Chapter 3

Solving Nonlinear Equations

Core Topics

Estimation of errors in numerical solutions (3.2).

Bisection method (3.3).

Regula falsi method (3.4).

Newton's method (3.5).

Secant method (3.6).

Fixed-point iteration method (3.7).

Use of MATLAB built-in Functions for solving nonlinear equations (3.8).

Complementary Topics

Equations with multiple roots (3.9).

System of nonlinear equations (3.10).

3.1 BACKGROUND

Equations need to be solved in all areas of science and engineering. An equation of one variable can be written in the form:

$$f(x) = 0 \tag{3.1}$$

A solution to the equation (also called a ***root*** of the equation) is a numerical value of x that satisfies the equation. Graphically, as shown in Fig. 3-1, the solution is the point where the function $f(x)$ crosses or touches the x-axis. An equation might have no solution or can have one or several (possibly many) roots.

When the equation is simple, the value of x can be determined analytically. This is the case when x can be written explicitly by applying mathematical operations, or when a known formula (such as the for-

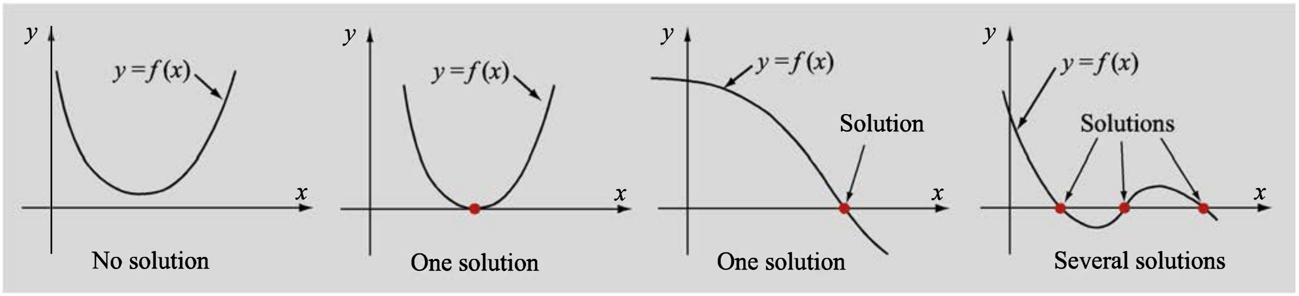


Figure 3-1: Illustration of equations with no, one, or several solutions.

mula for solving a quadratic equation) can be used to determine the exact value of x. In many situations, however, it is impossible to determine the root of an equation analytically. For example, the area of a segment A_S of a circle with radius r (shaded area in Fig. 3-2) is given by:

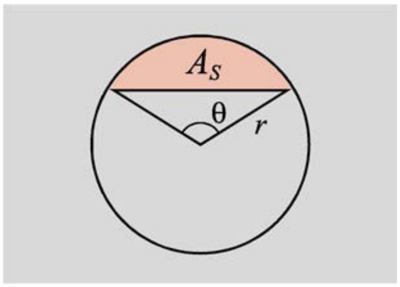


Figure 3-2: Segment of a circle.

$$A_S = \frac{1}{2}r^2(\theta - \sin\theta) \tag{3.2}$$

To determine the angle θ if A_S and r are given, Eq. (3.2) has to be solved for θ. Obviously, θ cannot be written explicitly in terms of A_S and r, and the equation cannot be solved analytically.

A numerical solution of an equation $f(x) = 0$ is a value of x that satisfies the equation approximately. This means that when x is substituted in the equation, the value of $f(x)$ is close to zero, but not exactly zero. For example, to determine the angle θ for a circle with $r = 3$ m and $A_S = 8$ m², Eq. (3.2) can be written in the form:

$$f(\theta) = 8 - 4.5(\theta - \sin\theta) = 0 \tag{3.3}$$

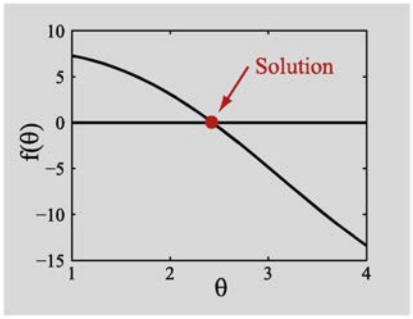


Figure 3-3: A plot of $f(\theta) = 8 - 4.5(\theta - \sin\theta)$.

A plot of $f(\theta)$ (Fig. 3-3) shows that the solution is between 2 and 3. Substituting $\theta = 2.4$ rad in Eq. (3.3) gives $f(\theta) = 0.2396$, and the solution $\theta = 2.43$ rad gives $f(\theta) = 0.003683$. Obviously, the latter is a more accurate, but not an exact, solution. It is possible to determine values of θ that give values of $f(\theta)$ that are closer to zero, but it is impossible to determine a numerical value of θ for which $f(\theta)$ is exactly zero. When solving an equation numerically, one has to select the desired accuracy of the solution.

Overview of approaches in solving equations numerically

The process of solving an equation numerically is different from the procedure used to find an analytical solution. An analytical solution is obtained by deriving an expression that has an exact numerical value. A numerical solution is obtained in a process that starts by finding an approximate solution and is followed by a numerical procedure in which a better (more accurate) solution is determined. An initial numerical solution of an equation $f(x) = 0$ can be estimated by plotting $f(x)$ versus x and looking for the point where the graph crosses the x-axis. It is also possible to write and execute a computer program that looks for a domain that contains a solution. Such a program looks for a solution by evaluating $f(x)$ at different values of x. It starts at one value of x and then changes the value of x in small increments. A change in the sign of $f(x)$ indicates that there is a root within the last increment. In most cases, when the equation that is solved is related to an application in science or engineering, the range of x that includes the solution can be estimated and used in the initial plot of $f(x)$, or for a numerical search of a small domain that contains a solution. When an equation has more than

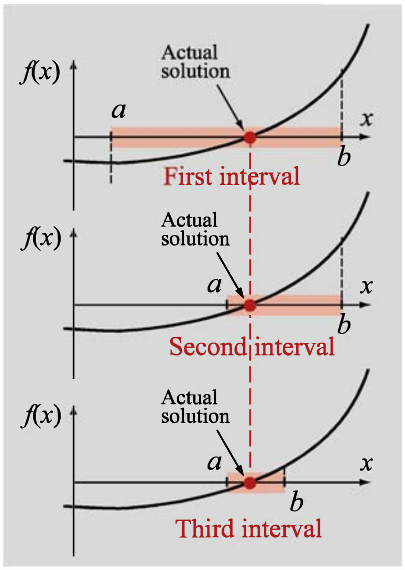


Figure 3-4: Illustration of a bracketing method.

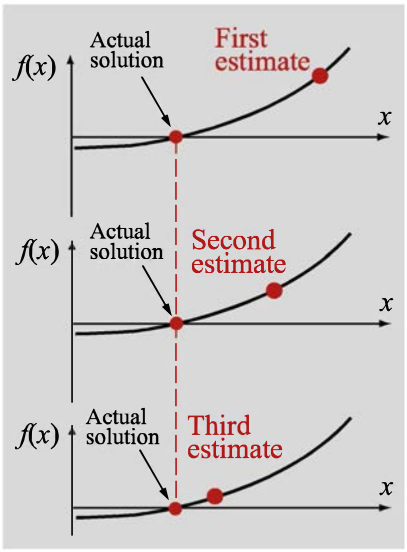


Figure 3-5: Illustration of an open method.

one root, a numerical solution is obtained one root at a time.

The methods used for solving equations numerically can be divided into two groups: ***bracketing methods*** and ***open methods***. In bracketing methods, illustrated in Fig. 3-4, an interval that includes the solution is identified. By definition, the endpoints of the interval are the upper bound and lower bound of the solution. Then, by using a numerical scheme, the size of the interval is successively reduced until the distance between the endpoints is less than the desired accuracy of the solution. In open methods, illustrated in Fig. 3-5, an initial estimate (one point) for the solution is assumed. The value of this initial guess for the solution should be close to the actual solution. Then, by using a numerical scheme, better (more accurate) values for the solution are calculated. Bracketing methods always converge to the solution. Open methods are usually more efficient but sometimes might not yield the solution.

As mentioned previously, since numerical solutions are generally not exact, there is a need for estimating the error. Several options are presented in Section 3.2. Sections 3.3 through 3.7 describe four numerical methods for finding a root of a single equation. Two bracketing methods, the bisection method and the regula falsi method, are presented in Sections 3.3 and 3.4, respectively. Three open methods, Newton's method, secant method, and fixed-point iteration, are introduced in the following three sections. Section 3.8 describes how to use MATLAB's built-in functions for obtaining numerical solutions, and Section 3.9 discusses how to deal with equations that have multiple roots. The last section in this chapter (3.10) deals with numerical methods for solving systems of nonlinear equations. The need to solve such systems arises in many problems in science and engineering and when numerical methods are used for solving ordinary differential equations (see Section 11.3).

3.2 ESTIMATION OF ERRORS IN NUMERICAL SOLUTIONS

Since numerical solutions are not exact, some criterion has to be applied in order to determine whether an estimated solution is accurate enough. Several measures can be used to estimate the accuracy of an approximate solution. The decision as to which measure to use depends on the application and has to be made by the person solving the equation.

Let x_{TS} be the true (exact) solution such that $f(x_{TS}) = 0$, and let x_{NS} be a numerically approximated solution such that $f(x_{NS}) = \varepsilon$ (where ε is a small number). Four measures that can be considered for estimating the error are:

***True error*:** The true error is the difference between the true solution, x_{TS}, and a numerical solution, x_{NS}:

$$TrueError = x_{TS} - x_{NS} \tag{3.4}$$

Unfortunately, however, the true error cannot be calculated because the true solution is generally not known.

***Tolerance in* $f(x)$:** Instead of considering the error in the solution, it is possible to consider the deviation of $f(x_{NS})$ from zero (the value of $f(x)$ at x_{TS} is obviously zero). The tolerance in $f(x)$ is defined as the absolute value of the difference between $f(x_{TS})$ and $f(x_{NS})$:

$$ToleranceInf = |f(x_{TS}) - f(x_{NS})| = |0 - \varepsilon| = |\varepsilon| \tag{3.5}$$

The tolerance in $f(x)$ then is the absolute value of the function at x_{NS}.

***Tolerance in the solution*:** A tolerance is the maximum amount by which the true solution can deviate from an approximate numerical solution. A tolerance is useful for estimating the error when bracketing methods are used for determining the numerical solution. In this case, if it is known that the solution is within the domain $[a, b]$, then the numerical solution can be taken as the midpoint between a and b:

$$x_{NS} = \frac{a+b}{2} \tag{3.6}$$

plus or minus a tolerance that is equal to half the distance between a and b:

$$Tolerance = \left|\frac{b-a}{2}\right| \tag{3.7}$$

Relative error*:** If x_{NS} is an estimated numerical solution, then the ***True Relative Error is given by:

$$TrueRelativeError = \left|\frac{x_{TS} - x_{NS}}{x_{TS}}\right| \tag{3.8}$$

This True Relative Error cannot be calculated since the true solution x_{TS} is not known. Instead, it is possible to calculate an ***Estimated Relative Error*** when two numerical estimates for the solution are known. This is the case when numerical solutions are calculated iteratively, where in each new iteration a more accurate solution is calculated. If $x_{NS}^{(n)}$ is the estimated numerical solution in the last iteration and $x_{NS}^{(n-1)}$ is the estimated numerical solution in the preceding iteration, then an Estimated Relative Error can be defined by:

$$EstimatedRelativeError = \left|\frac{x_{NS}^{(n)} - x_{NS}^{(n-1)}}{x_{NS}^{(n-1)}}\right| \tag{3.9}$$

When the estimated numerical solutions are close to the true solution, it is anticipated that the difference $x_{NS}^{(n)} - x_{NS}^{(n-1)}$ is small compared to the value of $x_{NS}^{(n)}$, and the Estimated Relative Error is approximately the same as the True Relative Error.

3.3 BISECTION METHOD

The bisection method is a bracketing method for finding a numerical solution of an equation of the form $f(x) = 0$ when it is known that within a given interval $[a, b]$, $f(x)$ is continuous and the equation has a solution. When this is the case, $f(x)$ will have opposite signs at the endpoints of the interval. As shown in Fig. 3-6, if $f(x)$ is continuous and

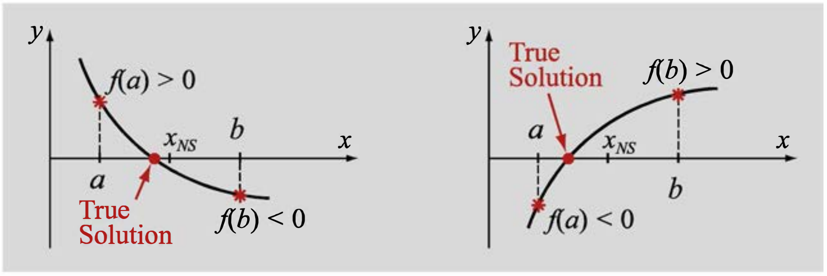


Figure 3-6: Solution of $f(x) = 0$ between $x = a$ and $x = b$.

has a solution between the points $x = a$ and $x = b$, then either $f(a) > 0$ and $f(b) < 0$ or $f(a) < 0$ and $f(b) > 0$. In other words, if there is a solution between $x = a$ and $x = b$, then $f(a)f(b) < 0$.

The process of finding a solution with the bisection method is illustrated in Fig. 3-7. It starts by finding points a and b that define an interval where a solution exists. Such an interval is found either by plotting $f(x)$ and observing a zero crossing, or by examining the function for sign change. The midpoint of the interval x_{NS1} is then taken as the first estimate for the numerical solution. The true solution is either in the section between points a and x_{NS1} or in the section between points x_{NS1} and b. If the numerical solution is not accurate enough, a new interval that contains the true solution is defined. The new interval is the half of the original interval that contains the true solution, and its midpoint is taken as the new (second) estimate of the numerical solution. The process continues until the numerical solution is accurate enough according to a criterion that is selected.

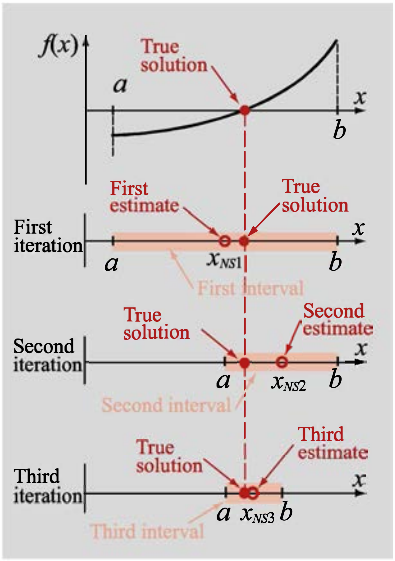


Figure 3-7: Bisection method.

