CHAPTER 1

Numerical Solution of Nonlinear Equations

1.1 INTRODUCTION

Many problems in engineering and science require the solution of nonlinear equations. Several examples of such problems drawn from the field of chemical engineering and from other application areas are discussed in this section. The methods of solution are developed in the remaining sections of the chapter, and specific examples of the solutions are demonstrated using the MATLAB software.

In thermodynamics, the pressure-volume-temperature relationship of real gases is described by the equation of state. There are several semitheoretical or empirical equations, such as Redlich-Kwong, Soave-Redlich-Kwong, and the Benedict-Webb-Rubin equations,

which have been used extensively in chemical engineering. For example, the Soave-Redlich-Kwong equation of state has the form

$$P = \frac{RT}{V - b} - \frac{a\alpha}{V(V + b)} \tag{1.1}$$

where P, V, and T are the pressure, specific volume, and temperature, respectively. R is the gas constant, α is a function of temperature, and a and b are constants, specific for each gas. Eq. (1.1) is a third-degree polynomial in V and can be easily rearranged into the canonical form for a polynomial, which is

$$Z^3 - Z^2 + (A - B - B^2)Z - AB = 0 \tag{1.2}$$

where $Z = PV/RT$ is the compressibility factor, $A = \alpha aP/R^2T^2$ and $B = bP/RT$. Therefore, the problem of finding the specific volume of a gas at a given temperature and pressure reduces to the problem of finding the appropriate root of a polynomial equation.

In the calculations for multicomponent separations, it is often necessary to estimate the minimum reflux ratio of a multistage distillation column. A method developed for this purpose by Underwood [1], and described in detail by Treybal [2], requires the solution of the equation

$$\sum_{j=1}^{n} \frac{\alpha_j z_{jF} F}{\alpha_j - \phi} - F(1 - q) = 0 \tag{1.3}$$

where F is the molar feed flow rate, n is the number of components in the feed, z_{jF} is the mole fraction of each component in the feed, q is the quality of the feed, α_j is the relative volatility of each component at average column conditions, and ϕ is the root of the equation. The feed flow rate, composition, and quality are usually known, and the average column conditions can be approximated. Therefore, ϕ is the only unknown in Eq. (1.3). Because this equation is a polynomial in ϕ of degree n, there are n possible values of ϕ (roots) that satisfy the equation.

The friction factor f for turbulent flow of an incompressible fluid in a pipe is given by the nonlinear Colebrook equation

$$\sqrt{\frac{1}{f}} = -0.86 \ln\left(\frac{\epsilon/D}{3.7} + \frac{2.51}{N_{Re}\sqrt{f}}\right) \tag{1.4}$$

where ϵ and D are roughness and inside diameter of the pipe, respectively, and N_{Re} is the

Reynolds number. This equation does not readily rearrange itself into a polynomial form; however, it can be arranged so that all the nonzero terms are on the left side of the equation as follows:

$$\sqrt{\frac{1}{f}} + 0.86 \ln\left(\frac{\epsilon/D}{3.7} + \frac{2.51}{N_{Re}\sqrt{f}}\right) = 0 \tag{1.5}$$

The method of differential operators is applied in finding analytical solutions of *n*th-order linear homogeneous differential equations. The general form of an *n*th-order linear homogeneous differential equation is

$$a_n \frac{d^n y}{dx^n} + a_{n-1} \frac{d^{n-1} y}{dx^{n-1}} + \ldots + a_1 \frac{dy}{dx} + a_0 y = 0 \tag{1.6}$$

By defining D as the differentiation with respect to x:

$$D = \frac{d}{dx} \tag{1.7}$$

Eq. (1.6) can be written as

$$[\, a_n D^n + a_{n-1} D^{n-1} + \ldots + a_1 D + a_0 \,] y = 0 \tag{1.8}$$

where the bracketed term is called the differential operator. In order for Eq. (1.8) to have a nontrivial solution, the differential operator must be equal to zero:

$$a_n D^n + a_{n-1} D^{n-1} + \ldots + a_1 D + a_0 = 0 \tag{1.9}$$

This, of course, is a polynomial equation in D whose roots must be evaluated in order to construct the complementary solution of the differential equation.

The field of process dynamics and control often requires the location of the roots of transfer functions that usually have the form of polynomial equations. In kinetics and reactor design, the simultaneous solution of rate equations and energy balances results in mathematical models of simultaneous nonlinear and transcendental equations. Methods of solution for these and other such problems are developed in this chapter.

1.2 TYPES OF ROOTS AND THEIR APPROXIMATION

All the nonlinear equations presented in Sec. 1.1 can be written in the general form

$$f(x) = 0 \tag{1.10}$$

where x is a single variable that can have multiple values (roots) that satisfy this equation. The function $f(x)$ may assume a variety of nonlinear functionalities ranging from that of a polynomial equation whose canonical form is

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0 = 0 \tag{1.11}$$

to the transcendental equations, which involve trigonometric, exponential, and logarithmic terms. The roots of these functions could be

1. Real and distinct
2. Real and repeated
3. Complex conjugates
4. A combination of any or all of the above.

The real parts of the roots may be positive, negative, or zero.

Fig. 1.1 graphically demonstrates all the above cases using fourth-degree polynomials. Fig. 1.1*a* is a plot of the polynomial equation (1.12):

$$x^4 + 6x^3 + 7x^2 - 6x - 8 = 0 \tag{1.12}$$

which has four real and distinct roots at -4, -2, -1, and 1, as indicated by the intersections of the function with the x axis. Fig. 1.1*b* is a graph of the polynomial equation (1.13):

$$x^4 + 7x^3 + 12x^2 - 4x - 16 = 0 \tag{1.13}$$

which has two real and distinct roots at -4 and 1 and two real and repeated roots at -2. The point of tangency with the x axis indicates the presence of the repeated roots. At this point $f(x) = 0$ and $f'(x) = 0$. Fig. 1.1*c* is a plot of the polynomial equation (1.14):

$$x^4 - 6x^3 + 18x^2 - 30x + 25 = 0 \tag{1.14}$$

which has only complex roots at $1 \pm 2i$ and $2 \pm i$. In this case, no intersection with the x axis of the Cartesian coordinate system occurs, as all of the roots are located in the complex plane. Finally, Fig. 1.1*d* demonstrates the presence of two real and two complex roots with

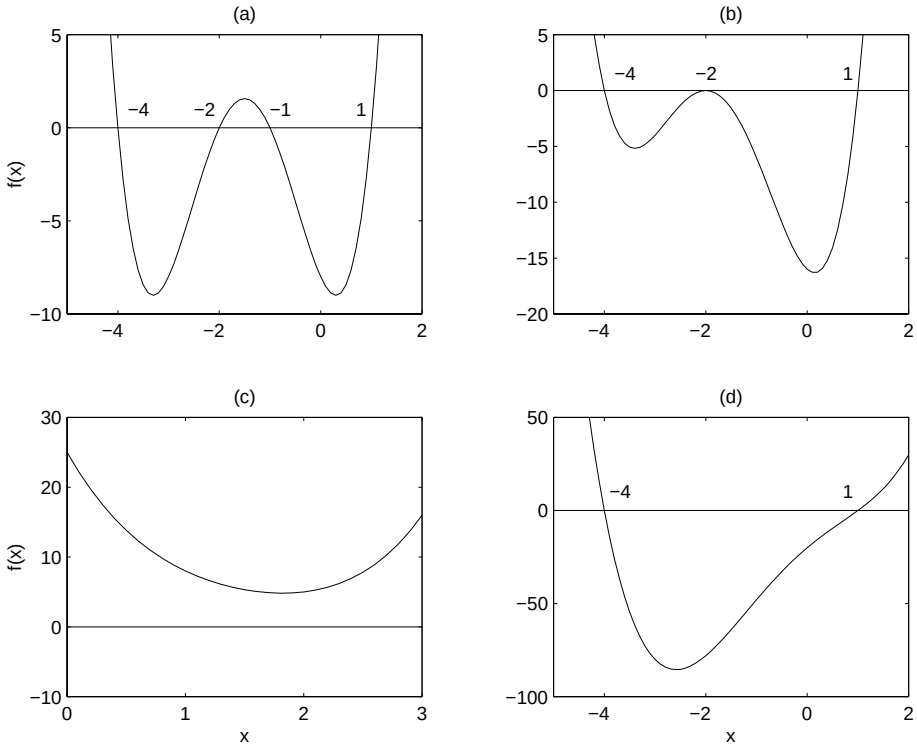

Figure 1.1 Roots of fourth-degree polynomial equations. (*a*) Four real distinct. (*b*) Two real and two repeated. (*c*) Four complex. (*d*) Two real and two complex.

the polynomial equation (1.15):

$$x^4 + x^3 - 5x^2 + 23x - 20 = 0 \tag{1.15}$$

whose roots are -4, 1, and $1 \pm 2i$. As expected, the function crosses the x axis only at two points: -4 and 1.

The roots of an nth-degree polynomial, such as Eq. (1.11), may be verified using Newton's relations, which are:

Newton's 1st relation:

$$\sum_{i=1}^{n} x_i = -\frac{a_{n-1}}{a_n} \tag{1.16}$$

where x_i are the roots of the polynomial.

Newton's 2nd relation:

$$\sum_{i,j=1}^{n} x_i x_j = \frac{a_{n-2}}{a_n} \tag{1.17}$$

Newton's 3rd relation:

$$\sum_{i,j,k=1}^{n} x_i x_j x_k = -\frac{a_{n-3}}{a_n} \tag{1.18}$$

Newton's *n*th relation:

$$x_1 x_2 x_3 \ldots x_n = (-1)^n \frac{a_0}{a_n} \tag{1.19}$$

where $i \neq j \neq k \neq \ldots$ for all the above equations which contain products of roots.

In certain problems it may be necessary to locate all the roots of the equation, including the complex roots. This is the case in finding the zeros and poles of transfer functions in process control applications and in formulating the analytical solution of linear *n*th-order differential equations. On the other hand, different problems may require the location of only one of the roots. For example, in the solution of the equation of state, the positive real root is the one of interest. In any case, the physical constraints of the problem may dictate the feasible region of search where only a subset of the total number of roots may be indicated. In addition, the physical characteristics of the problem may provide an approximate value of the desired root.

The most effective way of finding the roots of nonlinear equations is to devise iterative algorithms that start at an initial estimate of a root and converge to the exact value of the desired root in a finite number of steps. Once a root is located, it may be removed by synthetic division if the equation is of the polynomial form. Otherwise, convergence on the same root may be avoided by initiating the search for subsequent roots in different region of the feasible space.

For equations of the polynomial form, Descartes' rule of sign may be used to determine the number of positive and negative roots. This rule states: The number of positive roots is equal to the number of sign changes in the coefficients of the equation (or less than that by an even integer); the number of negative roots is equal to the number of sign repetitions in the coefficients (or less than that by an even integer). Zero coefficients are counted as positive [3]. The purpose of the qualifier, "less than that by an even integer," is to allow for the existence of conjugate pairs of complex roots. The reader is encouraged to apply Descartes' rule to Eqs. (1.12)-(1.15) to verify the results already shown.

If the problem to be solved is a purely mathematical one, that is, the model whose roots are being sought has no physical origin, then brute-force methods would have to be used to establish approximate starting values of the roots for the iterative technique. Two categories of such methods will be mentioned here. The first one is a truncation method applicable to

equation of the polynomial form. For example, the following polynomial

$$a_4 x^4 + a_3 x^3 + a_2 x^2 + a_1 x + a_0 = 0 \tag{1.20}$$

may have its lower powered terms truncated

$$a_4 x^4 + a_3 x^3 \simeq 0 \tag{1.21}$$

to yield an approximation of one of the roots

$$x \simeq -\frac{a_3}{a_4} \tag{1.22}$$

Alternatively, if the higher powered terms are truncated

$$a_1 x + a_0 \simeq 0 \tag{1.23}$$

the approximate root is

$$x \simeq -\frac{a_0}{a_1} \tag{1.24}$$

This technique applied to Soave-Redlich-Kwong equation [Eq. (1.2)] results in

$$Z = \frac{PV}{RT} \simeq 1 \tag{1.25}$$

This, of course, is the well-known *ideal gas law*, which is an excellent approximation of the pressure-volume-temperature relationship of real gases at low pressures. On the other end of the polynomial, truncation of the higher powered terms results in

$$Z \simeq \frac{AB}{A - B - B^2} \tag{1.26}$$

giving a value of Z very close to zero which is the case for liquids. In this case, the physical considerations of the problem determine that Eq. (1.25) or Eq. (1.26) should be used for gas phase or liquid phase, respectively, to initiate the iterative search technique for the real root.

Another method of locating initial estimates of the roots is to scan the entire region of search by small increments and to observe the steps in which a change of sign in the function $f(x)$ occurs. This signals that the function $f(x)$ crosses the x axis within the particular step. This search can be done easily in MATLAB environment using *fplot* function. Once the

function $f(x)$ is introduced in a MATLAB function *file_name.m*, the statement *fplot*('*file_name*',[*a, b*]) shows the plot of the function from $x = a$ to $x = b$. The values of *a* and *b* may be changed until the plot crosses the *x* axis.

The scan method may be a rather time-consuming procedure for polynomials whose roots lie in a large region of search. A variation of this search is the method of bisection that divides the interval of search by 2 and always retains that half of the search interval in which the change of sign has occurred. When the range of search has been narrowed down sufficiently, a more accurate search technique would then be applied within that step in order to refine the value of the root.

More efficient methods based on rearrangement of the function to $x = g(x)$ (*method of successive substitution*), linear interpolation of the function (*method of false position*), and the tangential descent of the function (*Newton-Raphson method*) will be described in the next three sections of this chapter.

MATLAB has its own built-in function *fzero* for root finding. The statement *fzero*('*file_name*',x_0) finds the root of the function $f(x)$ introduced in the user-defined MATLAB function *file_name.m*. The second argument x_0 is a starting guess. Starting with this initial value, the function *fzero* searches for change in the sign of the function $f(x)$. The calculation then continues with either bisection or linear interpolation method until the convergence is achieved.

1.3 THE METHOD OF SUCCESSIVE SUBSTITUTION

The simplest one-point iterative root-finding technique can be developed by rearranging the function $f(x)$ so that x is on the left-hand side of the equation

$$x = g(x) \tag{1.27}$$

The function $g(x)$ is a formula to predict the root. In fact, the root is the intersection of the line $y = x$ with the curve $y = g(x)$. Starting with an initial value of x_1, as shown in Fig. 1.2*a*, we obtain the value of x_2:

$$x_2 = g(x_1) \tag{1.28}$$

which is closer to the root than x_1 and may be used as an initial value for the next iteration. Therefore, general iterative formula for this method is

$$x_{n+1} = g(x_n) \tag{1.29}$$

which is known as the method of successive substitution or the method of $x = g(x)$.

A sufficient condition for convergence of Eq. (1.29) to the root x^* is that $|g'(x)| < 1$ for all x in the search interval. Fig. 1.2*b* shows the case when this condition is not valid and the method diverges. This analytical test is often difficult in practice. In a computer program it is easier to determine whether $|x_3 - x_2| < |x_2 - x_1|$ and, therefore, the successive x_n values converge. The advantage of this method is that it can be started with only a single point, without the need for calculating the derivative of the function.

