  it = it+1;  x0 = x1;
    d = feval(func,x0)/feval(dfunc,x0);
end
res = x0;
```

We will now find a root of the equation

$$x^3 - 10x^2 + 29x - 20 = 0$$

To use Newton's method we must define the function and its derivative as follows:

```
>> f = @(x) x.^3-10*x.^2+29*x-20;
>> df = @(x) 3*x.^2-20*x+29;
```

We may call the function `fnewton` as follows:

```
>> [x,it] = fnewton(f,df,7,0.00005)

x =
    5.0000

it =
     6
```

The progress of the iterations when solving $x^3 - 10x^2 + 29x - 20 = 0$ by Newton's method is shown in Figure 3.6.

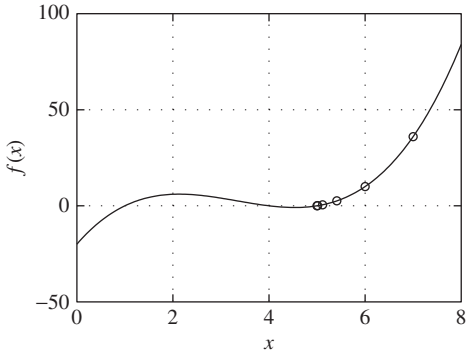

FIGURE 3.6 Plot of $x^3 - 10x^2 + 29x - 20 = 0$ with the iterates of Newton's method shown by $\circ$.

Table 3.3 Newton's Method to Solve $x^3 - 10x^2 + 29x - 20 = 0$ with an Initial Approximation of -2

x value	Error ε_r	$2\varepsilon_{r+1}/\varepsilon_r^2$	Approximate Second Derivative of g
−2.000000	3.000000	−0.320988	−0.395062
−0.444444	1.444444	−0.513956	−0.589028
0.463836	0.536164	−0.792621	−0.845260
0.886072	0.113928	−1.060275	−1.076987
0.993119	0.006881	−1.159637	−1.160775
0.999973	0.000027	−1.166639	−1.166643
1.000000	0.000000	−1.166639	−1.166667

Table 3.3 gives numerical results for this problem when Newton's method is used to seek a root, starting the iteration at -2. The second column of the table gives the current error ε_r by subtracting the known exact root from the current iterate. The third column contains the value of $2\varepsilon_{r+1}/\varepsilon_r^2$. This value tends to a constant as the process proceeds. From theoretical considerations, this value should approach the second derivative of the right side of the Newton iterative formula. This follows from (3.8) with $p = 2$. The final column contains the value of the second-order derivative of $g(x)$ calculated as follows. From (3.9) we have $g(x) = x - f(x)/f'(x)$. Thus from this we have

$$g'(x) = 1 - [\{f'(x)\}^2 - f''(x)f(x)]/[f'(x)]^2 = f''(x)f(x)/[f'(x)]^2$$

On differentiating again,

$$g''(x) = [\{f'(x)\}^2\{f'''(x)f(x) + f''(x)f'(x)\} - 2f'(x)\{f''(x)\}^2 f(x)]/[f'(x)]^4$$

Putting $x = a$, where a is the exact root, since $f(a) = 0$, we have

$$g''(a) = f''(a)/f'(a) \tag{3.10}$$

Thus we have a value for the second derivative of $g(x)$ when $x = a$. We note that as x approaches the root, the final column of Table 3.3, which uses this formula, gives an increasingly accurate approximation to the second derivative of $g(x)$. The table thus verifies our theoretical expectations.

We can find complex roots using Newton's method, providing our initial approximation is complex. For example, consider

$$\cos x - x = 0 \tag{3.11}$$

This equation has only one real root, which is $x = 0.7391$, but it has an infinity of complex roots. Figure 3.7 shows the distribution of the roots of (3.11) in the complex plane in the range $-30 < \mathrm{Re}(x) < 30$. Working with complex values presents no additional difficulty in the MATLAB environment since MATLAB implements complex arithmetic and so we can use the function `fnewton` without modification to deal with these cases.

Figure 3.8 illustrates the fact that it is difficult to predict which root we will find from a given starting value. This figure shows that the starting values $15 + \jmath 10$, $15.2 + \jmath 10$, $15.4 + \jmath 10$, $15.8 + \jmath 10$, and $16 + \jmath 10$, which are close together, lead to a sequence of iterations that converge to very different roots. In one case the complete trajectory is not shown because the complex part of the intermediate iterates is well outside the range of the graph.

Newton's method requires the first derivative of $f(x)$ to be supplied by the user. To make the procedure more self-contained we can use a standard approximation to the first derivative, which takes the form

$$f'(x_r) = \{f(x_r) - f(x_{r-1})\}/(x_r - x_{r-1}) \tag{3.12}$$

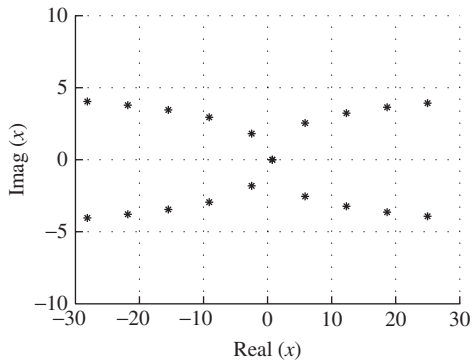

FIGURE 3.7 Plot showing the complex roots of $\cos x - x = 0$.

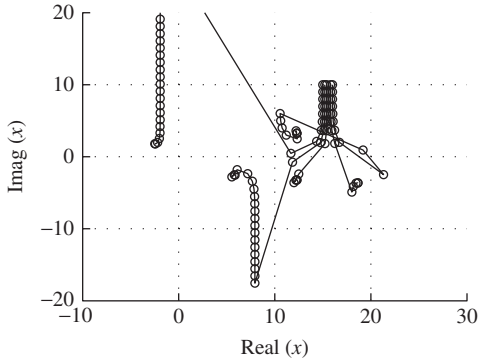

FIGURE 3.8 Plot of the iterates for five complex initial approximations for the solution of $\cos x - x = 0$ using Newton's method. Each iterate is shown by "$\circ$."

Substituting this result in (3.9) gives the new procedure for calculating the improvements to x as

$$x_{r+1} = [x_{r-1}f(x_r) - x_r f(x_{r-1})]/[f(x_r) - f(x_{r-1})] \tag{3.13}$$

This method does not require the calculation of the first derivative of $f(x)$ but does require that we know two initial approximations to the root, x_0 and x_1. Geometrically, we have simply approximated the slope of the tangent to the curve by the slope of a secant. For this reason the method is known as the *secant method*. The convergence of this method is slower than Newton's method. Another procedure similar to the secant method is called *regula falsi*. In this method two values of x that enclose the root are chosen to start the next iteration rather than the most recent pair of x values as in the secant method.

Newton's method and the secant method work well on a wide range of problems. However, for problems where the roots of an equation are close together or equal, the convergence may be slow. We now consider a simple adjustment to Newton's method that provides good convergence even with multiple roots.

3.8 Schroder's Method

In Section 3.2 we described how coincident roots present significant problems for most algorithms. In the case of Newton's method its performance is no longer quadratic for finding a coincident root and the procedure must be modified if it is to maintain this property. The iteration for Schroder's method for finding multiple roots has a form similar to that of Newton's method given in (3.9) except for the inclusion of a multiplying factor m. Thus

$$x_{r+1} = x_r - mf(x_r)/f'(x_r) \quad \text{where} \quad r = 0, 1, 2, \ldots \tag{3.14}$$

Here m is an integer equal to the multiplicity of the root to which we are trying to converge. Since the user may not know the value of m, it may have to be found experimentally.

It can be verified by some simple but lengthy algebraic manipulation that for a function $f(x)$ with multiple roots at $x = a$, $g'(a) = 0$. Here $g(x)$ is the right side of (3.14) and a is the exact root. This modification is sufficient to preserve the quadratic convergence of Newton's method

A MATLAB function for Schroder's method, `schroder`, is provided as follows:

