Detection is, or ought to be, an exact science, and should be treated in the same cold and unemotional manner. You have attempted to tinge it with romanticism, which produces much the same effect as if you worked a love-story or an elopement into the fifth proposition of Euclid.

— SIR ARTHUR CONAN DOYLE

Chapter 2

Single nonlinear algebraic equation

2.1 Introduction

In this chapter we consider the problem of finding the root of a single nonlinear algebraic *equation* of the form,

$$f(x) = 0 \tag{2.1}$$

where $f(x)$ is a continuous function of x and the equation is satisfied only at selected values of $x = r$, called the *roots*. The equation can be a simple polynomial as we saw with the Peng-Robinson equation of state in section 1.3.4 or a more complicated function as in the multicomponent flash example discussed in section 1.3.3. If the equation depends on other parameters, as is often the case, we will represent them as

$$f(x; p) = 0 \tag{2.2}$$

where p represents a set of known parameter values. Clearly, we can graph the function f vs. x for a range of values of x. The objective of such an exercise is to graphically locate the values of $x = r$ where the function crosses the x axis. While such a graphical approach has an intuitive appeal, it is difficult to generalize such methods to higher

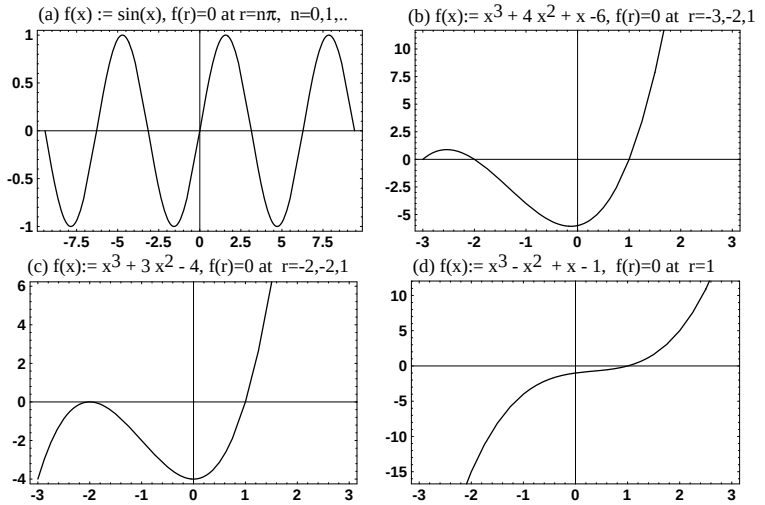

Figure 2.1: Graphs of some simple functions

dimensional systems. Hence we seek to construct computational algorithms that can be generalized and refined successively to handle a large class of nonlinear problems.

Before we embark on such a task, some of the potential problems are illustrated with specific examples. In the general nonlinear case, there is no way to know *a priori*, how many values of r can be found that will satisfy the equation, particularly if the entire range of x in $(-\infty, \infty)$ is considered. For example, the simple equation

$$f(x) := \sin(x) = 0$$

has infinitely many solutions given by $r = n\pi$ where n is any integer. The graph of the function shown in figure 2.1a illustrates this clearly. Often, the physical description of the problem that gave rise to the mathematical function, will also provide information on the range of values of x that are of interest. For example, in the multicomponent flash equation, the problem was so formulated that the dependent variable had the physical interpretation of fraction of feed in vapor; hence this fraction must be between $(0, 1)$. Although the mathematical equation may have many other roots, outside of this range, they would lack any physical meaning and hence would not be of interest.

Algebraic theory tells us that the total number of roots of a polynomial is equal to the degree of the polynomial, but not all of them may be real roots. Furthermore, if the coefficients of the polynomial are all real,

then any complex root must occur in pairs. Consider the three cubic equations given below:

$$f(x) := x^3 + 4x^2 + x - 6 = 0 \qquad r = -3, -2, 1$$

$$f(x) := x^3 + 3x^2 - 4 = 0, \qquad r = -2, -2, 1$$

$$f(x) := x^3 - x^2 + x - 1 = 0 \qquad r = 1$$

These are graphed in figures 2.1b,c,d respectively. In the first case there are three distinct roots. The function has a non-zero slope at each value of the root and such roots are called *simple* roots. In the second case we have a degeneracy or a non-simple root at $r = -2$. This problem manifests itself in a graphical representation with a zero slope of the function at the multiple root, $r = -2$. If the coefficients of the polynomial were slightly different, the curve could have moved slightly upward giving rise to two distinct roots or downwards yielding no roots in this region. Algebraically, we can see that the root is a multiple one with a multiplicity of 2 by factoring the function into $(x + 2)(x + 2)(x - 1)$. In the third case there is only a single *real* root.

We begin by constructing some simple algorithms that have an intuitive appeal. They are easy to represent graphically and symbolically so that one can appreciate the connection between the two representations. Subsequently we can refine the computational algorithms to meet the challenges posed by more difficult problems, while keeping the graphical representation as a visual aid.

There are essentially three key steps in any root finding algorithm. They are:

step 1: **Guess** one or more initial values for x.

step 2: **Iterate** using a scheme to improve the initial guess.

step 3: **Check convergence** - *i.e.*, has the improvement scheme of step 2 produced a result of desired accuracy?

The crux of the algorithm is often in the second step and the objective in devising various clever schemes is to get from the initial guess to the final result as quickly as possible.

2.2 Bisection method

The bisection algorithm is quite intuitive. A graphical illustration of this algorithm is shown in figure 2.2a. In step 1, we make two guesses x_1 and

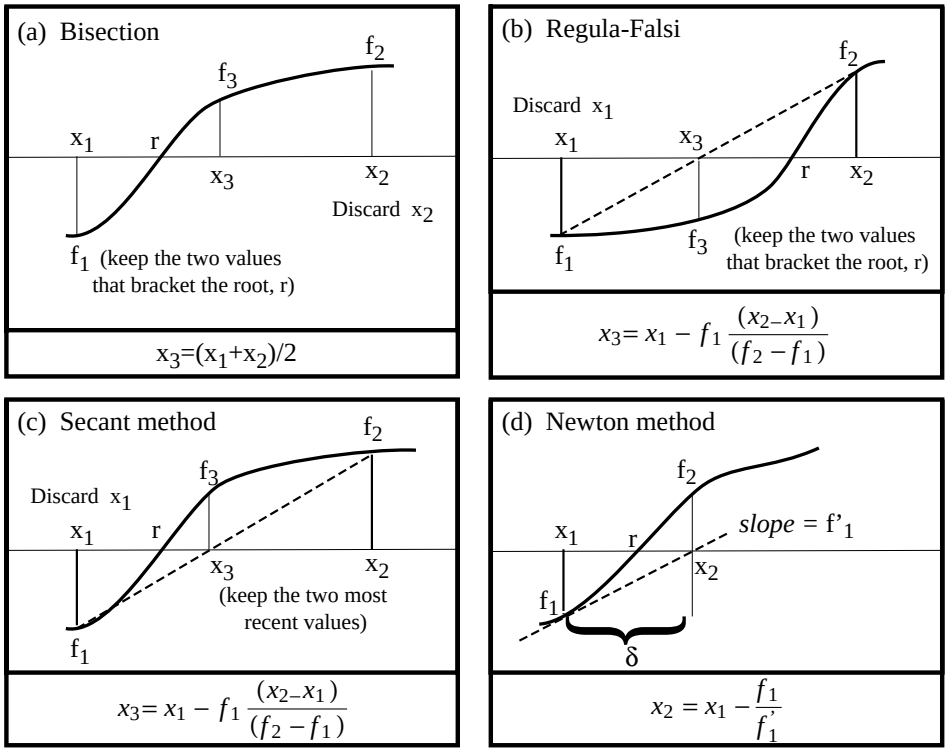

Figure 2.2: Graphical representation of some simple root finding algorithms

x_2 and calculate the functions values $f_1 = f(x_1)$ and $f_2 = f(x_2)$. If the function values have opposite signs, it implies that it passes through zero somewhere between x_1 and x_2 and hence we can proceed to the second step of producing a better estimate of the root, x_3. If there is no sign change, it might imply that there is no root between (x_1, x_2). So we have to make a set of alternate guesses. The scheme for producing a better estimate is also an extremely simple one of using the average of the two initial guesses, viz.

