

Applied Engineering Problem Solving (CHEN.3170)

Part II: Numerical Methods and Applications

Lesson 5: Root Finding and Polynomial Manipulations

In the remainder of this course, each new lesson will introduce a new class of problems that routinely occurs in engineering analysis. A set of brief examples will be given to motivate the need for the development of one or more methods for handling problems of this type. Then, with an established need, we will overview some of the basic strategies that are used to solve this particular class of problems. Finally, we will focus, where possible, on existing Matlab routines for solving the problem of interest, and illustrate the practical use of the available tools in a variety of realistic applications.

In Lesson 5 we will deal primarily with the subject of root finding -- that is, given some function, $f(x)$, we ask the question, “What are the values of x such that $f(x) = 0$?”, and our goal is to develop and apply some numerical methods that allow us to answer this question for a variety of situations. In practice, there are two general classes of problems that occur:

- I. Find the **real roots** of algebraic and transcendental equations, where we usually search for a single root based on its approximate location. The procedure may be repeated, with a new starting guess, if multiple roots are required.
- II. Find all the roots, **both real and complex**, of a polynomial equation of the form

$$f(x) = a_1x^n + a_2x^{n-1} + \cdots + a_nx + a_{n+1} = 0 \quad (1)$$

where, for an n^{th} order polynomial, there are n roots. For this class of problems, we are often interested in all n roots.

Also, since dealing with polynomials is so common, we will introduce a sequence of Matlab commands for performing a variety of polynomial operations (addition, subtraction, multiplication, and division -- as well as root finding).

The subjects of root finding and polynomial manipulation are discussed in parts of Chapters 8 and 9 in your Matlab text by Gilat, and also in some detail in Chapters 5 and 6 of your Numerical Methods textbook by Chapra. You should study these chapters of your texts, paying particular attention to the material in the two chapters in Chapra -- since he does a pretty nice job of highlighting the more important aspects of the key root finding methods.

Note: Chapter 7 of your text by Chapra introduces the subject of *Optimization Methods*, which are closely related to the root finding techniques discussed in Chapters 5 and 6. Although we will not formally discuss these methods as part of this course (simply not enough time), you should at least be aware that techniques for finding the minima of functions do exist. Thus, I suggest that you at least browse Chapter 7 in your text just to get a rough idea/overview of this interesting subject. Note also that we will briefly touch upon this subject again as part of the *Comprehensive Analysis of a Slanted Gate* illustrative demo at the end of this section of notes (see [slanted_gate_1.pdf](#) for more details).

Motivation

To motivate your study of this subject, let's identify a few situations where root-finding techniques are required. In particular, consider the following four problem scenarios:

Problem 1: Volume of Liquid in a Horizontal Cylindrical Tank

The volume of liquid in a horizontal cylindrical tank is given by

$$V = \left[R^2 \cos^{-1} \left(\frac{R-h}{R} \right) - (R-h) \sqrt{2Rh - h^2} \right] L \quad (2)$$

where R is the tank inside radius, L is the length, and h is the height of liquid.

- Consider a tank with $R = 2.5$ m and $L = 5$ m. If the fluid height is 3.0 m, what is the fluid volume in the tank?
- If another 1.5 m^3 of liquid is added to the tank in Part a, what will be the new value of fluid height?

Since eqn. (2) expresses the volume as an explicit function of fluid height, the solution to Part a is quite straightforward -- we simply put in all the known values (R , L , and h) on the RHS of eqn. (2) and solve for V . Now, Part b poses a completely different situation. Here we know R , L , and V , and we are asked to determine h . However, eqn. (2) cannot be written as an explicit equation of the form, $h = f(R, L, V)$. Instead, the best we can do is to write the implicit relationship, $f(R, L, V, h) = 0$, and with known R , L , and V , this can be written as $f(h) = 0$. So our goal is to find the value of h such that $f(h) = 0$. Thus, Problem 1b is indeed a classical root-finding problem, and clearly we need to develop some techniques for handling this situation.

Note: Just in case you are interested, eqn. (2) for the volume of fluid in a horizontal cylindrical tank can be developed using the notation associated with the geometry of a circle (see sketch). The area of the segment enclosed by the points ABCDA, A_{segment} , is given by

$$A_{\text{segment}} = \int r dr d\theta = \frac{R^2}{2} \int d\theta = \frac{R^2}{2} \theta$$

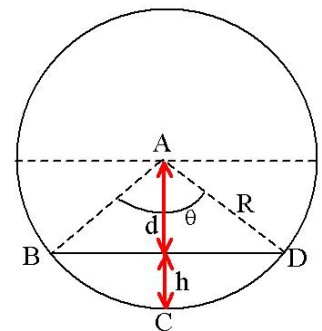
where θ can be written as

$$\cos\left(\frac{\theta}{2}\right) = \frac{d}{R} \quad \text{or} \quad \theta = 2 \cos^{-1}\left(\frac{d}{R}\right)$$

But $d = R-h$, so we can write θ as $\theta = 2 \cos^{-1}((R-h)/R)$, and the area of the segment becomes

$$A_{\text{segment}} = R^2 \cos^{-1}\left(\frac{R-h}{R}\right)$$

Now, the area occupied by the fluid, A_{fluid} , is enclosed by the points BCDB and this is given by the area of the segment defined above, A_{segment} , minus the area of the triangle, A_{triangle} , enclosed by the points ABDA. We can easily write the area of the triangle as



$$A_{\text{triangle}} = 2 \left(\frac{1}{2} \left(\frac{L_{BD}}{2} \right) d \right) = \left(\sqrt{R^2 - d^2} \right) d = \left(\sqrt{R^2 - (R-h)^2} \right) (R-h) = (R-h) \sqrt{2Rh - h^2}$$

so the area of interest can be written as

$$A_{\text{fluid}} = A_{\text{segment}} - A_{\text{triangle}} = R^2 \cos^{-1} \left(\frac{R-h}{R} \right) - (R-h) \sqrt{2Rh - h^2}$$

This area times the length of the horizontal cylinder, L , gives the desired volume -- which is the result stated in eqn. (2).

Problem 2: van der Waal's Equation of State

The formal relationship between pressure, temperature, and volume of a gas is called its equation of state. The simplest and most commonly used equation of state is the Ideal Gas Law,

$$Pv = RT \quad (3)$$

where P represents the pressure (atm), v refers to the molar volume (liters/gmole), R is the universal gas constant (0.08206 liters-atm/gmole-K), and T represents the absolute temperature (K).

However, several other relationships that are more accurate, but also more complicated, have also been developed. In particular, van der Waal's equation of state is given as

$$\left(P + \frac{a}{v^2} \right) (v - b) = RT \quad (4)$$

where a and b are constants for the gas of interest.

- a. The ideal gas law is certainly easier to use, but imagine that a particular application needs as much accuracy as possible. Thus, you are asked to provide a plot of molar volume versus temperature for several pressures of interest using the more complicated, but more accurate, van der Waal equation. In the application of interest, ammonia is used as the working fluid and the ranges of interest for the pressure and temperature are given as follows:

Pressure: 1, 3, and 5 atms and Temperature: $250 < T < 400$ K

with the van der Waal constants for ammonia as follows:

$a = 4.19 \text{ atm} \cdot (\text{liters/gmole})^2$ and $b = 0.0372 \text{ liters/gmole}$

- b. To compare how closely the van der Waal equation is to the Ideal Gas Law, one can compute and plot the compressibility factor, Z , which is given as,

$$Z = \frac{Pv}{RT} \quad (5)$$

Note that for the Ideal Gas Law, Z is always unity, but for other forms for the equation of state it will deviate somewhat from unity -- and this is a measure of the error involved if one decides to use the Ideal Gas Law instead of the van der Waal equation of state.

Provide a plot of this factor for ammonia using the van der Waal equation of state, and comment on the suitability of the simple Ideal Gas Law for ammonia over the range of pressures and temperatures of interest for this problem.