The procedure (or algorithm) for finding a numerical solution with the bisection method is summarized as follows:

Algorithm for the bisection method

1. Choose the first interval by finding points a and b such that a solution exists between them. This means that $f(a)$ and $f(b)$ have different signs such that $f(a)f(b) < 0$. The points can be determined by examining the plot of $f(x)$ versus x.

2. Calculate the first estimate of the numerical solution x_{NS1} by:

$$x_{NS1} = \frac{(a+b)}{2}$$

3. Determine whether the true solution is between a and x_{NS1}, or between x_{NS1} and b. This is done by checking the sign of the product $f(a) \cdot f(x_{NS1})$:

   If $f(a) \cdot f(x_{NS1}) < 0$, the true solution is between a and x_{NS1}.

   If $f(a) \cdot f(x_{NS1}) > 0$, the true solution is between x_{NS1} and b.

4. Select the subinterval that contains the true solution (a to x_{NS1}, or x_{NS1} to b) as the new interval $[a, b]$, and go back to step 2.

Steps 2 through 4 are repeated until a specified tolerance or error bound is attained.

When should the bisection process be stopped?

Ideally, the bisection process should be stopped when the true solution is obtained. This means that the value of x_{NS} is such that $f(x_{NS}) = 0$. In reality, as discussed in Section 3.1, this true solution generally cannot be found computationally. In practice, therefore, the process is stopped when the estimated error, according to one of the measures listed in Section 3.2, is smaller than some predetermined value. The choice of termination criteria may depend on the problem that is actually solved.

A MATLAB program written in a script file that determines a numerical solution by applying the bisection method is shown in the solution of the following example. (Rewriting this program in a form of a user-defined function is assigned as a homework problem.)

Example 3-1: Solution of a nonlinear equation using the bisection method.

Write a MATLAB program, in a script file, that determines the solution of the equation $8 - 4.5(x - \sin x) = 0$ by using the bisection method. The solution should have a tolerance of less than 0.001 rad. Create a table that displays the values of a, b, x_{NS}, $f(x_{NS})$, and the tolerance for each iteration of the bisection process.

SOLUTION

To find the approximate location of the solution, a plot of the function $f(x) = 8 - 4.5(x - \sin x)$ is made by using the `fplot` command of MATLAB. The plot (Fig. 3-8), shows that the solution is between $x = 2$ and $x = 3$. The initial interval is chosen as $a = 2$ and $b = 3$.

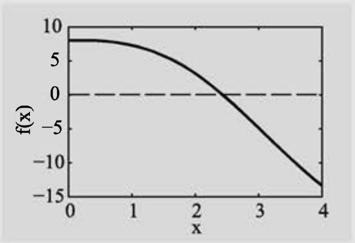


Figure 3-8: A plot of the function $f(x) = 8 - 4.5(x - \sin x)$.

A MATLAB program that solves the problem is as follows.

Program 3-1: Script file. Bisection method.

```
clear all
F = @ (x) 8-4.5*(x-sin(x));
```

Define $f(x)$ as an anonymous function.

```
a = 2; b = 3; imax = 20; tol = 0.001;
Fa = F(a); Fb = F(b);
if  Fa*Fb > 0
  disp('Error:The function has the same sign at points a and b.')
else
  disp('iteration   a       b   (xNS) Solution  f(xNS)  Tolerance')
  for i = 1:imax
      xNS = (a + b)/2;
      toli = (b - a)/2;
      FxNS = F(xNS);
      fprintf('%3i  %11.6f %11.6f %11.6f  %11.6f %11.6f\n', i, a, b, xNS, FxNS, toli)
      if FxNS = = 0
        fprintf('An exact solution x =%11.6f was found',xNS)
         break
      end
      if toli < tol
          break
      end
      if i = = imax
          fprintf('Solution was not obtained in %i iterations',imax)
          break
      end
      if F(a)*FxNS < 0
          b = xNS;
      else
          a = xNS;
      end
  end
end
```

Assign initial values to a and b, define max number of iterations and tolerance.

Stop the program if the function has the same sign at points a and b.

Calculate the numerical solution of the iteration, xNS.

Calculate the current tolerance.

Calculate the value of $f(xNS)$ of the iteration.

Stop the program if the true solution, $f(x) = 0$, is found.

Stop the iterations if the tolerance of the iteration is smaller than the desired tolerance.

Stop the iterations if the solution was not obtained and the number of the iteration reaches `imax`.

Determine whether the true solution is between a and xNS, or between xNS and b, and select a and b for the next iteration.

When the program is executed, the display in the Command Window is:

```
iteration      a           b      (xNS) Solution    f(xNS)      Tolerance
  1         2.000000    3.000000    2.500000    -0.556875    0.500000
  2         2.000000    2.500000    2.250000     1.376329    0.250000
  3         2.250000    2.500000    2.375000     0.434083    0.125000
  4         2.375000    2.500000    2.437500    -0.055709    0.062500
  5         2.375000    2.437500    2.406250     0.190661    0.031250
  6         2.406250    2.437500    2.421875     0.067838    0.015625
  7         2.421875    2.437500    2.429688     0.006154    0.007813
  8         2.429688    2.437500    2.433594    -0.024755    0.003906
  9         2.429688    2.433594    2.431641    -0.009295    0.001953
 10         2.429688    2.431641    2.430664    -0.001569    0.000977
```

The numerical solution.

The value of the function at the numerical solution.

The last tolerance (satisfies the prescribed tolerance).

The output shows that the solution with the desired tolerance is obtained in the 10th iteration.

Additional notes on the bisection method

- The method always converges to an answer, provided a root was trapped in the interval $[a, b]$ to begin with.
- The method may fail when the function is tangent to the axis and does not cross the x-axis at $f(x) = 0$.
- The method converges slowly relative to other methods.

3.4 REGULA FALSI METHOD

The regula falsi method (also called false position and linear interpolation methods) is a bracketing method for finding a numerical solution of an equation of the form $f(x) = 0$ when it is known that, within a given interval $[a, b]$, $f(x)$ is continuous and the equation has a solution. As illustrated in Fig. 3-9, the solution starts by finding an initial interval $[a_1, b_1]$ that brackets the solution. The values of the function at the endpoints are $f(a_1)$ and $f(b_1)$. The endpoints are then connected by a straight line, and the first estimate of the numerical solution, x_{NS1}, is the point where the straight line crosses the x-axis. This is in contrast to the bisection method, where the midpoint of the interval was taken as the solution. For the second iteration a new interval, $[a_2, b_2]$ is defined. The

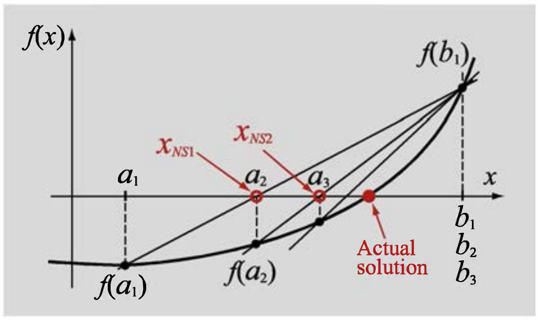


Figure 3-9: Regula Falsi method.

new interval is a subsection of the first interval that contains the solution. It is either $[a_1, x_{NS1}]$ (a_1 is assigned to a_2, and x_{NS1} to b_2) or $[x_{NS1}, b_1]$ (x_{NS1} is assigned to a_2, and b_1 to b_2). The endpoints of the second interval are next connected with a straight line, and the point where this new line crosses the x-axis is the second estimate of the solution, x_{NS2}. For the third iteration, a new subinterval $[a_3, b_3]$ is selected, and the iterations continue in the same way until the numerical solution is deemed accurate enough.

For a given interval $[a, b]$, the equation of a straight line that connects point $(b, f(b))$ to point $(a, f(a))$ is given by:

$$y = \frac{f(b) - f(a)}{b - a}(x - b) + f(b) \tag{3.10}$$

The point x_{NS} where the line intersects the x-axis is determined by substituting $y = 0$ in Eq. (3.10), and solving the equation for x:

$$x_{NS} = \frac{af(b) - bf(a)}{f(b) - f(a)} \tag{3.11}$$

The procedure (or algorithm) for finding a solution with the regula falsi method is almost the same as that for the bisection method.

Algorithm for the regula falsi method

1. Choose the first interval by finding points a and b such that a solution exists between them. This means that $f(a)$ and $f(b)$ have different signs such that $f(a)f(b) < 0$. The points can be determined by looking at a plot of $f(x)$ versus x.
2. Calculate the first estimate of the numerical solution x_{NS1} by using Eq. (3.11).
3. Determine whether the actual solution is between a and x_{NS1} or between x_{NS1} and b. This is done by checking the sign of the product $f(a) \cdot f(x_{NS1})$:
   If $f(a) \cdot f(x_{NS1}) < 0$, the solution is between a and x_{NS1}.
   If $f(a) \cdot f(x_{NS1}) > 0$, the solution is between x_{NS1} and b.
4. Select the subinterval that contains the solution (a to x_{NS1}, or x_{NS1} to b) as the new interval $[a, b]$, and go back to step 2.

Steps 2 through 4 are repeated until a specified tolerance or error bound is attained.

When should the iterations be stopped?

The iterations are stopped when the estimated error, according to one of the measures listed in Section 3.2, is smaller than some predetermined value.

Additional notes on the regula falsi method

- The method always converges to an answer, provided a root is initially trapped in the interval $[a, b]$.
- Frequently, as in the case shown in Fig. 3-9, the function in the interval $[a, b]$ is either concave up or concave down. In this case, one of the endpoints of the interval stays the same in all the iterations, while the other endpoint advances toward the root. In other words, the numerical solution advances toward the root only from one side. The convergence toward the solution could be faster if the other endpoint would also "move" toward the root. Several modifications have been introduced to the regula falsi method that make the subinterval in successive iterations approach the root from both sides (see Problem 3.18).

3.5 NEWTON'S METHOD

Newton's method (also called the Newton–Raphson method) is a scheme for finding a numerical solution of an equation of the form $f(x) = 0$ where $f(x)$ is continuous and differentiable and the equation is known to have a solution near a given point. The method is illustrated in Fig. 3.10.

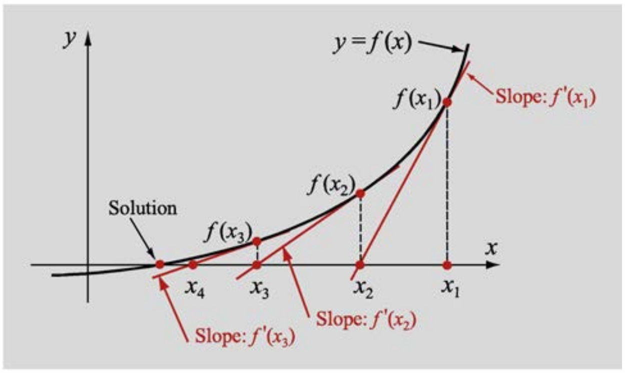


Figure 3-10: Newton's method.

The solution process starts by choosing point x_1 as the first estimate of the solution. The second estimate x_2 is obtained by taking the tangent line to $f(x)$ at the point $(x_1, f(x_1))$ and finding the intersection point of the tangent line with the x-axis. The next estimate x_3 is the intersection of the tangent line to $f(x)$ at the point $(x_2, f(x_2))$ with the x-axis, and so on. Mathematically, for the first iteration, the slope, $f'(x_1)$, of the tangent at point $(x_1, f(x_1))$ is given by:

$$f'(x_1) = \frac{f(x_1) - 0}{x_1 - x_2} \tag{3.12}$$

Solving Eq. (3.12) for x_2 gives:

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} \tag{3.13}$$

Equation 3.13 can be generalized for determining the "next" solution x_{i+1} from the present solution x_i:

$$\boxed{x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}} \tag{3.14}$$

Equation (3.14) is the general iteration formula for Newton's method. It is called an iteration formula because the solution is found by repeated application of Eq. (3.14) for each successive value of i.

Newton's method can also be derived by using Taylor series. Taylor series expansion of $f(x)$ about x_1 is given by:

$$f(x) = f(x_1) + (x - x_1)f'(x_1) + \frac{1}{2!}(x - x_1)^2 f''(x_1) + \ldots \tag{3.15}$$

If x_2 is a solution of the equation $f(x) = 0$ and x_1 is a point near x_2, then:

$$f(x_2) = 0 = f(x_1) + (x_2 - x_1) f'(x_1) + \frac{1}{2!}(x_2 - x_1)^2 f''(x_1) + \dots \quad (3.16)$$

By considering only the first two terms of the series, an approximate solution can be determined by solving Eq. (3.16) for x_2:

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} \quad (3.17)$$

The result is the same as Eq. (3.13). In the next iteration the Taylor expansion is written about point x_2, and an approximate solution x_3 is calculated. The general formula is the same as that given in Eq. (3.14).

Algorithm for Newton's method

1. Choose a point x_1 as an initial guess of the solution.
2. For $i = 1, 2, \dots$, until the error is smaller than a specified value, calculate x_{i+1} by using Eq. (3.14).

When are the iterations stopped?

Ideally, the iterations should be stopped when an exact solution is obtained. This means that the value of x is such that $f(x) = 0$. Generally, as discussed in Section 3.1, this exact solution cannot be found computationally. In practice therefore, the iterations are stopped when an estimated error is smaller than some predetermined value. A tolerance in the solution, as in the bisection method, cannot be calculated since bounds are not known. Two error estimates that are typically used with Newton's method are:

***Estimated relative error*:** The iterations are stopped when the estimated relative error (Eq. (3.9)) is smaller than a specified value ε:

$$\left|\frac{x_{i+1} - x_i}{x_i}\right| \le \varepsilon \quad (3.18)$$

Tolerance in $f(x)$**:** The iterations are stopped when the absolute value of $f(x_i)$ is smaller than some number δ:

$$|f(x_i)| \le \delta \quad (3.19)$$

The programming of Newton's method is very simple. A MATLAB user-defined function (called `NewtonRoot`) that finds the root of $f(x) = 0$ is listed in Fig. 3-11. The program consists of one loop in which the next solution `Xi` is calculated from the present solution `Xest` using Eq. (3.14). The looping stops if the error is small enough according to Eq. (3.18). To avoid the situation where the looping continues indefinitely (either because the solution does not converge or because of a programming error), the number of passes in the loop is limited to `imax`. The functions $f(x)$ and $f'(x)$ (that appear in Eq. (3.14)) have to be supplied as separate user-defined functions. They are entered in the arguments of `NewtonRoot` as function handles.