What else might it imply?

$$x_3 = \frac{x_1 + x_2}{2} \tag{2.3}$$

In figure 2.2a, (x_1, x_3) bracket the root r; hence we discard x_2, or better still, store the *value* of x_3 in the *variable* x_2 so that we are poised to repeat step 2. If the situation were such as the one shown in figure 2.2b,

then we would discard x_1 or better still, store the *value* of x_3 in the *variable* x_1 and repeat step 2. In either case, (x_1, x_2) will have better guesses than the original values.

The final step is to check if we are close enough to the desired root r so that we can terminate the repeated application of step 2. One test might be to check if the absolute difference between two successive values of x is smaller than a specified tolerance, *i.e.*,

$$|x_{i+1} - x_i| \leq \epsilon$$

Another test might be to check if the absolute value of the function f_i at the end of every iteration is below a certain tolerance, *i.e.*,

$$|f(x_i)| \leq \epsilon$$

In addition, we might wish to place a limit on the number of times step 2 is repeated. A MATLAB function, constructed in the form of a *m-file,* is shown in figure 2.3.

Example - Multicomponent flash equation

Let us consider the multicomponent, isothermal flash equation developed in section §1.3.3. The equation is

$$\boxed{\sum_{i}^{N} \frac{(1 - K_i) z_i}{(K_i - 1)\psi + 1} = 0} \tag{2.4}$$

where (K_i, z_i, N) are treated as known parameters and ψ is the unknown variable. Implementation of this function in MATLAB is shown below as an *m-file*.

See Appendix B for an introduction to MATLAB.

```
function r=bisect(Fun,x,tol,trace)
%BISECT find the root of "Fun" using bisection scheme
%       Fun   - the name of the external function
%       x     - vector of length 2, (initial guesses)
%       tol   - error criterion
%       trace - print intermediate results
%
%   Usage  bisect('flash',[0,1])
%          flash is the name of the external function.
%               [0,1] is the initial guess

%Check inputs
if nargin < 4, trace=0; end
if nargin < 3, tol=eps; trace=0; end
if (length(x) ~= 2)
 error('Please provide two initial guesses')
end

f = feval(Fun,x);        %Fun is assumed to accept a vector

if (prod(sign(f))) > 0,  %Check if roots are bracketed
 error('No sign change - no roots')
end;

for i = 1:100             %Set max limit on iterations
 x3 = (x(1) + x(2))/2;    %Update the guess
 f3 = feval(Fun,x3);      %Cal. f(x3)

%Check if x2 or x1 should be discarded
 if sign(f(1)*f3) < 0, x(2)=x3;  else x(1)=x3;  end;

 if abs(f3) < tol, r=x3; return; end  %Check convergence
 if trace, fprintf(1,'%3i %12.5f %12.5f\n', i,x3,f3); end
end
error('Exceeded maximum number of iterations')
```

Figure 2.3: MATLAB implementation of the bisection algorithm

```
function f=flash(psi)
% K is a vector of any length of equil ratios.
% z is the feed composition (same length as K)
% K, z are defined as global in main
% psi is the vapor fraction.

global K z
if ( length(K) ~= length(z) )
 error('Number of K values & compositions do not match')
end
n=length(psi);
for i = 1:n
 f(i)=sum( ((K-1).*z) ./ (1+(K-1)*psi(i)) );
end
```

Observe that this function, while being concise, is fairly general to handle any number of components N and a vector of guesses `psi` of any length and return a vector of function values, one corresponding to each element in the guessed variable `psi`. Assuming that you have such a function defined in a file named `flash.m`, you are encouraged to work through the following exercise using MATLAB.

» **global K z**
» **z=[.25 .25 .25 .25] %**define a 4-component system
» **K=[2 1.5 0.5 0.1] %**define equilibrium values
» **bisect('flash',[0,.1]) %**find the root using bisect — ans=0.0434
» **x=0:0.05:1; %**create a vector of equally spaced data
» **y=flash(x); %**evaluate the function
» **plot(x,y) %**plot the function

In this section, we have developed and implemented the bisection algorithm as a function in MATLAB and used it to solve an example problem from multicomponent flash. The function `bisect` can be used to solve any other root finding problem as long as you define the problem you want to solve as another MATLAB function along the lines of the example function `flash.m`.

While we managed to solve the problem, we did not concern ourselves with questions such as, (i) how many iterations did it take to converge? and, (ii) can we improve the iteration scheme in step 2 to reduce the number of iterations? By design, the bisection scheme will always

Is it always possible to find such a guess? Consider the pathological case of $r = -2$ in figure 2.1c!

converge provided an acceptable set of initial guesses have been chosen. This method, however, converges rather slowly and we attempt to devise algorithms that improve the rate of convergence.

2.3 Regula-falsi method

Instead of using the average of the two initial guesses as we did with the bisection scheme, we can attempt to approximate the function $f(x)$ by straight line (a *linear* approximation) since we know two points on the function $f(x)$. This is illustrated graphically in figure 2.2b with the dashed line approximating the function. We can then determine the root, x_3 of this linear function, $\tilde{f}(\tilde{x})$. The equation for the dashed straight line in figure 2.2b is

$$\tilde{f}(\tilde{x}) := \frac{\tilde{x} - x_1}{\tilde{f} - f_1} = \frac{x_2 - x_1}{f_2 - f_1}$$

Now we can determine the value of $\tilde{x} = x_3$ where the linear function $\tilde{f} = 0$ in the above equation. This results in,

$$\boxed{x_3 = x_1 - f_1 \frac{x_2 - x_1}{f_2 - f_1}} \tag{2.5}$$

which is used as the iterative equation in step 2. When we evaluate the original function at x_3, $f(x_3)$ clearly will not be zero (unless the scheme has converged) as shown in figure 2.2b; but x_3 will be closer to r than either x_1 or x_2. We can then retain x_3 and one of x_1 or x_2 in such a manner that the root r remains bracketed. This is achieved by following the same logic as in the bisection algorithm to discard the x value that does not bracket the root.

The MATLAB function `bisect` can be easily adapted to implement Regula-Falsi method by merely replacing equation (2.3) with equation (2.5) for step 2.