```
function [res, it] = schroder(func,dfunc,m,x,tol)
% Finds a multiple root of f(x) = 0 using Schroder's method.
% Example call: [res, it] = schroder(func,dfunc,m,x,tol)
% The user defined function func is the function f(x).
% The user defined function dfunc is df/dx.
% x is an initial starting value, tol is required accuracy.
% function has a root of multiplicity m.
% x is a starting value, tol is required accuracy.
it = 0; x0 = x;
d = feval(func,x0)/feval(dfunc,x0);
while abs(d)>tol
    x1 = x0-m*d; it = it+1; x0 = x1;
    d = feval(func,x0)/feval(dfunc,x0);
end
res = x0;
```

We will now use the function `schroder` to solve $(e^{-x} - x)^2 = 0$. In this case we must set the multiplying factor m to 2. We write the function `f` and its derivative `df` and call the function `schroder` as follows:

```
>> f = @(x) (exp(-x)-x).^2;
>> df = @(x) 2*(exp(-x)-x).*(-exp(-x)-1);
>> [x, it] = schroder(f,df,2,-2,0.00005)

x =
    0.5671

it =
     5
```

It is interesting to note that Newton's method took 17 iterations to solve this problem in contrast to the 5 required by Schroder's method.

When a function $f(x)$ is known to have repeated roots, an alternative to Schroder's approach is to apply Newton's method to the function $f(x)/f'(x)$ rather than to the function $f(x)$ itself. It can be easily shown by direct differentiation that if $f(x)$ has a root of any multiplicity then $f(x)/f'(x)$ will have the same root but with multiplicity 1. Thus the algorithm has the iterative form (3.9) but modified by replacing $f(x)$ with $f(x)/f'(x)$. The advantage of this approach is that the user does not have to know the multiplicity of

the root that is to be found. The considerable disadvantage is that both the first- and second-order derivatives must be supplied by the user.

3.9 Numerical Problems

We now consider the following problems that arise in solving single-variable nonlinear equations.

1. Finding good initial approximations
2. Ill-conditioned functions
3. Deciding on the most suitable convergence criteria
4. Discontinuities in the equation to be solved

These problems are now examined in detail.

1. Finding an initial approximation can be difficult for some nonlinear equations and a graph can be a considerable help in supplying such a value. The advantage of working in a MATLAB environment is that the script for the graph of the function can easily be generated and input can be taken from it directly. The function `plotapp` that is defined here finds an approximation to the root of a function supplied by the user in the range given by the parameters `rangelow` and `rangeup` using a step given by the `interval`.

```
function approx = plotapp(func,rangelow,interval,rangeup)
% Plots a function and allows the user to approximate a
% particular root using the cursor.
% Example call: approx = plotapp(func,rangelow,interval,rangeup)
% Plots the user defined function func in the range rangelow to
% rangeup using a step given by interval. Returns approx to root.
approx = [ ];
x = rangelow:interval:rangeup;
plot(x,feval(func,x))
hold on, xlabel('x'), ylabel('f(x)')
title(' ** Place cursor close to root and click mouse ** ')
grid on
% Use ginput to get approximation from graph using mouse
approx = ginput(1);
fprintf('Approximate root is %8.2f\n',approx(1)), hold off
```

The script that follows shows how this function may be used with the MATLAB function `fzero` to find a root of $x - \cos x = 0$.

```
% e3s303.m
g = @(x) x-cos(x);
approx = plotapp(g,-2,0.1,2);
```

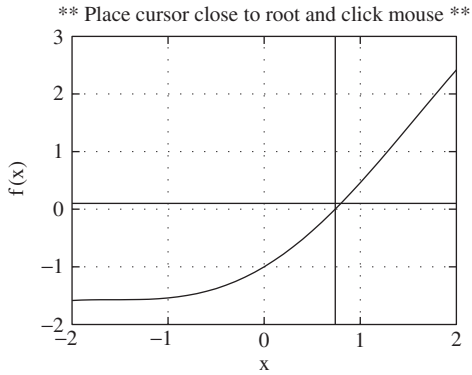

FIGURE 3.9 The cursor is shown close to the position of the root.