Program 3-2: User-defined function. Newton's method.

```
function Xs = NewtonRoot(Fun,FunDer,Xest,Err,imax)
% NewtonRoot finds the root of Fun = 0 near the point Xest using Newton's method.
% Input variables:
% Fun    Name of a user-defined function that calculates Fun for a given x.
% FunDer Name of a user-defined function that calculates the derivative
%        of Fun for a given x.
% Xest    Initial estimate of the solution.
% Err    Maximum error.
% imax    Maximum number of iterations
% Output variable:
% Xs      Solution

for i = 1:imax
    Xi = Xest - Fun(Xest)/FunDer(Xest);
    if abs((Xi - Xest)/Xest) < Err
        Xs = Xi;
        break
    end
    Xest = Xi;
end
if i = = imax
    fprintf('Solution was not obtained in %i iterations.\n',imax)
    Xs = ('No answer');
end
```

Eq. (3.14).

Eq. (3.18).

Figure 3-11: MATLAB function file for solving equation using the Newton's method.

Example 3-2 shows how Eq. (3.14) is used, and how to use the user-defined function `NewtonRoot` to solve a specific problem.

Example 3-2: Solution of equation using Newton's method.

Find the solution of the equation $8 - 4.5(x - \sin x) = 0$ (the same equation as in Example 3-1) by using Newton's method in the following two ways:

(*a*) Using a nonprogrammable calculator, calculate the first two iterations on paper using six significant figures.

(*b*) Use MATLAB with the function `NewtonRoot` that is listed in Fig. 3-11. Use 0.0001 for the maximum relative error and 10 for the maximum number of iterations.

In both parts, use $x = 2$ as the initial guess of the solution.

SOLUTION

In the present problem, $f(x) = 8 - 4.5(x - \sin x)$ and $f'(x) = -4.5(1 - \cos x)$.

(*a*) To start the iterations, $f(x)$ and $f'(x)$ are substituted in Eq. (3.14):

$$x_{i+1} = x_i - \frac{8 - 4.5(x_i - \sin x_i)}{-4.5(1 - \cos x_i)} \tag{3.20}$$

In the first iteration, $i = 1$ and $x_1 = 2$, and Eq. (3.20) gives:

$$x_2 = 2 - \frac{8 - 4.5(2 - \sin(2))}{-4.5(1 - \cos(2))} = 2.48517 \tag{3.21}$$

For the second iteration, $i = 2$ and $x_2 = 2.48517$, and Eq. (3.20) gives:

$$x_3 = 2.48517 - \frac{8 - 4.5(2.48517 - \sin(2.48517))}{-4.5(1 - \cos(2.48517))} = 2.43099 \tag{3.22}$$

(*b*) To solve the equation with MATLAB using the function `NewtonRoot`, the user must create user-defined functions for $f(x)$ and $f'(x)$. The two functions, called `FunExample2` and `FunDerExample2`, are:

```
function y = FunExample2(x)
y = 8 - 4.5*(x - sin(x));
```

and

```
function y = FunDerExample2(x)
y = -4.5 + 4.5*cos(x);
```

Once the functions are created and saved, the `NewtonRoot` function can be used in the Command Window:

The user-defined functions are entered as function handles.

```
>> format long
>> xSolution = NewtonRoot(@FunExample2,@FunDerExample2,2,0.0001,10)
xSolution =
   2.430465741723630
```

A comparison of the results from parts *a* and *b* shows that the first four digits of the solution (2.430) are obtained in the second iteration. (In part *b*, the solution process stops in the fourth iteration; see Problem 3.19.) This shows, as was mentioned before, that Newton's method usually converges fast. In Example 3-1 (bisection method), the first four digits are obtained only after 10 bisections.

Notes on Newton's method

- The method, when successful, works well and converges fast. When it does not converge, it is usually because the starting point is not close enough to the solution. Convergence problems typically occur when the value of $f'(x)$ is close to zero in the vicinity of the solution (where $f(x) = 0$). It is possible to show that Newton's method converges if the function $f(x)$ and its first and second derivatives $f'(x)$ and $f''(x)$ are all continuous, if $f'(x)$ is not zero at the solution, and if the starting value x_1 is near the actual solution. Illustrations of two cases where Newton's method does not converge (i.e., diverges) are shown in Fig. 3-12.

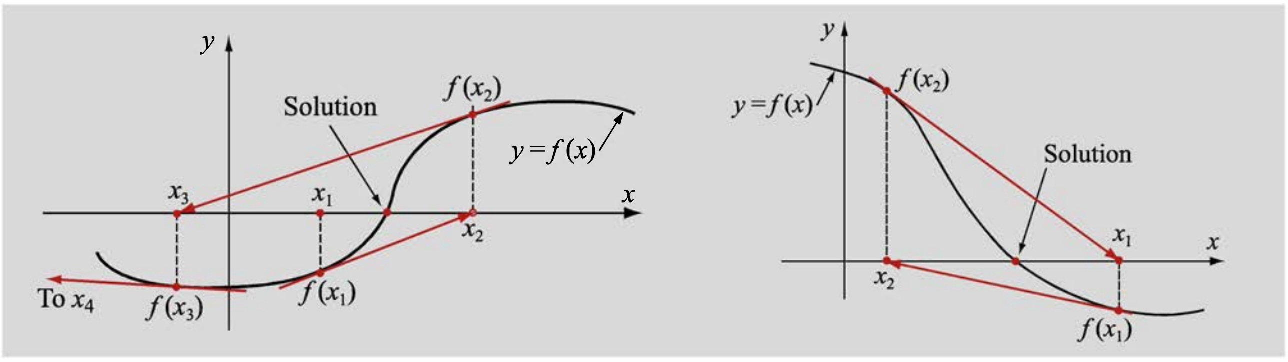


Figure 3-12: Cases where Newton's method diverges.

- A function $f'(x)$, which is the derivative of the function $f(x)$, has to be substituted in the iteration formula, Eq. (3.14). In many cases, it is simple to write the derivative, but sometimes it can be difficult to determine. When an expression for the derivative is not available, it might be possible to determine the slope numerically or to find a solution by using the secant method (Section 3.6), which is somewhat similar to Newton's method but does not require an expression for the derivative.

Next, Example 3-3 illustrates the effect that the starting point can have on a numerical solution with Newton's method.

Example 3-3: Convergence of Newton's method.

Find the solution of the equation $\frac{1}{x} - 2 = 0$ by using Newton's method. For the starting point (initial estimate of the solution) use:

(a) $x = 1.4$, (b) $x = 1$, and (c) $x = 0.4$

SOLUTION

The equation can easily be solved analytically, and the exact solution is $x = 0.5$.

For a numerical solution with Newton's method the function, $f(x) = \frac{1}{x} - 2$, and its derivative, $f'(x) = -\frac{1}{x^2}$, are substituted in Eq. (3.14):

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} = x_i - \frac{\frac{1}{x_i} - 2}{-\frac{1}{x_i^2}} = 2(x_i - x_i^2) \qquad (3.23)$$

(a) When the starting point for the iterations is $x_1 = 1.4$, the next two iterations, using Eq. (3.23), are:

$x_2 = 2(x_1 - x_1^2) = 2(1.4 - 1.4^2) = -1.12$ and $x_3 = 2(x_2 - x_2^2) = 2[(-1.12) - (-1.12)^2] = -4.7488$

These results indicate that Newton's method diverges. This case is illustrated in Fig. 3-13a.

(b) When the starting point for the iterations is $x = 1$, the next two iterations, using Eq. (3.23), are:

$$x_2 = 2(x_1 - x_1^2) = 2(1 - 1^2) = 0 \quad \text{and} \quad x_3 = 2(x_2 - x_2^2) = 2(0 - 0^2) = 0$$

From these results it looks like the solution converges to $x = 0$, which is not a solution. At $x = 0$, the function is actually not defined (it is a singular point). A solution is obtained from Eq. (3.23) because the equation was simplified. This case is illustrated in Fig. 3-13b.

(c) When the starting point for the iterations is $x = 0.4$, the next two iterations, using Eq. (3.23), are:

$$x_2 = 2(x_1 - x_1^2) = 2(0.4 - 0.4^2) = 0.48 \quad \text{and} \quad x_3 = 2(x_2 - x_2^2) = 2(0.48 - 0.48^2) = 0.4992$$

In this case, Newton's method converges to the correct solution. This case is illustrated in Fig. 3-13c. This example also shows that if the starting point is close enough to the true solution, Newton's method converges.

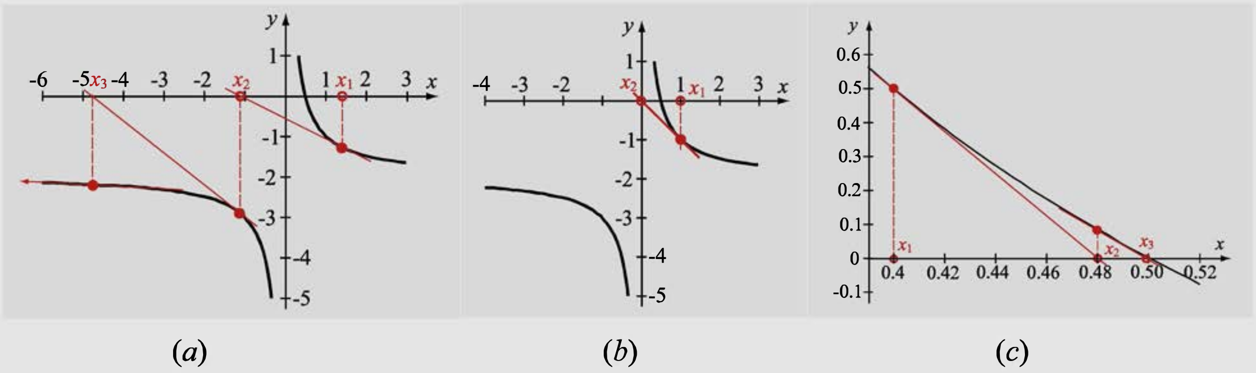


Figure 3-13: Solution with Newton's method using different starting points.

3.6 SECANT METHOD

The secant method is a scheme for finding a numerical solution of an equation of the form $f(x) = 0$. The method uses two points in the neighborhood of the solution to determine a new estimate for the solution (Fig. 3-14). The two points (marked as x_1 and x_2 in the figure) are used to define a straight line (secant line), and the point where the line intersects the x-axis (marked as x_3 in the figure) is the new estimate for the solution. As shown, the two points can be on one side of the solution

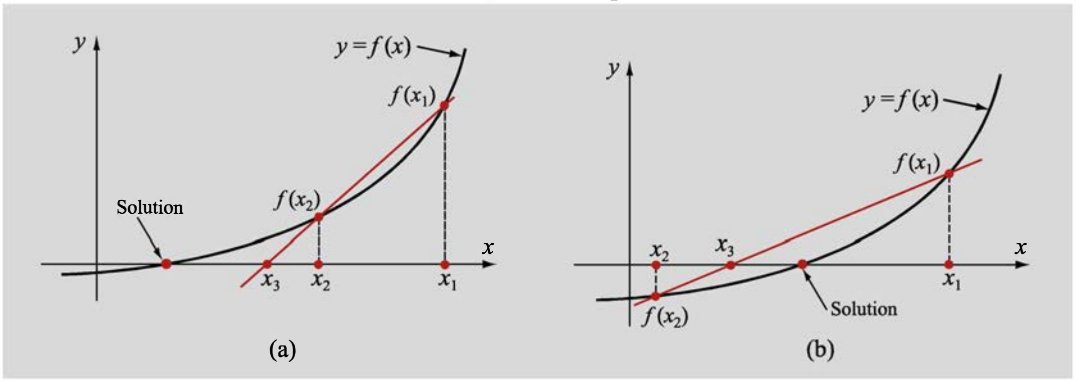


Figure 3-14: The secant method.

(Fig.3-14*a*) or the solution can be between the two points (Fig. 3-14*b*). The slope of the secant line is given by:

$$\frac{f(x_1)-f(x_2)}{x_1-x_2} = \frac{f(x_2)-0}{x_2-x_3} \tag{3.24}$$

which can be solved for x_3:

$$x_3 = x_2 - \frac{f(x_2)(x_1-x_2)}{f(x_1)-f(x_2)} \tag{3.25}$$

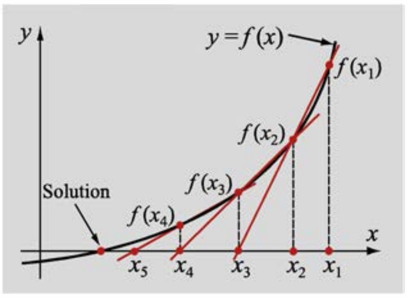


Figure 3-15: Secant method.

Once point x_3 is determined, it is used together with point x_2 to calculate the next estimate of the solution, x_4. Equation (3.25) can be generalized to an iteration formula in which a new estimate of the solution x_{i+1} is determined from the previous two solutions x_i and x_{i-1}.

$$x_{i+1} = x_i - \frac{f(x_i)(x_{i-1}-x_i)}{f(x_{i-1})-f(x_i)} \tag{3.26}$$

Figure 3-15 illustrates the iteration process with the secant method.

Relationship to Newton's method

Examination of the secant method shows that when the two points that define the secant line are close to each other, the method is actually an approximated form of Newton's method. This can be seen by rewriting Eq. (3.26) in the form:

$$x_{i+1} = x_i - \frac{f(x_i)}{\dfrac{f(x_{i-1})-f(x_i)}{(x_{i-1}-x_i)}} \tag{3.27}$$

This equation is almost identical to Eq. (3.14) of Newton's method. In Eq. (3.27), the denominator of the second term on the right-hand side of the equation is an approximation of the value of the derivative of $f(x)$ at x_i. In Eq. (3.14), the denominator is actually the derivative $f'(x_i)$. In the secant method (unlike Newton's method), it is not necessary to know the analytical form of $f'(x)$.

Programming of the secant method is very similar to that of Newton's method. Figure 3-16 lists a MATLAB user-defined function (called `SecantRoot`) that finds the root of $f(x) = 0$. The program consists of one loop in which the next solution `Xi` is calculated from the previous two solutions, `Xb` and `Xa`, using Eq. (3.26). The looping stops if the error is small enough according to Eq. (3.18). The function $f(x)$ (that is used in Eq. (3.26)) has to be supplied as a separate user-defined function. Its name is typed in the argument of `SecantRoot` as a function handle.

Program 3-3: User-defined function. Secant method.

```
function Xs = SecantRoot(Fun,Xa,Xb,Err,imax)
% SecantRoot finds the root of Fun = 0 using the secant method.
% Input variables:
% Fun    Name of a user-defined function that calculates Fun for a given x.
% a, b   Two points in the neighborhood of the root (on either side or the
%         same side of the root).
% Err     Maximum error.
% imax     Maximum number of iterations
% Output variable:
% Xs      Solution

for i = 1:imax
    FunXb = Fun(Xb);
    Xi = Xb - FunXb*(Xa-Xb)/(Fun(Xa)-FunXb);                    Eq. (3.26).
    if abs((Xi - Xb)/Xb) < Err                                  Eq. (3.18).
        Xs = Xi;
        break
    end
    Xa = Xb;
    Xb = Xi;
end
if i = = imax
    fprintf('Solution was not obtained in %i itera-
tions.\n',imax)
    Xs = ('No answer');
end
```

Figure 3-16: MATLAB function file for solving equation using the secant method.