Part b of this problem is straightforward once we have computed the molar volume for the pressures and temperatures indicated in Part a. However, writing eqn. (4) as an explicit relationship, $v = f(P, T)$, for the solution of Part a is not possible. Note, however, that for a given combination of P and T , eqn. (4) can be written in implicit form as

$$f(v) = \left(P + \frac{a}{v^2} \right) (v - b) - RT = 0 \quad (6)$$

In this form, we see that the question of interest is, “What is the value of v such that $f(v) = 0$?”. Thus, this problem is also a root finding problem!

Note: With a little algebra, eqn. (4) can be rewritten as a cubic polynomial in v , as follows:

$$(Pv^2 + a)(v - b) - RTv^2 = 0$$

or

$$Pv^3 - (Pb + RT)v^2 + av - ab = 0 \quad (7)$$

Thus, one could use eqn. (7) instead of eqn. (6), if desired. However, I would recommend eqn. (6) for this problem since it avoids the algebra needed to derive eqn. (7) and, in this form, we would look for only the real root that is close to the value given by the Ideal Gas Law in eqn. (3). If one uses eqn. (7), you might be tempted to find all three roots of the cubic polynomial, but then you would have to decide which one is the valid solution to the problem. Both approaches will work, but the first method that uses eqn. (6) is more efficient.

Problem 3: Implicit Solutions to IVPs

The implicit solution to the initial value problem (IVP) defined by

$$y' = \frac{-(2xy^3 + y^4)}{xy^3 - 2} \quad \text{with } y(0) = 1 \quad (8)$$

can be written as

$$u(x, y) = x^2 + xy + y^{-2} - 1 = 0 \quad (9)$$

Plot the solution, $y(x)$, over the domain $0 \leq x \leq 2$.

Obtaining the analytical solution to the given IVP is a lot of work (see below), but once you have a solution, you would think that creating a plot of $y(x)$ should be straightforward. However, for many situations, the solution to the IVP is in the form of an implicit relationship between the independent and dependent variables (x and y , for this problem). Although eqn. (9) is a valid solution to eqn. (8) (can you prove this?), we cannot write eqn. (9) in the form of an explicit solution, $y = f(x)$. Instead, as in the previous examples, an implicit relationship between x and y ,

given by $u(x,y) = 0$, is the best that we can do. Thus, once again, we see that a root finding technique is required so that, given x , we can “find y such that $u(y) = 0$ ”.

Note: The derivation of eqn. (9) was a bit of work, so I will include it here so that you won't have to struggle with it. We start by writing eqn. (8) as follows:

$$\frac{dy}{dx} = \frac{-(2xy^3 + y^4)}{xy^3 - 2}$$

and some algebraic manipulations gives

$$(xy^3 - 2)dy = -(2xy^3 + y^4)dx$$

$$\text{or} \quad (2xy^3 + y^4)dx + (xy^3 - 2)dy = 0 \quad (10)$$

This expression is in the form

$$M(x, y)dx + N(x, y)dy = 0 \quad (11)$$

The LHS of this equation is of the form of an exact differential

$$du = \frac{\partial u}{\partial x} dx + \frac{\partial u}{\partial y} dy \quad (12)$$

if $\partial M / \partial y = \partial N / \partial x$.

Thus, computing these partial derivatives gives

$$\frac{\partial M}{\partial y} = \frac{\partial}{\partial y}(2xy^3 + y^4) = 6xy^2 + 4y^3$$

and

$$\frac{\partial N}{\partial x} = \frac{\partial}{\partial x}(xy^3 - 2) = y^3$$

which says that the original expression is not exact!

One possible technique for resolving this situation is to find an integrating factor, $g(y)$, that, when used with the original equation, makes the LHS exact. (Note that one often tries an integrating factor, $g(x)$, first -- but, in this case, that approach failed. Thus, I tried to find an integrating factor that is only a function of y .) To develop an expression for $g(y)$, we multiply eqn. (11) by $g(y)$, giving

$$gMdx + gNdy = 0$$

Now, by definition, for this expression to be exact, the following relationship must be valid,

$$\frac{\partial(gM)}{\partial y} = \frac{\partial(gN)}{\partial x}$$

and, with the integrating factor only a function of y [i.e. $g(x,y) \rightarrow g(y)$], this becomes

$$g \frac{\partial M}{\partial y} + M \frac{dg}{dy} = g \frac{\partial N}{\partial x}$$

which reduces to

$$\frac{1}{g} \frac{dg}{dy} = \frac{1}{M} \left(\frac{\partial N}{\partial x} - \frac{\partial M}{\partial y} \right) \quad (13)$$

Now, if we can find some $g(y)$ that satisfies eqn. (13), it will be our desired integrating factor -- which, in turn, will make our original expression exact!

To determine $g(y)$ for the given problem, we simply substitute the appropriate expressions on the RHS of eqn. (13), or

$$\frac{1}{g} \frac{dg}{dy} = \frac{1}{2xy^3 + y^4} (y^3 - (6xy^2 + 4y^3)) = \frac{-(6xy^2 + 3y^3)}{2xy^3 + y^4}$$

which reduces to

$$\frac{1}{g} \frac{dg}{dy} = \frac{-3(2xy^2 + y^3)}{y(2xy^2 + y^3)} = -\frac{3}{y}$$

Thus, g is indeed only a function of y , and the solution of this separable ODE is given as

$$\frac{dg}{g} = -3 \frac{dy}{y}$$

which integrates to

$$\ln g = -3 \ln y = \ln y^{-3}$$

or

$$g(y) = y^{-3} \quad (14)$$

Now, with $g(y)$ known, we multiply the original ODE in eqn. (10) by $g(y)$, giving

$$(2x + y)dx + (x - 2y^{-3})dy = 0 \quad (15)$$

With the new correspondence between eqn. (15) and the $M(x,y)$ and $N(x,y)$ functions in eqn. (11), we can recheck to determine if the new equation is indeed exact. Doing this gives

$$\frac{\partial M}{\partial y} = \frac{\partial}{\partial y} (2x + y) = 1$$

and

$$\frac{\partial N}{\partial x} = \frac{\partial}{\partial x} (x - 2y^{-3}) = 1$$

Therefore, the ODE defined by eqn. (15) is now exact!

Now, since the LHS of eqn. (15) is exact, we know that $M = \partial u / \partial x$, which gives

$$u(x, y) = \int \frac{\partial u}{\partial x} \partial x = \int M(x, y) \partial x = \int (2x + y) \partial x = x^2 + xy + f(y) \quad (16)$$

where the constant of integration, $f(y)$, can be a function of y (because of the partial integration with respect to x).

But, we also know that $N = \partial u / \partial y$, which leads to the following equality

$$\frac{\partial u}{\partial y} = \frac{\partial}{\partial y} [x^2 + xy + f(y)] = x + \frac{df}{dy} = N(x, y) = x - 2y^{-3}$$

and, from this series of relationships, we have

$$\frac{df}{dy} = -2y^{-3}$$

which gives

$$f(y) = y^{-2} + c \quad (17)$$

where c is now just a simple constant of integration.

Finally, we can write the general solution to eqn. (8) as the combination of eqns. (16) and (17),

$$u(x, y) = x^2 + xy + y^{-2} + c = 0 \quad (18)$$

Now, to get the final result -- the unique solution -- we simply apply the initial condition, $y(0) = 1$, or

$$u(0, 1) = 0 + 0 + 1 + c = 0 \quad \text{or} \quad c = -1$$

Thus, the unique solution to the given IVP is

$$u(x, y) = x^2 + xy + y^{-2} - 1 = 0$$

and this is the implicit solution that was simply given in the original problem statement -- which completes our derivation.