```
% Use this approximation and fzero to find exact root
root = fzero(g,approx(1),0.00005);
fprintf('Exact root is %8.5f\n',root)
```

Figure 3.9 gives the graph of $x - \cos x = 0$ generated by `plotapp` and shows the crosshairs cursor generated by the `ginput` function close to the root. The call `ginput(1)` means only one point is taken. The cursor can be positioned over the intersection of the curve with the $f(x) = 0$ line. This provides a useful initial approximation, the accuracy of which depends on the scale of the graph. In this example an initial approximation was found to be 0.74 and the more exact value was found using `fzero` to be 0.73909.

2. Ill-conditioning in a nonlinear equation means that small changes in the coefficients of the equation lead to unexpectedly large errors in the solutions. An interesting example of a very ill-conditioned polynomial is Wilkinson's polynomial. The MATLAB function `poly(v)` generates the coefficients of a polynomial, beginning with the coefficient of the highest power, with roots that are equal to the elements of the vector `v`. Thus `poly(1:n)` generates the coefficients of the polynomial with the roots $1, 2, \ldots, n$, which is Wilkinson's polynomial of degree $n - 1$.
3. In the design of any numerical algorithm for the solution of nonlinear equations, the termination criterion is particularly important. There are two major indicators of convergence: the difference between successive iterates and the value of the function at the current iterate. Taken separately these indicators may be misleading. For example, some nonlinear functions are such that small changes in the independent variable value may lead to large changes in the function value. In this case it may be better to monitor both indicators.
4. The function $f(x) = \sin(1/x)$ is particularly difficult to plot, and $\sin(1/x) = 0$ is very difficult to solve since it has an infinite number of roots, all clustered between 1 and -1. The function has a discontinuity at $x = 0$. Figure 3.10 attempts to illustrate

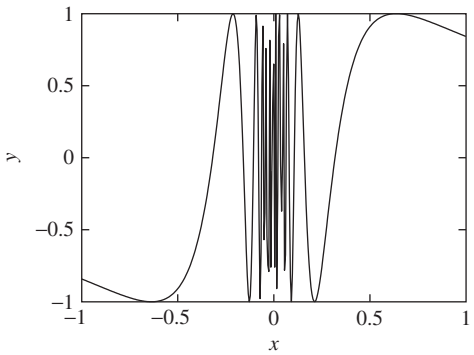

FIGURE 3.10 Plot of graph $f(x) = \sin(1/x)$. This plot is spurious in the range ± 0.2.

the behavior of this function. In fact, the graph shown does not truly represent the function and this plotting problem is discussed in more detail in Chapter 4. Near a discontinuity the function changes rapidly for small changes in the independent variable and some algorithms may have problems with this.

All the preceding points emphasize that algorithms for solving nonlinear equations need to be not only fast and efficient but robust as well. The next algorithm combines these properties and is relatively undemanding on the user.

3.10 The MATLAB Function `fzero` and Comparative Studies

Some problems may present particular difficulties for algorithms that in general work well. For example, algorithms that have fast ultimate convergence may initially diverge. One way to improve the reliability of an algorithm is to ensure that at each stage the root is confined to a known interval and the method of bisection, introduced in Section 3.3, may be used to provide an interval in which the root lies. Thus a method that combines bisection with a rapidly convergent procedure may be able to provide both rapid *and* reliable convergence.

The method of Brent combines inverse quadratic interpolation with bisection to provide a powerful method that has been found to be successful on a wide range of difficult problems. The method is easily implemented and a detailed description of the algorithm may be found in Brent (1971). Similar algorithms of comparable efficiency have been developed by Dekker (1969).

Experience with Brent's algorithm has shown it to be both reliable and efficient on a wide range of problems. A variation of this method is directly available in MATLAB and is called `fzero`. It may be used as follows:

```
x = fzero('funcname',x0,tol,trace);
```

where funcname is replaced by the name of any system function such as cos, sin, and so on, or the name of a function predefined by the user. The initial approximation is x0. The accuracy of the solution is set by tol and if trace is a value greater than 1, an output of the intermediate approximations is given. Only the first two parameters need be given and so an alternative call of this function is given by

```
x = fzero('funcname',x0);
```

To plot the function $(e^x - \cos x)^3$ and then determine some roots of $(e^x - \cos x)^3 = 0$ with tolerance 0.0005, initial approximations of 1.65 and -3, and no trace of the iterations, we use fzero as follows:

```
% e3s304.m
f = @(x) (exp(x)-cos(x)).^3;
x = -4:0.02:0.5;
plot(x,f(x)), grid on
xlabel('x'), ylabel('f(x)');
title('f(x) = (exp(x)-cos(x)).^3')
root = fzero(f,1.65,0.00005);
fprintf('A root of this equation is %6.4f\n',root)
root = fzero(f,-3,0.00005);
fprintf('A root of this equation is %6.4f\n',root)
```

The output and plot generated by this script are not given. However, the script is provided for reader experimentation.

Before we deal with the problem of finding many roots of a polynomial equation simultaneously, we present a comparative study of the MATLAB function fzero with the function fnewton. The following functions are considered:

1. $\sin(1/x) = 0$
2. $(x-1)^5 = 0$
3. $x - \tan x = 0$
4. $\cos\{(x^2+5)/(x^4+1)\} = 0$

The results of these comparative studies are given in Table 3.4. We see that fnewton is less reliable than fzero and that fzero produces accurate answers.

Table 3.4 Solution of Equations (1) through (4) with the Same Starting Point $x = -2$ and Accuracy $= 0.00005$

Function	1	2	3	4
fnewton	Fail	0.999795831	Fail	−1.352673831
fzero	−0.318309886	1.000000000	−1.570796327	−1.352678708

3.11 Methods for Finding All the Roots of a Polynomial

The problem of solving polynomial equations is a special one in that these equations contain only combinations of integer powers of x and no other functions. Because of their special structure, algorithms have been developed to find all of the roots of a polynomial equation simultaneously. The function `roots` is provided in MATLAB. This function sets up the companion matrix for the polynomial and determines its eigenvalues, which can be shown to be the roots of the polynomial. For a description of the companion matrix, see Appendix A.

The following sections describe the methods of Bairstow and Laguerre but do not give a detailed theoretical justification of them. We provide a MATLAB function for Bairstow's method.

3.11.1 Bairstow's Method

Consider the polynomial

$$a_0x^n + a_1x^{n-1} + a_2x^{n-2} + \cdots + a_n = 0 \tag{3.15}$$

Since this is a polynomial equation of degree n, it has n roots. A common approach for locating the roots of a polynomial is to find all its quadratic factors. These will have the form

$$x^2 + ux + v \tag{3.16}$$

where u and v are the constants we wish to determine. Once all the quadratic factors are found it is easy to solve the quadratics to find all the roots of the equation. We now outline the major steps used in Bairstow's method for finding these quadratic factors.

If $R(x)$ is the remainder after the division of polynomial (3.15) by the quadratic factor (3.16), then there will clearly exist constants b_0, b_1, b_2,... such that the following equality holds:

$$(x^2 + ux + v)(b_0x^{n-2} + b_1x^{n-3} + b_2x^{n-4} + \cdots + b_{n-2}) + R(x) = x^n + a_1x^{n-1} + a_2x^{n-2} + \cdots + a_n \tag{3.17}$$

where a_0 has been taken as one and $R(x)$ will have the form $rx + s$. To ensure that $x^2 + ux + v$ is an exact factor of the polynomial (3.15), the remainder $R(x)$ must be zero. For this to be true both r and s must be zero and we must adjust u and v until this is true. Thus since both r and s depend on u and v, the problem reduces to solving the equations

$$r(u, v) = 0$$
$$s(u, v) = 0$$

To solve these equations we use an iterative method that assumes some initial approximations u_0 and v_0. Then we require improved approximations u_1 and v_1 where $u_1 = u_0 + \Delta u_0$ and $v_1 = v_0 + \Delta v_0$ such that

$$r(u_1, v_1) = 0$$
$$s(u_1, v_1) = 0$$

or r and s are as close to zero as possible.

Now we wish to find the changes Δu_0 and Δv_0 that will result in this improvement. Consequently, we must expand the two equations

$$r(u_0 + \Delta u_0, v_0 + \Delta v_0) = 0$$
$$s(u_0 + \Delta u_0, v_0 + \Delta v_0) = 0$$

using a Taylor series expansion and neglecting higher powers of Δu_0 and Δv_0. This leads to two approximating linear equations for Δu_0 and Δv_0:

$$\begin{aligned} r(u_0, v_0) + (\partial r/\partial u)_0 \Delta u_0 + (\partial r/\partial v)_0 \Delta v_0 = 0 \\ s(u_0, v_0) + (\partial s/\partial u)_0 \Delta u_0 + (\partial s/\partial v)_0 \Delta v_0 = 0 \end{aligned} \tag{3.18}$$

The subscript 0 denotes that the partial derivatives are calculated at the point u_0, v_0. Once the corrections are found, the iteration can be repeated until r and s are sufficiently close to zero. The method we have used here is a two-variable form of Newton's method, which will be described in Section 3.12.

Clearly this method requires the first-order partial derivatives of r and s with respect to u and v. The form of these is not obvious; however, they may be determined using recurrence relations derived from equating coefficients in (3.17) and then differentiating them. The details of this derivation are not given here but a clear description of the process is given by Froberg (1969). Once the quadratic factor is found, the same process is applied to the residual polynomial with the coefficients b_i to obtain the remaining quadratic factors. The details of this derivation are not provided here but a MATLAB function `bairstow` is given next.