2.4 Secant method

The secant method retains the linear approximation procedure of the regula-falsi method, but differs by retaining the two most recent values of x viz. x_2 and x_3 and always discarding the oldest value x_1. This simple change from the Regula-Falsi scheme produces a dramatic difference in the rate of convergence. A MATLAB implementation of the

```
function r=secant(Fun,x,tol,trace)
%SECANT find the root of a function "Fun" using secant scheme
%       Fun   - the name of the external function
%       x     - vector of length 2, (initial guesses)
%       tol   - error criterion
%       trace - print intermediate results
%
%   Usage  secant('flash',[0,1])
%          Here flash is the name of the external function.
%               [0,1] is the initial guess

%Check inputs
if nargin < 4, trace=0; end
if nargin < 3, tol=eps; trace=0; end
if (length(x) ~= 2)
 error('Please provide two initial guesses')
end

f = feval(Fun,x);        %Fun is assumed to accept a vector

for i = 1:100            %Set max limit on iterations
 x3 = x(1) - f(1)*(x(2)-x(1))/(f(2)-f(1)) ; %Update(step 2)
 f3 = feval(Fun,x3);     %Cal. f(x3)

%Keep the last two values
 x(1) = x(2);f(1) = f(2); x(2) = x3; f(2) = f3;

 if abs(f3) < tol, r=x3; return; end %Check for convergence
 if trace, fprintf(1,'%3i %12.5f %12.5f\n', i,x3,f3); end
end
error('Exceeded maximum number of iterations')
```

Figure 2.4: MATLAB implementation of the secant algorithm

secant method is shown in figure 2.4. The convergence rate of the secant method was analyzed by Jeeves (1958).

2.5 Newton's method

The Newton method is by far the most powerful and widely used algorithm for finding the roots of nonlinear equations. A graphical representation of the algorithm is shown in figure 2.2d. This algorithm also relies on constructing a linear approximation of the function; But this is achieved by taking the tangent to the function at a given point. Hence this scheme requires only one initial guess, x_1. The linear function $\tilde{f}(\tilde{x})$ shown by dashed line in figure 2.2d is,

$$\tilde{f}(\tilde{x}) := \frac{\tilde{f} - f_1}{\tilde{x} - x_1}$$

The root of this linear equation, is at $(\tilde{x} = x_2, \tilde{f} = 0)$. Under these conditions one can solve the above equation for x_2 as,

$$\boxed{x_2 = x_1 - \frac{f_1}{f_1'}} \tag{2.6}$$

which forms the iterative process (step 2). Note that this algorithm requires that the derivative of the function be evaluated at every iteration, which can be a computationally expensive operation.

While we have relied on the geometrical interpretation so far in constructing the algorithms, we can also derive Newton's scheme from a Taylor series expansion of a function. This is an instructive exercise, for it will enable us to generalize the Newton's scheme to higher dimensional (*i.e.*, more than two equations) systems as well as provide some information on the rate of convergence of the iterative scheme.

The Taylor series representation of a function around a reference point, x_i is,

$$\boxed{f(x_i + \delta) = \sum_{k=0}^{\infty} \frac{f^{(k)}(x_i)}{k!} \delta^k} \tag{2.7}$$

where $f^{(k)}(x_i)$ is the $k-th$ derivative of the function at x_i and δ is a small displacement from x_i. While the infinite series expansion is an exact representation of the function, it requires all the higher order derivative of the function at the reference point. We can construct various levels of

approximate representations of the function, $\tilde{f}$ by *truncating* the series at *finite* terms. For example a three term expansion ($k = 0, 1, 2$) is

$$\tilde{f}(x_i + \delta) = f(x_i) + f'(x_i)\delta + f''(x_i)\frac{\delta^2}{2!} + \mathcal{O}(\delta^3)$$

where the symbol $\mathcal{O}(\delta^3)$ stands as a reminder of the higher order terms (three and above in this case) that have been neglected. The error introduced by such omission of higher order terms is called *truncation error*. In fact to derive the Newton scheme, we neglect the quadratic term $\mathcal{O}(\delta^2)$ also. In figure 2.2d, taking the reference point to be $x_i = x_1$, the displacement to be $\delta = x_{i+1} - x_i$, and recognising that $\tilde{f}(x_i + \delta) = 0$ we can rewrite the truncted two-term series as,

$$0 = f(x_i) + f'(x_i)(x_{i+1} - x_i) + \mathcal{O}(\delta^2)$$

which can be rearranged as

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

This is the same as equation (2.6).

2.6 Muller's method

This scheme can be thought of as an attempt to generalize secant method and is important at least from the point of illustrating such generalizations. Instead of making two guesses and constructing an approximate linear function as we did with the secant method, we can choose three inital guesses and construct a quadratic approximation to the original function and find the roots of the quadratic. A graphical representation of this is shown in figure 2.5. The three initial guesses are (x_0, x_1, x_2) and the corresponding function values are represented by (f_1, f_2, f_3) respectively. We construct a second degree polynomial as,

$$p_2(v) = av^2 + bv + c$$

where $v = (x - x_0)$. Note that the polynomial is represented as a function of a new independent variable v, which is merely a translation of the original independent variable x by x_0. An alternate view is to regard v as the distances measured from reference point x_0, so that $v = 0$ at this new origin. (a, b, c) are the coefficients of the quadratic that must be determined in such a way that $p_2(v)$ passes through the three

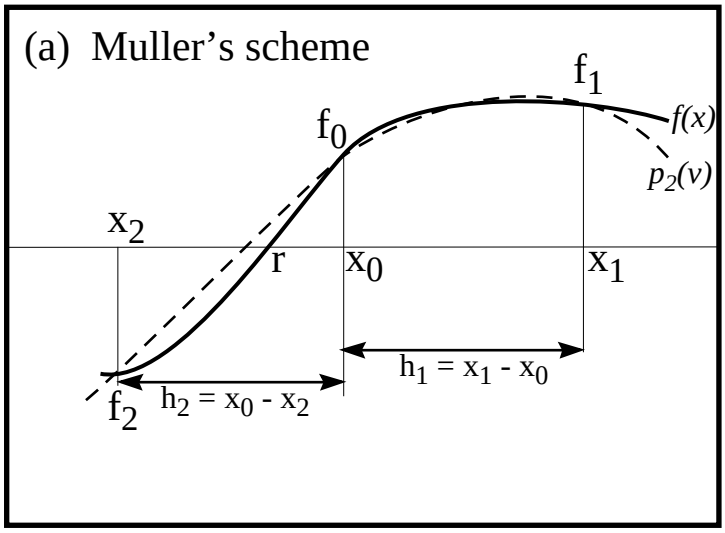

Figure 2.5: Graphical representation of Muller's scheme

data points (x_0, f_0), (x_1, f_1) and (x_2, f_2). Defining $h_1 = (x_1 - x_0)$ and $h_2 = (x_0 - x_2)$ and requiring that the polynomial pass through the three points, we get,

$$\begin{aligned} p_2(0) = a(0)^2 + b(0) + c &= f_0 \qquad \Rightarrow \qquad c = f_0 \\ p_2(h_1) = ah_1^2 + bh_1 + c &= f_1 \\ p_2(-h_2) = ah_2^2 - bh_2 + c &= f_2 \end{aligned}$$

The reason for coordinate shift should be clear by now. This enables c to be found directly. Solving the remaining two equations we obtain a and b as follows:

$$a = \frac{\gamma f_1 - f_0(1+\gamma) + f_2}{\gamma h_1^2 (1+\gamma)}$$

$$b = \frac{f_1 - f_0 - ah_1^2}{h_1}$$

where $\gamma = h_2/h_1$. So far we have only constructed an approximate representation of the original function, $f(x) \approx p_2(v)$. The next step is to find the roots of this approximate function, $p_2(v) = 0$. These are given by,

$$v = \tilde{r} - x_0 = \frac{-b \pm \sqrt{b^2 - 4ac}}{a2}$$

This can be rearranged as,

$$\boxed{\tilde{r} = x_0 - \frac{2c}{b \pm \sqrt{b^2 - 4ac}}} \tag{2.8}$$

Since $p_2(v)$ is a quadratic, there are clearly two roots $\tilde{r}$ in equation (2.8). In order the take the root closest to x_0 we choose the largest denominator in equation (2.8). In summary, the sequential procedure for impliment-ing Muller's scheme is as follows:

- Guess (x_0, x_1, x_2)
- Compute $(f_0 = f(x_0), f_1 = f(x_1), f_2 = f(x_2))$.
- Compute $h_1 = (x_1 - x_0)$, $h_2 = (x_0 - x_2)$, $\gamma = h_2/h_1$
- Compute $c = f(x_0)$.
- Compute $a = (\gamma f_1 - f_0(1 + \gamma) + f_2)/(\gamma h_1^2(1 + \gamma))$
- Compute $b = (f_1 - f_0 - ah_1^2)/(h_1)$.
- Compute the roots from equation (2.8).
- From (x_0, x_1, x_2) discard the point farthest from $\tilde{r}$ and substitue the new root in its place and repeat.