Note: All this detail was given here for two reasons:

1. As a refresher from your Differential Equations course, and
2. So you appreciate how easy the numerical solution of an IVP is relative to the analytical solution -- since we will revisit this example again in Lesson 8 when we are discussing numerical solutions to ODEs (if we get that far in the course...).

Problem 4: Roots of the Characteristic Equation

Find the general solution to the following 4th order ODE:

$$y^{(4)} + 3y' - 4y = 0 \quad (19)$$

Since this is a homogeneous linear constant coefficient system, you know from your Differential Equations class that a solution of the form $y \rightarrow e^{rx}$ is valid. Upon substitution into eqn. (19), this gives the characteristic equation,

$$r^4 + 3r - 4 = 0 \quad (20)$$

This is a 4th order polynomial that has 4 roots, r_1 , r_2 , r_3 , and r_4 , and, if the roots are distinct, the general solution can be written as

$$y(x) = c_1 e^{r_1 x} + c_2 e^{r_2 x} + c_3 e^{r_3 x} + c_4 e^{r_4 x} \quad (21)$$

Thus, our real goal here is to find all four roots of eqn. (20) -- since doing so will allow us to write eqn. (21) with explicit values for the r_i terms.

Clearly, this problem falls into our second class of root finding problems -- find all the roots of a polynomial equation -- and we will briefly discuss some general methods for addressing such problems later in this section of notes.

Root Finding Methods

Hopefully, the above four problem scenarios have demonstrated the need for the development of root finding techniques. Now, with an established need, the remainder of this section will highlight some basic strategies for solving this class of problems. Once we have a good handle on the basic solution methodology, we will overview some of Matlab's built-in capability for root finding. Finally, with the appropriate tools at our disposal, we will actually solve the four problems posed above!!!

To start our discussion of root finding methods, let's first focus on the problem of "finding real values of x such that $f(x) = 0$ ". There are two general approaches for finding real roots of nonlinear equations -- **Bracketing Methods** and **Open Methods** -- and they both have certain advantages and disadvantages. Bracketing methods require two initial guesses that are on either side of a root. This bracketing of the root is maintained as the solution algorithm continually reduces the size of the bracket containing the root until convergence is reached [i.e., when the value of $f(x_r) < \text{tol}$]. Bracketing methods are always convergent, but the rate of convergence is usually relatively slow. Open methods, on the other hand, use an initial guess or guesses that do not need to bracket a root. Information about the function and its derivative at the root estimate are usually used to extrapolate to a new root location. However, open methods are not guaranteed to converge, and the success of the method may be dependent on the goodness of the initial guess. Thus,

Bracketing Methods -- always converge, but are relatively slow, and

Open Methods -- may diverge, but they usually converge quite rapidly when they work.

Note, however, that at the expense of some additional programming detail, it is possible to combine a particular bracketing method and an open technique to give a **Hybrid Method** that

keeps the best qualities of both methods -- producing a fast method which always converges. In particular, the *fzero* routine in Matlab uses a hybrid approach to give a practical tool that is both robust and efficient.

We will now explore the basic ideas behind bracketing and open methods and then briefly overview how *fzero* incorporates elements from both these techniques to create a powerful hybrid method for general root finding applications.

Bracketing Methods

All bracketing methods are based on the fact that, if $f(x)$ is continuous, there is at least one zero crossing (i.e., a root) within the interval $a \leq x \leq b$ if $f(a)f(b) < 0$. This says that, if the product of the function evaluated at points a and b is negative (i.e., $f(a)$ and $f(b)$ have different signs), then there must be at least one zero crossing within the interval $[a,b]$. In fact, as seen in the sketches in Fig. 1, if there is a sign change in the function from point a to point b , then there must be an odd number of roots within the interval, and this knowledge can be used to develop an algorithm that will always converge on a root.

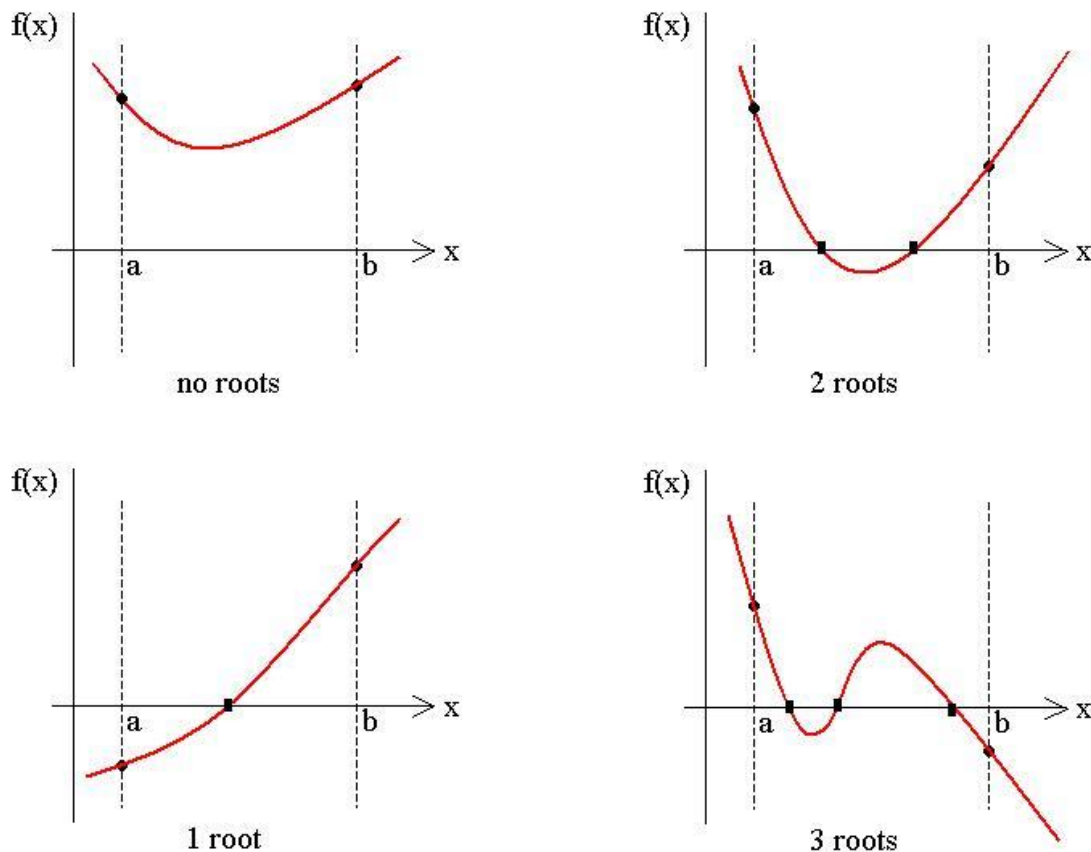


Fig. 1 Basic concept for Bracketing Methods.

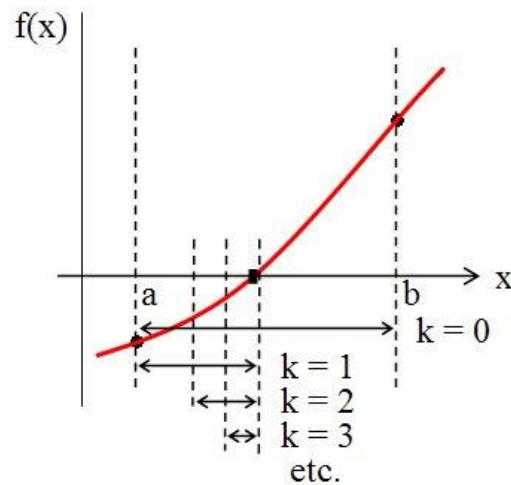
In particular, the **Bisection Method** is the most commonly employed bracketing algorithm. In this method, one simply estimates the root to be at the midpoint of the current interval,

$$x_r = a + (b - a)/2 = (a + b)/2 \quad (22)$$

Then, if $f(x_r)f(a) < 0$, the root is in the left half of the current interval, and we move the upper limit down to x_r (i.e., redefine $b = x_r$). If instead, $f(x_r)f(a) > 0$, the root is in the right half of the interval $[a,b]$, and we reset the lower limit to $a = x_r$. This sequence is repeated until $f(x_r)$ is less than some user-specified value, at which time the method is said to have converged.