```
function [rts,it] = bairstow(a,n,tol)
% Bairstow's method for finding the roots of a polynomial of degree n.
% Example call: [rts,it] = bairstow(a,n,tol)
% a is a row vector of REAL coefficients so that the
% polynomial is x^n+a(1)*x^(n-1)+a(2)*x^(n-2)+...+a(n).
% The accuracy to which the polynomial is satisfied is given by tol.
% The output is produced as an (n x 2) matrix rts.
% Cols 1 & 2 of rts contain the real & imag part of root respectively.
% The number of iterations taken is given by it.
it = 1;
```

```
while n>2
    %Initialise for this loop
    u = 1; v = 1; st = 1;
    while st>tol
        b(1) = a(1)-u; b(2) = a(2)-b(1)*u-v;
        for k = 3:n
            b(k) = a(k)-b(k-1)*u-b(k-2)*v;
        end
        c(1) = b(1)-u; c(2) = b(2)-c(1)*u-v;
        for k = 3:n-1
            c(k) = b(k)-c(k-1)*u-c(k-2)*v;
        end
        %calculate change in u and v
        c1 = c(n-1); b1 = b(n); cb = c(n-1)*b(n-1);
        c2 = c(n-2)*c(n-2); bc = b(n-1)*c(n-2);
        if n>3, c1 = c1*c(n-3); b1 = b1*c(n-3); end
        dn = c1-c2;
        du = (b1-bc)/dn; dv = (cb-c(n-2)*b(n))/dn;
        u = u+du; v = v+dv;
        st = norm([du dv]); it = it+1;
    end
    [r1,r2,im1,im2] = solveq(u,v,n,a);
    rts(n,1:2) = [r1 im1]; rts(n-1,1:2) = [r2 im2];
    n = n-2;
    a(1:n) = b(1:n);
end
% Solve last quadratic or linear equation
u = a(1); v = a(2);
[r1,r2,im1,im2] = solveq(u,v,n,a);
rts(n,1:2) = [r1 im1];
if n==2
    rts(n-1,1:2) = [r2 im2];
end
% ------------------------------------------------------
function [r1,r2,im1,im2] = solveq(u,v,n,a);
% Solves x^2 + ux + v = 0 (n ~= 1) or x + a(1) = 0 (n = 1).
% Example call: [r1,r2,im1,im2] = solveq(u,v,n,a)
% r1, r2 are real parts of the roots,
% im1, im2 are the imaginary parts of the roots.
% Called by function bairstow.
if n==1
    r1 = -a(1); im1 = 0; r2 = 0; im2 = 0;
```

```
else
    d = u*u-4*v;
    if d<0
        d = -d;
        im1 = sqrt(d)/2; r1 = -u/2; r2 = r1; im2 = -im1;
    elseif d>0
        r1 = (-u+sqrt(d))/2; im1 = 0; r2 = (-u-sqrt(d))/2; im2 = 0;
    else
        r1 = -u/2; im1 = 0; r2 = -u/2; im2 = 0;
    end
end
```

Note that the MATLAB function `solveq` is nested within the function `bairstow`. The function is not stored separately and so it can only be accessed by `bairstow`. We may now use `bairstow` to solve the specific polynomial equation

$$x^5 - 3x^4 - 10x^3 + 10x^2 + 44x + 48 = 0$$

In this case, we take the coefficient vector as `c` where `c = [-3 -10 10 44 48]` and if we require accuracy of four decimal places we take `tol` as 0.00005. The script uses `bairstow` to solve the given polynomial.

```
% e3s305.m
c = [-3 -10 10 44 48];
[rts, it] = bairstow(c,5,0.00005);
for i = 1:5
    fprintf('\nroot%3.0f Real part=%7.4f',i,rts(i,1))
    fprintf(' Imag part=%7.4f',rts(i,2))
end
fprintf('\n')
```

Note how `fprintf` is used to provide a clearer output from the matrix `rts`.