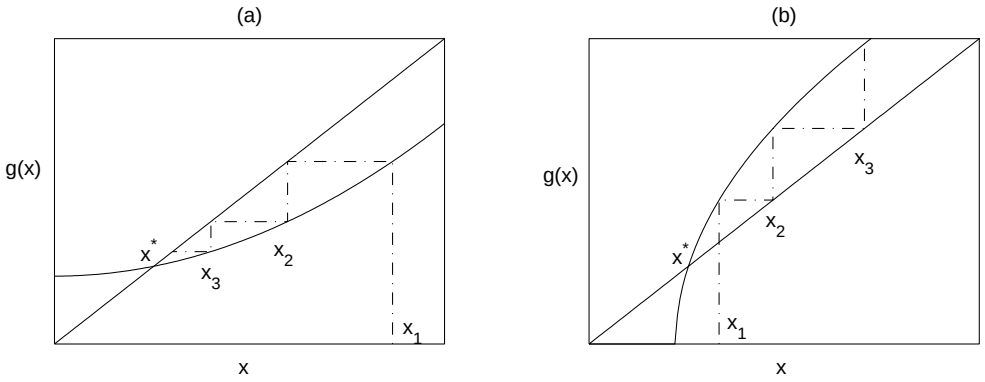

Figure 1.2 Use of $x = g(x)$ method. (*a*) Convergence. (*b*) Divergence.

1.4 THE WEGSTEIN METHOD

The Wegstein method may also be used for the solution of the equations of the form

$$x = g(x) \tag{1.27}$$

Starting with an initial value of x_1, we first obtain another estimation of the root from

$$x_2 = g(x_1) \tag{1.28}$$

As shown in Fig. 1.3, x_2 does not have to be closer to the root than x_1. At this stage, we estimate the function $g(x)$ with a line passing from the points $(x_1, g(x_1))$ and $(x_2, g(x_2))$

$$\frac{y - g(x_1)}{x - x_1} = \frac{g(x_2) - g(x_1)}{x_2 - x_1} \tag{1.30}$$

and find the next estimation of the root, x_3, from the intersection of the line (1.30) and the line $y = x$:

$$x_3 = \frac{x_1 g(x_2) - x_2 g(x_1)}{x_1 - g(x_1) - x_2 + g(x_2)} \tag{1.31}$$

It can be seen from Fig. 1.3*a* that x_3 is closer to the root than either x_1 and x_2. In the next iteration we pass the line from the points $(x_2, g(x_2))$ and $(x_3, g(x_3))$ and again evaluate the next estimation of the root from the intersection of this line with $y = x$. Therefore, the general iterative formula for the Wegstein method is

$$x_{n+1} = \frac{x_{n-1} g(x_n) - x_n g(x_{n-1})}{x_{n-1} - g(x_{n-1}) - x_n + g(x_n)} \qquad n \geq 2 \tag{1.32}$$

The Wegstein method converges, even under conditions in which the method of $x = g(x)$ does not. Moreover, it accelerates the convergence when the successive substitution method is stable Fig. 1.3*b*.

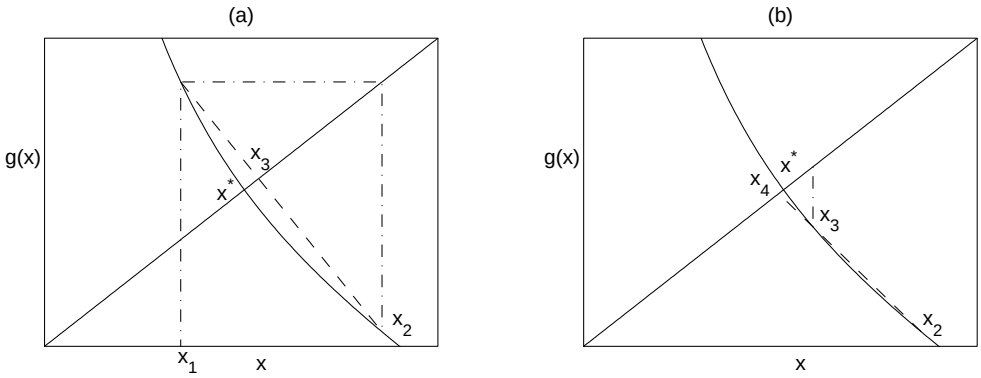

Figure 1.3 The Wegstein method.

1.5 THE METHOD OF LINEAR INTERPOLATION (METHOD OF FALSE POSITION)

This technique is based on linear interpolation between two points on the function that have been found by a scan to lie on either side of a root. For example, x_1 and x_2 in Fig. 1.4*a* are

positions on opposite sides of the root x^* of the nonlinear function $f(x)$. The points $(x_1, f(x_1))$ and $(x_2, f(x_2))$ are connected by a straight line, which we will call a chord, whose equation is

$$y(x) = ax + b \tag{1.33}$$

Because this chord passes through the two points $(x_1, f(x_1))$ and $(x_2, f(x_2))$, its slope is

$$a = \frac{f(x_2) - f(x_1)}{x_2 - x_1} \tag{1.34}$$

and its y intercept is

$$b = f(x_1) - ax_1 \tag{1.35}$$

Eq. (1.33) then becomes

$$y(x) = \left[\frac{f(x_2) - f(x_1)}{x_2 - x_1}\right]x + \left\{f(x_1) - \left[\frac{f(x_2) - f(x_1)}{x_2 - x_1}\right]x_1\right\} \tag{1.36}$$

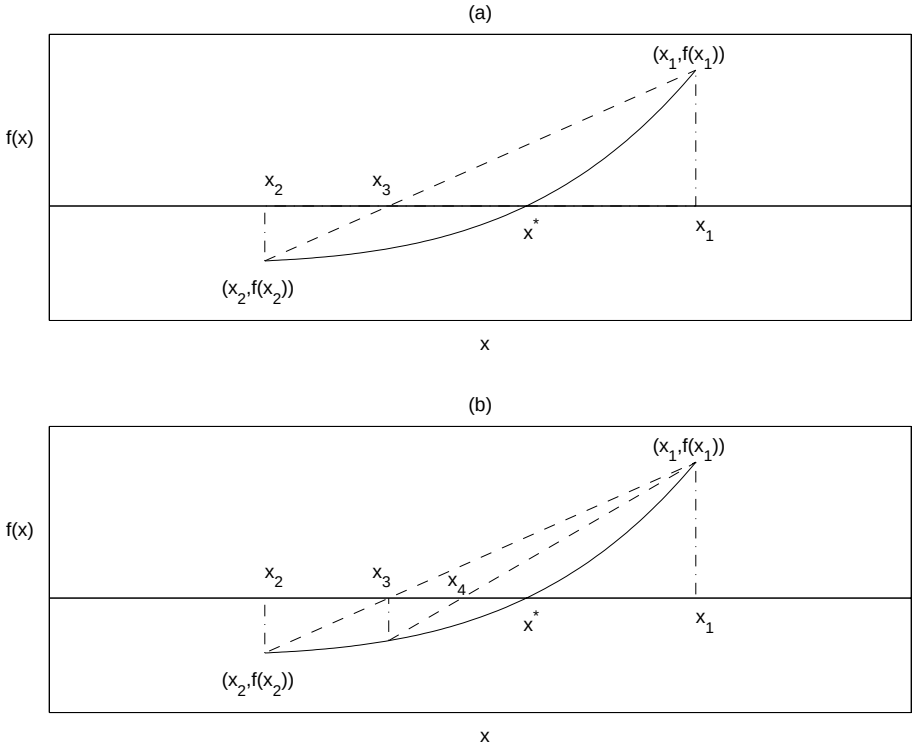

Figure 1.4 Method of linear interpolation.

Locating x_3 using Eq. (1.36), where $y(x_3) = 0$:

$$x_3 = x_1 - \frac{f(x_1)(x_2 - x_1)}{f(x_2) - f(x_1)} \tag{1.37}$$

Note that for the shape of curve chosen on Fig. 1.4, x_3 is nearer to the root x^* than either x_1 or x_2. This, of course, will not always be the case with all functions. Discussion of criteria for convergence will be given in the next section.

According to Fig. 1.4, $f(x_3)$ has the same sign as $f(x_2)$; therefore, x_2 may be replaced by x_3. Now repeating the above operation and connecting the points $(x_1, f(x_1))$ and $(x_3, f(x_3))$ with a new chord, as shown in Fig. 1.4*b*, we obtain the value of x_4:

$$x_4 = x_1 - \frac{f(x_1)(x_3 - x_1)}{f(x_3) - f(x_1)} \tag{1.38}$$

which is nearer to the root than x_3. For general formulation of this method, consider x^+ to be the value at which $f(x^+) > 0$ and x^- to be the value at which $f(x^-) < 0$. Next improved approximation of the root of the function may be calculated by successive application of the general formula

$$x_n = x^+ - \frac{f(x^+)(x^+ - x^-)}{f(x^+) - f(x^-)} \tag{1.39}$$

For the next iteration, x^+ or x^- should be replaced by x_n according to the sign of $f(x_n)$.

This method is known by several names: method of chords, linear interpolation, false position (*regula falsi*). Its simplicity of calculation (no need for evaluating derivatives of the function) gave it its popularity in the early days of numerical computations. However, its accuracy and speed of convergence are hampered by the choice of x_1, which forms the pivot point for all subsequent iterations.

1.6 THE NEWTON-RAPHSON METHOD

The best known, and possibly the most widely used, technique for locating roots of nonlinear equations is the *Newton-Raphson* method. This method is based on a Taylor series expansion of the nonlinear function $f(x)$ around an initial estimate (x_1) of the root:

$$f(x) = f(x_1) + f'(x_1)(x - x_1) + \frac{f''(x_1)(x - x_1)^2}{2!} + \frac{f'''(x_1)(x - x_1)^3}{3!} + \ldots \tag{1.40}$$

Because what is being sought is the value of x that forces the function $f(x)$ to assume zero value, the left side of Eq. (1.40) is set to zero, and the resulting equation is solved for x.

However, the right-hand side is an infinite series. Therefore, a finite number of terms must be retained and the remaining terms must be truncated. Retaining only the first two terms on the right-hand side of the Taylor series is equivalent to linearizing the function $f(x)$. This operation results in

$$x = x_1 - \frac{f(x_1)}{f'(x_1)} \tag{1.41}$$

that is, the value of x is calculated from x_1 by correcting this initial guess by $f(x_1)/f'(x_1)$. The geometrical significance of this correction is shown in Fig. 1.5*a*. The value of x is obtained by moving from x_1 to x in the direction of the tangent $f'(x_1)$ of the function $f(x)$.

Because the Taylor series was truncated, retaining only two terms, the new value x will not yet satisfy Eq. (1.10). We will designate this value as x_2 and reapply the Taylor series linearization at x_2 (shown in Fig. 1.5*b*) to obtain x_3. Repetitive application of this step converts Eq. (1.41) to an iterative formula:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{1.42}$$

In contrast to the method of linear interpolation discussed in Sec. 1.5, the Newton-Raphson method uses the newly found position as the starting point for each subsequent iteration.

In the discussion for both linear interpolation and Newton-Raphson methods, a certain shape of the function was used to demonstrate how these techniques converge toward a root in the space of search. However, the shapes of nonlinear functions may vary drastically, and convergence is not always guaranteed. As a matter of fact, divergence is more likely to occur, as shown in Fig. 1.6, unless extreme care is taken in the choice of the initial starting points.

To investigate the convergence behavior of the Newton-Raphson method, one has to examine the term $[-f(x_n)/f'(x_n)]$ in Eq. (1.42). This is the error term or correction term applied to the previous estimate of the root at each iteration. A function with a strong vertical trajectory near the root will cause the denominator of the error term to be large; therefore, the

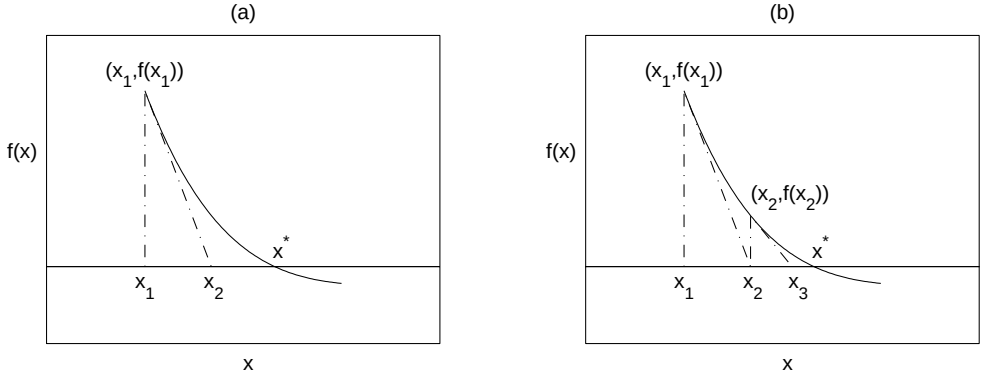

Figure 1.5 The Newton-Raphson method.

convergence will be quite fast. If, however, $f(x)$ is nearly horizontal near the root, the convergence will be slow. If at any point during the search, $f'(x) = 0$, the method would fail due to division by zero. Inflection points on the curve, within the region of search, are also troublesome and may cause the search to diverge.

A sufficient, but not necessary, condition for convergence of the Newton-Raphson method was stated by Lapidus [4] as follows: "If $f'(x)$ and $f''(x)$ do not change sign in the interval (x_1, x^*) and if $f(x_1)$ and $f''(x_1)$ have the same sign, the iteration will always converge to x^*." These convergence criteria may be easily programmed as part of the computer program which performs the Newton-Raphson search, and a warning may be issued or other appropriate action may be taken by the computer if the conditions are violated.

A more accurate extension of the Newton-Raphson method is Newton's 2nd-order method, which truncates the right-hand side of the Taylor series [Eq. (1.40)] after the third term to yield the equation:

$$\frac{f''(x_1)}{2!}(\Delta x_1)^2 + f'(x_1)\,\Delta x_1 + f(x_1) = 0 \tag{1.43}$$

where $\Delta x_1 = x - x_1$. This is a quadratic equation in Δx_1 whose solution is given by

$$\Delta x_1 = \frac{-f'(x_1) \pm \sqrt{[f'(x_1)]^2 - 2f''(x_1)f(x_1)}}{f''(x_1)} \tag{1.44}$$

The general iterative formula for this method would be

$$x_{n+1}^{+} = x_n - \frac{f'(x_n)}{f''(x_n)} + \frac{\sqrt{[f'(x_n)]^2 - 2f''(x_n)f(x_n)}}{f''(x_n)} \tag{1.45a}$$

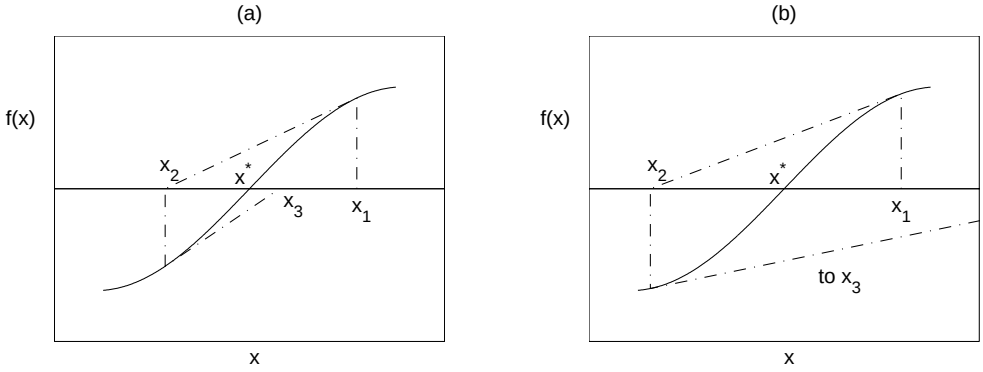

Figure 1.6 Choice of initial guesses affects convergence of Newton-Raphson method. (*a*) Convergence. (*b*) Divergence.

or

$$x_{n+1}^{-} = x_n - \frac{f'(x_n)}{f''(x_n)} - \frac{\sqrt{[f'(x_n)]^2 - 2f''(x_n)f(x_n)}}{f''(x_n)} \tag{1.45b}$$

The choice between (1.45*a*) and (1.45*b*) will be determined by exploring both values of x_{n+1}^{+} and x_{n+1}^{-} and determining which one results in the function $f(x_{n+1}^{+})$ or $f(x_{n+1}^{-})$ being closer to zero.