The sequence of steps for the **Bisection Method** is illustrated in the sketch shown below, which includes the function graph from the lower left plot in Fig. 1 with three interval-halving steps shown explicitly on the plot. In addition to this graphical representation, an algorithm for actual implementation of the method is outlined below:

1. Choose $b > a$ such that $f(a)f(b) < 0$ which guarantees at least one root within the interval $[a,b]$. Also set the convergence criterion, tol , and the maximum number of iterations, M .
2. Evaluate eqn. (22) and compute $f(x_r)$.
3. If $f(x_r)f(a) < 0$, set $b = x_r$ (root is in the first half interval). If not, set $a = x_r$.
4. Increment iteration counter, $k = k + 1$.
5. If $k \leq M$ and $|f(x_r)| > \text{tol}$, go to Step 2.
6. At this point, if $k-1 \leq M$, x_r is the desired estimate of the root -- use it as needed.



This sequence is fairly simple and easy to program and use -- and it works!!! This algorithm has been implemented into the **bisection.m** program listed in Table 1.

As an illustration of how to use the **Bisection Method**, let's use the **bisection.m** function to find the roots of

$$f(x) = e^{-x} - x \quad (23)$$

To do this we need to write a function file to evaluate eqn. (23) for any input x . This is a simple task, and the resultant function, **rroots_1.m**, is listed below:

```
%
% RROOTS_1.M   Function evaluation for sample root finding problem
%
% Function of interest:   f(x) = exp(-x) - x
%
%   function f = rroots_1(x)
%       f = exp(-x) - x;
%
% end of function
```

Table 1 Listing of the bisection.m Matlab routine.

```

%
% BISECTION.M Routine to implement the Bisection Method
%
% Inputs: f = function handle for user-supplied function.
%           The function f is of the form: function y = f(x)
%           a,b = lower and upper limits (where b > a)
%           tol = convergence criterion (tol > 100*eps)
%           M = maximum number of iterations (M >= 2)
%           display = intermediate results are displayed if > 0
%
% Outputs: x = estimated root of f(x) = 0
%           k = number of iterations performed (note that the number
%           of function evaluations is k + 2)
%
% Note that a and b must be selected such that f(a)*f(b) < 0 to ensure that
% there is a root in [a,b]
%
% File prepared by J. R. White, UMass-Lowell (last update: Nov. 2017)
%
function [x,k] = bisection(f,a,b,tol,M,display)
%
% check bounds, tolerance, # iterations, and display switch
fa = f(a); fb = f(b);
if fa*fb >= 0
    fprintf('\n Warning -- In bisection.m it is required that f(a)f(b) < 0.\n\n');
    return
end
if tol < 100*eps, tol = 100*eps; end
if M < 2, M = 2; end
if display > 0
    fprintf('\n\n Intermediate edit from the Bisection Method: \n\n');
    fprintf('      k      a      f(a)      b      f(b)      xr      f(xr)
sign(fa*fr)\n');
end
%
% find root
k = 1; err = tol + 1;
while (err > tol) && (k <= M)
    xr = a + (b-a)/2; fr = f(xr);
    if display > 0
        fprintf(' %3i %10.6f %10.6f %10.6f %10.6f %10.6f %10.6f %6i \n', ...
            k,a,fa,b,fb,xr,fr,sign(fa*fr));
    end
    if fa*fr < 0
        b = xr; fb = fr;
    else
        a = xr; fa = fr;
    end
    err = abs(fr); k = k + 1;
end
x = xr; k = k-1; % save outputs
if k == M
    fprintf('\n\n Warning: The maximum number of iterations was reached!!! \n');
end
%
% end of function

```

Now, to find the roots of the nonlinear function given in eqn. (23), we can simply call **bisection.m** as follows:

```

[xr,k] = bisection(@rroots_1,0,1,1e-6,30,1);
fprintf('\n Zero of f(x) = exp(-x) - x occurs at x = %12.7f \n',xr);
fprintf(' The value of f(x) at this point is = %12.7f \n',rroots_1(xr));
fprintf(' The number of function evaluations needed was = %6i \n',k+2);

```

where we have chosen the bounds to be $a = 0$ and $b = 1$, set the convergence limit to $\text{tol} = 10^{-6}$, set the maximum number of iterations to 30, and turned on the intermediate edit capability in **bisection.m**. The result from the above the commands (without the intermediate edit) is:

```
Zero of f(x) = exp(-x) - x occurs at x =      0.5671434
The value of f(x) at this point is =      -0.0000002
The number of function evaluations needed was =      22
```

Thus, the result is $x_r = 0.56714$ and this is usually all that is needed from a root finding routine. However, in this example, we are trying to illustrate the overall procedure associated with bracketing methods, in general, and the **Bisection Method**, in particular. As such, the intermediate edit capability was turned on and the resultant edit is reproduced below:

Intermediate edit from the Bisection Method:

k	a	f(a)	b	f(b)	xr	f(xr)	sign(fa*fr)
1	0.000000	1.000000	1.000000	-0.632121	0.500000	0.106531	1
2	0.500000	0.106531	1.000000	-0.632121	0.750000	-0.277633	-1
3	0.500000	0.106531	0.750000	-0.277633	0.625000	-0.089739	-1
4	0.500000	0.106531	0.625000	-0.089739	0.562500	0.007283	1
5	0.562500	0.007283	0.625000	-0.089739	0.593750	-0.041498	-1
6	0.562500	0.007283	0.593750	-0.041498	0.578125	-0.017176	-1
7	0.562500	0.007283	0.578125	-0.017176	0.570313	-0.004964	-1
8	0.562500	0.007283	0.570313	-0.004964	0.566406	0.001155	1
9	0.566406	0.001155	0.570313	-0.004964	0.568359	-0.001905	-1
10	0.566406	0.001155	0.568359	-0.001905	0.567383	-0.000375	-1
11	0.566406	0.001155	0.567383	-0.000375	0.566895	0.000390	1
12	0.566895	0.000390	0.567383	-0.000375	0.567139	0.000007	1
13	0.567139	0.000007	0.567383	-0.000375	0.567261	-0.000184	-1
14	0.567139	0.000007	0.567261	-0.000184	0.567200	-0.000088	-1
15	0.567139	0.000007	0.567200	-0.000088	0.567169	-0.000041	-1
16	0.567139	0.000007	0.567169	-0.000041	0.567154	-0.000017	-1
17	0.567139	0.000007	0.567154	-0.000017	0.567146	-0.000005	-1
18	0.567139	0.000007	0.567146	-0.000005	0.567142	0.000001	1
19	0.567142	0.000001	0.567146	-0.000005	0.567144	-0.000002	-1
20	0.567142	0.000001	0.567144	-0.000002	0.567143	-0.000000	-1

You should really take a few minutes to follow the basic algorithm, step by step, to assure that you understand what is happening here. We see that it takes 20 iterations to meet the specified tolerance for this problem, and careful examination of the above table shows how the lower or upper limit was reset on each step to always bracket the actual root within the bounds set by the current a and b values. Note that the last column, with either a +1 or -1, simply indicates whether the root is in the right or left half of the current interval and which bound needs to be adjusted. Overall, the method is easy to understand and to use, and it always works if the original bounds satisfy the constraint that $f(a)f(b) < 0$.