Note that Muller's method converges almost quadratically (as does Newton's scheme), but requires only one additional function evaluation at every iteration which is comparable to the computational load of the secant method. In particular derivative evaluation is not required, which is a major advantage as compared to Newton's method. Also, this scheme can converge to complex roots even while starting with real initial guesses as long as provision is made for handling complex arithmetic in the computer program. MATLAB handles complex arithmetic quite naturally.

2.7 Fixed point iteration

Another approach to construct an update scheme (for step 2) is to rearrange the given equation $f(x) = 0$ into a form,

$$x = g(x)$$

Then, starting with a guess x_i, we can evaluate $g(x_i)$ from the right hand side of the above equation and the result itself is regarded as a better estimate of the root, *i.e.*,

$$\boxed{x_{i+1} = g(x_i) \qquad i = 0, 1 \cdots} \tag{2.9}$$

Given, $f(x) = 0$ it is not difficult rewrite it in the form $x = g(x)$; nor is this process unique. For example, we can always let $g(x) = x + f(x)$. Such an iterative scheme need not always converge. Let us examine the possible behavior of the iterates with a specific example. In particular we will illustrate that different choices of $g(x)$ lead to different behavior. Consider the function

$$f(x) = x^2 - x - 6 = 0 \tag{2.10}$$

which has roots at $r = -2$ and $r = 3$. In the first case let us rewrite it as

$$x = \sqrt{x + 6}$$

A geometrical interpretation is that we are finding the intersection of two curves, $y = x$ (the left hand side) and $y = \sqrt{x + 6}$ (the right hand side). See figure 2.6 for a graphical illustration. Starting with an initial guess, say $x_0 = 4$, we compute $x_1 = g(x_0) = \sqrt{x_0 + 6}$. This is tantamount to stepping between the $y = x$ and $y = g(x)$ curves as shown in figure 2.6a. It is clear that the sequence will converge monotonically to the root $r = 3$. The table 2.1 shows the first ten iterates, starting with an inital guess of $x_0 = 5$.

Observe that the slope of the function at the root is $g'(r = 3) < 1$. We will show shortly that the condition for convergence is indeed $|g'(r)| < 1$. As an alternate formulation consider rewritting equation (2.10) as $x = g(x) = 6/(x - 1)$. Now, $g(x)$ has a singularity at $x = 1$. A graphical illustration is shown in figure 2.6b. Using this new $g(x)$, but starting at the same initial guess $x_0 = 4$ the sequence diverges initially in an oscillatory fashion around the root $r = 3$, but eventually is attracted to the other root at $r = -2$, also in an oscillatory fashion. Observe that the slopes at the two roots are: $g'(3) = -3/2$ and $g'(-2) = -2/3$. Both are negative and hence the oscillatory behavior. The one with absolute magnitude greater than unity diverges and the other with absolute magnitude less than unity converges. Finally consider the formulation $x = g(x) = (x^2 - 6)$. The behavior for this case is shown in figure 2.6c. For reasons indicated above, the sequence will not converge to either root! The following shows the MATLAB implementation for generating the iterative sequence for the first case. Enter this into a file called *g.m*.

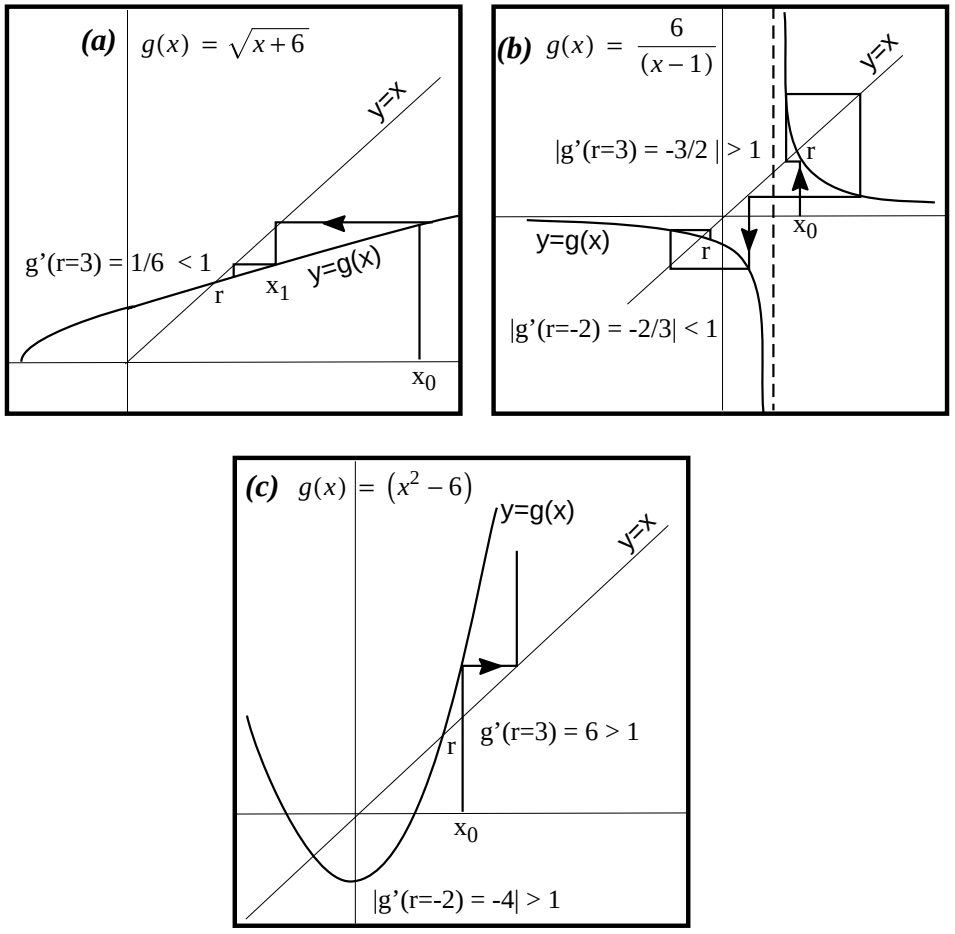

Figure 2.6: Graphical representation of fixed point scheme

Iteration Number	Iterate
x_1	5.0000000
x_2	3.3166248
x_3	3.0523147
x_4	3.0087065
x_5	3.0014507
x_6	3.0002418
x_7	3.0000403
x_8	3.0000067
x_9	3.0000011
x_{10}	3.0000002

Table 2.1: The first ten iterates of $x_{i+1} = \sqrt{x_i + 6}$ starting with $x_0 = 5$

```
function x=g(x)
for i=1:10
  fprintf(1,'%2i %12.5e\n',i,x);  %print the iterates
  x=sqrt(x+6);   %also try x=6/(x-1)  and x=(x^2-6) here
end
```

Invoke this function from within MATLAB with various initial guesses, *e.g.*, try initial guess of 5 by entering,