```
root  1 Real part= 4.0000 Imag part= 0.0000
root  2 Real part=-1.0000 Imag part=-1.0000
root  3 Real part=-1.0000 Imag part= 1.0000
root  4 Real part=-2.0000 Imag part= 0.0000
root  5 Real part= 3.0000 Imag part= 0.0000
```

As we have indicated, MATLAB provides a function `roots` to determine the roots of a polynomial. It is interesting to compare this function with Bairstow's method. Table 3.5 gives the results of this comparison applied to specific polynomials. The problems p1 through p5 are the polynomials:

$$\text{p1}: \quad x^5 - 3x^4 - 10x^3 + 10x^2 + 44x + 48 = 0$$

$$\text{p2}: \quad x^3 - 3.001x^2 + 3.002x - 1.001 = 0$$

Table 3.5 Time Required to Obtain All Roots (in Seconds)

	`roots`	`bairstow`
p1	7	33
p2	6	19
p3	6	14
p4	10	103
p5	11	37

$$\text{p3}: \quad x^4 - 6x^3 + 11x^2 + 2x - 28 = 0$$

$$\text{p4}: \quad x^7 + 1 = 0$$

$$\text{p5}: \quad x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1 = 0$$

Both methods determine the correct roots for all problems, although the function `roots` is more efficient.

3.11.2 Laguerre's Method

Laguerre's method provides a rapidly convergent procedure for locating the roots of a polynomial. The algorithm is interesting and for this reason it is described in this section. The method is applied to a polynomial in the form

$$p(x) = x^n + a_1 x^{n-1} + a_2 x^{n-2} + \cdots + a_n$$

Starting with an initial approximation x_1, we apply the following iterative formula to the polynomial $p(x)$:

$$x_{i+1} = x_i - np(x_i)/[p'(x_i) \pm \sqrt{\{h(x_i)\}}] \quad \text{for} \quad i = 1, 2, \ldots \tag{3.19}$$

where

$$h(x_i) = (n-1)[(n-1)\{p'(x_i)\}^2 - np(x_i)p''(x_i)]$$

and n is the degree of the polynomial. The sign taken in (3.19) is determined so that it is the same as the sign of $p'(x_i)$.

It is important to give some justification for using a formula with such a complex structure. The reader will notice that if the square root term were not present in (3.19), the iterative form would be similar to that of Newton's method, (3.9), and identical to that of Schroder's method, (3.14). Thus we would have a method with quadratic convergence for the roots of the polynomial. In fact, the more complex structure of (3.19) provides third-order convergence since the error is proportional to the cube of the previous error and

consequently provides faster convergence than Newton's method. Thus, given an initial approximation, the method will converge rapidly to a root of the polynomial, which we can denote by r.

To obtain the other roots of the polynomial we divide the polynomial $p(x)$ by the factor $(x-r)$, which provides another polynomial of degree $n-1$. We can then apply iteration (3.19) to this polynomial and repeat the whole procedure again. This is repeated until all roots are found to the required accuracy. The process of dividing by $(x-r)$ is known as deflation and can be performed in a simple and efficient way, described as follows.

Since we have a known factor $(x-r)$, then

$$a_0x^n + a_1x^{n-1} + a_2x^{n-2} + \cdots + a_n = (x-r)(b_0x^{n-1} + b_1x^{n-2} + b_2x^{n-3} + \cdots + b_{n-1}) \tag{3.20}$$

On equating coefficients of the powers of x on both sides we have

$$\begin{aligned} b_0 &= a_0 \\ b_i &= a_i + rb_{i-1} \quad \text{for} \quad i = 1, 2, \ldots, n-1 \end{aligned} \tag{3.21}$$

This process is known as synthetic division. Care must be taken here, particularly if the root is found to low accuracy, since ill-conditioning can magnify the effect of small errors in the coefficients of the deflated polynomial.

This completes the description of the method but a few important points should be noted. Assuming sufficient accuracy can be maintained in calculations, the method of Laguerre will converge for any value of the initial approximation. Convergence to complex roots and multiple roots can be achieved but at a slower rate because the convergence rate is linear. In the case of a complex root the value of the function $h(x_i)$ becomes negative and consequently the algorithm must be adjusted to deal with this situation. A key feature that should be considered is that the derivatives of the polynomial can be found efficiently by synthetic division.

To summarize the important features of the algorithm:

1. The algorithm is third order, thus providing rapid convergence to individual roots.
2. All roots of the polynomial can be found by using synthetic division.
3. Derivatives can be calculated efficiently using synthetic division.

3.12 Solving Systems of Nonlinear Equations

The methods considered so far have been concerned with finding one or all the roots of a nonlinear algebraic equation with one independent variable. We now consider methods for solving systems of nonlinear algebraic equations in which each equation is a function of a specified number of variables. We can write such a system in the form

$$f_i(x_1, x_2, \ldots, x_n) = 0 \quad \text{for} \quad i = 1, 2, 3, \ldots, n \tag{3.22}$$

A simple method for solving this system of nonlinear equations is based on Newton's method for the single equation. To illustrate this procedure we first consider a system of two equations in two variables:

$$\begin{aligned} f_1(x_1, x_2) &= 0 \\ f_2(x_1, x_2) &= 0 \end{aligned} \tag{3.23}$$

Given initial approximations x_1^0 and x_2^0 for x_1 and x_2, we may find new approximations x_1^1 and x_2^1 as follows:

$$\begin{aligned} x_1^1 &= x_1^0 + \Delta x_1^0 \\ x_2^1 &= x_2^0 + \Delta x_2^0 \end{aligned} \tag{3.24}$$

These approximations should be such that they drive the values of the functions closer to zero, so that

$$\begin{aligned} f_1(x_1^1, x_2^1) &\approx 0 \\ f_2(x_1^1, x_2^1) &\approx 0 \end{aligned}$$

or

$$\begin{aligned} f_1(x_1^0 + \Delta x_1^0,\ x_2^0 + \Delta x_2^0) &\approx 0 \\ f_2(x_1^0 + \Delta x_1^0,\ x_2^0 + \Delta x_2^0) &\approx 0 \end{aligned} \tag{3.25}$$

Applying a two-dimensional Taylor series expansion to (3.25) gives

$$\begin{aligned} f_1(x_1^0, x_2^0) + \{\partial f_1/\partial x_1\}^0 \Delta x_1^0 + \{\partial f_1/\partial x_2\}^0 \Delta x_2^0 + \cdots &\approx 0 \\ f_2(x_1^0, x_2^0) + \{\partial f_2/\partial x_1\}^0 \Delta x_1^0 + \{\partial f_2/\partial x_2\}^0 \Delta x_2^0 + \cdots &\approx 0 \end{aligned} \tag{3.26}$$

If we neglect terms involving powers of Δx_1^0 and Δx_2^0 higher than one, then (3.26) represents a system of two linear equations in two unknowns. The zero superscript means that the function is to be calculated at the initial approximation and Δx_1^0 and Δx_2^0 are the unknowns we wish to find. Having solved (3.26) we can obtain our new improved approximations and then repeat the process until we have obtained the accuracy we require. A common convergence criterion is to continue iterations until

$$\sqrt{(\Delta x_1^r)^2 + (\Delta x_2^r)^2} < \varepsilon$$

where r denotes the iteration number and ε is a small positive quantity preset by the user.

It is a simple step to generalize this procedure for any number of variables and equations. We may write the general system of equations as

$$\mathbf{f}(\mathbf{x}) = \mathbf{0}$$

where $\mathbf{f}$ denotes the column vector of n components $(f_1, f_2, \ldots, f_n)^\top$ and $\mathbf{x}$ is a column vector of n components $(x_1, x_2, \ldots, x_n)^\top$. Let $\mathbf{x}^{r+1}$ denote the value of $\mathbf{x}$ at the $(r+1)$th iteration; then

$$\mathbf{x}^{r+1} = \mathbf{x}^r + \Delta\mathbf{x}^r \quad \text{for} \quad r = 0, 1, 2, \ldots$$

If $\mathbf{x}^{r+1}$ is an improved approximation to $\mathbf{x}$, then

$$\mathbf{f}(\mathbf{x}^{r+1}) \approx \mathbf{0}$$

or

$$\mathbf{f}(\mathbf{x}^r + \Delta\mathbf{x}^r) \approx \mathbf{0} \tag{3.27}$$

Expanding (3.27) by using an n-dimensional Taylor series expansion gives

$$\mathbf{f}(\mathbf{x}^r + \Delta\mathbf{x}^r) = \mathbf{f}(\mathbf{x}^r) + \nabla\mathbf{f}(\mathbf{x}^r)\Delta\mathbf{x}^r + \cdots \tag{3.28}$$

where ∇ is a vector operator of partial derivatives with respect to each of the n components of $\mathbf{x}$. If we neglect higher-order terms in $(\Delta\mathbf{x}^r)^2$, this gives, by virtue of (3.27),

$$\mathbf{f}(\mathbf{x}^r) + \mathbf{J}_r\Delta\mathbf{x}^r \approx \mathbf{0} \tag{3.29}$$

where $\mathbf{J}_r = \nabla\mathbf{f}(\mathbf{x}^r)$. $\mathbf{J}_r$ is called the Jacobian matrix. The subscript r denotes that the matrix is evaluated at the point $\mathbf{x}^r$ and it can be written in component form as

$$\mathbf{J}_r = [\partial f_i(\mathbf{x}^r)/\partial x_j] \quad \text{for} \quad i = 1, 2, \ldots, n \quad \text{and} \quad j = 1, 2, \ldots, n$$

On solving (3.29) we have the improved approximation

$$\mathbf{x}^{r+1} = \mathbf{x}^r - \mathbf{J}_r^{-1}\mathbf{f}(\mathbf{x}^r) \quad \text{for} \quad r = 1, 2, \ldots$$

The matrix $\mathbf{J}_r$ may be singular and in this situation the inverse, $\mathbf{J}_r^{-1}$, cannot be calculated.

This is the general form of Newton's method. However, there are two major disadvantages with this method:

1. The method may not converge unless the initial approximation is a good one.
2. The method requires the user to provide the derivatives of each function with respect to each variable. The user must therefore provide n^2 derivatives and any computer implementation must evaluate the n functions and the n^2 derivatives at each iteration.

The MATLAB function `newtonmv` given here implements this method.