Open Methods

As indicated above, open methods tend to converge more rapidly than bracketing methods when they work but, unfortunately, they don't always work! The basic idea behind all open techniques is quite straightforward. Instead of looking for a zero crossing as done for bracketing methods, open methods search for a value of x where two functions are equal. To see this, let's start with the original statement that $f(x) = 0$, where our goal, of course, is to find a real value of x that satisfies this condition. Now, we can always write $f(x)$ as the difference between two functions, or

$$f(x) = h(x) - g(x) = 0 \quad (24)$$

which leads to

$$h(x) = g(x) \quad (25)$$

In this form, our goal is to find the real value of x such that functions $h(x)$ and $g(x)$ are equal.

Now, we are free to choose $h(x)$ and $g(x)$ as desired as long as eqn. (24) is satisfied. In most cases, we simply let $h(x) = x$, and eqn. (25) becomes

$$x = g(x) \quad (26)$$

This form of an equation, where the quantity of interest appears on both sides of the equation, suggests that an iterative process should be used to find x (the quantity of interest). To emphasize the iterative nature of the algorithm that will be used to solve eqn. (26) for x , we often add a subscript (or sometimes a superscript) to the dependent variable to display the iteration counter. After doing this, eqn. (26) becomes

$$x_{k+1} = g(x_k) \quad (27)$$

In this expression, $g(x)$ is called the iteration function and the method is often referred to as the One-Point Iteration or Fixed-Point Iteration scheme. The subscript k is the iteration counter.

This method is particularly simple, and it is often the method of choice when doing hand calculations. For example, if our goal is to find the root of $f(x)$ as given in eqn. (23), then we could solve it as

$$x = e^{-x} \quad \text{which gives} \quad x_{k+1} = e^{-x_k} = g_1(x_k) \quad (28a)$$

or as

$$x = -\ln x \quad \text{which gives} \quad x_{k+1} = -\ln x_k = g_2(x_k) \quad (28b)$$

If we make our first guess $x_1 = 1.0$, the following iteration table can be generated:

k		x_k	$g_1(x_k) = e^{-x_k}$		x_k	$g_2(x_k) = -\ln x_k$
1		1.0	0.36788		1.0	0.0
2		0.36788	0.69220		0.0	undefined
3		0.69220	0.50047		(we have divergence)	
4		0.50047	0.60625			
5		0.60625	0.54539			
6		0.54539	0.57962			
7		0.57962	0.56011			
8		0.56011	0.57115			
9		0.57115	0.56488			
10		0.56488	0.56843			

Clearly, we see that eqn. (28a) is converging where, after 10 manual iterations, we have two significant digits of accuracy (recall that the actual root occurs at $x_r = 0.56714$). However, although mathematically equivalent, eqn. (28b) fails miserably. This is a good example of the primary disadvantage associated with all open methods -- sometimes they work and sometimes they don't!

Now, the one-point iteration scheme outlined above is not usually implemented in this fashion for computer analysis because of the somewhat arbitrary nature for choosing the iteration function, $g(x)$. Instead, the most common one-point iteration algorithm used in practice, which has a specific representation for $g(x)$, is the **Newton-Raphson (NR) method** (often simply referred to as **Newton's method**). The iteration formula for this method is easily derived using a truncated Taylor series expansion for the original function, $f(x)$, evaluated at the root estimate, x_k . Rewriting eqn. (8) from the Lesson 4 Lecture Notes for the Taylor series using the current notation, gives

$$f_{k+1} = f_k + f'_k (x_{k+1} - x_k) + O(\Delta x^2) \quad (29)$$

where we have truncated the series after the 1st order term. Dropping the error term and solving this expression for x_{k+1} gives

$$x_{k+1} - x_k = \frac{f_{k+1} - f_k}{f'_k}$$

or

$$x_{k+1} = x_k + \frac{f_{k+1} - f_k}{f'_k} \quad (30)$$

Now, the next guess for a root, x_{k+1} , will be chosen so that, by definition, $f(x_{k+1}) = f_{k+1} = 0$ (i.e., if x_{k+1} is the root, then $f_{k+1} = 0$). Thus, eqn. (30) reduces to

$$x_{k+1} = x_k - \frac{f_k}{f'_k} \quad (31)$$

which is the iteration equation for **Newton's method**.

For general purpose use, it is often inconvenient to require the evaluation of both $f(x)$ and $f'(x)$ at each root estimate, x_k . Because of this, the **Newton method**, which formally requires user-defined functions to evaluate both $f(x_k)$ and $f'(x_k)$, is often replaced by the **Secant method** -- which is similar to the NR method except that f'_k is approximated by a finite difference formula. In particular, we can use a first-order backward approximation to f'_k (see eqn. (11) in the Lesson 4 Lecture Notes), or

$$f'_k = \frac{f_k - f_{k-1}}{x_k - x_{k-1}} \quad (32)$$

Now, substitution of eqn. (32) into eqn. (31) gives the **Secant method**,

$$x_{k+1} = x_k - f_k \left[\frac{x_k - x_{k-1}}{f_k - f_{k-1}} \right] \quad (33)$$

This iteration formula now requires two starting guesses like a bracketing technique but, since this is an open method, there is no requirement that the original two guesses bracket a root. However, it does make sense to provide reasonable guesses, as appropriate, for the problem of interest.

A formal algorithm for implementation of the **Secant method** is given below:

1. Make two guesses for the root, x_{k-1} and x_k , for $k = 1$. Also set the convergence criterion, tol , and the maximum number of iterations, M .
2. Evaluate eqn. (33) and compute $f(x_{k+1}) = f_{k+1}$.
3. Increment the iteration counter, $k = k+1$.
4. If $k \leq M$ and $|f(x_{k+1})| > \text{tol}$, go to Step 2.
5. At this point, if $k-1 \leq M$, x_{k+1} is the desired root estimate -- use it as desired.

This sequence is quite simple and easy to program. This algorithm for the **Secant method** has been implemented into the **secant.m** Matlab function file listed in Table 2.

To illustrate the use of the **secant.m** function and to show how the iteration sequence for an open method differs from a bracketing method, we can re-solve eqn. (23) using the **Secant method**. Of course, we will use the same equation file, **rroots_1.m**, as before for evaluating $f(x)$ (see pg. 10 of these notes). We can also call **secant.m** using a similar sequence of commands from the Matlab prompt as done previously for the **Bisection method** (by simply replacing the call to **bisection.m** with a call to **secant.m**):

```
[xr,k] = secant(@rroots_1,0,1,1e-6,20,1);
fprintf('\n Zero of f(x) = exp(-x) - x occurs at x = %12.7f \n',xr);
fprintf(' The value of f(x) at this point is = %12.7f \n',rroots_1(xr));
fprintf(' The number of function evaluations needed was = %6i \n',k+2);
```

where the two initial guesses are $x_0 = 0$ and $x_1 = 1$, the convergence criterion is again set to 10^{-6} , the maximum number of iterations is 20, and the intermediate edit switch has been activated. With these settings, the above commands give the following output (including the intermediate edit):

Intermediate edit from the Secant Method:

k	x_0	f_0	x_1	f_1	x_2	f_2
1	0.000000	1.000000	1.000000	-0.632121	0.612700	-0.070814
2	1.000000	-0.632121	0.612700	-0.070814	0.563838	0.005182
3	0.612700	-0.070814	0.563838	0.005182	0.567170	-0.000042
4	0.563838	0.005182	0.567170	-0.000042	0.567143	-0.000000

```
Zero of f(x) = exp(-x) - x occurs at x = 0.5671433
The value of f(x) at this point is = -0.0000000
The number of function evaluations needed was = 6
```

Clearly, the **Secant method** gives the correct result since it is the same as for the **Bisection method**. However, it arrived at the same answer with only 4 iterations (and 6 function evaluations), which is significantly faster than for the **Bisection method** (which required 22 function evaluations), and this is typical of the improved efficiency of an open method over a bracketing technique.