As an example, the function from Examples 3-1 and 3-2 is solved with the `SecantRoot` user-defined function. The two starting points are taken as $a = 2$ and $b = 3$.

```
>> format long
>> xSolution = Secant-
Root(@FunExample2,2,3,0.0001,10)
xSolution =
   2.430465726588755
```

The user-defined function `SecantRoot` is also used in the solution of Example 3-4.

3.7 FIXED-POINT ITERATION METHOD

Fixed-point iteration is a method for solving an equation of the form $f(x) = 0$. The method is carried out by rewriting the equation in the form:

$$x = g(x) \tag{3.28}$$

Obviously, when x is the solution of $f(x) = 0$, the left side and the right side of Eq. (3.28) are equal. This is illustrated graphically by plotting $y = x$ and $y = g(x)$, as shown in Fig. 3-17. The point of intersection of the two plots, called the ***fixed point***, is the solution. The numerical value of the solution is determined by an iterative process. It starts by taking a value of x near the fixed point as the first guess for the solution and substituting it in $g(x)$. The value of $g(x)$ that is obtained is the new (second) estimate for the solution. The second value is then substituted back in $g(x)$, which then gives the third estimate of the solution. The iteration formula is thus given by:

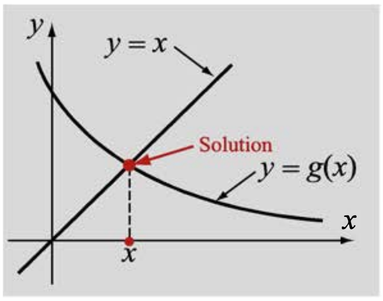


Figure 3-17: Fixed-point iteration method.

$$x_{i+1} = g(x_i) \tag{3.29}$$

The function $g(x)$ is called the ***iteration function***.

- When the method works, the values of x that are obtained are successive iterations that progressively converge toward the solution. Two such cases are illustrated graphically in Fig. 3-18. The solution process starts by choosing point x_1 on the x-axis and drawing a vertical line that intersects the curve $y = g(x)$ at point $g(x_1)$. Since $x_2 = g(x_1)$, a horizontal line is drawn from point $(x_1, g(x_1))$ toward the line $y = x$. The intersection point gives the location of x_2. From x_2 a vertical line is drawn toward the curve $y = g(x)$. The intersection point is now $(x_2, g(x_2))$, and $g(x_2)$ is also the value of x_3. From point $(x_2, g(x_2))$ a horizontal line is drawn again toward $y = x$, and

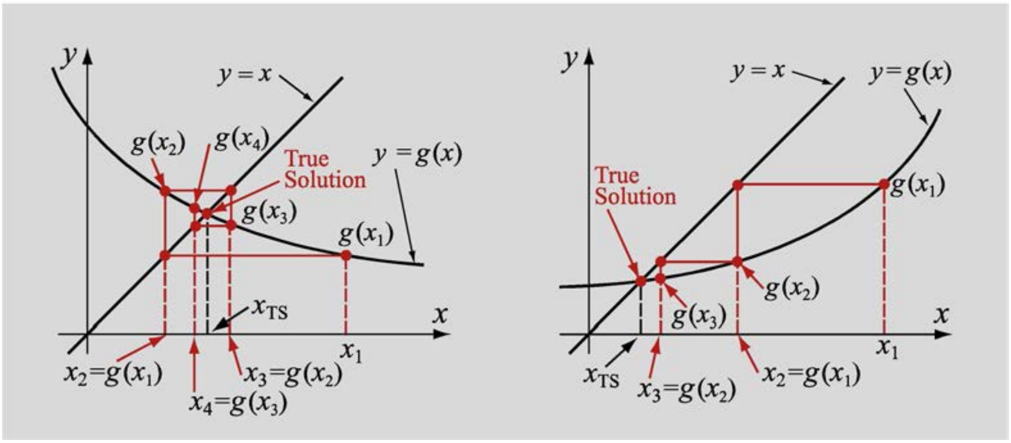


Figure 3-18: Convergence of the fixed-point iteration method.

the intersection point gives the location of x_3. As the process continues the intersection points converge toward the fixed point, or the true solution x_{TS}.

- It is possible, however, that the iterations will not converge toward the fixed point, but rather diverge away. This is shown in Fig. 3-19. The figure shows that even though the starting point is close to the solution, the subsequent points are moving farther away from the solution.

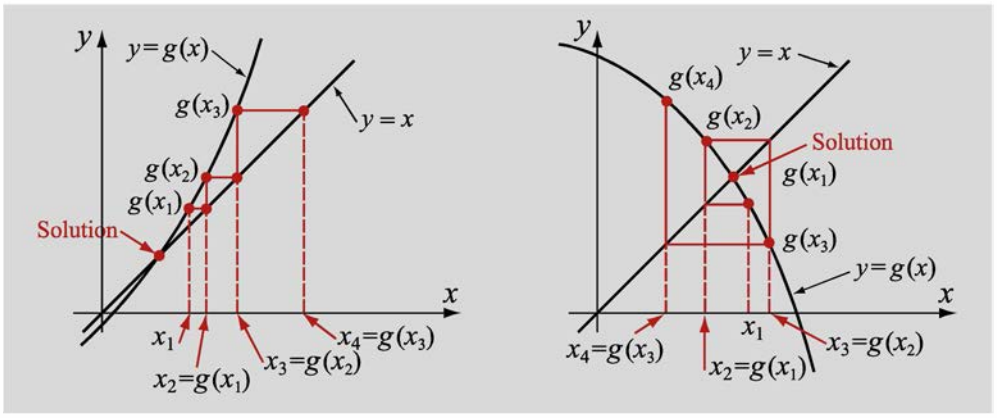


Figure 3-19: Divergence of the fixed-point iteration method.

- Sometimes, the form $f(x) = 0$ does not lend itself to deriving an iteration formula of the form $x = g(x)$. In such a case, one can always add and subtract x to $f(x)$ to obtain $x + f(x) - x = 0$. The last equation can be rewritten in the form that can be used in the fixed-point iteration method:

$$x = x + f(x) = g(x)$$

Choosing the appropriate iteration function g(x)

For a given equation $f(x) = 0$, the iteration function is not unique since it is possible to change the equation into the form $x = g(x)$ in different ways. This means that several iteration functions $g(x)$ can be written for the same equation. A $g(x)$ that should be used in Eq. (3.29) for the iteration process is one for which the iterations converge toward the solution. There might be more than one form that can be used, or it may be that none of the forms are appropriate so that the fixed-point iteration method cannot be used to solve the equation. In cases where there are multiple solutions, one iteration function may yield one root, while a different function yields other roots. Actually, it is possible to determine ahead of time if the iterations converge or diverge for a specific $g(x)$.

The fixed-point iteration method converges if, in the neighborhood of the fixed point, the derivative of $g(x)$ has an absolute value that is smaller than 1 (also called Lipschitz continuous):

$$|g'(x)| < 1 \tag{3.30}$$

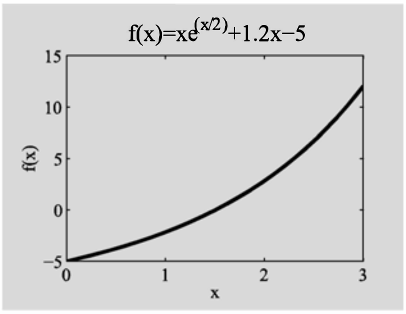


Figure 3-20: A plot of $f(x) = xe^{x/2} + 1.2x - 5$.

As an example, consider the equation:

$$xe^{0.5x} + 1.2x - 5 = 0 \tag{3.31}$$

A plot of the function $f(x) = xe^{0.5x} + 1.2x - 5$ (see Fig. 3-20) shows that the equation has a solution between $x = 1$ and $x = 2$.

Equation (3.31) can be rewritten in the form $x = g(x)$ in different ways. Three possibilities are discussed next.

***Case a*:**
$$x = \frac{5 - xe^{0.5x}}{1.2} \tag{3.32}$$

In this case, $g(x) = \frac{5 - xe^{0.5x}}{1.2}$ and $g'(x) = -(e^{0.5x} + 0.5xe^{0.5x})/1.2$. The values of $g'(x)$ at points $x = 1$ and $x = 2$, which are in the neighborhood of the solution, are:

$$g'(1) = -(e^{0.5 \cdot 1} + 0.5 \cdot 1e^{0.5 \cdot 1})/1.2 = -2.0609$$
$$g'(2) = -(e^{0.5 \cdot 2} + 0.5 \cdot 2e^{0.5 \cdot 2})/1.2 = -4.5305$$

***Case b*:**
$$x = \frac{5}{e^{0.5x} + 1.2} \tag{3.33}$$

In this case, $g(x) = \frac{5}{e^{0.5x} + 1.2}$ and $g'(x) = \frac{-5e^{0.5x}}{2(e^{0.5x} + 1.2)^2}$.

The values of $g'(x)$ at points $x = 1$ and $x = 2$, which are in the neighborhood of the solution, are:

$$g'(1) = \frac{-5e^{0.5 \cdot 1}}{2(e^{0.5 \cdot 1} + 1.2)^2} = -0.5079$$
$$g'(2) = \frac{-5e^{0.5 \cdot 2}}{2(e^{0.5 \cdot 2} + 1.2)^2} = -0.4426$$

***Case c*:**
$$x = \frac{5 - 1.2x}{e^{0.5x}} \tag{3.34}$$

In this case, $g(x) = \frac{5 - 1.2x}{e^{0.5x}}$ and $g'(x) = \frac{-3.7 + 0.6x}{e^{0.5x}}$.

The values of $g'(x)$ at points $x = 1$ and $x = 2$, which are in the neighborhood of the solution, are:

$$g'(1) = \frac{-3.7 + 0.6 \cdot 1}{e^{0.5 \cdot 1}} = -1.8802$$
$$g'(2) = \frac{-3.7 + 0.6 \cdot 2}{e^{0.5 \cdot 2}} = -0.9197$$

These results show that the iteration function $g(x)$ from Case b is the one that should be used since, in this case, $|g'(1)| < 1$ and $|g'(2)| < 1$. Substituting $g(x)$ from Case b in Eq. (3.29) gives:

$$x_{i+1} = \frac{5}{e^{0.5x_i} + 1.2} \tag{3.35}$$

Starting with $x_1 = 1$, the first few iterations are:

$$x_2 = \frac{5}{e^{0.5 \cdot 1} + 1.2} = 1.7552 \qquad x_3 = \frac{5}{e^{0.5 \cdot 1.7552} + 1.2} = 1.3869$$

$$x_4 = \frac{5}{e^{0.5 \cdot 1.3869} + 1.2} = 1.5622 \qquad x_5 = \frac{5}{e^{0.5 \cdot 1.5622} + 1.2} = 1.4776$$

$$x_6 = \frac{5}{e^{0.5 \cdot 1.4776} + 1.2} = 1.5182 \qquad x_7 = \frac{5}{e^{0.5 \cdot 1.5182} + 1.2} = 1.4986$$

As expected, the values calculated in the iterations are converging toward the actual solution, which is $x = 1.5050$.

On the contrary, if the function $g(x)$ from Case a is used in the iteration, the first few iterations are:

$$x_2 = \frac{5 - 1e^{0.5 \cdot 1}}{1.2} = 2.7927 \qquad x_3 = \frac{5 - 2.7927e^{0.5 \cdot 2.7927}}{1.2} = -5.2364$$

$$x_4 = \frac{5 - (-5.2364)e^{0.5 \cdot (-5.2364)}}{1.2} = 4.4849$$

$$x_5 = \frac{5 - 4.4849e^{0.5 \cdot 4.4849}}{1.2} = -31.0262$$

In this case, the iterations give values that diverge from the solution.

When should the iterations be stopped?

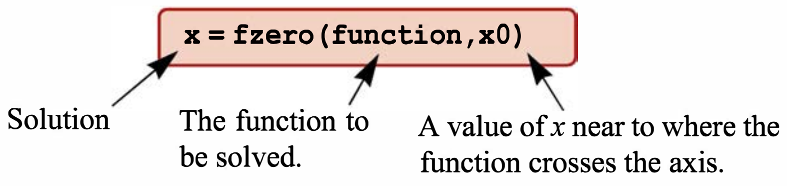
The true error (the difference between the true solution and the estimated solution) cannot be calculated since the true solution in general is not known. As with Newton's method, the iterations can be stopped either when the relative error or the tolerance in $f(x)$ is smaller than some predetermined value (Eqs. (3.18) or (3.19)).

3.8 USE OF MATLAB BUILT-IN FUNCTIONS FOR SOLVING NONLINEAR EQUATIONS

MATLAB has two built-in functions for solving equations with one variable. The `fzero` command can be used to find a root of any equation, and the `roots` command can be used for finding the roots of a polynomial.

3.8.1 The `fzero` Command

The `fzero` command can be used to solve an equation (in the form $f(x) = 0$) with one variable. The user needs to know approximately where the solution is, or if there are multiple solutions, which one is desired. The form of the command is:



```
x = fzero(function,x0)
```

Solution
The function to be solved.
A value of x near to where the function crosses the axis.

- `x` is the solution, which is a scalar.
- `function` is the function whose root is desired. It can be entered in three different ways:
  1. The simplest way is to enter the mathematical expression as a string.
  2. The function is first written as a user-defined function, and then the function handle is entered.
  3. The function is written as an anonymous function, and then its name (which is the name of the handle) is entered.
- The function has to be written in a standard form. For example, if the function to be solved is $xe^{-x} = 0.2$, it has to be written as $f(x) = xe^{-x} - 0.2 = 0$. If this function is entered into the `fzero` command as a string, it is typed as `'x*exp(-x)-0.2'`.
- When a function is entered as a string, it cannot include predefined variables. For example, if the function to be entered is $f(x) = xe^{-x} - 0.2$, it is not possible to first define `b=0.2` and then enter `'x*exp(-x)-b'`.
- `x0` can be a scalar or a two-element vector. If it is entered as a scalar, it has to be a value of x near the point where the function crosses the x-axis. If `x0` is entered as a vector, the two elements have to be points on opposite sides of the solution such that $f(x0(1))$ has a different sign than $f(x0(2))$. When a function has more than one solution, each solution can be determined separately by using the `fzero` function and entering values for `x0` that are near each of the solutions.