```
function [xv,it] = newtonmv(x,f,jf,n,tol)
% Newton's method for solving a system of n nonlinear equations
% in n variables.
% Example call: [xv,it] = newtonmv(x,f,jf,n,tol)
% Requires an initial approximation column vector x. tol is
% required accuracy. User must define functions f (system equations)
% and jf (partial derivatives). xv is the solution vector, the it
% parameter is number of iterations taken.
% WARNING. The method may fail, for example if initial estimates are poor.
it = 0; xv = x;
fr = feval(f,xv);
while norm(fr) > tol
    Jr = feval(jf,xv);  xv = xv-Jr\fr;
    fr = feval(f,xv);  it = it+1;
end
```

Figure 3.11 illustrates the following system of two equations in two variables:

$$\begin{aligned} x^2 + y^2 &= 4 \\ xy &= 1 \end{aligned} \tag{3.30}$$

To solve the system (3.30) we define the MATLAB function by `f` and its Jacobian by `Jf` and then call `newtonmv` using initial approximations for the roots $x = 3$ and $y = -1.5$ and a tolerance of 0.00005 as follows:

```
>> f = @(v) [v(1)^2+v(2)^2-4; v(1)*v(2)-1];
>> Jf = @(v) [2*v(1) 2*v(2); v(2) v(1)];
>> [rootvals,iter] = newtonmv([3 -1.5]',f,Jf,2,0.00005)
```

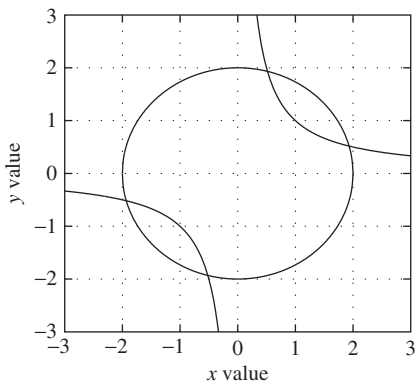

FIGURE 3.11 Plot of system (3.30). Intersections show roots.

This results in the MATLAB output

```
rootvals =
    1.9319
    0.5176

iter =
     5
```

The solution is $x = 1.9319$ and $y = 0.5176$. Clearly the user must supply a large amount of information for this function. The next section attempts to deal with this problem.

3.13 Broyden's Method for Solving Nonlinear Equations

The method of Newton described in Section 3.12 does not provide a practical procedure for solving any but the smallest systems of nonlinear equations. As we have seen, the method requires the user to provide not only the function definitions but also the definitions of the n^2 partial derivatives of the functions. Thus, for a system of 10 equations in 10 unknowns, the user must provide 110 function definitions!

To deal with this problem a number of techniques have been proposed but the group of methods that appears most successful is the class known as the quasi-Newton methods. The quasi-Newton methods avoid the calculation of the partial derivatives by obtaining approximations to them involving only the function values. The set of derivatives of the functions evaluated at any point $\mathbf{x}^r$ may be written in the form of the Jacobian matrix

$$\mathbf{J}_r = [\partial f_i(\mathbf{x}^r)/\partial x_j] \text{ for } i = 1,2,\ldots,n \quad \text{and} \quad j = 1,2,\ldots,n \tag{3.31}$$

The quasi-Newton methods provide an updating formula, which gives successive approximations to the Jacobian for each iteration. Broyden and others have shown that under specified circumstances these updating formulae provide satisfactory approximations to the inverse Jacobian. The structure of the algorithm suggested by Broyden is

1. Input an initial approximation to the solution. Set the counter r to zero.
2. Calculate or assume an initial approximation to the inverse Jacobian $\mathbf{B}^r$.
3. Calculate $\mathbf{p}^r = -\mathbf{B}^r\mathbf{f}^r$ where $\mathbf{f}^r = \mathbf{f}(\mathbf{x}^r)$.
4. Determine the scalar parameter t_r such that $||\mathbf{f}(\mathbf{x}^r + t_r\mathbf{p}^r)|| < ||\mathbf{f}^r||$ where the symbols $||\quad||$ denote that the norm of the vector is to be taken.
5. Calculate $\mathbf{x}^{r+1} = \mathbf{x}^r + t_r\mathbf{p}^r$.

6. Calculate $\mathbf{f}^{r+1} = \mathbf{f}(\mathbf{x}^{r+1})$. If $||\mathbf{f}^{r+1}|| < \varepsilon$ (where ε is a small preset positive quantity), then exit. If not continue with step 7.

7. Use the updating formula to obtain the required approximation to the Jacobian

$$\mathbf{B}^{r+1} = \mathbf{B}^r - (\mathbf{B}^r\mathbf{y}^r - \mathbf{p}^r)(\mathbf{p}^r)^\top\mathbf{B}^r/\{(\mathbf{p}^r)^\top\mathbf{B}^r\mathbf{y}^r\} \text{ where } \mathbf{y}^r = \mathbf{f}^{r+1} - \mathbf{f}^r.$$

8. Set $i = i + 1$ and return to step 3.

The initial approximation to the inverse Jacobian **B** is usually taken as a scalar multiple of the unit matrix. The success of this algorithm depends on the nature of the functions to be solved and on the closeness of the initial approximation to the solution. In particular, step 4 may present major problems. It may be very expensive in computer time and to avoid this t_r is sometimes set as a constant, usually 1 or smaller. This may reduce the stability of the algorithm but speeds it up.

It should be noted that other updating formulae have been suggested and it is fairly easy to replace the Broyden formula by others in the preceding algorithm. In general, the problem of solving a system of nonlinear equations is a very difficult one. There is no algorithm that is guaranteed to work for all systems of equations. For large systems of equations the available algorithms tend to require large amounts of computer time to obtain accurate solutions.

The MATLAB function `broyden` implements Broyden's method. It should be noted that this avoids the difficulty of implementing step 4 by taking $t_r = 1$.