Also of interest here is the actual sequence of intermediate results from the **Secant method** for this problem. As before, I strongly encourage you to review this short table of results, step by step, so that you have a good handle of the actual computational algorithm for the **Secant method!!!**

Table 2 Listing of the secant.m Matlab routine.

```
%
% SECANT.M Routine to implement the Secant Method
%
% Inputs: f = function handle for user-supplied function.
%          The function f is of the form: function y = f(x)
%          x0,x1 = two initial guesses for the root location
%          tol = convergence criterion (tol > 100*eps)
%          M = maximum number of iterations (M >= 2)
%          display = intermediate results are displayed if > 0
%
% Outputs: x = estimated root of f(x) = 0
%          k = number of iterations performed (note that the number
%            of function evaluations is k + 2)
%
% File prepared by J. R. White, UMass-Lowell (last update: Nov. 2017)
%
% function [x,k] = secant(f,x0,x1,tol,M,display)
%
% check tolerance, # iterations, initial guesses, and display switch
% if tol < 100*eps, tol = 100*eps; end
% if M < 2, M = 2; end
% if abs(x1-x0) < tol
%     fprintf('The Secant Method requires that x1 be different from x0.\n');
%     return
% end
% if display > 0
%     fprintf('\n\n Intermediate edit from the Secant Method: \n\n');
%     fprintf('      k      x0      f0      x1      f1      x2      f2\n');
% end
%
% find root
% f0 = f(x0); f1 = f(x1);
% k = 1; err = tol + 1;
% while (err > tol) && (k <= M);
%     x2 = x1 - (x1-x0)*f1/(f1-f0); f2 = f(x2);
%     if display > 0
%         fprintf(' %3i %10.6f %10.6f %10.6f %10.6f %10.6f %10.6f \n', ...
%             k,x0,f0,x1,f1,x2,f2);
%     end
%     err = abs(f2); k = k + 1;
%     x0 = x1; f0 = f1; x1 = x2; f1 = f2;
% end
% x = x2; k = k-1; % save outputs
% if k == M
%     fprintf('\n\n Warning: The maximum number of iterations was reached!!! \n');
% end
%
% end of function
```

The fzero Routine

Matlab has a built-in function, **fzero**, for finding real roots of nonlinear equations. It is a particularly useful hybrid method that incorporates a combination of a bracketing method for guaranteed convergence and an open method (actually two open methods) for overall fast operation. The algorithm implemented within **fzero** is quite complicated (at least relative to the

simple ones we have discussed here), but the formal details are not really necessary for practical application of the routine. Chapra does a nice job of summarizing the basic hybrid root finding strategy within *fzero* in Section 6.5 of your text, and he also provides a detailed example of its use. I suggest that a review of this material, a quick look at the output that you get by typing *help fzero* in Matlab, and a few practical examples should be sufficient for most users to become proficient with using *fzero* for realistic applications.

In particular, the most common syntax for using *fzero* is given below:

set **xg** -- where **xg** is a single guess for the root or a two element vector that contains the lower and upper limits that bracket the root. If you only supply a single guess, *fzero* automatically searches for a bracket near the supplied guess.

set **options** -- where **options** is a particular data structure that allows the user to set the convergence tolerance and the output edit level. In most cases this is not needed.

solve for the root, **xr**, with the following command:

xr = fzero(@function_name, xg, options, p1, p2,...)

In the above command, **function_name** is the name of a user-defined Matlab function file that evaluates $f(x)$ for any scalar input x (and **@function_name** generates a function handle for use in *fzero*), and **p1, p2, ...** are additional optional parameters that can be passed to the function file.

As a particular example, let's solve for the root of the function given by eqn. (23) using *fzero*, since we are already familiar with this function with the **Bisection** and **Secant methods**. The needed command sequence can be input at the Matlab prompt as:

```
xr = fzero(@rroots_1,[0 1]);
fprintf('\n Zero of f(x) = exp(-x) - x occurs at x = %12.7f \n',xr);
fprintf(' The value of f(x) at this point is = %12.7f \n',rroots_1(xr));
```

with the following result echoed back to the screen,

```
Zero of f(x) = exp(-x) - x occurs at x = 0.5671433
The value of f(x) at this point is = 0.0000000
```

Here, we have used the built-in default **options**, did not require any additional parameters, and set the brackets directly within the *fzero* command. This format is the easiest and most common structure to use. And, of course, it gives the correct root for this problem!!!

Overall, the *fzero* function is easy to use, it works every time (if you have a well-behaved function), and it is quite efficient for most problems. In short, unless you have a very special problem that requires special attention, there is no reason to use anything but *fzero* for finding real roots of nonlinear equations. I have used it successfully, in every case, for the past 30+ years!!!

Polynomial Roots and Other Such Things...

As indicated previously (see Problem 4 under the Motivation Section), finding all the roots of polynomial equations is also an important task that needs to be addressed as part of many engineering analysis and design applications. In addition, performing a variety of algebraic manipulations with polynomials, such as addition, subtraction, multiplication, and division, as well as simple polynomial evaluation, are quite routine for many applications. Thus, we need to

provide some techniques and tools for performing these operations. Unfortunately, however, although polynomial evaluation and algebraic manipulation are quite straightforward, polynomial root finding is not -- and some fairly advanced techniques are needed to accomplish this task. At this point in our study of numerical methods we do not have all the necessary background so, in this course, we will not go into any real details of the polynomial root finding methods. Instead, we will only briefly overview the basic ideas and then focus on using the built-in capability within Matlab! Thus, we will leave the formal methodology behind Matlab's **roots** command for another day -- possibly as individual study for the interested student, or as part of a more advanced numerical methods course taken as a senior elective or as part of your graduate studies...

To start our discussion of polynomials, we first recall that an n^{th} order polynomial can always be written as

$$f(x) = a_1x^n + a_2x^{n-1} + \cdots + a_nx + a_{n+1} \quad (34)$$

Note, in particular, that **we have written the polynomial with decreasing powers of x** -- since this is consistent with the way Matlab works with these functions. Now, if we know that $f(x)$ is a polynomial function, we can summarize the function by simply listing the coefficients in eqn. (34),

$$f = [a_1 \ a_2 \ a_3 \ \cdots \ a_n \ a_{n+1}]$$

This sequence of numbers forms a row vector and, of course, this is a simple structure to represent in Matlab.

Now, with the polynomial coefficients stored in a row vector, the addition or subtraction of two polynomials is quite simple. For example, let's consider the following three polynomials:

$$f_1 = x^5 - 3x^4 - 10x^3 + 10x^2 + 44x + 48$$

$$f_2 = x^2 + 2x + 2$$

$$f_3 = x^2 + 2x + 1$$

To add f_2 and f_3 , the Matlab sequence is

```
>> f2 = [1 2 2]; f3 = [1 2 1];
>> f2+f3
```

which gives

```
ans =
     2     4     3
```

and, as expected, this is the proper representation of

$$f_2 + f_3 = 2x^2 + 4x + 3$$

Note that, to add or subtract polynomials of different order, we need to pad the lower order polynomial with the appropriate number of leading zeros (since the vectors must be the same size). For example,

```
>> f1 = [1 -3 -10 10 44 48]; f2 = [0 0 0 1 2 2];
>> f1-f2
```

gives

```
ans =
     1     -3    -10     9    42    46
```

which is consistent with expectations.

Now, polynomial multiplication and division, although conceptually straightforward, are not so easy to do via hand calculations -- you probably remember the dreaded long multiplication and division problems from your junior high school days. Of course, Matlab makes these tasks easy for us with the **conv** (multiplication) and **deconv** (division) commands. To illustrate these functions, let's do some simple cases with Matlab and by hand (just to jog your memory).