Usage of the `fzero` command is illustrated next for solving the equation in Examples 3-1 and 3-2. The function $f(x) = 8 - 4.5(x - \sin x)$ is first defined as an anonymous function named `FUN`. Then the name `FUN` is entered as an input argument in the function `fzero`.

```
>> format long
>> FUN = @ (x) 8-4.5*(x-sin(x))
FUN =
    @(x)8-4.5*(x-sin(x))
>> sol=fzero(FUN,2)
sol =
   2.430465741723630
```

$f(x)$ is written as an anonymous function.

The name `FUN` of the anonymous function is entered in `fzero`.

3.8.2 The `roots` Command

The `roots` command can be used to find the roots of a polynomial. The form of the command is:

`r = roots(p)`

`r` is a column vector with the roots of the polynomial.

`p` is a row vector with the coefficients of the polynomial.

3.9 EQUATIONS WITH MULTIPLE SOLUTIONS

Many nonlinear equations of the form $f(x) = 0$ have multiple solutions or roots. As an example, consider the following equation:

$$f(x) = \cos(x)\cosh(x) + 1 \tag{3.36}$$

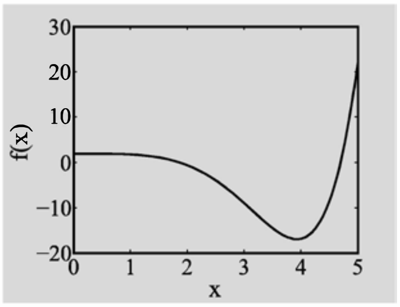


Figure 3-21: A plot of Eq. (3.36).

A plot of this function using MATLAB is shown in Fig. 3-21 over the interval $[0, 5]$. As can be seen, the function has zero crossings between $x = 1$ and $x = 2$, and between $x = 4$ and $x = 5$. Existence of multiple roots is typical of nonlinear equations. A general strategy for finding the roots in the interval $[0, 5]$ is:

- Determine the approximate location of the roots by defining smaller intervals over which the roots exist. This can be done by plotting the function (as shown in Fig. 3-21) or by evaluating the function over a set of successive points and looking for sign changes of the function.
- Apply any of the methods described in Sections 3.3 through 3.7 over a restricted subinterval. For example, the first root that is contained within the interval $[1, 2]$ can be found by the bisection method or a similar method with $a = 1$ and $b = 2$. Alternatively, a starting value or initial guess can be used with Newton's method or fixed-point iteration method to determine the root. The `fzero` MATLAB built-in function can also be used to find the root. The process can then be repeated over the next interval $[4, 5]$ to find the next root.

The next example presents the solution of the function in Eq. (3.36) in a practical situation.

Example 3-4: Solution of equation with multiple roots.

The natural frequencies, ω_n, of free vibration of a cantilever beam are determined from the roots of the equation:

$$f(k_nL) = \cos(k_nL)\cosh(k_nL) + 1 = 0 \tag{3.37}$$

where L is the length of the beam, and the frequency ω_n is given by:

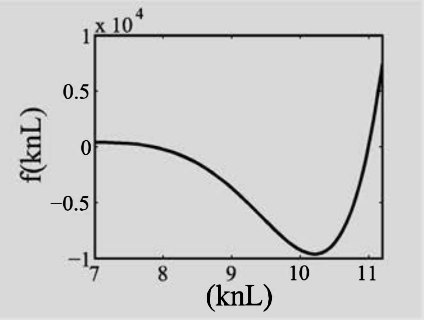
$$\omega_n = (k_nL)^2\sqrt{\frac{EI}{\rho AL^4}}$$

in which E is the elastic modulus, I is the moment of inertia, A is the cross-sectional area, and ρ is the density per unit length.

(*a*) Determine the value of the first root by defining smaller intervals over which the roots exist and using the secant method.

(*b*) Write a MATLAB program in a script file that determines the value of k_nL for the first four roots.

SOLUTION



Equation (3.37) is identical to Eq. (3.36), and a plot that shows the location of the first two roots is presented in Fig. 3-21. The location of the next two roots is shown in the figure on the right where the function is plotted over the interval $[7, 11.2]$. It shows that the third root is around 8 and the fourth root is near 11.

(*a*) The value of the first root is determined by using the `SecantRoot` user-defined function that is listed in Fig. 3-16. First, however, a user-defined function for the function $f(k_nL)$ in Eq. (3.37) is written (the function name is `FunExample3`):

```
function y = FunExample3(x)
y = cos(x)*cosh(x) + 1;
```

The first root is between 1 and 2. To find its solution numerically, the user-defined function `SecantRoot` (listed in Fig. 3.16) is used with $a = 1$, $b = 2$, $Err = 0.0001$, and $imax = 10$:

```
>> FirstSolution = SecantRoot(@FunExample3,1,2,0.0001,10)
FirstSolution =
   1.875104064602412
```

`format long` is used in MATLAB.

(*b*) Next, a MATLAB program that automatically finds the four roots is written. The program evaluates the function over a set of successive intervals and looks for sign changes. It starts at $k_nL = 0$ and uses an increment of 0.2 up to a value of $k_nL = 11.2$. If a change in sign is detected, the root within that interval is determined by using MATLAB's built-in `fzero` function.

```
clear all
F = @ (x) cos(x)*cosh(x)+1;
Inc = 0.2;
```

```
i = 1;
KnLa = 0;                                 Define the left point of the first increment.
KnLb = KnLa + Inc;                        Define the right point of the first increment.
while KnLb <= 11.2
  if F(KnLa)*F(KnLb) < 0                  Check for a sign change in the value of the function.
      Roots(i) = fzero(F,[KnLa,KnLb]);    Determine the root within the interval if a sign
      i = i + 1;                          change was detected.
  end
  KnLa = KnLb;                            Define the left point of the next increment.
  KnLb = KnLa + Inc;                      Define the right point of the next increment.
end
Roots
```

When the program is executed, the display in the Command Window is:

```
Roots =
 1.875104068711961  4.694091132974175  7.854757438237613 10.995540734875467
```

These are the values of the first four roots.

3.10 SYSTEMS OF NONLINEAR EQUATIONS

A system of nonlinear equations consists of two, or more, nonlinear equations that have to be solved simultaneously. For example, Fig. 3-22 shows a catenary (hanging cable) curve given by the equation $y = \frac{1}{2}(e^{x/2} + e^{(-x)/2})$ and an ellipse specified by the equation $\frac{x^2}{5^2} + \frac{y^2}{3^2} = 1$. The point of intersection between the two curves is given by the solution of the following system of nonlinear equations:

$$f_1(x, y) = y - \frac{1}{2}(e^{x/2} + e^{(-x)/2}) = 0 \qquad (3.38)$$

$$f_2(x, y) = 9x^2 + 25y^2 - 225 = 0 \qquad (3.39)$$

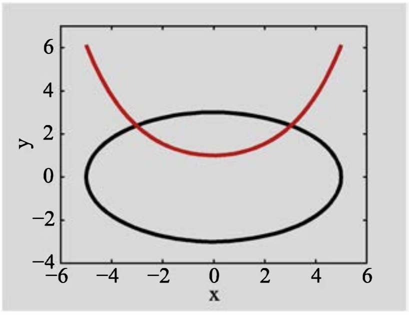


Figure 3-22: A plot of Eq. (3.38) and Eq. (3.39).

Analysis of many problems in science and engineering requires solution of systems of nonlinear equations. In addition, as shown in Chapter 11, one of the popular numerical methods for solving nonlinear ordinary differential equations (the finite difference method) requires the solution of a system of nonlinear algebraic equations.

In this section, two methods for solving systems of nonlinear equations are presented. The Newton method (also called the Newton–Raphson method), suitable for solving small systems, is described in Section 3.10.1. The fixed-point iteration method, which can also be used for solving large systems, is discussed in Section 3.10.2.

3.10.1 Newton's Method for Solving a System of Nonlinear Equations

Newton's method for solving a system of nonlinear equations is an extension of the method used for solving a single equation (Section 3.5). The method is first derived in detail for the solution of a system of two nonlinear equations. Subsequently, a general formulation is presented for the case of a system of n nonlinear equations.

Solving a system of two nonlinear equations

A system of two equations with two unknowns x and y can be written as:

$$\begin{aligned} f_1(x, y) &= 0 \\ f_2(x, y) &= 0 \end{aligned} \tag{3.40}$$

The solution process starts by choosing an estimated solution x_1 and y_1. If x_2 and y_2 are the true (unknown) solutions of the system and are sufficiently close to x_1 and y_1, then the values of f_1 and f_2 at x_2 and y_2 can be expressed using a Taylor series expansion of the functions $f_1(x, y)$ and $f_2(x, y)$ about (x_1, y_1) (see Section 2.7.2):

$$f_1(x_2, y_2) = f_1(x_1, y_1) + (x_2 - x_1)\left.\frac{\partial f_1}{\partial x}\right|_{x_1, y_1} + (y_2 - y_1)\left.\frac{\partial f_1}{\partial y}\right|_{x_1, y_1} + \ldots \tag{3.41}$$

$$f_2(x_2, y_2) = f_2(x_1, y_1) + (x_2 - x_1)\left.\frac{\partial f_2}{\partial x}\right|_{x_1, y_1} + (y_2 - y_1)\left.\frac{\partial f_2}{\partial y}\right|_{x_1, y_1} + \ldots \tag{3.42}$$

Since x_2 and y_2 are close to x_1 and y_1, approximate values for $f_1(x_2, y_2)$ and $f_2(x_2, y_2)$ can be calculated by neglecting the higher-order terms. Also, since $f_1(x_2, y_2) = 0$ and $f_2(x_2, y_2) = 0$, Eqs. (3.41) and (3.42) can be rewritten as:

$$\left.\frac{\partial f_1}{\partial x}\right|_{x_1, y_1} \Delta x + \left.\frac{\partial f_1}{\partial y}\right|_{x_1, y_1} \Delta y = -f_1(x_1, y_1) \tag{3.43}$$

$$\left.\frac{\partial f_2}{\partial x}\right|_{x_1, y_1} \Delta x + \left.\frac{\partial f_2}{\partial y}\right|_{x_1, y_1} \Delta y = -f_2(x_1, y_1) \tag{3.44}$$

where $\Delta x = x_2 - x_1$ and $\Delta y = y_2 - y_1$. Since all the terms in Eqs. (3.43) and (3.44) are known, except the unknowns Δx and Δy, these equations are a system of two linear equations. The system can be solved by using Cramer's rule (see Section 2.4.6):

$$\Delta x = \frac{-f_1(x_1, y_1)\left.\frac{\partial f_2}{\partial y}\right|_{x_1, y_1} + f_2(x_1, y_1)\left.\frac{\partial f_1}{\partial y}\right|_{x_1, y_1}}{J(f_1(x_1, y_1), f_2(x_1, y_1))} \tag{3.45}$$

$$\Delta y = \frac{-f_2(x_1, y_1)\left.\frac{\partial f_1(x_1, y_1)}{\partial x}\right|_{x_1, y_1} + f_1(x_1, y_1)\left.\frac{\partial f_2}{\partial x}\right|_{x_1, y}}{J(f_1(x_1, y_1), f_2(x_1, y_1))} \tag{3.46}$$

where

$$J(f_1, f_2) = det\begin{bmatrix} \frac{\partial f_1}{\partial x} & \frac{\partial f_1}{\partial y} \\ \frac{\partial f_2}{\partial x} & \frac{\partial f_2}{\partial y} \end{bmatrix} \tag{3.47}$$

is the Jacobian (see Section 2.6.3). Once Δx and Δy are known, the values of x_2 and y_2 are calculated by:

$$\begin{aligned} x_2 &= x_1 + \Delta x \\ y_2 &= y_1 + \Delta y \end{aligned} \tag{3.48}$$

Obviously, the values of x_2 and y_2 that are obtained are not the true solution since the higher-order terms in Eqs. (3.41) and (3.42) were neglected. Nevertheless, these values are expected to be closer to the true solution than x_1 and y_1.

The solution process continues by using x_2 and y_2 as the new estimates for the solution and using Eqs. (3.43) and (3.44) to determine new Δx and Δy that give x_3 and y_3. The iterations continue until two successive answers differ by an amount smaller than a desired value.

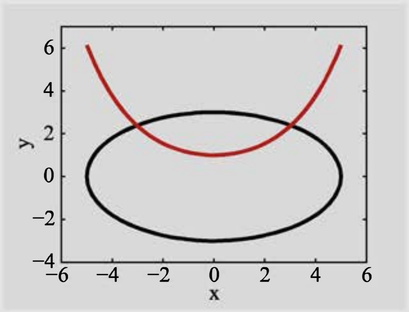
An application of Newton's method is illustrated in Example 3-5 where the intersection point between the catenary curve and the ellipse in Fig. 3-22 is determined.

Example 3-5: Solution of a system of nonlinear equations using Newton's method.

The equations of the catenary curve and the ellipse, which are shown in the figure, are given by:

$$f_1(x, y) = y - \frac{1}{2}(e^{x/2} + e^{(-x)/2}) = 0 \tag{3.49}$$

$$f_2(x, y) = 9x^2 + 25y^2 - 225 = 0 \tag{3.50}$$



Use Newton's method to determine the point of intersection of the curves that resides in the first quadrant of the coordinate system.

SOLUTION

Equations (3.49) and (3.50) are a system of two nonlinear equations. The points of intersection are given by the solution of the system. The solution with Newton's method is obtained by using Eqs. (3.43) and (3.44). In the present problem, the partial derivatives in the equations are given by:

$$\frac{\partial f_1}{\partial x} = -\frac{1}{4}(e^{x/2} - e^{(-x)/2}) \quad \text{and} \quad \frac{\partial f_1}{\partial y} = 1 \tag{3.51}$$

$$\frac{\partial f_2}{\partial x} = 18x \quad \text{and} \quad \frac{\partial f_2}{\partial y} = 50y \tag{3.52}$$

The Jacobian is given by:

$$J(f_1, f_2) = det\begin{bmatrix} \frac{\partial f_1}{\partial x} & \frac{\partial f_1}{\partial y} \\ \frac{\partial f_2}{\partial x} & \frac{\partial f_2}{\partial y} \end{bmatrix} = \det\begin{bmatrix} -\frac{1}{4}(e^{x/2} - e^{(-x)/2}) & 1 \\ 18x & 50y \end{bmatrix} = -\frac{1}{4}(e^{x/2} - e^{(-x)/2})50y - 18x \tag{3.53}$$

Substituting Eqs. (3.51)–(3.53) in Eqs. (3.45) and (3.46) gives the solution for Δx and Δy.
The problem is solved in the MATLAB program that is listed below. The order of operations in the program is:

- The solution is initiated by the initial guess, $x_i = 2.5$, $y_i = 2.0$.
- The iterations start. Δy and Δx are determined by substituting x_i and y_i in Eqs. (3.45) and (3.46).
- $x_{i+1} = x_i + \Delta x$, and $y_{i+1} = y_i + \Delta y$ are determined.
- If the estimated relative error (Eq. (3.9)) for both variables is smaller than 0.001, the iterations stop. Otherwise, the values of x_{i+1} and y_{i+1} are assigned to x_i and y_i, respectively, and the next iteration starts.