An alternative to the above exploration will be to treat Eq. (1.43) as another nonlinear equation in Δx and to apply the Newton-Raphson method for its solution:

$$F(\Delta x) = \frac{f''(x_1)}{2!}(\Delta x)^2 + f'(x_1)\Delta x + f(x_1) = 0 \tag{1.46}$$

where

$$\Delta x_{n+1} = \Delta x_n - \frac{F(\Delta x_n)}{F'(\Delta x_n)} \tag{1.47}$$

Two nested Newton-Raphson algorithms would have to be programmed together as follows:

1. Assume a value of x_1.
2. Calculate Δx_1 from Eq. (1.44).
3. Calculate Δx_2 from Eq. (1.47).
4. Calculate x_2 from $x_2 = x_1 + \Delta x_2$.
5. Repeat steps 2 to 4 until convergence is achieved.

Example 1.1: Solution of the Colebrook Equation by Successive Substitution, Linear Interpolation, and Newton-Raphson Methods. Develop MATLAB functions to solve nonlinear equations by the successive substitution method, the linear interpolation, and the Newton-Raphson root-finding techniques. Use these functions to calculate the friction factor from the Colebrook equation [Eq. (1.4)] for flow of a fluid in a pipe with $\epsilon/D = 10^{-4}$ and $N_{Re} = 10^5$. Compare these methods with each other.

Method of Solution: Eqs. (1.29), (1.39), and (1.42) are used for the method of $x = g(x)$, linear interpolation, and Newton-Raphson, respectively. The iterative procedure stops when the difference between two succeeding approximations of the root is less than the convergence criterion (default value is 10^{-6}), or when the number of iterations reaches 100, whichever is satisfied first. The program may show the convergence results numerically and/or graphically, if required, to illustrate how each method arrives at the answer.

Program Description: Three MATLAB functions called *XGX.m*, *LI.m*, and *NR.m* are developed to find the root of a general nonlinear equation using successive substitution [the

method of $x = g(x)$], linear interpolation, and Newton-Raphson methods, respectively. The name of the nonlinear function subject to root finding is introduced in the input arguments; therefore, these MATLAB functions may be applied to any problem.

Successive substitution method (XGX.m): This function starts with initialization section in which input arguments are evaluated and initial values for the main iteration are introduced. The first argument is the name of the MATLAB function in which the function $g(x)$ is described. The second argument is a starting value and has to be a scalar. By default, convergence is assumed when two succeeding iterations result in root approximations with less than 10^{-6} in difference. If another value for convergence is desired, it may be introduced to the function by the third argument. A value of 1 as the fourth argument makes the function show the results of each iteration step numerically. If this value is set to 2, the function shows the results numerically and graphically. The third and fourth arguments are optional. Every additional argument that is introduced after the fourth argument is passed directly to the function $g(x)$. In this case, if it is desired to use the default values for the third and fourth arguments, an empty matrix should be entered in their place. For solution of the problem, the values of N_{Re} and ϵ/D are passed to the Colebrook function by introducing them in fifth and sixth arguments.

The next section in the function is the main iteration loop, in which the iteration according to Eq. (1.29) takes place and the convergence is checked. In the case of the Colebrook equation, Eq. (1.4) is rearranged to solve for f: The right-hand side of this equation is taken as $g(f)$ and is introduced in the MATLAB function *Colebrookg.m*. Numerical results of the calculations are also shown, if requested, in each iteration of this section.

$$f = \frac{1}{\left[0.86 \ln\left(\frac{\epsilon/D}{3.7} + \frac{2.51}{N_{Re}\sqrt{f}}\right)\right]^2} = g(f)$$

At the end, the MATLAB function plots the function as well as the results of the calculation, if required, to illustrate the convergence procedure.

Linear interpolation method (LI.m): This function consists of the same parts as the *XGX.m* function. The number of input arguments is one more than that of *XGX.m*, because the linear interpolation method needs two starting points. Special care should be taken to introduce two starting values in which the function have opposite signs. Eq. (1.5) is used without change as the function the root of which is to be located. This function is contained in a MATLAB function called *Colebrook.m*.

Newton-Raphson method (NR.m): The structure of this function is the same as that of the two previous functions. The derivative of the function is taken numerically to reduce the inputs. It is also more applicable for complicated functions. The reader may simply introduce the derivative function in another MATLAB function and use it instead of numerical derivation. In the case of the Colebrook equation, the same MATLAB function *Colebrook.m*, which represents Eq. (1.5), may be used with this function to calculate the value of the friction factor.

The MATLAB program *Example1_1.m* finds the friction factor from the Colebrook equation by three different methods of root finding. The program first asks the user to input the values for N_{Re} and ϵ/D. It then asks for the method of solution of the Colebrook equation, name of the *m*-file that contains the Colebrook equation, and the initial value(s) to start the method. The program then calculates the friction factor by the selected method and continues asking for the method of solution until the user enters 0.

WARNING: The original MATLAB Version 5.2 had a "bug." The command

linspace(0,0,100)

which is used in *LI.m*, *NR.m*, and *Nrpoly.m* in Chapter 1, would not work properly in the MATLAB installations of Version 5.2 which have not yet been corrected. A patch which corrects this problem is available on the website of Math Works, Inc.:

http://www.mathworks.com

If you have Version 5.2, you are strongly encouraged to download and install this patch, if you have not done so already.

Program

Example1_1.m

```
% Example1_1.m
% This program solves the problem posed in Example 1.1.
% It calculates the friction factor from the Colebrook equation
% using the Successive Substitution, the Linear Interpolation,
% and the Newton-Raphson methods.

clear
clc
clf

disp('Calculating the friction factor from the Colebrook equation')

% Input
Re = input('\n Reynolds No.       = ');
e_over_D = input(' Relative roughness = ');

method = 1;
while method
   fprintf('\n')
   disp('  1 ) Successive substitution')
   disp('  2 ) Linear Interpolation')
   disp('  3 ) Newton Raphson')
   disp('  0 ) Exit')
   method = input('\n Choose the method of solution : ');
   if method
```

```
  fname = input('\n Function containing the Colebrook equation : ');
   end
   switch method
   case 1                                % Successive substitution
      x0 = input(' Starting value = ');
      f = xgx(fname,x0,[],2,Re,e_over_D);
      fprintf('\n f = %6.4f\n',f)
   case 2                                % Linear interpolation
      x1 = input(' First starting value = ');
      x2 = input(' Second starting value = ');
      f = LI(fname,x1,x2,[],2,Re,e_over_D);
      fprintf('\n f = %6.4f\n',f)
   case 3                                % Newton-Raphson
      x0 = input(' Starting value = ');
      f = NR(fname,x0,[],2,Re,e_over_D);
      fprintf('\n f = %6.4f\n',f)
   end
end
```

XGX.m

```
function x = XGX(fnctn,x0,tol,trace,varargin)
%XGX Finds a zero of a function by x=g(x) method.
%
%   XGX('G',X0) finds the intersection of the curve y=g(x)
%   with the line y=x. The function g(x) is  described by the
%   M-file G.M. X0 is a starting guess.
%
%   XGX('G',X0,TOL,TRACE) uses tolerance TOL for convergence
%   test. TRACE=1 shows the calculation steps numerically and
%   TRACE=2 shows the calculation steps both numerically and
%   graphically.
%
%   XGX('G',X0,TOL,TRACE,P1,P2,...) allows for additional
%   arguments which are passed to the function G(X,P1,P2,...).
%   Pass an empty matrix for TOL or TRACE to use the default
%   value.
%
%   See also FZERO, ROOTS, NR, LI

% (c) by N. Mostoufi & A. Constantinides
% January 1, 1999

% Initialization
if nargin < 3 | isempty(tol)
   tol = 1e-6;
end
if nargin < 4 | isempty(trace)
   trace = 0;
end
if tol == 0
   tol = 1e-6;
```

```
end
if (length(x0) > 1) | (~isfinite(x0))
   error('Second argument must be a finite scalar.')
end
if trace
   header = ' Iteration       x            g(x)';
   disp(' ')
   disp(header)
   if trace == 2
      xpath = [x0];
      ypath = [0];
   end
end

x = x0;
x0 = x + 1;
iter = 1;
itermax = 100;

% Main iteration loop
while abs(x - x0) > tol & iter <= itermax
   x0 = x;
   fnk = feval(fnctn,x0,varargin{:});

   % Next approximation of the root
   x = fnk;

   % Show the results of calculation
   if trace
      fprintf('%5.0f   %13.6g %13.6g\n',iter, [x0 fnk])
      if trace == 2
         xpath = [xpath x0 x];
         ypath = [ypath fnk x];
      end
   end
   iter = iter + 1;
end

if trace == 2
   % Plot the function and path to the root
   xmin = min(xpath);
   xmax = max(xpath);
   dx = xmax - xmin;
   xi = xmin - dx/10;
   xf = xmax + dx/10;
   yc = [];
   for xc = xi : (xf - xi)/99 : xf
      yc=[yc feval(fnctn,xc,varargin{:})];
   end
   xc = linspace(xi,xf,100);
   plot(xc,yc,xpath,ypath,xpath(2),ypath(2),'*', ...
```

```
      x,fnk,'o',[xi xf],[xi,xf],'--')
   axis([xi xf min(yc) max(yc)])
   xlabel('x')
   ylabel('g(x) [-- : y=x]')
   title('x=g(x) : The function and path to the root ...
      (* : initial guess ; o : root)')
end

if iter >= itermax
   disp('Warning : Maximum iterations reached.')
end
```

LI.m

```
function x = LI(fnctn,x1,x2,tol,trace,varargin)
%LI Finds a zero of a function by the linear interpolation method.
%
%   LI('F',X1,X2) finds a zero of the function described by the
%   M-file F.M. X1 and X2 are starting points where the function
%   has different signs at these points.
%
%   LI('F',X1,X2,TOL,TRACE) uses tolerance TOL for convergence
%   test. TRACE=1 shows the calculation steps numerically and
%   TRACE=2 shows the calculation steps both numerically and
%   graphically.
%
%   LI('F',X1,X2,TOL,TRACE,P1,P2,...) allows for additional
%   arguments which are passed to the function F(X,P1,P2,...).
%   Pass an empty matrix for TOL or TRACE to use the default
%   value.
%
%   See also FZERO, ROOTS, XGX, NR

% (c) by N. Mostoufi & A. Constantinides
% January 1, 1999

% Initialization
if nargin < 4 | isempty(tol)
   tol = 1e-6;
end
if nargin < 5 | isempty(trace)
   trace = 0;
end
if tol == 0
   tol = 1e-6;
end
if (length(x1) > 1) | (~isfinite(x1)) | (length(x2) > 1) | ...
      (~isfinite(x2))
   error('Second and third arguments must be finite scalars.')
end
if trace
   header = ' Iteration       x            f(x)';
```

```
   disp(' ')
   disp(header)
end
f1 = feval(fnctn,x1,varargin{:});
f2 = feval(fnctn,x2,varargin{:});

iter = 0;
if trace
   % Display initial values
   fprintf('%5.0f   %13.6g %13.6g \n',iter, [x1 f1])
   fprintf('%5.0f   %13.6g %13.6g \n',iter, [x2 f2])
   if trace == 2
      xpath = [x1 x1 x2 x2];
      ypath = [0 f1 f2 0];
   end
end

if f1 < 0
   xm = x1;
   fm = f1;
   xp = x2;
   fp = f2;
else
   xm = x2;
   fm = f2;
   xp = x1;
   fp = f1;
end

iter = iter + 1;
itermax = 100;
x = xp;
x0 = xm;

% Main iteration loop
while abs(x - x0) > tol & iter <= itermax
   x0 = x;
   x = xp - fp * (xm - xp) / (fm - fp);
   fnk = feval(fnctn,x,varargin{:});

   if fnk < 0
      xm = x;
      fm = fnk;
   else
      xp = x;
      fp = fnk;
   end

   % Show the results of calculation
   if trace
      fprintf('%5.0f   %13.6g %13.6g \n',iter, [x fnk])
```

```
      if trace == 2
         xpath = [xpath xm xm xp xp];
         ypath = [ypath 0 fm fp 0];
      end
   end
   iter = iter + 1;
end

if trace == 2
   % Plot the function and path to the root
   xmin = min(xpath);
   xmax = max(xpath);
   dx = xmax - xmin;
   xi = xmin - dx/10;
   xf = xmax + dx/10;
   yc = [];
   for xc = xi : (xf - xi)/99 : xf
      yc=[yc feval(fnctn,xc,varargin{:})];
   end
   xc = linspace(xi,xf,100);
   ax = linspace(0,0,100);
   plot(xc,yc,xpath,ypath,xc,ax,xpath(2:3),ypath(2:3),'*',x,fnk,'o')
   axis([xi xf min(yc) max(yc)])
   xlabel('x')
   ylabel('f(x)')
   title('Linear Interpolation : The function and path to the root
(* : initial guess ; o : root)')
end

if iter >= itermax
   disp('Warning : Maximum iterations reached.')
end
```