Dividing f_1 by f_3 gives

$$\begin{array}{r}
 x^3 - 5x^2 - x + 17 \\
 x^2 + 2x + 1 \overline{) x^5 - 3x^4 - 10x^3 + 10x^2 + 44x + 48} \\
 \underline{x^5 + 2x^4 + x^3} \\
 -5x^4 - 11x^3 + 10x^2 \\
 \underline{-5x^4 - 10x^3 - 5x^2} \\
 -x^3 + 15x^2 + 44x \\
 \underline{-x^3 - 2x^2 - x} \\
 17x^2 + 45x + 48 \\
 \underline{17x^2 + 34x + 17} \\
 11x + 31
 \end{array}$$

where we see that the answer is $x^3 - 5x^2 - x + 17$ with a remainder of $11x + 31$. Thus, f_3 is not a factor of f_1 -- since the remainder is nonzero.

And, multiplying f_2 by f_3 gives the following 4th order polynomial

$$\begin{array}{r}
 x^2 + 2x + 2 \\
 x^2 + 2x + 1 \overline{) x^2 + 2x + 2} \\
 \underline{x^2 + 2x + 1} \\
 2x^3 + 4x^2 + 4x \\
 x^4 + 2x^3 + 2x^2 \\
 \underline{x^4 + 4x^3 + 7x^2 + 6x + 2}
 \end{array}$$

Now, these two operations in Matlab are rather trivial (and certainly easier to type than the above hand examples):

```
>> f1 = [1 -3 -10 10 44 48]; f2 = [1 2 2]; f3 = [1 2 1];
>> [Q,R] = deconv(f1,f3)
Q =
     1     -5     -1     17
```

```
R =
    0     0     0     0    11    31

>> conv(f2,f3)
ans =
     1     4     7     6     2
```

which give the same results as above!!!

For the division example, note that, if the denominator is a factor of the numerator, then we expect the remainder to be zero. For example, evaluating f_1/f_2 gives

```
>> [QQ,RR] = deconv(f1,f2)
QQ =
     1     -5     -2     24
RR =
     0     0     0     0     0     0
```

Since RR is zero, f_2 is indeed a factor of f_1 -- a quadratic factor!

Evaluating polynomials is also an easy task. Of course we could write a simple function file to evaluate $f(x)$ at a vector of x values. However, Matlab has a built-in function, ***polyval***, that does this quite efficiently using a technique known as Horner's Rule. This method is simply a clever way to efficiently evaluate the polynomial. An algorithm for **Horner's Rule** is given below:

1. Define polynomial coefficients: $a = [a_1 \ a_2 \ \dots \ a_n \ a_{n+1}]$
2. Define vector of independent variables: $x = \text{linspace}(x_0, x_f, Nx)$
3. Determine length of coefficient array: $M = \text{length}(a)$
4. Initialize function: $f = \text{zeros}(\text{size}(x))$
5. Evaluate polynomial: $\text{for } i = 1:M, \ f = f.*x + a(i); \ \text{end}$

To see exactly what this algorithm does, let's assume a 3rd order polynomial ($M = 4$). Thus, there will be four passes through the ***for ... end*** loop with the following results after each pass:

```
i = 1,      f = a1
i = 2,      f = a1x + a2
i = 3,      f = (a1x + a2)x + a3
i = 4,      f = [(a1x + a2)x + a3]x + a4
```

and, written out in detail, we have

$$f = a_1x^3 + a_2x^2 + a_3x + a_4$$

Thus, we see that, after M passes, we have the desired polynomial evaluated at all desired x values!

This algorithm is really quite efficient, and it has been implemented within Matlab's ***polyval*** function. Therefore, if you need to evaluate a polynomial, you should definitely use ***polyval*** -- that is, don't re-invent the wheel each time it is needed!

As an example that uses ***polyval***, the following code plots $f_1(x)$ over the range $[-3,5]$:

```
a = [1 -3 -10 10 44 48]
x = linspace(-3,5,101);  f = polyval(a,x);
subplot(2,1,1), plot(x,f, 'LineWidth',2), grid, axis([-3 5 -200 200]);
```

```
title('PRoots: Plot of f(x) = x^5 - 3x^4 - 10x^3 + 10x^2 + 44x + 48')
xlabel('x value'),ylabel('f(x)')
```

Another common task involves the evaluation of derivatives of polynomials. This, of course, is easy to do by hand, but Matlab has a built-in function, *polyder*, that is also easy to use. As an example, the derivative of $f_1(x)$ can be plotted as follows (assumes **a** and **x** have been defined):

```
ap = polyder(a); fp = polyval(ap,x);
subplot(2,1,2),plot(x,fp,'LineWidth',2),grid,axis([-3 5 -200 200]);
title('PRoots: Plot of f'(x)')
xlabel('x value'),ylabel('f'(x)')
```

The resulting plot from the above two polynomial evaluation examples is given in Fig. 2.

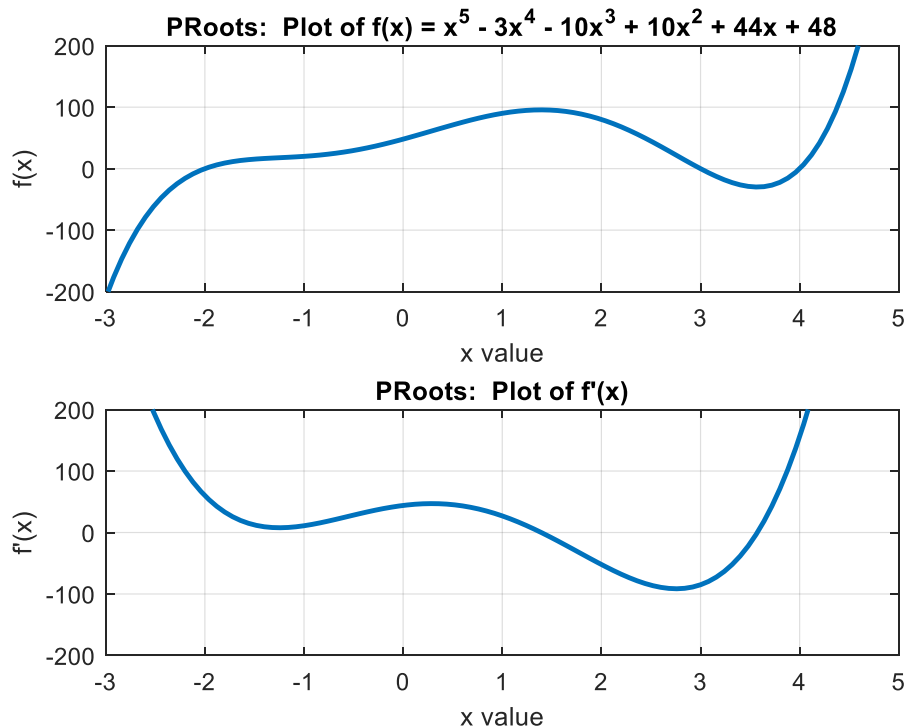


Fig. 2 Plot of $f_1(x)$ and its first derivative.

As defined above, $f_1(x)$ is a 5th order polynomial and its derivative is of order 4. Clearly, we expect that f_1 should have 5 roots and its derivative should have 4 (an n^{th} order polynomial will have n roots). However, from Fig. 2 there are only 3 zero crossings in the upper plot (at -2, 3, and 4) and only 2 zeros in the lower subplot for f_1' . This implies that there is a set of complex conjugate roots for each case. To find all the roots with Matlab, we can simply use the built-in **roots** command. For example, the roots of $f_1(x) = 0$ are

```
>> a = [1 -3 -10 10 44 48]; roots(a)
ans =
    4.0000
    3.0000
   -2.0000
  -1.0000 + 1.0000i
  -1.0000 - 1.0000i
```

where, as expected, we see 3 real roots at the locations indicated in Fig. 2 and a pair of complex conjugate roots (note that these are also the roots of the quadratic factor, $x^2 + 2x + 2$, as discovered above). Thus, with Matlab, finding polynomial roots is quite simple.