The program also displays the solution and the error at each iteration.

```
%   Solution of Chapter 3 Example 5
F1 = @ (x,y) y - 0.5*(exp(x/2) + exp(-x/2));
F2 = @ (x,y) 9*x^2 + 25*y^2 - 225;
F1x = @ (x) -(exp(x/2) - exp(-x/2))/4;
F2x = @ (x) 18*x;
F2y = @ (y) 50*y;
Jacob = @ (x,y) -(exp(x/2) - exp(-x/2))/4*50*y - 18*x;
xi = 2.5; yi = 2; Err = 0.001;
for i = 1:5
   Jac = Jacob(xi,yi);
   Delx = (-F1(xi,yi)*F2y(yi) + F2(xi,yi))/Jac;
   Dely = (-F2(xi,yi)*F1x(xi) + F1(xi,yi)*F2x(xi))/Jac;
   xip1 = xi + Delx;
   yip1 = yi + Dely;
   Errx = abs((xip1 - xi)/xi);
   Erry = abs((yip1 - yi)/yi);
   fprintf('i =%2.0f x = %-7.4f y = %-7.4f Error in x = %-7.4f Error in y = %-
7.4f\n',i,xip1,yip1,Errx,Erry)
   if Errx < Err & Erry < Err
       break
   else
       xi = xip1; yi = yip1;
   end
end
```

Assign the initial estimate of the solution.

Start the iterations.

Calculate Δx and Δy with Eqs. (3.45) and (3.46).

Calculate x_{i+1} and y_{i+1}.

If the error is not small enough, assign x_{i+1} to x_i, and y_{i+1} to y_i.

When the program is executed, the display in the Command Window is:

```
i = 1 x = 3.1388  y = 2.4001  Error in x = 0.25551 Error in y = 0.20003
i = 2 x = 3.0339  y = 2.3855  Error in x = 0.03340 Error in y = 0.00607
i = 3 x = 3.0312  y = 2.3859  Error in x = 0.00091 Error in y = 0.00016
```

These results show that the values converge quickly to the solution.

Solving a system of n nonlinear equations

Newton's method can easily be generalized to the case of a system of *n* nonlinear equations. With *n* unknowns, $x_1, x_2, \ldots, x_n$, a system of *n* simultaneous nonlinear equations has the form:

$$\begin{aligned} f_1(x_1, x_2, \ldots, x_n) &= 0 \\ f_2(x_1, x_2, \ldots, x_n) &= 0 \\ &\ldots \\ f_n(x_1, x_2, \ldots, x_n) &= 0 \end{aligned} \tag{3.54}$$

The value of the functions at the next approximation of the solution, $x_{1,\,i+1}, x_{2,\,i+1}, \ldots, x_{n,\,i+1}$, is then obtained using a Taylor series expansion about the current value of the approximation of the solution, $x_{1,\,i}, x_{2,\,i}, \ldots, x_{n,\,i}$. Following the same procedure that led to Eqs. (3.43) and (3.44) results in the following system of *n* linear equations for the unknowns $\Delta x_1, \Delta x_2, \ldots, \Delta x_n$:

$$\begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix} \begin{bmatrix} \Delta x_1 \\ \Delta x_2 \\ \cdots \\ \Delta x_n \end{bmatrix} = \begin{bmatrix} -f_1 \\ -f_2 \\ \cdots \\ -f_n \end{bmatrix} \tag{3.55}$$

(The determinant of the matrix of the partial derivatives of the functions on the left-hand side of the equation is called the Jacobian, Section 2.6.3.) Once the system in Eq. (3.55) is solved, the new approximate solution is obtained from:

$$\begin{aligned} x_{1,\,i+1} &= x_{1,\,i} + \Delta x_1 \\ x_{2,\,i+1} &= x_{2,\,i} + \Delta x_2 \\ &\ldots \\ x_{n,\,i+1} &= x_{n,\,i} + \Delta x_n \end{aligned} \tag{3.56}$$

As with Newton's method for a single nonlinear equation, convergence is not guaranteed. Newton's iterative procedure for solving a system of nonlinear equations will likely converge provided the following three conditions are met:

(*i*) The functions $f_1, f_2, \ldots, f_n$ and their derivatives must be continuous and bounded near the solution (root).

(*ii*) The Jacobian must be nonzero, that is, $J(f_1, f_2, \ldots, f_n) \neq 0$, near the solution.

(*iii*) The initial estimate (guess) of the solution must be sufficiently close to the true solution.

Newton's method for solving a system of n nonlinear equations is summarized in the following algorithm:

Algorithm for Newton's method for solving a system of nonlinear equations

Given a system of n nonlinear equations,

1. Estimate (guess) an initial solution, $x_{1,i}, x_{2,i}, \ldots, x_{n,i}$.

2. Calculate the Jacobian and the value of the fs on the right-hand side of Eq. (3.55).

3. Solve Eq. (3.55) for $\Delta x_1, \Delta x_2, \ldots, \Delta x_n$.

4. Calculate a new estimate of the solution, $x_{1,i+1}, x_{2,i+1}, \ldots, x_{n,i+1}$, using Eq. (3.56).

5. Calculate the error. If the new solution is not accurate enough, assign the values of $x_{1,i+1}, x_{2,i+1}, \ldots, x_{n,i+1}$ to $x_{1,i}, x_{2,i}, \ldots, x_{n,i}$, and start a new iteration beginning with Step *2*.

Additional comments on Newton's method for solving a system of nonlinear equations

- The method, when successful, converges fast. When it does not converge, it is usually because the initial guess is not close enough to the solution.
- The partial derivatives (the elements of the Jacobian matrix) have to be determined. This can be done analytically or numerically (numerical differentiation is covered in Chapter 8). However, for a large system of equations, the determination of the Jacobian might be difficult.
- When the system of equations consists of more than three equations, the solution of Eq. (3.55) has to be done numerically. Methods for solving systems of linear equations are described in Chapter 4.

3.10.2 Fixed-Point Iteration Method for Solving a System of Nonlinear Equations

The fixed-point iteration method discussed in Section 3.7 for solving a single nonlinear equation can be extended to the case of a system of nonlinear equations. A system of n nonlinear equations with the unknowns, $x_1, x_2, \ldots, x_n$, has the form:

$$\begin{aligned} f_1(x_1, x_2, \ldots, x_n) &= 0 \\ f_2(x_1, x_2, \ldots, x_n) &= 0 \\ &\ldots \\ f_n(x_1, x_2, \ldots, x_n) &= 0 \end{aligned} \tag{3.57}$$

The system can be rewritten in the form:

$$\begin{aligned} x_1 &= g_1(x_1, x_2, \ldots, x_n) \\ x_2 &= g_2(x_1, x_2, \ldots, x_n) \\ &\ldots \\ x_n &= g_n(x_1, x_2, \ldots, x_n) \end{aligned} \tag{3.58}$$

where the gs are the iteration functions. The solution process starts by guessing a solution, $x_{1,1}, x_{2,1}, \ldots, x_{n,1}$, which is substituted on the right-hand side of Eqs. (3.58). The values that are calculated by Eqs. (3.58) are the new (second) estimate of the solution, $x_{1,2}, x_{2,i+1}, \ldots, x_{n,i+1}$. The new estimate is substituted back on the right-hand side of Eqs. (3.58) to give a new solution, and so on. When the method works, the new estimates of the solution converge toward the true solution. In this case, the process is continued until the desired accuracy is achieved. For example, the estimated relative error is calculated for each of the variables, and the iterations are stopped when the largest relative error is smaller than a specified value.

Convergence of the method depends on the form of the iteration functions. For a given problem there are many possible forms of iteration functions since rewriting Eqs. (3.57) in the form of Eqs. (3.58) is not unique. In general, several forms might be appropriate for one solution, or in the case where several solutions exist, different iteration functions need to be used to find the multiple solutions. When using the fixed-point iteration method, one can try various forms of iteration functions, or it may be possible in some cases to determine ahead of time if the solution will converge for a specific choice of gs.

The fixed-point iteration method applied to a set of simultaneous nonlinear equations will converge under the following sufficient (but not necessary) conditions:

(*i*) $g_1, \ldots, g_n, \frac{\partial g_1}{\partial x_1}, \ldots, \frac{\partial g_1}{\partial x_n}, \frac{\partial g_2}{\partial x_1}, \ldots, \frac{\partial g_2}{\partial x_n}, \ldots, \frac{\partial g_n}{\partial x_n}$ are continuous in the neighborhood of the solution.

(*ii*)
$$\left|\frac{\partial g_1}{\partial x_1}\right| + \left|\frac{\partial g_1}{\partial x_2}\right| + \ldots + \left|\frac{\partial g_1}{\partial x_n}\right| \le 1$$
$$\left|\frac{\partial g_2}{\partial x_1}\right| + \left|\frac{\partial g_2}{\partial x_2}\right| + \ldots + \left|\frac{\partial g_2}{\partial x_n}\right| \le 1$$
$$\ldots$$
$$\left|\frac{\partial g_n}{\partial x_1}\right| + \left|\frac{\partial g_n}{\partial x_2}\right| + \ldots + \left|\frac{\partial g_n}{\partial x_n}\right| \le 1$$

(*iii*) The initial guess, $x_{1,\,1}, x_{2,\,1}, \ldots, x_{n,\,1}$, is sufficiently close to the solution.

3.11 PROBLEMS

Problems to be solved by hand

Solve the following problems by hand. When needed, use a calculator, or write a MATLAB script file to carry out the calculations. If using MATLAB, do not use built-in functions for solving nonlinear equations.

3.1 The tolerance, tol, of the solution in the bisection method is given by $tol = \frac{1}{2}(b_n - a_n)$, where a_n and b_n are the endpoints of the interval after the *n*th iteration. The number of iterations *n* that are required for obtaining a solution with a tolerance that is equal to or smaller than a specified tolerance can be determined before the solution is calculated. Show that *n* is given by:

$$n \ge \frac{\log(b-a) - \log(tol)}{\log 2}$$

where a and b are the endpoints of the starting interval and tol is a user-specified tolerance.

3.2 Determine the root of $f(x) = x - 2e^{-x}$ by:

(*a*) Using the bisection method. Start with $a = 0$ and $b = 1$, and carry out the first three iterations.

(*b*) Using the secant method. Start with the two points, $x_1 = 0$ and $x_2 = 1$, and carry out the first three iterations.

(*c*) Using Newton's method. Start at $x_1 = 1$ and carry out the first three iterations.

3.3 The location $\bar{x}$ of the centroid of an arc of a circle is given by:

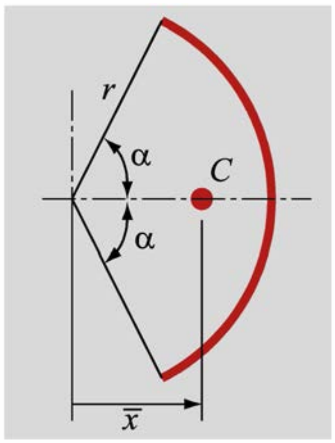
$$\bar{x} = \frac{r\sin\alpha}{\alpha}$$

Determine the angle α for which $\bar{x} = \frac{3r}{4}$.

First, derive the equation that must be solved and then determine the root using the following methods:

(*a*) Use the bisection method. Start with $a = 0.5$ and $b = 1.5$, and carry out the first four iterations.

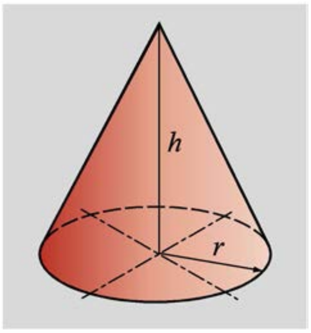
(*b*) Use the secant method. Start with the two points $\alpha_1 = 0.5$ and $\alpha_2 = 1.5$, and carry out the first four iterations.



3.4 The lateral surface area, S, of a cone is given by:

$$S = \pi r \sqrt{r^2 + h^2}$$

where r is the radius of the base and h is the height. Determine the radius of a cone that has a surface area of 1800 m^2 and a height of 25 m. Solve by using the fixed-point iteration method with $r = S/(\pi\sqrt{r^2 + h^2})$ as the iteration function. Start with $r = 17$ m and calculate the first four iterations.



3.5 Determine the fourth root of 200 by finding the numerical solution of the equation $x^4 - 200 = 0$. Use Newton's method. Start at $x = 8$ and carry out the first five iterations.

3.6 Determine the positive root of the polynomial $x^3 + 0.6x^2 + 5.6 - 4.8$.
(*a*) Plot the polynomial and choose a point near the root for the first estimate of the solution. Using Newton's method, determine the approximate solution in the first four iterations.
(*b*) From the plot in part (*a*), choose two points near the root to start the solution process with the secant method. Determine the approximate solution in the first four iterations.

3.7 The equation $1.2x^3 + 2x^2 - 20x - 10 = 0$ has a root between $x = -4$ and $x = -5$. Use these values for the initial two points and calculate the next four estimates for the solution using the secant method.

3.8 Find the root of the equation $\sqrt{x} + x^2 = 7$ using Newton's method. Start at $x = 7$ and carry out the first five iterations.

3.9 The equation $x^3 - x - e^x - 2 = 0$ has a root between $x = 2$ and $x = 3$.
(*a*) Write four different iteration functions for solving the equation using the fixed-point iteration method.
(*b*) Determine which $g(x)$ from part (*a*) could be used according to the condition in Eq. (3.30).
(*c*) Carry out the first five iterations using the $g(x)$ determined in part (*b*), starting with $x = 2$.

3.10 The equation $f(x) = x^2 - 5x^{1/3} + 1 = 0$ has a root between $x = 2$ and $x = 2.5$. To find the root by using the fixed-point iteration method, the equation has to be written in the form $x = g(x)$. Derive two possible forms for $g(x)$ — one by solving for x from the first term of the equation, and the next by solving for x from the second term of the equation.
(*a*) Determine which form should be used according to the condition in Eq. (3.30).
(*b*) Carry out the first five iterations using both forms of $g(x)$ to confirm your determination in part (*a*).

3.11 The equation $f(x) = 2x^3 - 4x^2 - 4x - 20 = 0$ has a root between $x = 3$ and $x = 4$. Find the root by using the fixed-point iteration method. Determine the appropriate form of $g(x)$ according to Eq. (3.30). Start the iterations with $x = 2.5$ and carry out the first five iterations.