» **g(5)**

2.8 Error analysis and convergence acceleration

A simple error analysis can be developed for the fixed point iterative scheme which will provide not only a criterion for convergence, but also clues for accelerating convergence with very little additional computational effort. We are clearly moving away from the realm of intuition to the realm of analysis! Consider the fixed point iteration $x_{i+1} = g(x_i)$. After convergence to the root r we will have $r = g(r)$. Subtracting the two equations we get,

$$(x_{i+1} - r) = g(x_i) - g(r)$$

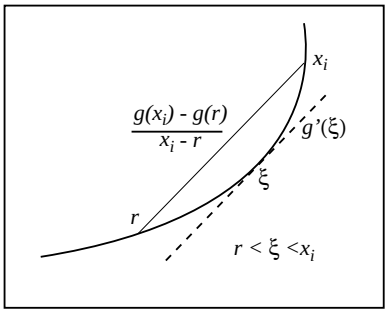

Figure 2.7: Graphical illustration of mean value theorem

Multiplying and diving the right hand side by $(x_i - r)$,

$$(x_{i+1} - r) = \frac{g(x_i) - g(r)}{(x_i - r)}(x_i - r)$$

Now the difference $e_i = (x_i - r)$ can be interpreted as the error at iterative step i and hence the above equation can be written as,

$$\boxed{e_{i+1} = g'(\xi)\, e_i} \tag{2.11}$$

where we have used the mean value theorem to replace slope of the chord by the tangent to the curve at some suitable value of $x = \xi$, *i.e.*,

$$\frac{(g(x_i) - g(r))}{(x_i - r)} = g'(\xi)$$

A geometrical interpretation of the mean value theorem is shown in figure 2.7. From equation (2.11), it is clear the error will decrease with every iteration if the slope $|g'(\xi)| < 1$; otherwise the error will be amplified at every iteration. Since the error in the current step is proportional to that of the previous step, we conclude that the rate of convergence of the fixed point iteration is linear. The development has been reasonably rigorous so far. We now take a more pragmatic step and assume that $g'(\xi) = K$ is a constant in the neighbourhodd of the root r. Then we have the sequence,

Will K be the same constant at every iteration?

$$e_2 = Ke_1, \qquad e_3 = Ke_2 = K^2 e_1, \qquad e_4 = Ke_3 = K^3 e_1 \qquad \cdots$$

and hence we can write a general error propagation solution as,

$$\boxed{e_n = K^{n-1} e_1 \qquad \text{or} \qquad x_n - r = K^{n-1} e_1} \tag{2.12}$$

It should be clear now that $e_n \to 0$ as $n \to \infty$ only if $|K| < 1$. . We refer to equation (2.12) as the error propagation solution since it provides a solution of estimating the error at any step n, provided the error at the first step e_1 and K are known.

Well, if we know the error in the first step, none of this analysis would be necessary! $r = x_1 - e_1$ would do it!

We can develop a convergence acceleration scheme using the error solution (2.12) to estimate the three unknowns (r, K, e_1) in the second form of equation (2.12). Once we have generated three iterates, (x_n, x_{n+1}, x_{n+2}), we can use equation (2.12) to write down,

$$\begin{aligned} x_n &= r + K^{n-1} e_1 \\ x_{n+1} &= r + K^n e_1 \\ x_{n+2} &= r + K^{n+1} e_1 \end{aligned} \tag{2.13}$$

Now we have three equations in three unkowns which can be solved to estimate r (and K, e_1 as well). If K were to remain a true constant with every iteration, r would be the correct root; since K is not a constant in general, r is only an *estimate* of the root, hopefully a better estimate than any of (x_n, x_{n+1}, x_{n+2}). Now let us proceed to construct a solution for r from the above three equations. We will define a first order forward difference operator Δ as,

$$\Delta x_n = x_{n+1} - x_n$$

Think of the symbol Δ as defining a new rule of operation just like a derivative operator $\frac{d}{dx}$ defines a rule. When Δ operates on x_n it is computed using the rule shown on the right hand side. Now, if we apply the operator Δ to x_{n+1} we should have,

$$\Delta x_{n+1} = x_{n+2} - x_{n+1}.$$

If we apply the Δ operator twice (which is equivalent to defining higher order derivatives), we should get,

$$\Delta(\Delta x_n) = \Delta^2 x_n = (\Delta x_{n+1}) - (\Delta x_n) = x_{n+2} - 2x_{n+1} + x_n.$$

You can verify that using equation (2.13) in the above definitions, we get,

$$\frac{(\Delta x_n)^2}{\Delta^2 x_n} = K^{n-1} e_1 = x_n - r$$

and hence r is given by,

$$r = x_n - \frac{(\Delta x_n)^2}{\Delta^2 x_n} = x_n - \frac{x_{n+1}^2 - 2x_n x_{n+1} + x_n^2}{x_{n+2} - 2x_{n+1} + x_n}$$

Thus the three iterates (x_n, x_{n+1}, x_{n+2}) can be plugged into the right hand side of the above equation to get a better estimate of r.

Example of convergence acceleration

Let us apply this convergence acceleration procedure to the first three iterates of table 2.1.

$$\begin{aligned}
x_1 &= 5.0000000 \\
x_2 &= 3.3166248 \\
x_3 &= 3.0523147 \\
\Delta x_1 &= (x_2 - x_1) = -1.6833752 \\
\Delta x_2 &= (x_3 - x_2) = -0.26431010 \\
\Delta^2 x_1 &= (\Delta x_2 - \Delta x_1) = 1.41906510 \\
r &= x_1 - \frac{(\Delta x_1)^2}{\Delta^2 x_1} = 5.000000 - \frac{(-1.6833752)^2}{1.41906510} = 3.0030852
\end{aligned}$$

Compare this with the fourth iterate produce in the original sequence $x_4 = 3.0087065$.

2.8.1 Convergence of Newton scheme

The Newton scheme given by equation (2.6) can be thought of as a fixed point iteration scheme where $g(x)$ has been specified in a special manner as,

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

Hence,

$$g'(x) = 1 - \frac{(f')^2 - ff''}{(f')^2} = \left| \frac{ff''}{(f')^2} \right| < 1$$

Since $f(r) = 0$ by definition, $g'(r) = 0$ (barring any pathological situation such as $f'(r) = 0$) and the inequality should hold near the root r. Thus the Newton method is guaranteed to converge as long as we have a good initial guess. Having progressed this far, we can take the next

step and ask the question about the rate of convergence of the Newton method. A Taylor series expansion of $g(x)$ around r is,

$$g(x_n) = g(r) + g'(r)(x_n - r) + \frac{g''(r)}{2}(x_n - r)^2 + \cdots$$

Recognizing that $e_{n+1} = x_{n+1} - r = g(x_n) - r$, $e_n = (x_n - r)$ and $g'(r) = 0$, the truncated Taylor series expansion can be rearranged as,

$$e_{n+1} = \frac{g''(r)}{2} e_n^2$$

which shows that the error at any step goes down as the square of the previous step - *i.e.*, quadratically! This manifests itself in the form of doubling accuracy at every iteration.