NR.m

```
function x = NR(fnctn,x0,tol,trace,varargin)
%NR Finds a zero of a function by the Newton-Raphson method.
%
%   NR('F',X0) finds a zero of the function described by the
%   M-file F.M. X0 is a starting guess.
%
%   NR('F',X0,TOL,TRACE) uses tolerance TOL for convergence
%   test. TRACE=1 shows the calculation steps numerically and
%   TRACE=2 shows the calculation steps both numerically and
%   graphically.
%
%   NR('F',X0,TOL,TRACE,P1,P2,...) allows for additional
%   arguments which are passed to the function F(X,P1,P2,...).
%   Pass an empty matrix for TOL or TRACE to use the default
%   value.
%
%   See also FZERO, ROOTS, XGX, LI
```

```
% (c) by N. Mostoufi & A. Constantinides
% January 1, 1999

% Initialization
if nargin < 3 | isempty(tol)
   tol = 1e-6;
end
if nargin < 4 | isempty(trace)
   trace = 0;
end
if tol == 0
   tol = 1e-6;
end
if (length(x0) > 1) | (~isfinite(x0))
   error('Second argument must be a finite scalar.')
end

iter = 0;
fnk = feval(fnctn,x0,varargin{:});
if trace
   header = ' Iteration       x            f(x)';
   disp(' ')
   disp(header)
   fprintf('%5.0d   %13.6g %13.6g \n',iter, [x0 fnk])
   if trace == 2
      xpath = [x0 x0];
      ypath = [0 fnk];
   end
end

x = x0;
x0 = x + 1;
itermax = 100;

% Main iteration loop
while abs(x - x0) > tol & iter <= itermax
   iter = iter + 1;
   x0 = x;

   % Set dx for differentiation
   if x ~= 0
      dx = x/100;
   else
      dx = 1/100;
   end

   % Differentiation
   a = x - dx;  fa = feval(fnctn,a,varargin{:});
   b = x + dx;  fb = feval(fnctn,b,varargin{:});
   df = (fb - fa)/(b - a);
```

```
   % Next approximation of the root
   if df == 0
      x = x0 + max(abs(dx),1.1*tol);
   else
      x = x0 - fnk/df;
   end

   fnk = feval(fnctn,x,varargin{:});
   % Show the results of calculation
   if trace
      fprintf('%5.0d   %13.6g %13.6g \n',iter, [x fnk])
      if trace == 2
         xpath = [xpath x x];
         ypath = [ypath 0 fnk];
      end
   end
end

if trace == 2
   % Plot the function and path to the root
   xmin = min(xpath);
   xmax = max(xpath);
   dx = xmax - xmin;
   xi = xmin - dx/10;
   xf = xmax + dx/10;
   yc = [];
   for xc = xi : (xf - xi)/99 : xf
      yc = [yc feval(fnctn,xc,varargin{:})];
   end
   xc = linspace(xi,xf,100);
   ax = linspace(0,0,100);
   plot(xc,yc,xpath,ypath,xc,ax,xpath(1),ypath(2),'*',x,fnk,'o')
   axis([xi xf min(yc) max(yc)])
   xlabel('x')
   ylabel('f(x)')
   title('Newton-Raphson : The function and path to the root
     (* : initial guess ; o : root)')
end

if iter >= itermax
   disp('Warning : Maximum iterations reached.')
end
```

Colebrook.m

```
function y = Colebrook(f, Re, e)
% Colebrook.m
% This function evaluates the value of Colebrook equation to be
% solved by the linear interpolation or the Newton-Raphson method.

y = 1/sqrt(f) + 0.86*log(e/3.7 + 2.51/Re/sqrt(f));
```

Colebrookg.m

```
function y = clbrkg(f, Re, e)
% Colebrookg.m
% This function evaluates the value of the rearranged Colebrook
% equation to be solved by x=g(x) method.

y=1/(0.86*log(e/3.7+2.51/Re/sqrt(f)))^2;
```

Input and Results

```
>>Example1_1

Calculating the friction factor from the Colebrook equation

 Reynolds No.       = 1e5
 Relative roughness = 1e-4

  1 ) Successive substitution
  2 ) Linear Interpolation
  3 ) Newton-Raphson
  0 ) Exit

 Choose the method of solution : 1

 Function containing the Colebrook equation : 'Colebrookg'
 Starting value = 0.01

 Iteration       x            g(x)
    1            0.01     0.0201683
    2       0.0201683     0.0187204
    3       0.0187204     0.0188639
    4       0.0188639     0.0188491
    5       0.0188491     0.0188506
    6       0.0188506     0.0188505

f = 0.0189

  1 ) Successive substitution
  2 ) Linear Interpolation
  3 ) Newton-Raphson
  0 ) Exit

 Choose the method of solution : 2

 Function containing the Colebrook equation : 'Colebrook'
 First  starting value = 0.01
 Second starting value = 0.03

 Iteration       x            f(x)
    0            0.01    2.9585
    0            0.03   -1.68128
```

```
    1       0.0227528  -0.723985
    2       0.0202455  -0.282098
    3       0.0193536  -0.105158
    4       0.0190326  -0.0385242
    5       0.0189165  -0.0140217
    6       0.0188744  -0.00509133
    7       0.0188592  -0.00184708
    8       0.0188536  -0.000669888
    9       0.0188516  -0.000242924
   10      0.0188509  -8.80885e-005

f = 0.0189

  1 ) Successive substitution
  2 ) Linear Interpolation
  3 ) Newton-Raphson
  0 ) Exit

 Choose the method of solution : 3

 Function containing the Colebrook equation : 'Colebrook'
 Starting value = 0.01

 Iteration       x            f(x)
    0            0.01   2.9585
    1       0.0154904   0.825216
    2       0.0183977   0.0982029
    3       0.0188425   0.00170492
    4       0.0188505  6.30113e-007
    5       0.0188505  3.79075e-011

f = 0.0189

  1 ) Successive substitution
  2 ) Linear Interpolation
  3 ) Newton-Raphson
  0 ) Exit

 Choose the method of solution : 0
```

Discussion of Results: All three methods are applied to finding the root of the Colebrook equation for the friction factor. Graphs of the step-by-step path to convergence are shown in Figs. E1.1*a*, *b*, and *c* for the three methods. It can be seen that Newton-Raphson converges faster than the other two methods. However, the Newton-Raphson method is very sensitive to the initial guess, and the method may converge to the other roots of the equation, if a different starting point is used. The reader may test other starting points to examine this sensitivity. The convergence criterion in all the above MATLAB functions is $|x_n - x_{n-1}| < 10^{-6}$.

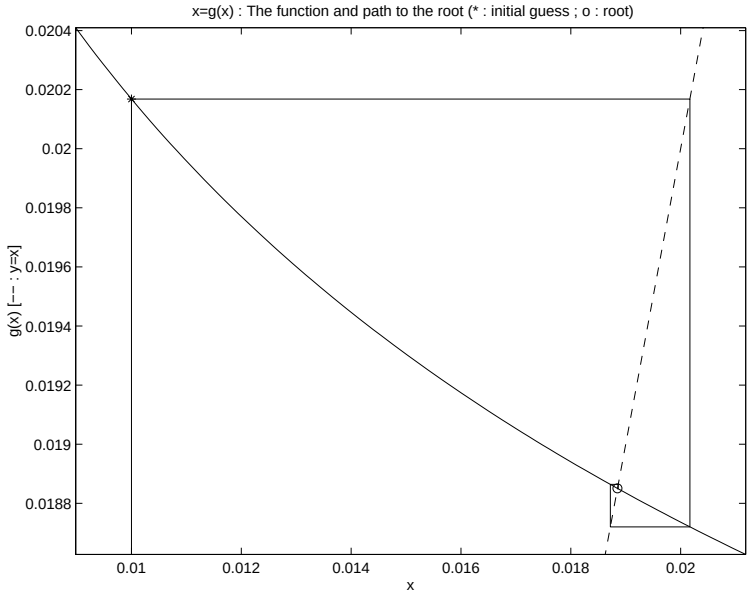

Figure E1.1a Solution using the method of successive substitution.

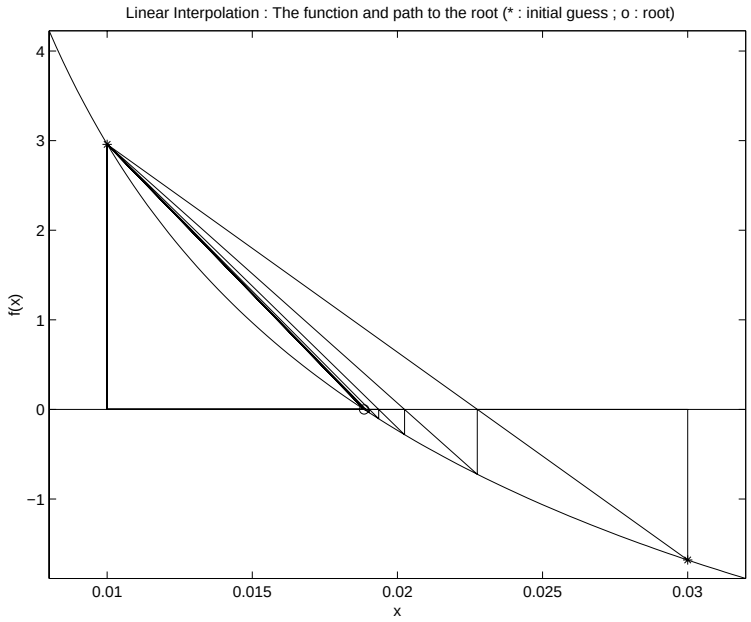

Figure E1.1*b* Solution using the method of linear interpolation.

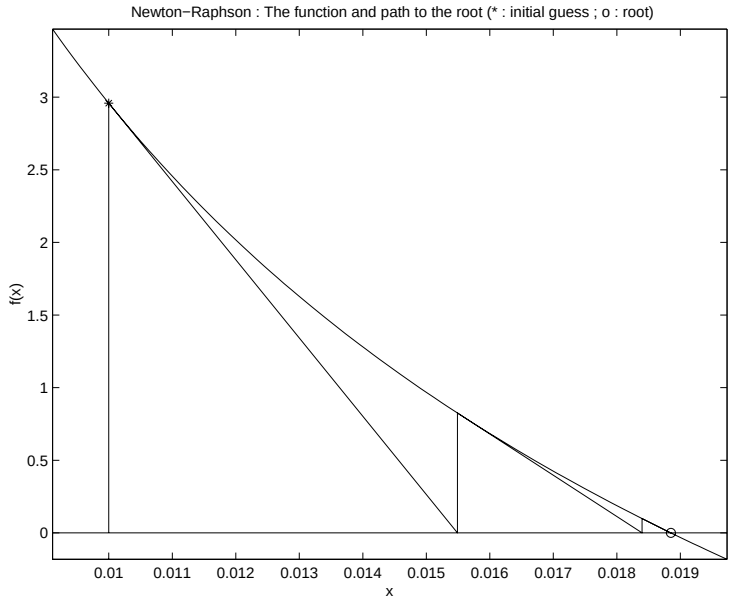

Figure E1.1*c* Solution using the Newton-Raphson method.

Example 1.2: Finding a Root of an *n*th-Degree Polynomial by Newton-raphson Method Applied to the Soave-Redlich-Kwong Equation of State. Develop a MATLAB function to calculate a root of a polynomial equation by Newton-Raphson method. Calculate the specific volume of a pure gas, at a given temperature and pressure, by using the Soave-Redlich-Kwong equation of state

$$P = \frac{RT}{V - b} - \frac{a\alpha}{V(V + b)}$$

The equation constants, a and b, are obtained from

$$a = \frac{0.4278\, R^2\, T_C^2}{P_C}$$

$$b = \frac{0.0867\, R\, T_C}{P_C}$$

where T_C and P_C are critical temperature and pressure, respectively. The variable α is an empirical function of temperature:

$$\alpha = \left[1 + S\left(1 - \sqrt{\frac{T}{T_C}}\right)\right]^2$$

The value of S is a function of the acentric factor, ω, of the gas:

$$S = 0.48508 + 1.55171\,\omega - 0.15613\,\omega^2$$

The physical properties of *n*-butane are:

$$T_C = 425.2\ K, \qquad P_C = 3797\ kPa, \qquad \omega = 0.1931$$

and the gas constant is:

$$R = 8314\ J/kmol.K.$$

Calculate the specific volume of *n*-butane vapor at 500 K and at temperatures from 1 to 40 atm. Compare the results graphically with the ones obtained from using the ideal gas law. What conclusion do you draw from this comparison?

Method of Solution: Eq. (1.42) is used for Newton-Raphson evaluation of the root. For finding the gas specific volume from the Soave-Redlich-Kwong equation of state, Eq. (1.2), which is a third-degree polynomial in compressibility factor, is solved. Starting value for the iterative method is $Z = 1$, which is the compressibility factor of the ideal gas.

Program Description: The MATLAB function *NRpoly.m* calculates a root of a polynomial equation by Newton-Raphson method. The first input argument of this function is the vector of coefficients of the polynomial, and the second argument is an initial guess of the root. The function employs MATLAB functions *polyval* and *polyder* for evaluation of the polynomial and its derivative at each point. The reader can change the convergence criterion by introducing a new value in the third input argument. The default convergence criterion is 10^{-6}. The reader may also see the results of the calculations at each step numerically and graphically by entering the proper value as the fourth argument (1 and 2, respectively). The third and fourth arguments are optional.

MATLAB program *Example1_2.m* solves the Soave-Redlich-Kwong equation of state by utilizing the *NRpoly.m* function. In the beginning of this program, temperature, pressure range and the physical properties of *n*-butane are entered. The constants of the Soave-Redlich-Kwong equation of state are calculated next. The values of A and B [used in Eq. (1.2)] are also calculated in this section. Evaluation of the root is done in the third part of the program. In this part, the coefficients of Eq. (1.2) are first introduced and the root of the equation, closest to the ideal gas, is determined using the above-mentioned MATLAB function *NRpoly*. The last part of the program, *Example1_2.m*, plots the results of the calculation both for Soave-Redlich-Kwong and ideal gas equations of state. It also shows some of the numerical results.