3.12 Determine the positive root of the equation $\cos x - 0.8x^2 = 0$ by using the fixed-point iteration method. Carry out the first five iterations.

3.13 Solve the following system of nonlinear equations:

$$-2x^3 + 3y^2 + 42 = 0$$
$$5x^2 + 3y^3 - 69 = 0$$

(*a*) Use Newton's method. Start at $x = 1,\ y = 1$, and carry out the first five iterations.

(*b*) Use the fixed-point iteration method. Use the iteration functions $y = \left(\frac{-5x^2 + 69}{3}\right)^{1/3}$ and $x = \left(\frac{3y^2 + 42}{2}\right)^{1/3}$. Start at $x = 1,\ y = 1$, and carry out the first five iterations.

3.14 Solve the following system of nonlinear equations:

$$x^2 + 2x + 2y^2 - 26 = 0$$
$$2x^3 - y^2 + 4y - 19 = 0$$

(*a*) Use Newton's method. Start at $x = 1,\ y = 1$, and carry out the first five iterations.
(*b*) Use the fixed-point iteration method. Start at $x = 1,\ y = 1$, and carry out the first five iterations.

Problems to be programmed in MATLAB

Solve the following problems using the MATLAB environment. Do not use MATLAB's built-in functions for solving nonlinear equations.

3.15 In the program of Example 3-1 the iterations are executed in the for-end loop. In the loop, the anonymous function F is used twice (once in the command `FxNS = F(xNS)` and once in the command `if F(a)*FxNS < 0`). Rewrite the program such that the anonymous function F is used inside the loop only once. Execute the new program and show that the output is the same as in the example.

3.16 Write a MATLAB user-defined function that solves for a root of a nonlinear equation $f(x) = 0$ using the bisection method. Name the function `Xs = BisectionRoot(Fun,a,b)`. The output argument `Xs` is the solution. The input argument `Fun` is a name for the function that calculates $f(x)$ for a given x (it is a dummy name for the function that is imported into `BisectionRoot`); `a` and `b` are two points that bracket the root. The iterations should stop when the tolerance in $f(x)$ (Eq. (3.5)) is smaller than 0.000001. The program should check if points `a` and `b` are on opposite sides of the solution. If not, the program should stop and display an error message. Use `BisectionRoot` to solve Problem 3.2.

3.17 Determining the square root of a number p, $\sqrt{p}$, is the same as finding a solution to the equation $f(x) = x^2 - p = 0$. Write a MATLAB user-defined function that determines the square root of a positive number by solving the equation using Newton's method. Name the function `Xs = SquareRoot(p)`. The output argument `Xs` is the answer, and the input argument `p` is the number whose square root is determined. The program should include the following features:

- It should check if the number is positive. If not, the program should stop and display an error message.
- The starting value of x for the iterations should be $x = p$.
- The iterations should stop when the estimated relative error (Eq. (3.9)) is smaller than 1×10^{-6}.
- The number of iterations should be limited to 20. If a solution is not obtained in 20 iterations, the program should stop and display an error message.

Use the function `SquareRoot` to determine the square root of (*a*) 729, (*b*) 1500, and (*c*) –72.

3.18 Determining the natural logarithm of a number p, $\ln p$, is the same as finding a solution to the equation $f(x) = e^x - p = 0$. Write a MATLAB user-defined function that determines the natural logarithm of a number by solving the equation using the bisection method. Name the function `X = Ln(p)`. The output argument `X` is the value of $\ln p$, and the input argument `p` is the number whose natural logarithm is determined. The program should include the following features:

- The starting values of a and b (see Section 3.3) are $a = e^0$ and $b = p$, respectively, if $b > e^1$, and $a = -1/p$ and $b = e^0$, respectively, if $b < e^1$.
- The iterations should stop when the tolerance (Eq. (3.7)) is smaller than 1×10^{-6}.
- The number of iterations should be limited to 100. If a solution is not obtained in 100 iterations, the function stops and displays an error message.
- If zero or a negative number is entered for `p`, the program stops and displays an error message.

Use the function `Ln` to determine the natural logarithm of (*a*) 510, (*b*) 1.35, (*c*) 1, and (*c*) -7.

3.19 A new method for solving a nonlinear equation $f(x) = 0$ is proposed. The method is similar to the bisection method. The solution starts by finding an interval $[a, b]$ that brackets the solution. The first estimate of the solution is the midpoint between $x = a$ and $x = b$. Then the interval $[a, b]$ is divided into four equal sections. The section that contains the root is taken as the new interval for the next iteration.

Write a MATLAB user-defined function that solves a nonlinear equation with the proposed new method. Name the function `Xs = QuadSecRoot(Fun,a,b)`, where the output argument `Xs` is the solution. The input argument `Fun` is a name for the function that calculates $f(x)$ for a given x (it is a dummy name for the function that is imported into `QuadSecRoot`), `a` and `b` are two points that bracket the root. The iterations should stop when the tolerance, Eq. (3.7), is smaller than $10^{-6}x_{NS}$ (x_{NS} is the current estimate of the solution, Eq. (3.6)).

Use the user-defined `QuadSecRoot` function to solve the equations in Problems 3.2 and 3.3. For the initial values of a and b, take the values that are listed in part (*a*) of the problems.

3.20 Write a MATLAB user-defined function that solves a nonlinear equation $f(x) = 0$ with the regula falsi method. Name the function `Xs = RegulaRoot(Fun,a,b,ErrMax)`, where the output argument `Xs` is the solution. The input argument `Fun` is a name for the function that calculates $f(x)$ for a given x (it is a dummy name for the function that is imported into `RegulaRoot`), `a` and `b` are two points that bracket the root, and `ErrMax` the maximum error according to Eq. (3.9).

The program should include the following features:

- Check if points `a` and `b` are on opposite sides of the solution. If not, the program should stop and display an error message.
- The number of iterations should be limited to 100 (to avoid an infinite loop). If a solution with the required accuracy is not obtained in 100 iterations, the program should stop and display an error message.

Use the function `RegulaRoot` to solve the equation in Problem 3.3 (use $a = 0.1$, $b = 1.4$).

3.21 A new method for solving a nonlinear equation $f(x) = 0$ is proposed. The method is a combination of the bisection and the regula falsi methods. The solution starts by defining an interval $[a, b]$ that brackets the solution. Then estimated numerical solutions are determined once with the bisection method and once with the regula falsi method. (The first iteration uses the bisection method.) Write a MATLAB user-defined function that solves a nonlinear equation $f(x) = 0$ with this new method. Name the function `Xs = BiRegRoot(Fun,a,b,ErrMax)`, where the output argument `Xs` is the solution. The input argument `Fun` is a name for the function that calculates $f(x)$ for a given x (it is a dummy name for the function that is imported into `BiRegRoot`), `a` and `b` are two points that bracket the root, and `ErrMax` is the maximum error according to Eq. (3.9).

The program should include the following features:

- Check if points `a` and `b` are on opposite sides of the solution. If not, the program should stop and display an error message.
- The number of iterations should be limited to 100 (to avoid an infinite loop). If a solution with the required accuracy is not obtained in 100 iterations, the program should stop and display an error message.

Use the function `RegulaRoot` to solve the equation in Problem 3.3 (use $a = 0.1$, $b = 1.4$). For `ErrMax` use 0.00001.

3.22 Modify the function `NewtonRoot` that is listed in Fig. 3-11, such that the input will have three arguments. Name the function `Xs = NewtonSol(Fun,FunDer,Xest)`. The output argument `Xs` is the solution, and the input arguments `Fun`, `FunDer`, and `Xest` are the same as in `NewtonRoot`. The iterations should stop when the estimated relative error (Eq. (3.9)) is smaller than 10^{-6}. The number of iterations should be limited to 100 (to avoid an infinite loop). If a solution with the required accuracy is not obtained in 100 iterations, the program should stop and display an error message. Use the function `NewtonSol` to solve the equation that is solved in Example 3-2.

3.23 Modify the function `NewtonRoot` that is listed in Fig. 3-11, such that the output will have three arguments. Name the function `[Xs,FXs,iact] = NewtonRootMod(Fun,FunDer,Xest,Err,imax)`. The first output argument is the solution, the second is the value of the function at the solution, and the third is the actual number of iterations that are performed to obtain the solution. Use the function `NewtonRootMod` to solve the equation that is solved in Example 3-2.

3.24 Steffensen's method is a scheme for finding a numerical solution of an equation of the form $f(x) = 0$ that is similar to Newton's method but does not require the derivative of $f(x)$. The solution process starts by choosing a point x_i, near the solution, as the first estimate of the solution. The next estimates of the solution x_{i+1} are calculated by:

$$x_{i+1} = x_i - \frac{f(x_i)^2}{f(x_i + f(x_i)) - f(x_i)}$$

Write a MATLAB user-defined function that solves a nonlinear equation with Steffensen's method. Name the function `Xs = SteffensenRoot(Fun,Xest)`, where the output argument `Xs` is the numerical solution. The input argument `Fun` is a name for the function that calculates $f(x)$ for a given x (it is a dummy name for the function that is imported into `SteffensenRoot`), and `Xest` is the initial estimate of the solution. The iterations should stop when the estimated relative error (Eq. (3.9)) is smaller than 10^{-6}. The number of iterations should be limited to 100 (to avoid an infinite loop). If a solution with the required accuracy is not obtained in 100 iterations, the program should stop and display an error message.

Use the function `SteffensenRoot` to solve Problems 3.2 and 3.3.

3.25 Write a user-defined MATLAB function that solves for all the real roots in a specified domain of a nonlinear function $f(x) = 0$ using the bisection method. Name the function `R=BisecAll-Roots(fun,a,b,TolMax)`. The output argument `R` is a vector whose elements are the values of the roots. The input argument `Fun` is a name for a function that calculates $f(x)$ for a given x. (It is a dummy name for the function that is imported into `BisecAllRoots`.) The arguments `a` and `b` define the domain, and `TolMax` is the maximum tolerance that is used by the bisection method when the value of each root is calculated. Use the following algorithm:

1. Divide the domain $[a, b]$ into 10 equal subintervals of length h such that $h = (b-a)/10$.
2. Check for a sign change of $f(x)$ at the endpoints of each subinterval.
3. If a sign change is identified in a subinterval, use the bisection method for determining the root in that subinterval.
4. Divide the domain $[a, b]$ into 100 equal subintervals of length h such that $h = (b-a)/100$.
5. Repeat step 2. If a sign change is identified in a subinterval, check if it contains a root that was already obtained. If not, use the bisection method for determining the root in that subinterval.
6. If no new roots have been identified, stop the program.
7. If one or more new roots have been identified, repeat steps 4–6, wherein each repetition the number of subintervals is multiplied by 10.

Use the function `BisecAllRoots`, with `TolMax` value of 0.0001, to find all the roots of the equation $x^4 - 5.5x^3 - 7.2x^2 + 43x + 36 = 0$.

3.26 Examine the differences between the True Relative Error, Eq. (3.8), and the Estimated Relative Error, Eq. (3.9), by numerically solving the equation $f(x) = 0.5e^{(2+x)} - 40 = 0$. The exact solution of the equation is $x = \ln(80) - 2$. Write a MATLAB program in a script file that solves the equation by using Newton's method. Start the iterations at $x = 4$, and execute 11 iterations. In each iteration, calculate the True Relative Error (TRE) and the Estimated Relative Error (ERE). Display the results in a four-column table (create a 2-dimensional array), with the number of iterations in the first column, the estimated numerical solution in the second, and TRE and ERE in the third and fourth columns, respectively.

Problems in math, science, and engineering

Solve the following problems using the MATLAB environment. As stated, use the MATLAB programs that are presented in the chapter, programs developed in previously solved problems, or MATLAB's built-in functions.

3.27 When calculating the payment of a mortgage, the relationship between the loan amount, *Loan*, the monthly payment, *MPay*, the duration of the loan in months *Months*, and the annual interest rate, *Rate*, is given by the equation (annuity equation):

$$MPay = \frac{Loan \cdot Rate}{12\left(1 - \dfrac{1}{\left(1 + \dfrac{Rate}{12}\right)^{Months}}\right)}$$

Determine the rate of a 20 years, \$300,000 loan if the monthly payment is \$1684.57.

(*a*) Use the user-defined function `SteffensenRoot` from Problem 3.24.

(*b*) Use MATLAB's built-in function `fzero`.

3.28 The operation of Resistance Temperature Detector (RTD) is based on the fact that the electrical resistance of the detector material changes with temperature. For Nickel which is sometimes used in such detectors, the resistance, R_T, at temperature T (°C) as a function of temperature is given by:

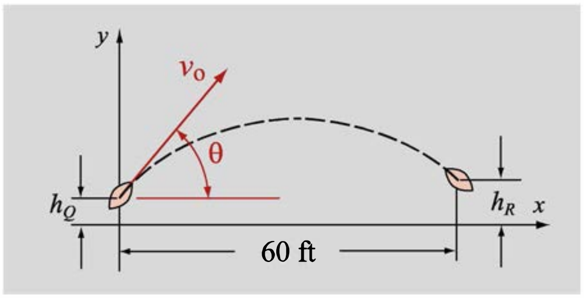
$$R_T = R_0(1 + AT + BT^2 + CT^4 + DT^6)$$

where R_0 is the resistance of the detector at 0°C and $A = 5.485 \times 10^{-3}$, $B = 6.65 \times 10^{-6}$, $C = 2.805 \times 10^{-11}$, and $D = -2 \times 10^{-17}$ are constants. Consider a detector with $R_0 = 100\Omega$ and determine the temperature when its resistance is $300\,\Omega$.

(*a*) Use the user-defined function `NewtonSol` given in Problem 3.22.

(*b*) Use MATLAB's built-in `fzero` function.

3.29 A quarterback throws a pass to his wide receiver running a route. The quarterback releases the ball at a height of h_Q. The wide receiver is supposed to catch the ball straight down the field 60 ft away at a height of h_R. The equation that describes the motion of the football is the familiar equation of projectile motion from physics:



$$y = x\tan(\theta) - \frac{1}{2}\frac{x^2 g}{v_o^2}\frac{1}{\cos^2(\theta)} + h_Q$$

where x and y are the horizontal and vertical distance, respectively, $g = 32.2$ ft/s^2 is the acceleration due to gravity, v_o is the initial velocity of the football as it leaves the quarterback's hand, and θ is the angle the football makes with the horizontal just as it leaves the quarterback's throwing hand. For $v_o = 50$ ft/s, $x = 60$ ft, $h_Q = 6.5$ ft, and $h_R = 7$ ft, find the angle θ at which the quarterback must launch the ball.

(*a*) Use the user-defined function `BisectionRoot` that was developed in Problem 3.16.

(*b*) Use MATLAB built-in function `fzero`.