2.9 Deflation technique

Having found a root, r, of $f(x) = 0$, if we are interested in finding additional roots of $f(x) = 0$, we can start with a different initial guess and hope that the new initial guess lies within the region of attraction of a root different from r. Choosing a different initial guess does not guarantee that the iteration scheme will not be attracted to the root already discovered. In order to ensure that we stay away from the known root, r, we can choose to *deflate* the original function by constructing a modified function,

$$g(x) = f(x)/(x - r)$$

which does not have r as a root. For a single equation the concepts are best illustrated with a graphical example. Consider the illustration in figure 2.8a where the original function,

$$f(x) := (x - 2)\sin(2x)e^{-0.8x}$$

can be seen to have several roots including one at $r = 2$. A sketch of the deflated function, $g(x) = f(x)/(x - 2)$ is shown in figure 2.8b. Since $r = 2$ turns out to be a *simple root* of $f(x) = 0$, the deflated function $g(x) = 0$ does not contain the already discovered root at $x = 2$. Hence starting with a different initial guess and applying an iterative method like the secant or Newton scheme on the function $g(x)$ will result in convergence to another root. This process can obviously be repeated by deflating successively found roots. For example if we know two roots r_1 and r_2 then a new function can be constructed as

$$h(x) = \frac{f(x)}{(x - r_1)(x - r_2)}.$$

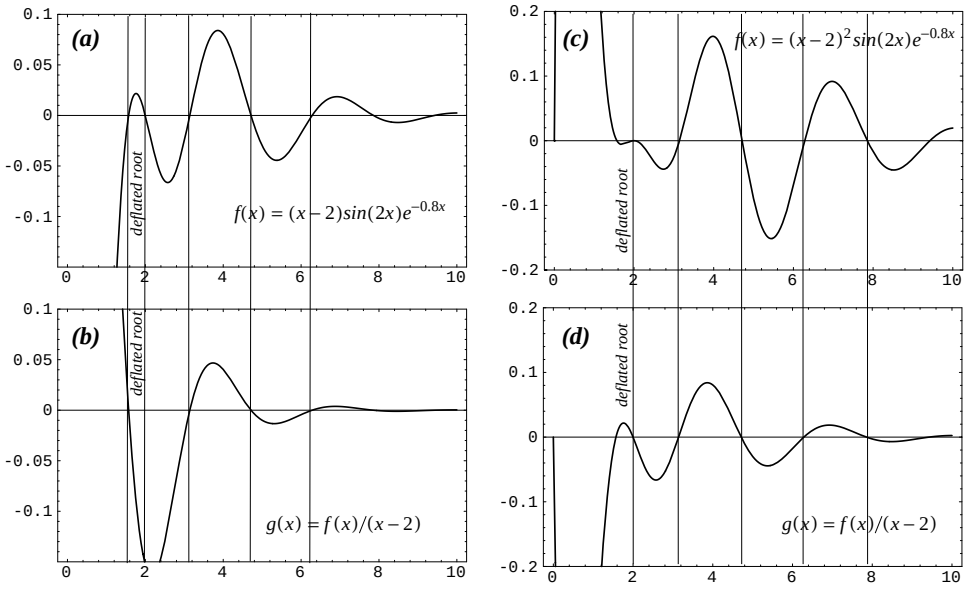

Figure 2.8: Graphical illustration deflation technique

The successive application of this approach is of course susceptible to propagation of round off errors. For example if the roots r_1, r_2 are known to only a few significant digits, then the definition of the deflated function $h(x)$ will inherit these errors and hence the roots of $h(x) = 0$ will not be as accurate as those of the original equation $f(x) = 0$.

Can you think of a way to alleviate this problem?

Another problem of the deflation technique pertains to non-simple roots. A sketch of the function,

$$f(x) := (x-2)^2 \sin(2x)e^{-0.8x}$$

is shown in figure 2.8c. It is immediately clear that $r = 2$ is a *double root* - *i.e.*, occurs with a multiplicity of two. Hence the deflated function $g(x) = f(x)/(x-2)$ still has $r = 2$ as a *simple root* as seen in figure 2.8d.

What computational problem might this pose?

2.10 Parameter continuation

2.10.1 Euler-Newton continuation

2.10.2 Homotopy continuation

2.11 Software tools

2.11.1 MATLAB

The MATLAB function for determining roots of a polynomial is called `roots`. You can invoke it by entering,

» **roots(c)**

where `c` is a vector containing the coefficients of the polynomial in the form,

$$p_n(x) = c_1 x^n + c_2 x^{n-1} + \cdots + c_n x + c_{n+1}.$$

Let us consider the factored form of the polynomial $p_3(x) = (x+2)(x+i)(x-i)$ so that we know the roots are at $(-2, \pm i)$. To check whether MATLAB can find the roots of this polynomial we need to construct the coefficients of the expanded polynomial. This can be done with the convolve fucntion `conv(f1,f2)` as follows.

```
» f1 = [1 2] %Here we define coeff of (x+2) as [1 2]
» f2 = [1 i] %Here we define coeff of (x+i) as [1 i]
» f3 = [1 -i] %Here we define coeff of (x-i) as [1 -i]
» c=conv(conv(f1,f2),f3) % c contains coeff of polynomial
» r=roots(c) %returns roots of polynomial defined by c
```

Note that the function `roots` finds *all* of the roots of a polynomial, including complex ones.

The MATLAB function for finding a *real* root of any real, single nonlinear algebraic equation (not necessarily a polynomial) is called `fzero`. You can invoke it by entering,

» **fzero('fn',x)**

where `fn` is the name of a *m-file* that defines the function, `x` is the initial guess for the root. This `fzero` is not based on a very robust algorithm. If the function you want to solve has singularities, or multiple roots, the scheme fails to converge, often without producing any appropriate error or warning messages. Hence use with caution. After it produces an

answer check that it is the correct result by evaluating the function at the root. As an example try the multicomponent flash problem considered previously. You are encouraged to try the following steps during a MATLAB session.

```
»global K z;              % define K,z to be global
»K=[2 1.5 0.5 0.2];       % define K values
»z=[.25 .25 .25 .25];     % define z values
»root=fzero('flash',.5)   % solve x0=0.5, ⇒ ans=0.0949
»flash(root)              % check solution
»root=fzero('flash',-.85) % solve x0=-0.85, ⇒ ans=-1.0000
»flash(root)              % check solution
»root=fzero('flash',1.85) % solve x0=1.85, ⇒ ans=1.6698
»flash(root)              % check solution
»root=fzero('flash',1.90) % solve x0=1.90, ⇒ ans=2.0
»flash(root)              % check solution
```

Note that the desired result is `root=0.0949203`. But starting with different initial guesses, MATLAB produces different results! Why? Try plotting the function over the range $\psi \in [-5, 5]$ in MATLAB and see if you can understand MATLAB behavior! (Clue: sign change)

2.11.2 Mathematika

Mathematika is another powerful software package for mathematical analysis including symbolic processing. It also has an inteactive environment; *i.e.*, commands, functions are executed as soon as you enter them. If you invoke Mathematica using a Graphical User Interface (GUI) then, the ploting functions will display the graphs. Otherwise you are limited to the use of computational features of Mathematica. A complete reference to Mathematica can be found in the book by Wolfram (1988).

Example of solving the flash equation with Mathematica

The Mathematica function for finding the roots of an equation is called `Solve`. In MATLAB we constructed a *m-file* to define a function. In Mathematica this is done in one line within the workspace. Anything delimited by (* *) is treated as comments and ignored by Mathematica. You may wish to work through the following exercise.

```
(* Define the multicomponent flash function f[psi]
   for a 4-component system.
   K[[i]], z[[i]]   are called lists in Mathematica.
                    Treat them as arrays.
```

Observe that first letter of all Mathematica functions is in uppercase.