Program

Example1_2.m

```
% Example1_2.m
% This program solves the problem posed in Example 1.2.
% It calculates the real gas specific volume from the
% SRK equation of state using the Newton-Raphson method
% for calculating the roots of a polynomial.

clear
clc
clf

% Input data
P = input(' Input the vector of pressure range (Pa) = ');
T = input(' Input temperature (K) = ');
R = 8314;    % Gas constant (J/kmol.K)
Tc = input(' Critical temperature (K) = ');
Pc = input(' Critical pressure (Pa) = ');
omega = input(' Acentric factor = ');

% Constants of Soave-Redlich-Kwong equation of state
a = 0.4278 * R^2 * Tc^2 / Pc;
b = 0.0867 * R * Tc / Pc;
sc = [-0.15613, 1.55171, 0.48508];
s = polyval(sc,omega);
alpha = (1 + s * (1 - sqrt(T/Tc)))^2;
A = a * alpha * P / (R^2 * T^2);
B = b * P / (R * T);

for k = 1:length(P)
   % Defining the polynomial coefficients
   coef = [1, -1, A(k)-B(k)-B(k)^2, -A(k)*B(k)];
   v0(k) = R * T / P(k);   % Ideal gas specific volume
   vol(k) = NRpoly(coef , 1) * R * T / P(k);   % Finding the root
end

% Show numerical results
fprintf('\nRESULTS:\n');
fprintf('Pres. =  %5.2f  Ideal gas vol. =%7.4f',P(1),v0(1));
fprintf('  Real gas vol. =%7.4f\n',vol(1));
for k=10:10:length(P)
   fprintf('Pres. = %5.2f  Ideal gas vol. =%7.4f',P(k),v0(k));
   fprintf('  Real gas vol. =%7.4f\n',vol(k));
end

% plotting the results
loglog(P/1000,v0,'.',P/1000,vol)
xlabel('Pressure, kPa')
```

```
ylabel('Specific Volume, m^3/kmol')
legend('Ideal','SRK')
```

NRpoly.m

```
function x = NRpoly(c,x0,tol,trace)
%NRPOLY Finds a root of polynomial by the Newton-Raphson method.
%
%   NRPOLY(C,X0) computes a root of the polynomial whose
%   coefficients are the elements of the vector C.
%   If C has N+1 components, the polynomial is
%   C(1)*X^N + ... + C(N)*X + C(N+1).
%   X0 is a starting point.
%
%   NRPOLY(C,X0,TOL,TRACE) uses tolerance TOL for convergence
%   test. TRACE=1 shows the calculation steps numerically and
%   TRACE=2 shows the calculation steps both numerically and
%   graphically.
%
%   See also ROOTS, NR, LI, XGX, FZERO.

% (c) by N. Mostoufi & A. Constantinides
% January 1, 1999

% Initialization
if nargin < 3 | isempty(tol)
   tol = 1e-6;
end
if nargin < 4 | isempty(trace)
   trace = 0;
end
if tol == 0
   tol = 1e-6;
end
if (length(x0) > 1) | (~isfinite(x0))
   error('Second argument must be a finite scalar.')
end

iter = 0;
fnk = polyval(c,x0);   % Function
if trace
  header = ' Iteration         x           f(x)';
  disp(header)
  disp([sprintf('%5.0f   %13.6g %13.6g ',iter, [x0 fnk])])
  if trace == 2
     xpath = [x0 x0];
     ypath = [0 fnk];
  end
end
```

```
x = x0;
x0 = x + .1;
maxiter = 100;

% Solving the polynomial by Newton-Raphson method
while abs(x0 - x) > tol & iter < maxiter
   iter = iter + 1;
   x0 = x;
   fnkp = polyval(polyder(c),x0);% Derivative
   if fnkp ~= 0
      x = x0 - fnk / fnkp; % Next approximation
   else
      x = x0 + .01;
   end

   fnk = polyval(c,x);  % Function
   % Show the results of calculation
   if trace
      disp([sprintf('%5.0f   %13.6g %13.6g ',iter, [x fnk])])
      if trace == 2
         xpath = [xpath x x];
         ypath = [ypath 0 fnk];
      end
   end
end

if trace == 2
   % Plot the function and path to the root
   xmin = min(xpath);
   xmax = max(xpath);
   dx = xmax - xmin;
   xi = xmin - dx/10;
   xf = xmax + dx/10;
   yc = [];
   for xc = xi : (xf - xi)/99 : xf
      yc = [yc polyval(c,xc)];
   end
   xc = linspace(xi,xf,100);
   ax = linspace(0,0,100);
   plot(xc,yc,xpath,ypath,xc,ax,xpath(1),ypath(2),'*',x,fnk,'o')
   axis([xi xf min(yc) max(yc)])
   xlabel('x')
   ylabel('f(x)')
   title('Newton-Raphson : The function and path to the root (* :
initial guess ; o : root)')
end

if iter == maxiter
   disp('Warning : Maximum iterations reached.')
end
```

Input and Results

```
>>Example1_2

 Input the vector of pressure range (Pa) : [1:40]*101325
 Input temperature (K) : 500
 Critical temperature (K) : 425.2
 Critical pressure (Pa) : 3797e3
 Acentric factor : 0.1931

Pres. =  101325.00  Ideal gas vol. =41.0264  Real gas vol. =40.8111
Pres. = 1013250.00  Ideal gas vol. = 4.1026  Real gas vol. = 3.8838
Pres. = 2026500.00  Ideal gas vol. = 2.0513  Real gas vol. = 1.8284
Pres. = 3039750.00  Ideal gas vol. = 1.3675  Real gas vol. = 1.1407
Pres. = 4053000.00  Ideal gas vol. = 1.0257  Real gas vol. = 0.7954
```

Discussion of Results: In this example we use the *Example1_2.m* program to calculate the specific volume of a gas using the Soave-Redlich-Kwong equation of state. Because this equation can be arranged in the canonical form of a third-degree polynomial, the function *NRpoly.m* can be used. Additional information such as temperature, pressure, and physical properties are entered by the user through the program.

Above the critical temperature, the Soave-Redlich-Kwong equation of state has only one real root that is of interest, the one located near the value given by the ideal gas law. Therefore, the latter, which corresponds to $Z = 1$, is used as the initial guess of the root.

Direct comparison between the Soave-Redlich-Kwong and ideal gas volumes is made in Fig. E1.2. It can be seen from this figure that the ideal gas equation overestimates gas volumes and, as expected from thermodynamic principles, the deviation from ideality increases as the pressure increases.

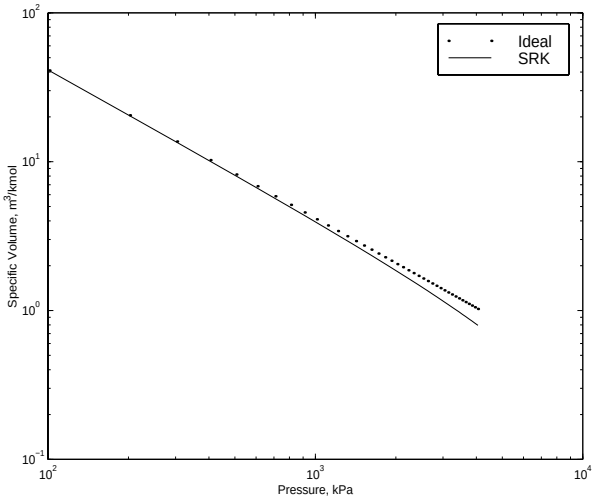

Figure E1.2 Graphical comparison between the Soave-Redlich-Kwong and the ideal gas equations of state.

1.7 SYNTHETIC DIVISION ALGORITHM

If the nonlinear equation being solved is of the polynomial form, each real root (located by one of the methods already discussed) can be removed from the polynomial by synthetic division, thus reducing the degree of the polynomial to (n - 1). Each successive application of the synthetic division algorithm will reduce the degree of the polynomial further, until all real roots have been located.

A simple computational algorithm for synthetic division has been given by Lapidus [4]. Consider the fourth-degree polynomial

$$f(x) = a_4 x^4 + a_3 x^3 + a_2 x^2 + a_1 x + a_0 = 0 \tag{1.48}$$

whose first real root has been determined to be x^*. This root can be factored out as follows:

$$f(x) = (x - x^*)(b_3 x^3 + b_2 x^2 + b_1 x + b_0) = 0 \tag{1.49}$$

In order to determine the coefficients (b_i) of the third-degree polynomial first multiply out Eq. (1.49) and rearrange in descending power of x:

$$f(x) = b_3 x^4 + (b_2 - b_3 x^*) x^3 + (b_1 - b_2 x^*) x^2 + (b_0 - b_1 x^*) x - b_0 x^* \tag{1.50}$$

Equating Eqs. (1.48) and (1.50), the coefficients of like powers of x must be equal to each other, that is,

$$\begin{aligned} a_3 &= b_2 - b_3 x^* \\ a_2 &= b_1 - b_2 x^* \\ a_1 &= b_0 - b_1 x^* \end{aligned} \tag{1.51}$$

Solving Eqs. (1.51) for b_i we obtain:

$$\begin{aligned} b_3 &= a_4 \\ b_2 &= a_3 + b_3 x^* \\ b_1 &= a_2 + b_2 x^* \\ b_0 &= a_1 + b_1 x^* \end{aligned} \tag{1.52}$$

In general notation, for a polynomial of *n*th-degree, the new coefficients after application of synthetic division are given by

$$\begin{aligned} b_{n-1} &= a_n \\ b_{n-1-r} &= a_{n-r} + b_{n-r}x^* \end{aligned} \tag{1.53}$$

where $r = 1, 2, \ldots, (n - 1)$. The polynomial is then reduced by one degree

$$n_{j+1} = n_j - 1 \tag{1.54}$$

where j is the iteration number, and the newly calculated coefficients are renamed as shown by Eq. (1.55):

$$\begin{aligned} b_{n-1} &= a_n \\ b_{n-1-r} &= a_{n-r} + b_{n-r}x^* \end{aligned} \tag{1.55}$$

This procedure is repeated until all real roots are extracted. When this is accomplished, the remainder polynomial will contain the complex roots. The presence of a pair of complex roots will give a quadratic equation that can be easily solved by quadratic formula. However, two or more pairs of complex roots require the application of more elaborate techniques, such as the eigenvalue method, which is developed in the next section.

1.8 THE EIGENVALUE METHOD

The concept of eigenvalues will be discussed in Chap. 2 of this textbook. As a preview of that topic, we will state that a square matrix has a *characteristic polynomial* whose roots are called the *eigenvalues* of the matrix. However, root-finding methods that have been discussed up to now are not efficient techniques for calculating eigenvalues [5]. There are more efficient eigenvalue methods to find the roots of the characteristic polynomial (see Sec. 2.8).

It can be shown that Eq. (1.11) is the characteristic polynomial of the $(n \times n)$ companion matrix $\boldsymbol{A}$, which contains the coefficients of the original polynomial as `shown in Eq. (1.56).` Therefore, finding the eigenvalues of $\boldsymbol{A}$ is equivalent to locating the roots of the polynomial in Eq. (1.11).

MATLAB has its own function, *roots.m*, for calculating all the roots of a polynomial equation of the form in Eq. (1.11). This function accomplishes the task of finding the roots of the polynomial equation [Eq. (1.11)] by first converting the polynomial to the companion matrix $\boldsymbol{A}$ shown in Eq. (1.56). It then uses the built-in function *eig.m*, which calculates the eigenvalues of a matrix, to evaluate the eigenvalues of the companion matrix, which are also the roots of the polynomial Eq. (1.11):

$$\boldsymbol{A} = \begin{bmatrix} -\frac{a_{n-1}}{a_n} & -\frac{a_{n-2}}{a_n} & \cdots & -\frac{a_1}{a_n} & -\frac{a_0}{a_n} \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & & & & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix} \tag{1.56}$$

Example 1.3: Solution of *n*th-Degree Polynomials and Transfer Functions Using the Newton-Raphson Method with Synthetic Division and Eigenvalue Method. Consider the isothermal continuous stirred tank reactor (CSTR) shown in Fig. E1.3.

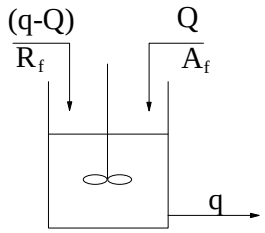

Figure E1.3 The continuous stirred tank reactor.

Components *A* and *R* are fed to the reactor at rates of *Q* and (*q* - *Q*), respectively. The following complex reaction scheme develops in the reactor:

$$A + R \rightarrow B$$

$$B + R \rightarrow C$$

$$C + R \leftrightarrows D$$

$$D + R \rightarrow E$$

This problem was analyzed by Douglas [6] in order to illustrate the various techniques for designing simple feedback control systems. In his analysis of this system, Douglas made the following assumptions:

1. Component *R* is present in the reactor in sufficiently large excess so that the reaction rates can be approximated by first-order expressions.
2. The feed compositions of components *B*, *C*, *D*, and *E* are zero.
3. A particular set of values is chosen for feed concentrations, feed rates, kinetic rate constant, and reactor volume.
4. Disturbances are due to changes in the composition of component *R* in the vessel.

The control objective is to maintain the composition of component *C* in the reactor as close as possible to the steady-state design value, despite the fact that disturbances enter the system. This objective is accomplished by measuring the actual composition of *C* and using the difference between the desired and measured values to manipulate the inlet flow rate *Q* of component *A*.

Douglas developed the following transfer function for the reactor with a proportional control system:

$$K_C \frac{2.98\,(s + 2.25)}{(s + 1.45)(s + 2.85)^2(s + 4.35)} = -1$$

where K_C is the gain of the proportional controller. This control system is stable for values of K_C that yield roots of the transfer function having negative real parts.

Using the Newton-Raphson method with synthetic division or eigenvalue method, determine the roots of the transfer function for a range of values of the proportional gain K_C and calculate the critical value of K_C above which the system becomes unstable. Write the program so that it can be used to solve *n*th-degree polynomials or transfer functions of the type shown in the above equation.

Method of Solution: In the Newton-Raphson method with synthetic division, Eq. (1.42) is used for evaluation of each root. Eqs. (1.53)-(1.55) are then applied to perform synthetic division in order to extract each root from the polynomial and reduce the latter by one degree. When the *n*th-degree polynomial has been reduced to a quadratic

$$a_2 x^2 + a_1 x + a_0 = 0$$

the program uses the quadratic solution formula

$$x_{1,2} = \frac{-a_1 \pm \sqrt{a_1^2 - 4a_2a_0}}{2\,a_2}$$

to check for the existence of a pair of complex roots. In the eigenvalue method, the MATLAB function *roots* may be used directly.

The numerator and the denominator of the transfer function are multiplied out to yield

$$\frac{(2.98\,s + 6.705)\,K_C}{s^4 + 11.50\,s^3 + 47.49\,s^2 + 83.0632\,s + 51.2327} = -1$$

A first-degree polynomial is present in the numerator and a fourth-degree polynomial in the denominator. To convert this to the canonical form of a polynomial, we multiply through by the denominator and rearrange to obtain

$$[s^4 + 11.50\,s^3 + 47.49\,s^2 + 83.0632\,s + 51.2327] + [2.98\,s + 6.705]\,K_C = 0$$

It is obvious that once a value of K_C is chosen, the two bracketed terms of this equation can be added to form a single fourth-degree polynomial whose roots can be evaluated.

When $K_C = 0$, the transfer function has the following four negative real roots, which can be found by inspection of the original transfer function:

$$s_1 = -1.45 \qquad s_2 = -2.85 \qquad s_3 = -2.85 \qquad s_4 = -4.35$$

These are called the poles of the open-loop transfer function.

The value of K_C that causes one or more of the roots of the transfer function to become positive (or have positive real parts) is called the critical value of the proportional gain. This critical value is calculated as follows:

1. A range of search for K_C is established.
2. The bisection method is used to search this range.
3. All the roots of the transfer function are evaluated at each step of the bisection search.
4. The roots are checked for positive real part. The range of K_C, over which the change from negative to positive roots occurs, is retained.
5. Steps 2-4 are repeated until successive values of K_C change by less than a convergence criterion, ϵ.

Program Description: The MATLAB function *NRsdivision.m* calculates all roots of a polynomial by the Newton-Raphson method with synthetic division as described in the Method of Solution. Unlike other functions employing the Newton-Raphson method, this function does not need a starting value as one of the input arguments. Instead, the function generates a starting point at each step according to Eq. (1.22). Only polynomials that have no more than a pair of complex roots can be handled by this function. If the polynomial has more than a pair of complex roots, the function *roots* should be used instead. The function is written in general form and may be used in other programs directly.

The MATLAB program *Example1_3.m* does the search for the desired value of K_C by the bisection method. At the beginning of the program, the user is asked to enter the coefficients of the numerator and the denominator of the transfer function (in descending *s* powers). The numerator and the denominator may be of any degree with the limitation that the numerator cannot have a degree greater than that of the denominator. The user should also enter the range of search and method of root finding. It is good practice to choose zero for the minimum value of the range; thus, poles of the open-loop transfer function are evaluated in the first step of the search. The maximum value must be higher than the critical value, otherwise the search will not arrive at the critical value.