```
function [xv,it] = broyden(x,f,n,tol)
% Broyden's method for solving a system of n nonlinear equations
% in n variables.
% Example call: [xv,it] = broyden(x,f,n,tol)
% Requires an initial approximation column vector x. tol is required
% accuracy. User must define function f.
% xv is the solution vector, parameter it is number of iterations
% taken. WARNING. Method may fail, for example, if initial estimates
% are poor.
fr = zeros(n,1); it = 0; xv = x;
Br = eye(n); %Set initial Br
fr = feval(f, xv);
while norm(fr)>tol
    it = it+1; pr = -Br*fr; tau = 1;
    xv = xv+tau*pr;
    oldfr = fr; fr = feval(f,xv);
    % Update approximation to Jacobian using Broyden's formula
    y = fr - oldfr; oldBr = Br;
    oyp = oldBr*y-pr; pB = pr'*oldBr;
    for i = 1:n
        for j = 1:n
            M(i,j) = oyp(i)*pB(j);
```

```
        end
    end
    Br = oldBr-M./(pr'*oldBr*y);
end
```

To solve the system (3.30) using Broyden's method we call `broyden` as follows:

```
>> f = @(v) [v(1)^2+v(2)^2-4; v(1)*v(2)-1];
>> [x, iter] = broyden([3 -1.5]',f,2,0.00005)
```

This results in

```
x =
    0.5176
    1.9319

iter =
    36
```

This is a correct root of system (3.30) but it is not the same root as that found by Newton's method, even though the starting values for the iteration are the same.

As a second example we consider the following system of equations, which are taken from the *MATLAB User's Guide* (1989):

$$\begin{aligned} \sin x + y^2 + \log_e z &= 7 \\ 3x + 2y - z^3 &= -1 \\ x + y + z &= 5 \end{aligned} \tag{3.32}$$

The function g, which implements (3.32), is given here

```
>> g = @(p) [sin(p(1))+p(2)^2+log(p(3))-7; 3*p(1)+2^p(2)-p(3)^3+1;
             p(1)+p(2)+p(3)-5];
```

The result of solving (3.32) is given next. The starting values used are $x = 0$, $y = 2$, and $z = 2$.

```
>> x = broyden([0 2 2]',g,3,0.00005)

x =
    0.5991
    2.3959
    2.0050
```

This shows that the method is successful for two problems and does not require the evaluation of the partial derivatives. The reader may be interested in applying the function `newtonmv` to this problem. Nine first-order partial derivatives will be required.

3.14 Comparing the Newton and Broyden Methods

We end our discussion of the solution of nonlinear systems of equations by comparing the performance of the functions `broyden` and `newtonmv`, developed in Sections 3.12 and 3.13, when solving the system (3.30). The following script calls both functions and provides the number of iterations required for convergence.

```
>> f = @(v) [v(1)^2+v(2)^2-4; v(1)*v(2)-1];
>> [x,it] = broyden([3 -1.5]',f,2,0.00005)

x =
    0.5176
    1.9319

it =
    36

>> J = @(v) [2*v(1) 2*v(2);v(2) v(1)];
>> [x,it] = newtonmv([3,-1.5]',f,J,2,0.00005)