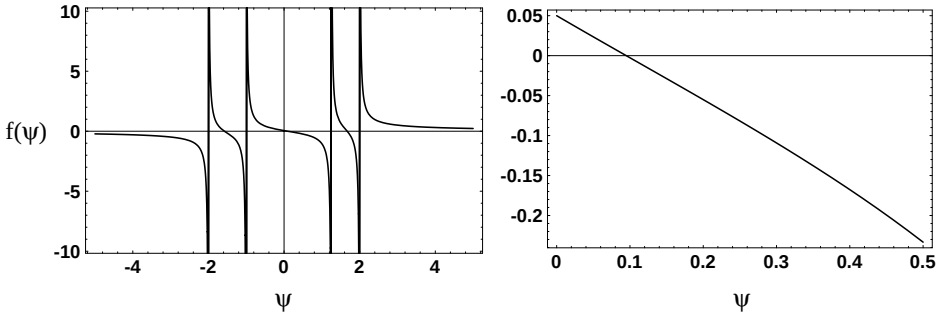

Figure 2.9: Multicomponent flash function

```
   Sum[f(i), {i,N}] is a function that sums f(i)
                    over the index range {1,N}         *)

f[psi_]:=Sum[(K[[i]]-1)*z[[i]]/((K[[i]]-1)*psi+1), {i,4}]

K={2., 1.5, .5, .2}      (* Define K values *)
z={.25, .25, .25, .25}   (* Define z values *)
Solve[f[x] == 0, x ]     (* Solve f(x)=0    *)
Plot[f[x],{x,-5,5}, Frame -> True]  (* Plot f[x] over 5 to 5 *)
Plot[f[x],{x,0,.5}, Frame -> True]  (* Plot f[x] over 0 to 0.5 *)
```

Note that no initial guess is needed. Mathematica finds all the three roots. They are:

```
{{x -> -1.57727}, {x -> 0.0949203}, {x -> 1.66984}}
```

Ofcourse, only the second root is of interest. Others do not have any physical relevance. Note also that while ploting functions, Mathematica samples the function at sufficient number of data points (*i.e.*, x-values) to provide a smooth function. Graphs of the flash equation produced by Mathematica are shown in figure 2.9.

2.12 Exercise problems

2.12.1 Multicomponent, isothermal flash model

A sketch of a multicomponent flash process is shown in figure 1.5. The following equation, which was derived in Chapter 1, models the multicomponent flash process. This is a single non-linear algebraic equation

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

Table 2.2: Feed composition & equilibrium ratio of a natural gas mixture

in the unknown ψ, which represents the fraction of feed that goes into the vapor phase.

$$\sum_{i=1}^{N} \frac{(1-K_i)z_i}{(K_i-1)\psi+1} = 0 \tag{2.14}$$

This equation has several roots, but not all of them have any physical meaning. Only the root for $\psi \in [0,1]$ is of interest.

For a bonus point you may choose to find the number of roots for the test data above!

The test data in Table 2.2, (taken from Katz etal) relate to the flashing of a natural gas stream at 1600 psia and 120^oF. Determine the fraction ψ using the *secant* algorithm given in figure 2.4 and another root finding function that is provided in MATLAB named *fzero*.

2.12.2 Compressible flow in a pipe

In a fluid mechanics course you might come across the Weymouth equation, which is used for relating the pressure drop vs. flow rate in a pipeline carrying compressible gases. It is given by,

$$Q_o = 433.54 \frac{T_o}{P_o}\left[\frac{(P_1^2-P_2^2)}{L\ \sigma\ T}\right]^{0.5} d^{2.667}\eta \tag{2.15}$$

where

Q_o is the gas flow rate = 2000000 SCFD
T_o is the standard temperature = 520^oR

P_o is the standard pressure = 14.7 psia
P_1 is the upstream pressure, (?), psia
P_2 is the downstream pressure, (21.7), psia
L is the length of pipe = 0.1894 miles
σ is the specific gravity of gas (air=1) = 0.7
T is the actual gas temperature = $530^o R$
d is the diameter of the pipe, (?) inches
η is the efficiency = 0.7 (a fudge factor!)

1. If the diameter of the pipe is 4.026 inches, determine the upstream pressure using the *secant* (initial guess of [5, 45]) and *fzero* (initial guess of 25) functions. Compare the *flops* which stands for the floating point operations.

2. Suppose the maximum pressure the pipeline can withstand is only 24.7 psia. Other conditions remaining the same as in previous part, determine the diameter of the pipe that should be used using the *secant* (initial guess of [4, 8]) and *fzero* (initial guess of 6) functions. Compare the *flops*.

2.12.3 A model for separation processes

Consider a stagewise separation process shown in Figure 1.2. A model for this process was developed in Chapter 1. The variables of interest are $(L, V, x_0, x_1, x_2, \cdots, x_n, y_1, y_2, y_3, \cdots, y_{n+1})$. Under the assumption of linear equilibrium model, $y_i = Kx_i$ it is possible to successively eliminate all of the variables and obtain the following single, analytical expression relating the input, (x_0, y_{n+1}), the output, x_n, the separation factor $S = L/KV$ and the number of stages n.

$$\boxed{\frac{[x_0 - x_n]}{[x_0 - y_{n+1}/K]} = \frac{\left[(1/S)^{n+1} - (1/S)\right]}{[(1/S)^{n+1} - 1]}} \tag{2.16}$$

The equation is called the Kremser-Brown-Souders equation. We have a single equation relating six variables, viz. $(x_0, x_n, y_{n+1}, K, S, n)$. Given any five of these variables, we can solve for the 6th one. Your task is to formulate this problem as a root finding problem of the type

$$f(x; \boldsymbol{p}) = 0$$

where the unknown variable is associated with x and $\boldsymbol{p}$ is the parameter set consisting of the remaining five known values. Write an m-file to

represent the function in the form

$$f(x;\boldsymbol{p}) = \frac{(x_0 - x_n)}{(x_0 - y_{n+1}/K)} - \frac{\left[(1/S)^{n+1} - (1/S)\right]}{\left[(1/S)^{n+1} - 1\right]} = 0 \tag{2.17}$$

1. In a typical *design problem* you might be given the flow rates, (say $L = 10, V = 10$), the inlet compositions (say, $x_0 = 0.8, y_{n+1} = 0$) and a specified recovery ($x_n = 0.1615$). Your task is to determine the number of stages (n) required to meet the specifications. Take the equilibrium ratio to be $K = 0.8$. Here the unknown variable x is associated with n and the others form the parameter set. Solve for n using *secant* and *bisection* methods using initial guesses of $[10, 30]$. Report the number of iterations required for convergence to the MATLAB built-in convergence tolerance of $eps = 10^{-16}$. You can use the *secant.m* and *bisect.m* algorithms outlined in figures 2.4,2.3. You must construct a m-file to define the function represented by equation (2.17) in terms of the unknown. Here is a sample function for the first case.

```
function f=KBS1(x)
% Kremser-Brown-Souders equation
% number of stages is unknown i.e. solve for x=n

K=0.8; L=10; V=10;
x0 = 0.8; ynp1= 0; xn=0.1615; S=L/K/V; %Known values

m=length(x);
for i=1:m
 n=x(i);
 f(i) = (x0-xn)/(x0-ynp1/K) - ( (1/S)^(n+1) - (1/S)) ...
/ ( (1/S)^(n+1) - 1);
end
```

Make sure that you understand what the above function does! In the next two parts you will have to modify this function to solve for a different unknown! Create a file named `KBS1.m` and enter the above function. Then to solve the problem from within MATLAB enter

» **secant('KBS1',[10,30],eps,1)**

You may wish to plot the function to graphically locate the root using

```
» x=10:1:30;
» y=KBS1(x);
» plot(x,y)
```

or, you can do the same in just one line using

```
» plot([10:1:30],KBS1(10:1:30))
```

2. In a *performance analysis* problem, you will be analyzing an existing process with a known number of stages (say, $n = 10$). Suppose $x_0 = 0.8, y_{n+1} = 0, L = 10, x_n = 0.01488$. Find the amount of gas V that can be processed. Use an initial guess of $[5, 20]$ and $[5, 30]$ with both *secant* and *bisect* algorithms. Record and comment on your observations.