Stability of the system is examined at the minimum, maximum, and midpoints of the range of search of K_C. That half of the interval in which the change from negative to positive (stable to unstable system) occurs is retained by the bisection algorithm. This new interval is bisected again and the evaluation of the system stability is repeated, until the convergence criterion, which is $|\Delta K_C| < 0.001$, is met.

In order to determine whether the system is stable or unstable, the two polynomials are combined, as shown in the Method of Solution, using K_C as the multiplier of the polynomial from the numerator of the transfer function. Function *NRsdivision* (which uses the Newton-Raphson method with synthetic division algorithm) or function *roots* (which uses the eigenvalue algorithm) is called to calculate the roots of the overall polynomial function and the sign of all roots is checked for positive real parts. A flag named stbl indicates that the system is stable (all negative roots; stbl = 1) or unstable (positive root; stbl = 0).

Program

Example1_3.m

```
% Example1_3.m
% Solution to the problem posed in Example 1.3. It calculates the
% critical value of the constant of a proportional controller above
% which the system of chemical reactor becomes unstable. This program
% evaluate all roots of the denominator of the transfer function using
% the Newton-Raphson method with synthetic division or eigenvalue
% methods.

clear
clc

% Input data
num = input(' Vector of coefficients of the numerator polynomial =
');
denom = input (' Vector of coefficients of the denominator
polynomial = ');
disp(' ')
Kc1 = input(' Lower limit of the range of search = ');
Kc2 = input(' Upper limit of the range of search = ');
disp(' ')
disp(' 1 ) Newton-Raphson with synthetic division')
disp(' 2 ) Eigenvalue method')
method = input(' Method of root finding = ');

iter = 0;
n1 = length(num);
n2 = length(denom);
c(1:n2-n1) = denom (1:n2-n1);
```

```
% Main loop
while abs(Kc1 - Kc2) > 0.001
   iter = iter + 1;
   if iter == 1
      Kc = Kc1;                          % Lower limit
   elseif iter == 2
      Kc = Kc2;                          % Upper limit
   else
      Kc = (Kc1 + Kc2) / 2;              % Next approximation
   end

   % Calculation of coefficients of canonical form of polynomial
   for m = n2-n1+1 : n2;
      c(m) = denom(m) + Kc * num(m-n2+n1);
   end

   % Root finding
   switch method
   case 1                 %Newton-Raphson with synthetic division
      root = NRsdivision(c);
   case 2                                % Eigenvalue method
      root = roots (c);
   end
   realpart = real (root);
   imagpart = imag (root);

   % Show the results of calculations of this step
   fprintf('\n Kc = %6.4f\n Roots = ',Kc)
   for k = 1:length(root)
      if isreal(root(k))
         fprintf('%7.5g   ',root(k))
      else
         fprintf('%6.4g',realpart(k))
         if imagpart(k) >= 0
            fprintf('+%5.4gi   ',imagpart(k))
         else
            fprintf('-%5.4gi   ',abs(imagpart(k)))
         end
      end
   end
   disp(' ')
   % Determining stability or unstability of the system
   stbl = 1;
   for m = 1 : length(root)
      if realpart(m) > 0
         stbl = 0;                       % System is unstable
         break;
      end
   end
```

```
   if iter == 1
      stbl1 = stbl;
   elseif iter == 2
      stbl2 = stbl;
      if stbl1 == stbl2
         error('Critical value is outside the range of search.')
         break
      end
   else
      if stbl == stbl1
         Kc1 = Kc;
      else
         Kc2 = Kc;
      end
   end
end
```

NRsdivision.m

```
function x = NRsdivision(c,tol)
%NRSDIVISION Finds polynomial roots.
%
%   The function NRSDIVISION(C) evaluates the roots of a
%   polynomial equation whose coefficients are given in the
%   vector C.
%
%   NRSDIVISION(C,TOL) uses tolerance TOL for convergence
%   test. Using the second argument is optional.
%
%   The polynomial may have no more than a pair of complex
%   roots. A root of nth-degree polynomial is determined by
%   Newton-Raphson method. This root is then extracted from
%   the polynomial by synthetic division. This procedure
%   continues until the polynomial reduces to a quadratic.
%
%   See also ROOTS, NRpoly, NR

% (c) by N. Mostoufi & A. Constantinides
% January 1, 1999

% Initialization
if nargin < 2 | isempty(tol)
   tol = 1e-6;
end
if tol == 0
   tol = 1e-6;
end

n = length(c) - 1;                    % Degree of the polynomial
a = c;
```

```
% Main loop
for k = n : -1 : 3
   x0 = -a(2)/a(1);
   x1=x0+0.1;
   iter = 0;
   maxiter = 100;

   % Solving the polynomial by Newton-Raphson method
   while abs(x0 - x1) > tol & iter < maxiter
      iter = iter + 1;
      x0 = x1;
      fnk = polyval(a,x0);              % Function
      fnkp = polyval(polyder(a),x0);    % Derivative
      if fnkp ~= 0
         x1 = x0 - fnk / fnkp;          % Next approximation
      else
         x1 = x0 + 0.01;
      end
   end

   x(n-k+1) = x1;                       % the root

   % Calculation of new coefficients
   b(1) = a(1);
   for r = 2 : k
      b(r) = a(r) + b(r-1) * x1;
   end

   if iter == maxiter
      disp('Warning : Maximum iteration reached.')
   end

   clear a
   a = b;
   clear b
end

% Roots of the remaining quadratic polynomial
delta = a(2) ^ 2 - 4 * a(1) * a(3);
x(n-1) = (-a(2) - sqrt(delta)) / (2 * a(1));
x(n) = (-a(2) + sqrt(delta)) / (2 * a(1));

x=x';
```

Input and Results

```
>>Example1_3

 Vector of coefficients of the numerator polynomial = [2.98, 6.705]
```

```
 Vector of coefficients of the denominator polynomial = [1, 11.5,
47.49, 83.0632, 51.2327]

 Lower limit of the range of search = 0
 Upper limit of the range of search = 100

 1 ) Newton-Raphson with synthetic division
 2 ) Eigenvalue method
 Method of root finding = 1

 Kc = 0.0000
 Roots =   -4.35   -2.8591   -2.8409     -1.45

 Kc = 100.0000
 Roots =  -9.851    -2.248   0.2995+5.701i   0.2995-5.701i

 Kc = 50.0000
 Roots = -8.4949   -2.2459   -0.3796+4.485i   -0.3796-4.485i

 Kc = 75.0000
 Roots = -9.2487   -2.2473   -0.001993+5.163i   -0.001993-5.163i

 Kc = 87.5000
 Roots = -9.5641   -2.2477   0.1559+5.445i   0.1559-5.445i

 Kc = 81.2500
 Roots = -9.4104   -2.2475   0.07893+5.308i   0.07893-5.308i

 Kc = 78.1250
 Roots = -9.3306   -2.2474    0.039+5.237i    0.039-5.237i

 Kc = 76.5625
 Roots =   -9.29   -2.2473   0.01864+5.2i   0.01864-5.2i

 Kc = 75.7812
 Roots = -9.2694   -2.2473   0.00836+5.182i   0.00836-5.182i

 Kc = 75.3906
 Roots = -9.2591   -2.2473   0.003192+5.173i   0.003192-5.173i

 Kc = 75.1953
 Roots = -9.2539   -2.2473   0.0006016+5.168i   0.0006016-5.168i

 Kc = 75.0977
 Roots = -9.2513   -2.2473   -0.0006953+5.166i   -0.0006953-5.166i

 Kc = 75.1465
 Roots = -9.2526   -2.2473   -4.667e-005+5.167i   -4.667e-005-5.167i
```

```
Kc = 75.1709
Roots = -9.2533   -2.2473   0.0002775+5.167i   0.0002775-5.167i

Kc = 75.1587
Roots = -9.2529   -2.2473   0.0001154+5.167i   0.0001154-5.167i

Kc = 75.1526
Roots = -9.2528   -2.2473   3.438e-005+5.167i   3.438e-005-5.167i

Kc = 75.1495
Roots = -9.2527   -2.2473   -6.147e-006+5.167i   -6.147e-006-5.167i

Kc = 75.1511
Roots = -9.2527   -2.2473   1.412e-005+5.167i   1.412e-005-5.167i

Kc = 75.1503
Roots = -9.2527   -2.2473   3.985e-006+5.167i   3.985e-006-5.167i
```

Discussion of Results: The range of search for the proportional gain (K_C) is chosen to be between 0 and 100. A convergence criterion of 0.001 is used and may be changed by the user if necessary. The bisection method evaluates the roots at the low end of the range ($K_C = 0$) and finds them to have the predicted values of

$$-4.3500 \quad -2.8591 \quad -2.8409 \quad \text{and} \quad -1.4500$$

The small difference between the two middle roots and their actual values is due to rounding off the coefficients of the denominator polynomial. This deviation is very small in comparison with the root itself and it can be ignored. At the upper end of the range ($K_C = 100$) the roots are

$$-9.8510 \quad -2.2480 \quad \text{and} \quad 0.2995 \pm 5.7011i$$

The system is unstable because of the positive real components of the roots. At the midrange ($K_C = 50$) the system is still stable because all the real parts of the roots are negative. The bisection method continues its search in the range 50-100. In a total of 19 evaluations, the algorithm arrives at the critical value of K_C in the range

$$75.1495 < K_C < 75.1503$$

In the event that the critical value of the gain was outside the limits of the original range of search, the program would have detected this early in the search and would have issued a warning and stopped running.

1.9 NEWTON'S METHOD FOR SIMULTANEOUS NONLINEAR EQUATIONS

If the mathematical model involves two (or more) simultaneous nonlinear equations in two (or more) unknowns, the Newton-Raphson method can be extended to solve these equations simultaneously. In what follows, we will first develop the Newton-Raphson method for two equations and then expand the algorithm to a system of k equations.

The model for two unknowns will have the general form

$$\begin{aligned} f_1(x_1, x_2) &= 0 \\ f_2(x_1, x_2) &= 0 \end{aligned} \tag{1.57}$$

where f_1 and f_2 are nonlinear functions of variables x_1 and x_2. Both these functions may be expanded in two-dimensional Taylor series around an initial estimate of $x_1^{(1)}$ and $x_2^{(1)}$:

$$\begin{aligned} f_1(x_1, x_2) &= f_1(x_1^{(1)}, x_2^{(1)}) + \frac{\partial f_1}{\partial x_1}\Big|_{x^{(1)}}(x_1 - x_1^{(1)}) + \frac{\partial f_1}{\partial x_2}\Big|_{x^{(1)}}(x_2 - x_2^{(1)}) + \ldots \\ f_2(x_1, x_2) &= f_2(x_1^{(1)}, x_2^{(1)}) + \frac{\partial f_2}{\partial x_1}\Big|_{x^{(1)}}(x_1 - x_1^{(1)}) + \frac{\partial f_2}{\partial x_2}\Big|_{x^{(1)}}(x_2 - x_2^{(1)}) + \ldots \end{aligned} \tag{1.58}$$

The superscript (1) will be used to designate the iteration number of the estimate.

Setting the left sides of Eqs. (1.58) to zero and truncating the second-order and higher derivatives of the Taylor series, we obtain the following equations:

$$\begin{aligned} \frac{\partial f_1}{\partial x_1}\Big|_{x^{(1)}}(x_1 - x_1^{(1)}) + \frac{\partial f_1}{\partial x_2}\Big|_{x^{(1)}}(x_2 - x_2^{(1)}) &= -f_1(x_1^{(1)}, x_2^{(1)}) \\ \frac{\partial f_2}{\partial x_1}\Big|_{x^{(1)}}(x_1 - x_1^{(1)}) + \frac{\partial f_2}{\partial x_2}\Big|_{x^{(1)}}(x_2 - x_2^{(1)}) &= -f_2(x_1^{(1)}, x_2^{(1)}) \end{aligned} \tag{1.59}$$

If we define the correction variable δ as

$$\begin{aligned} \delta_1^{(1)} &= x_1 - x_1^{(1)} \\ \delta_2^{(1)} &= x_2 - x_2^{(1)} \end{aligned} \tag{1.60}$$

then Eqs. (1.59) simplify to

$$\begin{aligned} \frac{\partial f_1}{\partial x_1}\Big|_{x^{(1)}} \delta_1^{(1)} + \frac{\partial f_1}{\partial x_2}\Big|_{x^{(1)}} \delta_2^{(1)} &= -f_1(x_1^{(1)}, x_2^{(1)}) \\ \frac{\partial f_2}{\partial x_1}\Big|_{x^{(1)}} \delta_1^{(1)} + \frac{\partial f_2}{\partial x_2}\Big|_{x^{(1)}} \delta_2^{(1)} &= -f_2(x_1^{(1)}, x_2^{(1)}) \end{aligned} \tag{1.61}$$

Eqs. (1.61) are a set of simultaneous linear algebraic equations, where the unknowns are $\delta_1^{(1)}$ and $\delta_2^{(1)}$. These equations can be written in matrix format as follows:

$$\begin{bmatrix} \dfrac{\partial f_1}{\partial x_1}\Big|_{x^{(1)}} & \dfrac{\partial f_1}{\partial x_2}\Big|_{x^{(1)}} \\ \dfrac{\partial f_2}{\partial x_1}\Big|_{x^{(1)}} & \dfrac{\partial f_2}{\partial x_2}\Big|_{x^{(1)}} \end{bmatrix} \begin{bmatrix} \delta_1^{(1)} \\ \\ \delta_2^{(1)} \end{bmatrix} = - \begin{bmatrix} f_1^{(1)} \\ \\ f_2^{(1)} \end{bmatrix} \tag{1.62}$$

Because this set contains only two equations in two unknowns, it can be readily solved by the application of Cramer's rule (see Chap. 2) to give the first set of values for the correction vector:

$$\begin{aligned} \delta_1^{(1)} &= -\frac{\left[f_1 \dfrac{\partial f_2}{\partial x_2} - f_2 \dfrac{\partial f_1}{\partial x_2} \right]}{\left[\dfrac{\partial f_1}{\partial x_1} \dfrac{\partial f_2}{\partial x_2} - \dfrac{\partial f_1}{\partial x_2} \dfrac{\partial f_2}{\partial x_1} \right]} \\ \delta_2^{(1)} &= -\frac{\left[f_2 \dfrac{\partial f_1}{\partial x_1} - f_1 \dfrac{\partial f_2}{\partial x_1} \right]}{\left[\dfrac{\partial f_1}{\partial x_1} \dfrac{\partial f_2}{\partial x_2} - \dfrac{\partial f_1}{\partial x_2} \dfrac{\partial f_2}{\partial x_1} \right]} \end{aligned} \tag{1.63}$$

The superscripts, indicating the iteration number of the estimate, have been omitted from the right-hand side of Eqs. (1.63) in order to avoid overcrowding.

The new estimate of the solution may now be obtained from the previous estimate by adding to it the correction vector:

$$x_i^{(n+1)} = x_i^{(n)} + \delta_i^{(n)} \tag{1.64}$$

This equation is merely a rearrangement and generalization to the $(n + 1)$st iteration of Eqs. (1.60).