3.30 The van der Waals equation gives a relationship between the pressure P (in atm.), volume V (in L), and temperature T (in K) for a real gas:

$$P = \frac{nRT}{V - nb} - \frac{n^2 a}{V^2}$$

where n is the number of moles, $R = 0.08206$ (L atm)/(mole K) is the gas constant, and a (in L^2 atm/mole^2) and b (in L/mole) are material constants.

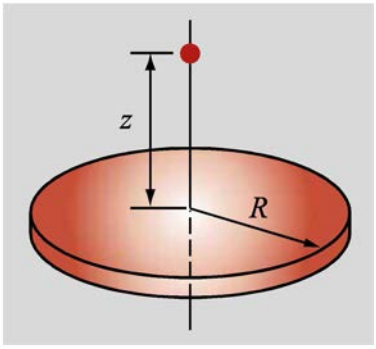
Consider 1.5 moles of nitrogen ($a = 1.39$ L^{2}atm/mole^2, $b = 0.03913$ L/mole) at 25° C stored in a pressure vessel. Determine the volume of the vessel if the pressure is 13.5 atm.

(*a*) Use the user-defined function `BisectionRoot` given in Problem 3.16.

(*b*) Use the user-defined function `SecantRoot` given in Program 3-3. Use 0.0001 for `Err`.

(*c*) Use MATLAB's built-in `fzero` function.

3.31 The force F acting between a particle with a charge q and a round disk with a radius R and a charge Q is given by the equation:



$$F = \frac{Qqz}{2\varepsilon_0}\left(1 - \frac{z}{\sqrt{z^2 + R^2}}\right)$$

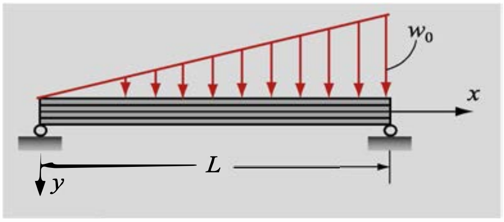
where $\varepsilon_0 = 0.885 \times 10^{-12}$ $\mathrm{C^2/(Nm^2)}$ is the permittivity constant and z is the distance to the particle. Determine the distance z if $F = 0.3\,\mathrm{N}$, $Q = 9.4 \times 10^{-6}$ C, and $q = 2.4 \times 10^{-5}$ C, and $R = 0.1$ m.

(*a*) Use the user-defined function `BisectionRoot` that was developed in Problem 3.16 with a starting interval of `[0.1, 0.2]`.

(*b*) Use the user-defined function `SteffensenRoot` from Problem 3.24.

(*c*) Use MATLAB's built-in function `fzero`.

3.32 A simply supported I-beam is loaded with a distributed load, as shown. The deflection, y, of the center line of the beam as a function of the position, x, is given by the equation:



$$y = \frac{w_0 x}{360LEI}(7L^4 - 10L^2x^2 + 3x^4)$$

where $L = 4$ m is the length, $E = 70$ GPa is the elastic modulus, $I = 52.9 \times 10^{-6}\ \mathrm{m}^4$ is the moment of inertia, and $w_0 = 20$ kN/m.

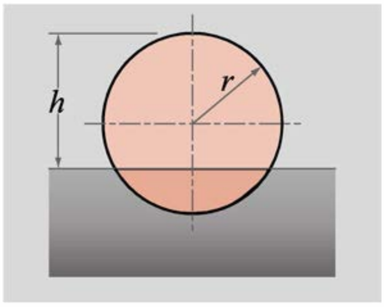
Find the position x where the deflection of the beam is maximum, and determine the deflection at this point. (The maximum deflection is at the point where $\frac{dy}{dx} = 0$.)

(*a*) Use the user-defined function `NewtonSol` given in Problem 3.22.

(*b*) Use the user-defined function `SecantRoot` given in Program 3-3. Use 0.0001 for `Err`, 1.5 for `Xa`. and 2.5 for `Xb`.

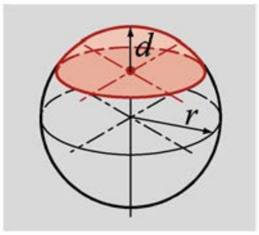
(*c*) Use MATLAB's built-in `fzero` function.

3.33 According to Archimedes' principle, the buoyancy force acting on an object that is partially immersed in a fluid is equal to the weight that is displaced by the portion of the object that is submerged.



A spherical float with a mass of $m_f = 70$kg and a diameter of 90 cm is placed in the ocean (density of sea water is approximately $\rho = 1030$ kg/m^3. The height, h, of the portion of the float that is above water can be determined by solving an equation that equates the mass of the float to the mass of the water that is displaced by the portion of the float that is submerged:

$$\rho V_{cap} = m_f \tag{3.59}$$



where, for a sphere of radius r, the volume of a cap of depth d is given by:

$$V_{cap} = \frac{1}{3}\pi d^2(3r - d)$$

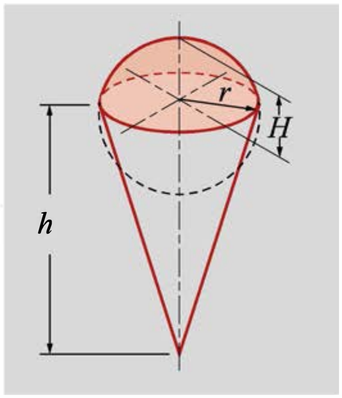
Write Eq. (3.59) in terms of h and solve for h.

(*a*) Use the user-defined function `NewtonRoot` given in Program 3-2. Use 0.0001 for `Err`, and 0.8 for `Xest`.
(*b*) Use MATLAB's built-in `fzero` function.

3.34 An ice cream drum is made of a waffle cone filled with ice cream such that the ice cream above the cone forms a spherical cap. The volume of the ice cream is given by:

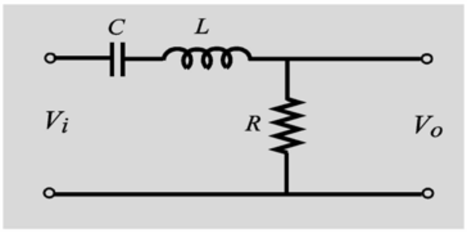
$$V = \pi\left(\frac{r^2h}{3} + \frac{r^2H}{2} + \frac{H^3}{6}\right)$$

Determine H if $V = 1/3$ U.S. pint (1 U.S pint = 28.875 in^3), $h = 4$ in., $r = 1.1$ in.
(*a*) Use the user-defined function `NewtonSol` given in Problem 3.22.
(*b*) Use the user-defined function `SteffensenRoot` from Problem 3.24.
(*c*) Use MATLAB's built-in `fzero` function.



3.35 A bandpass filter passes signals with frequencies that are within a certain range. In this filter the ratio of the magnitudes of the voltages is given by

$$RV = \left|\frac{V_o}{V_i}\right| = \frac{\omega RC}{\sqrt{(1-\omega^2 LC)^2 + (\omega RC)^2}}$$



where ω is the frequency of the input signal. Given $R = 1000\ \Omega$, $L = 11$ mH, and $C = 8\ \mu$F, determine the frequency range that corresponds to $RV \geq 0.87$.
(*a*) Use the user-defined function `BisectionRoot` that was developed in Problem 3.16.
(*b*) Use the user-defined function `SteffensenRoot` from Problem 3.24.
(*c*) Use MATLAB's built-in function `fzero`.

3.36 Determining the value of $65/17$ is the same as calculating the root of the function $f(x) = 17x - 65$. Determine the root, to accuracy of five decimal points, with the bisection method. Use the user-defined function `BisectionRoot` from problem 3.16. Compare the result with the value calculated with a calculator.

3.37 The power output of a solar cell varies with the voltage it puts out. The voltage V_{mp} at which the output power is maximum is given by the equation:

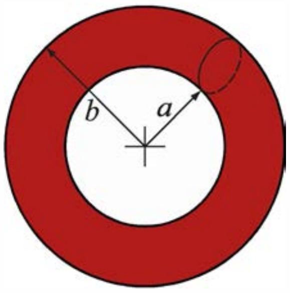
$$e^{(qV_{mp}/k_BT)}\left(1 + \frac{qV_{mp}}{k_BT}\right) = e^{(qV_{OC}/k_BT)}$$

where V_{OC} is the open circuit voltage, T is the temperature in Kelvin, $q = 1.6022 \times 10^{-19}$ C is the charge on an electron, and $k_B = 1.3806 \times 10^{-23}$ J/k is Boltzmann's constant. For $V_{OC} = 0.5$ V and room temperature ($T = 297$ K), determine the voltage V_{mp} at which the power output of the solar cell is a maximum.
(*a*) Write a program in a script file that uses the fixed-point iteration method to find the root. For starting point, use $V_{mp} = 0.5$ V. To terminate the iterations, use Eq. (3.18) with $\varepsilon = 0.001$.
(b) Use MATLAB's `fzero` built-in function.

3.38 The volume V of a torus-shaped water tube is given by:

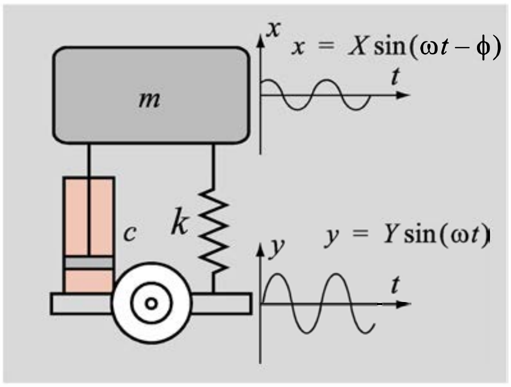
$$V = \frac{1}{4}\pi^2(r_1 + r_2)(r_2 - r_1)^2$$

where r_1 and r_2 are the inside and outside radii, respectively, as shown in the figure. Determine r_1 if $V = 2500\,\text{in}^2$ and $r_2 = 18$ in.



(*a*) Use the user-defined function `NewtonSol` given in Problem 3.22.
(*b*) Use the user-defined function `SteffensenRoot` from Problem 3.24.
(*c*) Use MATLAB's built-in `fzero` function.

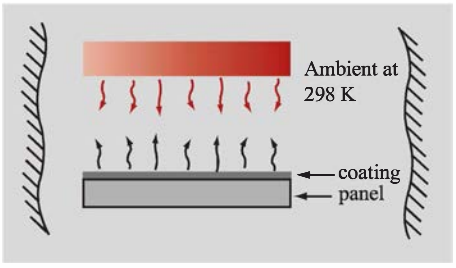
3.39 A simplified model of the suspension of a car consists of a mass, *m*, a spring with stiffness, *k*, and a dashpot with damping coefficient, *c*, as shown in the figure. A bumpy road can be modeled by a sinusoidal up-and-down motion of the wheel $y = Y\sin(\omega t)$. From the solution of the equation of motion for this model, the steady-state up-and-down motion of the car (mass) is given by $x = X\sin(\omega t - \phi)$. The ratio between amplitude X and amplitude Y is given by:



$$\frac{X}{Y} = \sqrt{\frac{\omega^2 c + k^2}{(k - m\omega^2) + (\omega c)^2}}$$

Assuming $m = 2500$ kg, $k = 300$ kN/m, and $c = 36 \times 10^3$ N-s/m, determine the frequency ω for which $X/Y = 0.4$. Rewrite the equation such that it is in the form of a polynomial in ω and solve.
(*a*) Use the user-defined function `BisectionRoot` that was developed in Problem 3.16.
(*b*) Use MATLAB's built-in `fzero` function.

3.40 A coating on a panel surface is cured by radiant energy from a heater. The temperature of the coating is determined by radiative and convective heat transfer processes. If the radiation is treated as diffuse and gray, the following nonlinear system of simultaneous equations determine the unknowns J_h, T_h, J_c, T_c:



$$5.67 \times 10^{-8} T_c^4 + 17.41 T_c - J_c = 5188.18$$
$$J_c - 0.71 J_h + 7.46 T_c = 2352.71$$
$$5.67 \times 10^{-8} T_h^4 + 1.865 T_h - J_h = 2250$$
$$J_h - 0.71 J_c + 7.46 T_h = 11093$$

where J_h and J_c are the radiosities of the heater and coating surfaces, respectively, and T_h and T_c are the respective temperatures.
(*a*) Show that the following iteration functions can be used for solving the nonlinear system of equations with the fixed-point iteration method:

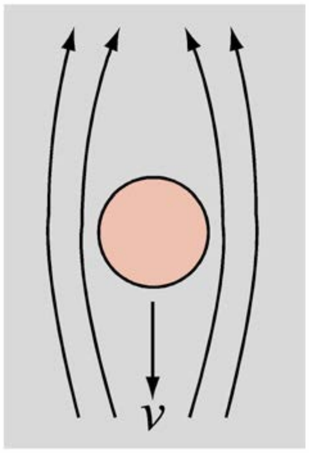
$$T_c = \left[\frac{J_c - 17.41 T_c + 5188.18}{5.67 \times 10^{-8}}\right]^{1/4} \qquad T_h = \left[\frac{2250 + J_h - 1.865 T_h}{5.67 \times 10^{-8}}\right]^{1/4}$$

$$J_c = 2352.71 + 0.71 J_h - 7.46 T_c \qquad J_h = 11093 + 0.71 J_c - 7.46 T_h$$

(*b*) Solve the nonlinear system of equations with the fixed-point iteration method using the iteration functions from part (*a*). Use the following initial values: $T_h = T_c = 298$ K, $J_c = 3000\ \text{W/m}^2$, and $J_h = 5000\ \text{W/m}^2$. Carry out 100 iterations, and plot the respective values to observe their convergence. The final answers should be $T_c = 481$ K, $J_c = 6222\ \text{W/m}^2$, $T_h = 671$ K, $J_h = 10504\ \text{W/m}^2$.

3.41 If a basketball is dropped from a helicopter, its velocity as a function of time $v(t)$ can be modeled by the equation:

$$v(t) = \sqrt{\frac{2mg}{\rho A C_d}}\left(1 - e^{-\sqrt{\frac{\rho g C_d A}{2m}}\,t}\right)$$



where $g = 9.81\ \text{m/s}^2$ is the gravitation of the Earth, $C_d = 0.5$ is the drag coefficient, $\rho = 1.2\ \text{kg/m}^3$ is the density of air, m is the mass of the basketball in kg, and $A = \pi r^2$ is the projected area of the ball ($r = 0.117$ m is the radius). Note that initially the velocity increases rapidly, but then due to the resistance of the air, the velocity increases more gradually. Eventually the velocity approaches a limit that is called the terminal velocity. Determine the mass of the ball if at $t = 5$ s the velocity of the ball was measured to be 19.5 m/s.

(*a*) Use the user-defined function `SecantRoot` given in Program 3-3. Use 0.0001 for `Err`.

(*b*) MATLAB's built-in `fzero` function.