3. In another variation of the *performance analysis* problem, the amount of gas to be processed ($V = 10$) may be given. You will have to determine the exit composition x_n. Take $n = 20, x_0 = 0.8, y_{n+1} = 0, L = 10$. Try initial guesses of $[0, .2]$ and $[0, 1]$ on both bisection and secant algorithms. Record and comment on your observations.

2.12.4 Peng-Robinson Equation of State

The phase behavior of fluids can be predicted with the help of equations of state. The one developed by Peng & Robinson is particularly well tuned, accurate and hence is widely used. The equation is given below.

$$P = \frac{RT}{(V-b)} - \frac{a(T)}{V(V+b)+b(V-b)} \tag{2.18}$$

where

$$a(T) = 0.45724\frac{R^2T_c^2}{P_c}\alpha(T_r,\omega), \qquad b = 0.0778\frac{RT_c}{P_c}, \qquad \sqrt{\alpha} = 1 + m(1-\sqrt{T_r})$$
$$m = 0.37464 + 1.54226\omega - 0.26992\omega^2, \qquad T_r = T/T_c, \text{ and} \qquad Z = PV/RT.$$

Whenever (P, T) are given it is convenient to write equation (2.18) as a cubic equation in Z

$$Z^3 - (1-B)Z^2 + (A - 3B^2 - 2B)Z - (AB - B^2 - B^3) = 0 \qquad (2.19)$$

where $A = aP/(R^2T^2), B = bP/(RT)$.

Use equation (2.19) to compute the density of CO_2 in gmole/lit at $P = 20.684MPa$ and $T = 299.82^oK$. The critical properties required for CO_2 are $T_c = 304.2^oK, P_c = 7.3862MPa$ and $\omega = 0.225$, $R = 8314Pa\ m^3/kmol\ ^oK$.

a) Use the function *roots(c)* in MATLAB to find all the roots of the cubic equation (2.19) in terms of Z. In MATLAB, how does the function *roots* differ from the function *fzero*?

b) Use the *secant* method to find the *real* roots of the above equation.

c) After finding the compressibility Z from each of the above methods, convert it into molar density and compare with the experimental value of $20.814gmole/lit$

d) Consider the case where you are given (P,V) and you are asked to find T. Develop and implement the Newton iteration to solve for this case. Use the above equation to compute the temperature of CO_2 in ^oK at $P = 20.684 \times 10^6 Pa$ and $V = .04783lit/gmole$.

   Compare the number of iterations required to obtain a solution to a tolerance of $||f|| < 10^{-15}$ using an initial guess of $T = 250$ by Newton method with that required by the `secant` method with an initial guess of [200,310].

e) Suppose that you are given (T,V) and you are asked to find P, which form of equation will you choose? Eqn. (2.18) or Eqn.(2.19)? What method of solution would you recommend?

2.12.5 Transient heat conduction in semi-infinite slab

Many engineering problems can be cast in the form of determining the roots of a nonlinear algebraic equation. One such example arises in determining the time required to cool a solid body at a given point to a predetermined temperature level.

Consider a semi-infinite solid, initially at a temperature of $T_i = 200^oC$ and one side of it is suddenly exposed to an ambient temperature of $T_a = 70^oC$. The heat transfer coefficient between the solid and surroundings is $h = 525W/(m^{2o}C)$. The thermal conductivity of the solid is $k = 215W/m^oC$ and the thermal diffusivity of the solid is $\alpha = 8.4 \times$

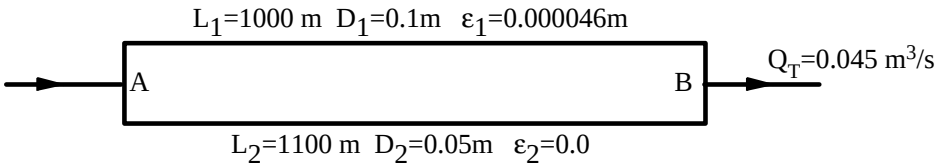

Figure 2.10: Turbulent flow in a parallel pipe

$10^{-5} m^2/s$. Determine the time required to cool the solid at a distance of $x = 4cm$ measured from the exposed surface, to $T = 120^o C$. The temperature profile as a function of time and distance is given by the following expression.

$$\theta = 1 - erf(\xi) - \left[e^{(hx/k+\tau)}\right][1 - erf(\xi + \sqrt{\tau})]$$

where the dimensionless temperature, $\theta = \frac{(T-T_i)}{(T_a-T_i)}$, and $\xi = \frac{x}{2\sqrt{\alpha t}}$ and $\tau = \frac{h^2 \alpha t}{k^2}$, t is the time, x is the distance and erf is the error function.

2.12.6 Turbulent flow in a parallel pipeline system

Consider the flow of an incompressible ($\rho = 1000 kg/m^3$), Newtonian fluid ($\mu = 0.001 Pa \cdot s$) in a parallel pipe system shown in figure 2.10. The lengths, diameters, roughness for the pipes as well as the total flow rate are as shown in figure 2.10. Your task is to determine the individual flow rates in each of the pipe segments 1 and 2. The equation to be satisfied is obtained based on the fact that the pressure drop between points A and B is the same. The equation is

$$f_1(v_1)\frac{L_1}{D_1}\frac{v_1^2}{2} = f_2(v_2)\frac{L_2}{D_2}\frac{v_2^2}{2} \tag{2.20}$$

where v_1, v_2 are the velocities in the two pipes and f_1, f_2 are the friction factors given by the Churchill equation.

$$f_i(v_i) = 8 * \left[\left(\frac{8}{Re_i}\right)^{12} + \frac{1}{(A+B)^{1.5}}\right]^{1/12}$$

where

$$A = \left[2.457 \ln\left(\frac{1}{(1/Re_i)^{0.9} + 0.27(\epsilon_i/D_i)}\right)\right]^{16} \qquad B = \left[\frac{37530}{Re_i}\right]^{16} \text{ and } Re_i = \frac{D_i v_i \rho}{\mu}$$

Finally the mass balance equation provides another constraint as,

$$\frac{\pi}{4}(D_1^2 v_1 + D_2^2 v_2) = Q_T$$

This problem can be formulated as two equation in two unknowns (v_1, v_2), but your task is to pose this as a single equation in one unknown, v_1, by rearranging equation 2.20 as,

$$F(v_1) = f_1(v_1)\frac{L_1}{D_1}\frac{v_1^2}{2} - f_2(v_2)\frac{L_2}{D_2}\frac{v_2^2}{2} = 0$$

i.e., for a given guess of v_1, write a m-file that will calculate $F(v_1)$. Then carryout the following calculations.

- Solve the problem using *secant* algorithm with initial guess of [4.5,5.5]. [Ans:$v_1 = 4.8703$]
- Suppose the total flow rate is increased to 0.09 m^3/s, what will be the new velocities in the pipes.
- Consider the case where $L_1 = 1000, L_2 = 900$ and $D_1 = 0.1, D_2 = 0.09$, other values being the same. Is there is flow rate Q_T for which the velocities in both pipes will be the same? If so what is it? [Ans:$Q_T = 0.0017$]
- Plot v_1 vs Q_T and v_2 vs Q_T for the case above over a suitable range of Q_T.
- Discuss the pros and cons of implementing the Newton method for this problem.