The method just described for two nonlinear equations is readily expandable to the case of k simultaneous nonlinear equations in k unknowns:

$$\begin{aligned} f_1(x_1, \ldots, x_k) &= 0 \\ &\vdots \\ f_k(x_1, \ldots, x_k) &= 0 \end{aligned} \tag{1.65}$$

The linearization of this set by the application of the Taylor series expansion produces Eq. (1.66).

$$\begin{bmatrix} \dfrac{\partial f_1}{\partial x_1} & \cdots & \dfrac{\partial f_1}{\partial x_k} \\ & \cdots\cdots & \\ \dfrac{\partial f_k}{\partial x_1} & \cdots & \dfrac{\partial f_k}{\partial x_k} \end{bmatrix} \begin{bmatrix} \delta_1 \\ \vdots \\ \delta_k \end{bmatrix} = - \begin{bmatrix} f_1 \\ \vdots \\ f_k \end{bmatrix} \tag{1.66}$$

In matrix/vector notation this condenses to

$$\boldsymbol{J}\,\boldsymbol{\delta} = -\boldsymbol{f} \tag{1.67}$$

where $\boldsymbol{J}$ is the *Jacobian* matrix containing the partial derivatives, $\boldsymbol{\delta}$ is the correction vector, and $\boldsymbol{f}$ is the vector of functions. Eq. (1.67) represents a set of linear algebraic equations whose solution will be discussed in Chap. 2.

Strongly nonlinear equations are likely to diverge rapidly. To prevent this situation, relaxation is used to stabilize the iterative solution process. If $\boldsymbol{\delta}$ is the correction vector without relaxation, then relaxed change is $\rho\boldsymbol{\delta}$ where ρ is the relaxation factor:

$$\boldsymbol{x}^{(n+1)} = \boldsymbol{x}^{(n)} + \rho\,\boldsymbol{\delta} \tag{1.68}$$

A typical value for ρ is 0.5. A value of zero inhibits changes and a value of one is equivalent to no relaxation. Relaxation reduces the correction made to the variable from one iteration to the next and may eliminate the tendency of the solution to diverge.

Example 1.4: Solution of Nonlinear Equations in Chemical Equilibrium Using Newton's Method for Simultaneous Nonlinear Equations. Develop a MATLAB function

to solve *n* simultaneous nonlinear equations in *n* unknowns. Apply this function to find the equilibrium conversion of the following reactions:

$$2A + B \leftrightarrows C \qquad K_1 = \frac{C_C}{C_A^2 C_B} = 5\times10^{-4}$$

$$A + D \leftrightarrows C \qquad K_2 = \frac{C_C}{C_A C_D} = 4\times10^{-2}$$

Initial concentrations are

$$C_{A,0} = 40 \qquad C_{B,0} = 15 \qquad C_{C,0} = 0 \qquad C_{D,0} = 10$$

All concentrations are in kmol/m^3.

Method of Solution: Eq. (1.67) is applied to calculate the correction vector in each iteration and Eq. (1.68) is used to estimate the new relaxed variables. The built-in MATLAB function *inv* is used to invert the Jacobian matrix.

The variables of this problem are the conversions, x_1 and x_2, of the above reactions. The concentrations of the components can be calculated from these conversions and the initial concentrations

$$C_A = C_{A,0} - 2x_1 C_{B,0} - x_2 C_{D,0} = 40 - 30x_1 - 10x_2$$

$$C_B = (1 - x_1) C_{B,0} = 15 - 15x_1$$

$$C_C = C_{C,0} + x_1 C_{B,0} + x_2 C_{D,0} = 15x_1 + 10x_2$$

$$C_D = (1 - x_2) C_{D,0} = 10 - 10x_2$$

The set of equations that are functions of x_1 and x_2 are

$$f_1(x_1, x_2) = \frac{C_C}{C_A^2 C_B} - 5\times10^{-4} = 0$$

$$f_2(x_1, x_2) = \frac{C_C}{C_A C_D} - 4\times10^{-2} = 0$$

The values of x_1 and x_2 are to be calculated by the program so that $f_1 = f_2 = 0$.

Program Description: The MATLAB function *Newton.m* solves a set of nonlinear equations by Newton's method. The first part of the program is initialization in which the convergence criterion, the relaxation factor, and other initial parameters needed by the program are being set. The main iteration loop comes next. In this part, the components of

the Jacobian matrix are calculated by numeric differentiation [Eq. (1.66)], and new estimates of the roots are calculated according to Eq. (1.68). This procedure continues until the convergence criterion is met or a maximum iteration limit is reached. The convergence criterion is $max(|x_i^{(n)} - x_i^{(n-1)}|) < \epsilon$.

The MATLAB program *Example1_4.m* is written to solve the particular problem of this example. This program simply takes the required input data from the user and calls *Newton.m* to solve the set of equations. The program allows the user to repeat the calculation and try new initial values and relaxation factor without changing the problem parameters.

The set of equations of this example are introduced in the MATLAB function *Ex1_4_func.m*. It is important to note that the function *Newton.m* should receive the function values at each point as a column vector. This is considered in the function *Ex1_4_func*.

Program

Example1_4.m

```
% Example1_4.m
% Solution to the problem posed in Example 1.4. It calculates the
% equilibrium concentration of the components of a system of two
% reversible chemical reactions using the Newton's method.

clear
clc

% Input data
c0 = input(' Vector of initial concentration of A, B, C, and D = ');
K1 = input(' 2A + B = C      K1 = ');
K2 = input('  A + D = C      K2 = ');
fname = input(' Name of the file containing the set of equations =
');

repeat = 1;
while repeat
   % Input initial values and relaxation factor
   x0 = input('\n\n Vector of initial guesses = ');
   rho = input(' Relaxation factor = ');

   % Solution of the set of equations
   [x,iter] = Newton(fname,x0,rho,[],c0,K1,K2);

   % Display the results
   fprintf('\n Results :\n x1 = %6.4f  ,  x2 = %6.4f',x)
   fprintf('\n Solution reached after %3d iterations.\n\n',iter)
   repeat = input(' Repeat the calculations (0 / 1) ? ');
end
```

Newton.m

```
function [xnew , iter] = Newton(fnctn , x0 , rho , tol , varargin)
%NEWTON   Solves a set of equations by Newton's method.
%
%   NEWTON('F',X0) finds a zero of the set of equations
%   described by the M-file F.M. X0 is a vector of starting
%   guesses.
%
%   NEWTON('F',X0,RHO,TOL) uses relaxation factor RHO and
%   tolerance TOL for convergence test.
%
%   NEWTON('F',X0,RHO,TOL,P1,P2,...) allows for additional
%   arguments which are passed to the function F(X,P1,P2,...).
%   Pass an empty matrix for TOL or TRACE to use the default
%   value.

% (c) by N. Mostoufi & A. Constantinides
% January 1, 1999

% Initialization
if nargin < 4 | isempty(tol)
   tol = 1e-6;
end
if nargin < 3 | isempty(rho)
   rho = 1;
end

x0 = (x0(:).')'; % Make sure it's a column vector
nx = length(x0);
x = x0 * 1.1;
xnew = x0;
iter = 0;
maxiter = 100;

% Main iteration loop
while max(abs(x - xnew)) > tol & iter < maxiter
   iter = iter + 1;
   x = xnew;
   fnk = feval(fnctn,x,varargin{:});

   % Set dx for derivation
   for k = 1:nx
      if x(k) ~= 0
         dx(k) = x(k) / 100;
      else
         dx(k) = 1 / 100;
      end
   end
```

```
   % Calculation of the Jacobian matrix
   a = x;
   b = x;
   for k = 1 : nx
      a(k) = a(k) - dx(k);  fa = feval(fnctn,a,varargin{:});
      b(k) = b(k) + dx(k);  fb = feval(fnctn,b,varargin{:});
      jacob(:,k) = (fb - fa) / (b(k) - a(k));
      a(k) = a(k) + dx(k);
      b(k) = b(k) - dx(k);
   end

   % Next approximation of the roots
   if det(jacob) == 0
      xnew = x + max([abs(dx), 1.1*tol]);
   else
      xnew = x - rho * inv(jacob) * fnk;
   end
end

if iter >= maxiter
   disp('Warning : Maximum iterations reached.')
end
```

Ex1_4_func.m

```
function f = Ex1_4_func(x,c0,K1,K2)
% Evaluation of set of equations for example 1-4.
% c0(1) = ca0 / c0(2) = cb0 / c0(3) = cc0 / c0(4) = cd0

ca = c0(1) - 2*x(1)*c0(2) - x(2)*c0(4);
cb = (1 - x(1))*c0(2);
cc = c0(3) + x(1)*c0(2) + x(2)*c0(4);
cd = (1 - x(2))*c0(4);

f(1) = cc / ca^2 / cb - K1;
f(2) = cc / ca / cd - K2;
f = f';   % Make it a column vector.
```

Input and Results

```
>>Example1_4

Vector of initial concentration of A, B, C, and D = [40, 15, 0, 10]
 2A + B = C      K1 = 5e-4
  A + D = C      K2 = 4e-2
 Name of the file containing the set of equations = 'Ex1_4_func'

 Vector of initial guesses = [0.1, 0.9]
 Relaxation factor = 1
```

```
 Results :
 x1 = 0.1203  ,  x2 = 0.4787
 Solution reached after   8 iterations.

 Repeat the calculations (0 / 1) ? 1


 Vector of initial guesses = [0.1, 0.1]
 Relaxation factor = 1
 Warning : Maximum iterations reached.

 Results :
 x1 = 32129631755678440000000000.0000  ,  x2 =
-69732070564209780000000000.0000
 Solution reached after 100 iterations.

 Repeat the calculations (0 / 1) ? 1


 Vector of initial guesses = [0.1, 0.1]
 Relaxation factor = 0.7

 Results :
 x1 = 0.1203  ,  x2 = 0.4787
 Solution reached after  18 iterations.

 Repeat the calculations (0 / 1) ? 0
```

Discussion of Results: Three runs are made to test the sensitivity in the choice of initial guesses and the effectiveness of the relaxation factor. In the first run, initial guesses are set to $x_1^{(0)} = 0.1$ and $x_2^{(0)} = 0.9$. With these guesses, the method converges in 8 iterations. By default, the convergence criterion is $max(|\Delta x_i|) < 10^{-6}$. This value may be changed through the fourth input argument of the function *Newton*.

In the third run, the initial guesses are set to $x_1^{(0)} = 0.1$ and $x_2^{(0)} = 0.1$. The maximum number of iterations defined in the function *Newton* is 100, and the method does not converge in this case, even in 100 iterations. This test shows high sensitivity of Newton's method to the initial guess. Introducing the relaxation factor $\rho = 0.7$ in the next run causes the method to converge in only 18 iterations. This improvement in the speed of convergence was obtained by using a fixed relaxation factor. A more effective way of doing this would be to adjust the relaxation factor from iteration to iteration. This is left as an exercise for the reader.

PROBLEMS

1.1 Evaluate all the roots of the polynomial equations, (a)-(g) given below, by performing the following steps:

(i) Use Descartes' rule to predict how many positive and how many negative roots each polynomial may have.

(ii) Use the Newton-Raphson method with synthetic division to calculate the numerical values of the roots. To do so, first apply the MATLAB function *roots.m* and then the *NRsdivision.m*, which was developed in this chapter. Why does the *NRsdivision.m* program fail to arrive at the answers of some of these polynomials? What is the limitation of this program?

(iii) Classify these polynomials according to the four categories described in Sec. 1.2.

(a) $x^4 - 16x^3 + 96x^2 - 256x + 256 = 0$
(b) $x^4 - 32x^2 + 256 = 0$
(c) $x^4 + 3x^3 + 12x - 16 = 0$
(d) $x^4 + 4x^3 + 18x^2 - 20x + 125 = 0$
(e) $x^5 - 8x^4 + 35x^3 - 106x^2 + 170x - 200 = 0$
(f) $x^4 - 10x^3 + 35x^2 - 5x + 24 = 0$
(g) $x^6 - 8x^5 + 11x^4 + 78x^3 - 382x^2 + 800x - 800 = 0$

1.2 Evaluate roots of the following transcendental equations.

(a) $\sin x - 2\exp(-x^2) = 0$
(b) $ax - a^x = 0$ for $a = 2$, e, or 3
(c) $\ln(1 + x^2) - \sqrt{|x|} = 0$
(d) $e^{-x}/(1 + \cos x) - 1 = 0$

1.3 Repeat Example 1.2 by using the Benedict-Webb-Rubin (BWR) and the Patel-Teja (PT) equations of state. Compare the results with those obtained in Example 1.2.

Benedict-Webb-Rubin equation of state:

$$P = \frac{RT}{V} + \frac{B_0RT - A_0 - (C_0/T^2)}{V^2} + \frac{bRT - a}{V^3} + \frac{a\alpha}{V^6} + \frac{c}{V^3T^2}\left(1 + \frac{\gamma}{V^2}\right)e^{-\gamma/V^2}$$

where A_0, B_0, C_0, a, b, c, α, and γ are constants. When P is in atmosphere, V is in liters per mole, and T is in Kelvin, the values of constants for *n*-butane are:

$A_0 = 10.0847$	$B_0 = 0.124361$	$C_0 = 0.992830\times10^6$
$a = 1.88231$	$b = 0.0399983$	$c = 0.316400\times10^6$
$\alpha = 1.10132\times10^{-3}$	$\gamma = 3.400\times10^{-2}$	$R = 0.08206$

Patel-Teja equation of state:

$$P = \frac{RT}{V - b} - \frac{a}{V(V + b) + c(V - b)}$$

where a is a function of temperature, and b and c are constants

$$a = \Omega_a(R^2T_c^2/P_c)\left[1 + F(1 - \sqrt{T_R})\right]^2$$

$$b = \Omega_b(RT_c/P_c)$$

$$c = \Omega_c(RT_c/P_c)$$

where

$$\Omega_c = 1 - 3\zeta_c$$

$$\Omega_a = 3\zeta_c^2 + 3(1 - 2\zeta_c)\Omega_b + \Omega_b^2 + 1 - 3\zeta_c$$

and Ω_b is the smallest positive root of the cubic

$$\Omega_b^3 + (2 - 3\zeta_c)\Omega_b^2 + 3\zeta_c^2\Omega_b - \zeta_c^3 = 0$$

F and ζ_c are functions of the acentric factor given by the following quadratic correlations

$$F = 0.452413 + 1.30982\,\omega - 0.295937\,\omega^2$$

$$\zeta_c = 0.329032 - 0.076799\,\omega + 0.0211947\,\omega^2$$

Use the data given in Example 1.2 for *n*-butane to calculate the parameters of PT equation.

1.4 F moles per hour of an n-component natural gas stream is introduced as feed to the flash vaporization tank shown in Fig. P1.4. The resulting vapor and liquid streams are withdrawn at the rate of V and L moles per hour, respectively. The mole fractions of the components in the feed, vapor, and liquid are designated by z_i, y_i, and x_i, respectively (i = 1, 2, . . ., n). Assuming vapor-liquid equilibrium and steady-state operation, we have:

Overall balance	$F = L + V$	
Individual component balances	$z_iF = x_iL + y_iV$	$i = 1, 2, \ldots, n$
Equilibrium relations	$K_i = y_i/x_i$	$i = 1, 2, \ldots, n$

Here, K_i is the equilibrium constant for the ith component at the prevailing temperature and pressure in the tank. From these equations and the fact that

$$\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i = 1$$

derive the following equation:

$$\sum_{i=1}^{n} \frac{z_i K_i F}{V(K_i - 1) + F} = 1$$

Using the data given in Table P1.4, solve the above equation for V. Also calculate the values of L, the x_i, and the y_i by using the first three equations given above. The test data in Table P1.4 relates to flashing of a natural gas stream at 11 MPa and 48°C. Assume that F = 100 mol/h.