x =
    1.9319
    0.5176

it =
     5
```

Note that although a correct solution is found in each case, it is a different root.

The first-order partial derivatives are required for the Newton method and this requires a considerable effort on the part of the user. Solving the previous problem demonstrates that the relatively simple form of the function `broyden` is attractive since it relieves the user of this effort.

In Sections 3.12 and 3.13 two relatively simple algorithms were provided for the solution of a very difficult problem. They cannot always be guaranteed to work and for large problems will converge only slowly.

3.15 Summary

The user wishing to solve nonlinear equations will find that this is an area that can present particular difficulties. It is always possible to devise or meet problems that particular algorithms either cannot solve or take a long time to solve. For example, it is just not possible for many algorithms to find the roots of the apparently trivial problem $x^{20} = 0$ very accurately. However, the algorithms described, if used with care, provide ways of solving a wide

range of problems. MATLAB is well suited for this study because it allows interactive experimentation and graphical insights into the behavior of methods and functions. The reader is referred to Section 9.6 for applications of the symbolic toolbox for solving nonlinear equations. The algorithms `solve`, `fnewtsym`, and `newtmvsym` are described and applied in that section.

Problems

3.1. Omar Khayyam (who lived in the twelfth century) solved, by geometric means, a cubic equation with the form

$$x^3 - cx^2 + b^2x + a^3 = 0$$

The positive roots of this equation are the x coordinates of points of intersection in the first quadrant of the circle and parabola given in the following:

$$x^2 + y^2 - (c - a^3/b^2)x + 2by + b^2 - ca^3/b^2 = 0$$
$$xy = a^3/b$$

For $a = 1$, $b = 2$, and $c = 3$ use MATLAB to plot these two functions and note the x coordinates of the points of intersection. Using the MATLAB function `fzero`, solve the cubic equation and hence verify Omar Khayyam's method. *Hint*: You may find it helpful to use the MATLAB function `ginput`.

3.2. Use the MATLAB function `fnewton` to find a root of

$$x^{1.4} - \sqrt{x} + 1/x - 100 = 0$$

given an initial approximation 50. Use an accuracy of 10^{-4}.

3.3. Find the two real roots of $|x^3| + x - 6 = 0$ using the MATLAB function `fnewton`. Use initial approximations -1 and 1 and an accuracy of 10^{-4}. Plot the function using MATLAB to verify that the equation has only two real roots. *Hint:* Take care in finding the derivative of the function.

3.4. Explain why it is relatively difficult to find the root of $\tan x - c = 0$ when c is large. Use the MATLAB function `fnewton`, with initial approximations 1.3 and 1.4 and accuracy 10^{-4}, to find a root of this equation when $c = 5$ and $c = 10$. Compare the number of iterations required in both cases. *Hint*: A MATLAB `plot` will be useful.

3.5. Find a root of the polynomial $x^5 - 5x^4 + 10x^3 - 10x^2 + 5x - 1 = 0$ correct to four decimal places by using the MATLAB function `schroder` with $n = 5$ and a starting value $x_0 = 2$. Use MATLAB function `fnewton` to solve the same problem. Compare the result and the number of iterations using both methods. Use an accuracy of 5×10^{-7}.

3.6. Use the simple iterative method to solve the equation $x^{10} = e^x$. Express the equation in the form $x = f(x)$ in different ways and start the iterations with the initial approximation $x = 1$. Compare the efficiency of the formulae you have devised and check your answer(s) using the MATLAB function `fnewton`.

3.7. The historic Kepler's equation has the form $E - e\sin E = M$. Solve this equation for $e = 0.96727464$, the eccentricity of Halley's comet, and $M = 4.527594 \times 10^{-3}$. Use the MATLAB function `fnewton`, with an accuracy of 0.00005 and a starting value of 1.

3.8. Examine the performance of the function `fzero` for solving $x^{11} = 0$ with an initial value of -1.5 and also 1. Use an accuracy of 1×10^{-5}.

3.9. The smallest positive root of the equation

$$1 - x + x^2/(2!)^2 - x^3/(3!)^2 + x^4/(4!)^2 - \cdots = 0$$

is 1.4458. By considering in turn only the first four, five, and six terms in the series, show that a root of the truncated series approaches this result. Use the MATLAB function `fzero` to derive these results, with an initial value of 1 and an accuracy of 10^{-4}.

3.10. Reduce the following system of equations to one equation in terms of x and solve the resulting equation using the MATLAB function `fnewton`.

$$e^{x/10} - y = 0$$
$$2\log_e y - \cos x = 2$$

Use the MATLAB function `newtonmv` to solve these equations directly and compare your results. Use an initial approximation $x = 1$ for `fnewton` and approximations $x = 1$, $y = 1$ for `newtonmv` and accuracy 10^{-4} in both cases.

3.11. Solve the pair of equations that follow using the MATLAB function `broyden`, with the starting point $x = 10$, $y = -10$ and accuracy 10^{-4}.

$$2x = \sin\{(x+y)/2\}$$
$$2y = \cos\{(x-y)/2\}$$

3.12. Solve the two equations that follow using the MATLAB functions `newtonmv` and `broyden` with the starting point $x = 1, y = 2$ and accuracy 10^{-4}.

$$x^3 - 3xy^2 = 1/2$$
$$3x^2y - y^3 = \sqrt{3}/2$$

3.13. The polynomial equation

$$x^4 - (13+\varepsilon)x^3 + (57+8\varepsilon)x^2 - (95+17\varepsilon)x + 50 + 10\varepsilon = 0$$

has roots 1, 2, 5, $5+\varepsilon$. Use the functions `bairstow` and `roots` to find all the roots of this polynomial for $\varepsilon = 0.1, 0.01$, and 0.001. What happens as ε becomes smaller? Use an accuracy of 10^{-5}.

3.14. Employ the MATLAB function `bairstow` to find all the roots of the following polynomial using an accuracy requirement of 10^{-4}.

$$x^5 - x^4 - x^3 + x^2 - 2x + 2 = 0$$

3.15. Use the MATLAB function `roots` to find all the roots of the equation

$$t^3 - 0.5 - \sqrt{(3/2)}\imath = 0 \quad \text{where} \quad \imath = \sqrt{-1}$$

Compare with the exact solution

$$\cos\{(\pi/3 + 2\pi k)/3\} + \imath \sin\{(\pi/3 + 2\pi k)/3\} \quad \text{for} \quad k = 0, 1, 2$$

Use an accuracy of 10^{-4}.

3.16. An outline algorithm for the Illinois method for finding a root of $f(x) = 0$ (Dowell and Jarrett, 1971) is as follows:

For $k = 0, 1, 2, \ldots$
$x_{k+1} = x_k - f_k / f[x_{k-1}, x_k]$
if $f_k f_{k+1} > 0$ set $x_k = x_{k-1}$ and $f_k = g f_{k-1}$
where $f_k = f(x_k)$, $f[x_{k-1}, x_k] = (f_k - f_{k-1})/(x_k - x_{k-1})$
and $g = 0.5$.

Write a MATLAB function to implement this method. Note that the *regula falsi* method is similar but differs in that g is taken as 1.

3.17. The following iterative formulae can be used to solve the equation $x^2 - a = 0$:

$$x_{k+1} = (x_{k+1} + a/x_k)/2, \quad k = 0, 1, 2, \ldots$$

and

$$x_{k+1} = (x_{k+1} + a/x_k)/2 - (x_k - a/x_k)^2/(8x_k), \quad k = 0, 1, 2, \ldots$$

These iterative formulae are second- and third-order methods, respectively, for solving this equation. Write a MATLAB script to implement them and compare the number of iterations required to obtain the square root of 100.112 to five decimal places. For the purpose of illustration, use an initial approximation of 1000.

3.18. Show how MATLAB can be used to study chaotic behavior by considering the iteration

$$x_{k+1} = g(x_k) \quad \text{for} \quad k = 0, 1, 2, \ldots$$

where

$$g(x) = cx(1-x)$$

for different values of the constant c. This simple iteration arises from an attempt to solve a simple quadratic equation. However, its behavior is complex and for some values of c is chaotic. Write a MATLAB script to plot the value of the iterates against the iterate number for this function and study the behavior of the iterations for $c = 2.8, 3.25, 3.5$, and 3.8. Use an initial value of $x_0 = 0.7$.

3.19. For the functions solved in Problems 3.2, 3.3, and 3.7, use the MATLAB function `plotapp`, given in Section 3.9, to find approximate solutions for these functions.

3.20. It can be shown that the cubic polynomial equation

$$x^3 - px - q = 0$$

will have real roots if the inequality $p^3/q^2 > 27/4$ is satisfied. Select five pairs of values for p and q for which this inequality is satisfied and hence, using the MATLAB function `roots`, verify in each case that the roots of the equation are real.

3.21. In the sixteen century the mathematician Ioannes Colla suggested the following problem: Divide 10 into three parts such that they shall be in continued proportion to each other and the product of the first two shall be 6. Taking x, y, and z as three parts, this problem can be stated as

$$x + y + z = 10,\ x/y = y/z,\ xy = 6$$

Now by simple manipulation these equations can be expressed in terms of the specific variable y as

$$y^4 + 6y^2 - 60y + 36 = 0$$

Clearly if we can solve this equation for y then we can easily find the other variables x and z from the original equations. Use the MATLAB function `roots` to find values for y and hence solve Colla's problem.

3.22. The natural frequencies of a simply supported beam are given by the roots of the equation

$$c_1^2 - x^4 c_3^2 = 0$$

where

$$c_1 = (\sinh(x) + \sin(x))/(2x)$$

and

$$c_3 = (\sinh(x) - \sin(x))/(2x^3)$$

Substituting for c_1 and c_3 gives

$$((\sinh(x)+\sin(x))/(2x))^2 - x^4((\sinh(x)-\sin(x))/(2x^3))^2 = 0$$

When searching for the roots of this equation no difficulty is found in determining the root for trial values of x providing x is small (say $x < 10$). For values of $x > 25$ the process becomes erratic. The roots of this equation are actually $x = k\pi$ where k is a positive integer. Use the MATLAB function `fzero` with initial approximations $x = 5$ and $x = 30$ to obtain a solution close to these initial approximations for this equation. For the purpose of this exercise, do not simplify this equation.

Why are the results so poor? If you simplify the preceding equation, which equation do you obtain and what is its solution?