What would be a good value V_0 for starting the iteration? Base this answer on your observations of the data given in Table P1.4.

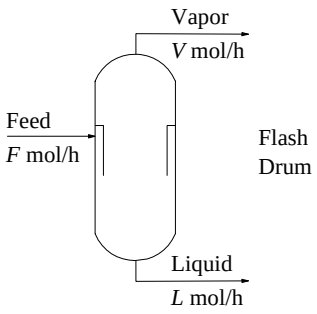

Figure P1.4 Flash drum.

Table P1.4

Component	*i*	z_i	K_i
Methane	1	0.8345	3.090
Carbon dioxide	2	0.0046	1.650
Ethane	3	0.0381	0.720
Propane	4	0.0163	0.390
i-Butane	5	0.0050	0.210
n-Butane	6	0.0074	0.175
Pentanes	7	0.0287	0.093
Hexanes	8	0.0220	0.065
Heptanes+	9	0.0434	0.036
		1.0000	

1.5 The Underwood equation for multicomponent distillation is given as

$$\left(\sum_{j=1}^{n} \frac{\alpha_j z_{jF} F}{\alpha_j - \phi} \right) - F(1 - q) = 0$$

where F = molar feed flow rate
n = number of components in the feed
z_{jF} = mole fraction of each component in the feed
q = quality of the feed
α_j = relative volatility of each component at average column conditions
ϕ = root of the equation

It has been shown by Underwood that (n - 1) of the roots of this equation lie between the values of the relative volatilities as shown below:

$$\alpha_n < \phi_{n-1} < \alpha_{n-1} < \phi_{n-2} < \ldots < \alpha_3 < \phi_2 < \alpha_2 < \phi_1 < \alpha_1$$

Evaluate the (n - 1) roots of this equation for the case shown in Table P1.5.

Table P1.5

Component in feed	Mole fraction, z_{jF}	Relative volatility, α_j
C_1	0.05	10.00
C_2	0.05	5.00
C_3	0.10	2.05
C_4	0.30	2.00
C_5	0.05	1.50
C_6	0.30	1.00
C_7	0.10	0.90
C_8	0.05	0.10
	1.00	

F = 100 mol/h q = 1.0 (saturated liquid)

1.6 Carbon monoxide from a water gas plant is burned with air in an adiabatic reactor. Both the carbon monoxide and air are being fed to the reactor at 25°C and atmospheric pressure. For the reaction:

$$CO + \frac{1}{2}O_2 \leftrightarrows CO_2$$

the following standard free energy change (at 25°C) has been determined:

$$\Delta G^0_{T_0} = -257\ kJ/(g\ mol\ of\ CO)$$

The standard enthalpy change at 25°C has been measured as

$$\Delta H^0_{T_0} = -283\ kJ/(g\ mol\ of\ CO)$$

The standard states for all components are the pure gases at 1 atm.

Calculate the adiabatic flame temperature and the conversion of CO for the following two cases:

(a) 0.4 mole of oxygen per mole of CO is provided for the reaction.

(b) 0.8 mole of oxygen per mole of CO is provided for the reaction.

The constant pressure heat capacities for the various constituents in J/(gmol.K) with T in Kelvin are all of the form

$$C_{p_i} = A_i + B_i T_K + C_i T_K^2$$

For the gases involved here, the constants are as shown in Table P1.6.

Table P1.6

Gas	A	B	C
CO	26.16	8.75×10^{-3}	-1.92×10^{-6}
O_2	25.66	12.52×10^{-3}	-3.37×10^{-6}
CO_2	28.67	35.72×10^{-3}	-10.39×10^{-6}
N_2	26.37	7.61×10^{-3}	-1.44×10^{-6}

Hint: Combine the material balance, enthalpy balance, and equilibrium relationship to form two nonlinear algebraic equations in two unknowns: the temperature and conversion.

1.7 Consider the three-mode feedback control of a stirred-tank heater system (Fig. P1.7).

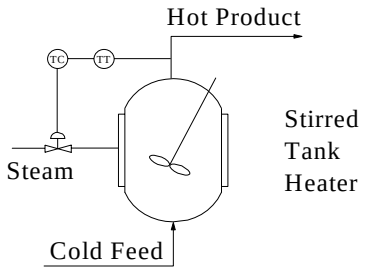

Figure P1.7 Stirred tank heater.

The measured output variable is the feedstream temperature [7]. Using classical methods (i.e., deviation variables, linearization, and Laplace transforms) the overall closed-loop transfer function for the control system is given by

$$\frac{\bar{T}}{\bar{T}_i} = \frac{(\tau_I s)(\tau_v s + 1)(\tau_m s + 1)}{(\tau_I s)(\tau_P s + 1)(\tau_v s + 1)(\tau_m s + 1) + K(\tau_I s + 1 + \tau_D \tau_I s^2)}$$

where τ_I = reset time constant
τ_D = derivative time constant
$K = K_P K_v K_m K_c$
K_P= first-order process static gain
K_v = first-order valve constant
K_m= first-order measurement constant
K_c= proportional gain for the three-mode controller
$\bar{T}$ = Laplace transform of the output temperature deviation
$\bar{T}_i$ = Laplace transform of the input load temperature deviation
τ_P, τ_m, τ_v = first-order time constants for the process, measurement device, and process valve, respectively.

For a given set of values, the stability of the system can be determined from the roots of the characteristic polynomial (i.e, the polynomial in the denominator of the overall transfer function). Thus:

$$\tau_I \tau_P \tau_m \tau_v s^4 + (\tau_I \tau_P \tau_m + \tau_I \tau_P \tau_v + \tau_I \tau_m \tau_v) s^3 + (K\tau_I \tau_D + \tau_I \tau_P + \tau_I \tau_v + \tau_I \tau_m) s^2 + (\tau_I + K\tau_I) s + K = 0$$

For the following set of parameter values, find the four roots to the characteristic polynomial when K_c is equal to its "critical" value:

$\tau_I = 10$	$\tau_D = 1$	$\tau_P = 10$	$\tau_m = 5$	$\tau_v = 5$
$K_P = 10$	$K_v = 2$	$K_m = 0.09$	$K = 1.8K_c$	

1.8 In the analytical solution of some parabolic partial differential equations in cylindrical coordinates, it is necessary to calculate roots of the Bessel function first (for example, see Problem 6.11). Find the first N root of the first and the second kind. Use the following approximations for evaluating the initial guesses:

$$J_n(x) \simeq \sqrt{\frac{2}{\pi x}} \cos\left(x - \frac{n\pi}{2} - \frac{\pi}{4}\right)$$

$$Y_n(x) \simeq \sqrt{\frac{2}{\pi x}} \sin\left(x - \frac{n\pi}{2} - \frac{\pi}{4}\right)$$

Which method of root finding do you recommend?

1.9 A direct-fired tubular reactor is used in the thermal cracking of light hydrocarbons or naphthas for the production of olefins, such as ethylene (see Fig. P1.9). The reactants are preheated in the convection section of the furnace, mixed with steam, and then subjected to high temperatures in the radiant section of the furnace. Heat transfer in the radiant section of the furnace takes place through three mechanisms: radiation, conduction, and convection. Heat is transferred by radiation from the walls of the furnace to the surface of the tubes that carry the reactants, and it is transferred through the walls of the tubes by conduction and finally to the fluid inside the tubes by convection [8].

The three heat-transfer mechanisms are quantified as follows:

1. *Radiation*: The Stefan-Boltzmann law of radiation may be written as

$$\frac{dQ}{dA_o} = \sigma\phi(T_R^4 - T_o^4)$$

where dQ/dA_o is the rate for heat transfer per unit outside surface area of the tubes, T_R is the "effective" furnace radiation temperature, and T_o is the temperature on the outside surface of the tube. In furnaces with tube banks irradiated from both sides, a reasonable approximation is

$$T_R = T_G$$

where T_G is the temperature of the flue gas in the reactor. Therefore, the Stefan-Boltzmann equation is revised to

$$\frac{dQ}{dA_o} = \sigma\phi(T_G^4 - T_o^4)$$

σ is the Stefan-Boltzmann constant and ϕ is the tube geometry emissivity factor, which depends on the tube arrangement and tube surface emissivity. For single rows of tubes irradiated from both sides:

$$\frac{1}{\phi} = \frac{1}{\epsilon} - 1 + \frac{\pi}{2\Omega}$$

$$\Omega = \frac{S}{D_o} + \arctan\sqrt{\left(\frac{S}{D_o}\right)^2 - 1} - \sqrt{\left(\frac{S}{D_o}\right)^2 - 1}$$

where ϵ is the emissivity of the outside surface of the tube and S is the spacing (pitch) of the tubes (center-to-center) and D_o is the outside diameter of the tubes.

2. *Conduction*: Conduction through the tube wall is given by Fourier's equation:

$$\frac{dQ}{dA_o} = \frac{k_t}{t_t}(T_o - T_i)$$

where T_i is the temperature on the inside surface of the tube, k_t is the thermal conductivity of the tube material, and t_t is the thickness of the tube wall.

3. *Convection*: Convection through the fluid film inside the tube is expressed by

$$\frac{dQ}{dA_o} = h_i\left(\frac{D_i}{D_o}\right)(T_i - T_f)$$

where D_i is the inside diameter of the tube, T_f is the temperature of the fluid in the tube, and h_i is the heat-transfer film coefficient on the inside of the tube. The film coefficient may be approximated from the Dittus-Boelter equation [9]:

$$h_i = 0.023\left(\frac{k_f}{D_i}\right)Re_f^{0.8}Pr_f^{0.4}$$

where Re_f is the Reynolds number, Pr_f is the Prandtl number, and k_f is the thermal conductivity of the fluid.

Conditions vary drastically along the length of the tube, as the temperature of the fluid inside the tube rises rapidly. The rate of heat transfer is the highest at the entrance conditions and lowest at the exit conditions of the fluid.

Calculate the rate of heat transfer (dQ/dA_o), the temperature on the outside surface of the tube (T_o), and the temperature on the inside surface of the tube (T_i) at a point along the length of the tube where the following conditions exist:

T_G = 1200°C	T_f = 800°C	ϵ = 0.9	$\sigma = 5.7\times10^{-8}$ W/m^2.K^4
S = 0.20 m	D_i = 0.10 m	D_o = 0.11 m	t_t = 0.006 m
Re_f = 388,000	Pr_f = 0.660	k_t = 21.6 W/m.K	k_f = 0.175 W/m.K

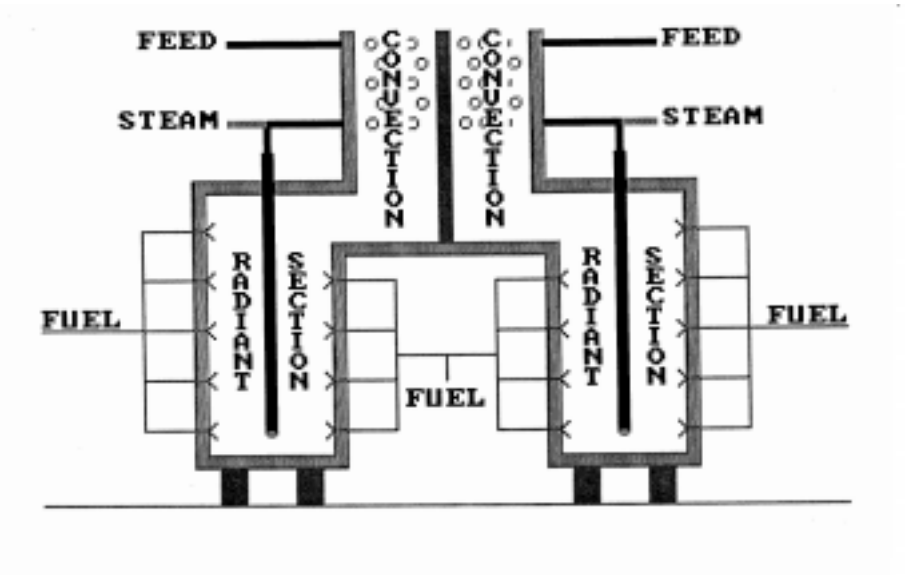

Figure P1.9 Pyrolysis reactor.

1.10 The elementary reaction A → B + C is carried out in a continuous stirred tank reactor (CSTR). Pure A enters the reactor at a flow rate of 12 mol/s and a temperature of 25°C. The reaction is exothermic and cooling water at 50°C is used to absorb the heat generated. The energy balance for this system, assuming constant heat capacity and equal heat capacity of both sides of the reaction, can be written as

$$-F_{A_0} X \Delta H_R = F_{A_0} C_{p_A} (T - T_0) + UA(T - T_a)$$

where F_{A_0} = molar flow rate, mol/s
X = conversion
ΔH_R = heat of reaction, J/mol A
C_{p_A} = heat capacity of A, J/mol.K
T = reactor temperature, °C
T_0 = reference temperature, 25°C
T_a = cooling water temperature, 20°C
U = overall heat transfer coefficient, W/m^2.K
A = heat transfer area, m^2

For a first-order reaction the conversion can be calculated from

$$X = \frac{\tau k}{1 + \tau k}$$

where τ is the residence time of the reactor in seconds and k is the specific reaction rate in s^{-1} defined by the Arrhenius formula:

$$k = 650 \exp[-3800/(T + 273)]$$

Solve the energy balance equation for temperature and find the steady-state operating temperatures of the reactor and the conversions corresponding to these temperatures. Additional data are:

ΔH_R = -1500 kJ/mol τ = 10 s C_{p_A} = 4500 J/mol.K UA/F_{A_0} = 700 W.s/mol.K

REFERENCES

1. Underwood, A. J. V., *Chem. Eng. Prog.*, vol. 44, 1948, p. 603.

2. Treybal, R. E., *Mass Transfer Operations*, 3rd ed., McGraw-Hill, New York, 1980.

3. Salvadori, M. G., and Baron, M. L., *Numerical Methods in Engineering*, Prentice Hall, Englewood Cliffs, NJ, 1961.

4. Lapidus, L., *Digital Computation for Chemical Engineering*, McGraw-Hill, New York, 1962.

5. Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P., *Numerical Recipes in FORTRAN*, 2nd ed., Cambridge University Press, Cambridge, U.K., 1992.

6. Douglas, J. M., *Process Dynamics and Control*, vol. 2, Prentice Hall, Englewood Cliffs, NJ, 1972.

7. Davidson, B. D., private communication, Department of Chemical and Biochemical Engineering, Rutgers University, Piscataway, NJ, 1984.

8. Constantinides, A., *Applied Numerical Methods with Personal Computers*, McGraw-Hill, New York, 1987.

9. Bennett, C. O., and Meyers, J. E., *Momentum, Heat, and Mass Transfer*, McGraw-Hill, New York, 1973.