Note: There are several different techniques for finding roots of polynomials. Many of the methods try to find linear or quadratic factors of the original polynomial (e.g., Muller's method and Bairstow's method). Once a factor is found, one can divide by the factor to reduce the order by 1 or 2 (this process is called polynomial deflation). Using this approach, one can eventually break an n^{th} order polynomial into a series of linear and quadratic factors -- from which the roots are easily obtained.

Another approach, which is the one implemented in the Matlab **roots** command, uses well established techniques for finding the eigenvalues of a particular matrix -- referred to as the **companion matrix** -- where one can show that the eigenvalues of the companion matrix and the roots of the original polynomial are identical. We have already briefly discussed the subject of eigenvalues and eigenvectors of a matrix in an earlier lesson and we are familiar with the use of Matlab's **eig** command to find these quantities. In particular, since the **eig** function is highly optimized, Matlab's **roots** command first forms the companion matrix and then calls the built-in **eig** function to find the associated eigenvalues -- which gives us the roots of the original polynomial.

Well, if all this seems a little confusing, don't be too concerned. For now, I only require that you to be able to use the **roots** function properly -- since we do not have time in this course to go into the details of the methods...

To complete our brief discussion of polynomial roots, we should also note that a polynomial can also be written in its factored form,

$$f(x) = (x - r_1)(x - r_2)(x - r_3) \cdots (x - r_{n-1})(x - r_n) \quad (35)$$

and multiplying out all the factors will give the form of the polynomial written in eqn. (34). As you might expect, Matlab has a built-in function, **poly**, that will convert the factored form given in eqn. (35) (represented by a column vector of roots) into the coefficient form given in eqn. (34) (which is written as a row vector of coefficients). For example, the following sequence of Matlab commands takes $f_1(x)$ written in coefficient form, finds its roots, and then uses these roots to reconstruct the coefficient form of the polynomial:

```
>> a = [1 -3 -10 10 44 48],    r = roots(a),    p = poly(r)
a =
     1     -3    -10     10     44     48
r =
     4.0000
     3.0000
    -2.0000
    -1.0000 + 1.0000i
    -1.0000 - 1.0000i
p =
     1.0000    -3.0000   -10.0000    10.0000    44.0000    48.0000
```

And, of course, the initial and final coefficient forms of the polynomials are identical!

Solution of the Motivation Problems

Now that we have reviewed several basic methods for root finding and have discussed, in particular, Matlab's *fzero* and *roots* commands, we should be able to solve the problems presented earlier to motivate the need for such techniques. In particular, several Matlab script and function files were written to solve the four problems posed earlier. Each problem is briefly discussed in the following subsections (please refer, as needed, to the problem descriptions given at the beginning of this lesson):

Problem 1 Solution

This problem deals with the volume of fluid in a horizontal cylindrical tank. There were two parts to the problem. For Part a, the height, h , radius, R , and length, L , are given, and the explicit expression for the volume of fluid, V , was simply evaluated as given in eqn. (2) in the first part of **horizontal_cyl_1.m** (see Table 3 for a listing of the Matlab files for this problem). The program output for this part of the problem is given below:

```
Lesson 5 Prob. #1a --> Height of fluid in tank = 3.00000 m
--> Volume of fluid in tank = 61.50354 m^3
```

In Part b, an additional 1.5 m^3 of fluid is added and the goal is to determine the new fluid height. Thus, the volume is now known ($V_b = 63.00354 \text{ m}^3$), and the implicit form of eqn. (2), $f(h) = 0$, was implemented in function file, **horizontal_cyl_1a.m**. This file was called by *fzero* to find a zero of $f(h)$ within the bracket $[0, 2R]$. This range makes physical sense for this problem since the height of fluid must be within these limits. The result from the call to *fzero* is given below:

```
Lesson 5 Prob. #1b --> Volume of fluid in tank = 63.00354 m^3
--> Height of fluid in tank = 3.06140 m
```

Thus, we see that an additional 1.5 m^3 of fluid increases the fluid height from 3.0 m to about 3.06 m (a 6 cm increase).

The Matlab implementation for this problem was quite straightforward and a listing of the associated files is given in Table 3.

Table 3 Listing of the programs to solve Problem 1.

```
%
% HORIZONTAL_CYL_1.M      Find the volume & height of fluid
%                          in a horizontal cylindrical tank
%
% The volume of fluid in a horizontal cylindrical tank is given by:
%   V = L*(R*R*acos((R-h)/R)-(R-h)*sqrt(2*R*h-h*h))
% where V = volume, R = radius, h = height of fluid, and L = length of tank.
%
% Problem description:
%   a. If h, R, and L are known, find V.  -- this is explicit
%   b. If V, R, and L are known, find h.  -- this is implicit
%
% The explicit problem is simple and uses a single explicit evaluation. The
% implicit problem uses fzero, with the function file HORIZONTAL_CYL_1A.M to
% find the value of h that satisfies the implicit form,  $f(h) = 0$ , of the above
% equation.
%
% File prepared by J. R. White, UMass-Lowell (last update: Nov. 2017)
%
```

```

clear all, close all
global R L Vol
%
% Part a: Find Volume of Fluid in a Horizontal Cylindrical Tank
R = 2.5; L = 5; % radius and length of tank (m)
ha = 3.0; % height of fluid for Part a of problem (m)
Va = L*(R*R*acos((R-ha)/R)-(R-ha)*sqrt(2*R*ha-ha*ha));
fprintf('\n\n Lesson 5 Prob. #1a --> Height of fluid in tank = %8.5f m \n ',ha);
fprintf(' --> Volume of fluid in tank = %8.5f m^3 \n ',Va);
%
% Part b: Find Height of Fluid in a Horizontal Cylindrical Tank
Vb = 1.5+Va; % volume after adding another 1.5 m^3 of fluid
fprintf('\n\n Lesson 5 Prob. #1b --> Volume of fluid in tank = %8.5f m^3 \n ',Vb);
Vol = Vb; % volume of fluid to pass to function subroutine used by fzero
hb = fzero(@horizontal_cyl_1a,[0 2*R]);
fprintf(' --> Height of fluid in tank = %8.5f m \n ',hb);
%
% end of problem

%
% HORIZONTAL_CYL_1A.M Function evaluation for Root Finding Exercise
%
% The volume of fluid in a horizontal cylindrical tank is given by:
%  $V = L*(R*R*acos((R-h)/R)-(R-h)*sqrt(2*R*h-h*h))$ 
% where V = volume, R = radius, h = height of fluid, and L = length of tank.
%
%
function f = horizontal_cyl_1a(h)
global R L Vol
f = Vol - L*(R*R*acos((R-h)/R)-(R-h)*sqrt(2*R*h-h*h));
%
% end of function

```

Problem 2 Solution

In this problem, the van der Waal equation of state is compared to the ideal gas law for ammonia over a range of temperatures and pressures. The ideal gas law is the simple explicit relationship given in eqn. (3) and van der Waal's equation is a more complicated implicit expression that interrelates the molar volume, v , the pressure, P , and the gas temperature, T , as given in eqn. (4). Since we want to plot the equation of state for a range of temperatures at three different pressures, a double loop is needed with a separate call to *fzero* to evaluate the implicit form of eqn. (4) for each P and T combination. In the call to *fzero*, the value of v obtained from the ideal gas law is used as a guess for the root finding algorithm. Once the molar volume is determined, the compressibility factor, Z , is easily determined via eqn. (5).

The Matlab program listed in Table 4, **eqofst_1.m**, implements the above solution algorithm. The first part of the main program sets the desired pressure and temperature values. Then, in separate code segments, the ideal gas law and van der Waal's equation of state are evaluated, where we see that the implicit form of the equation of state requires a little more effort than the explicit ideal gas law representation (note that an anonymous function was used here). Finally, the output section creates the desired plot of molar volume versus temperature for the three given values of P and a graphical representation of the compressibility factor for the temperatures and pressures of interest in this problem. The Matlab generated plots are reproduced below as Fig. 3.

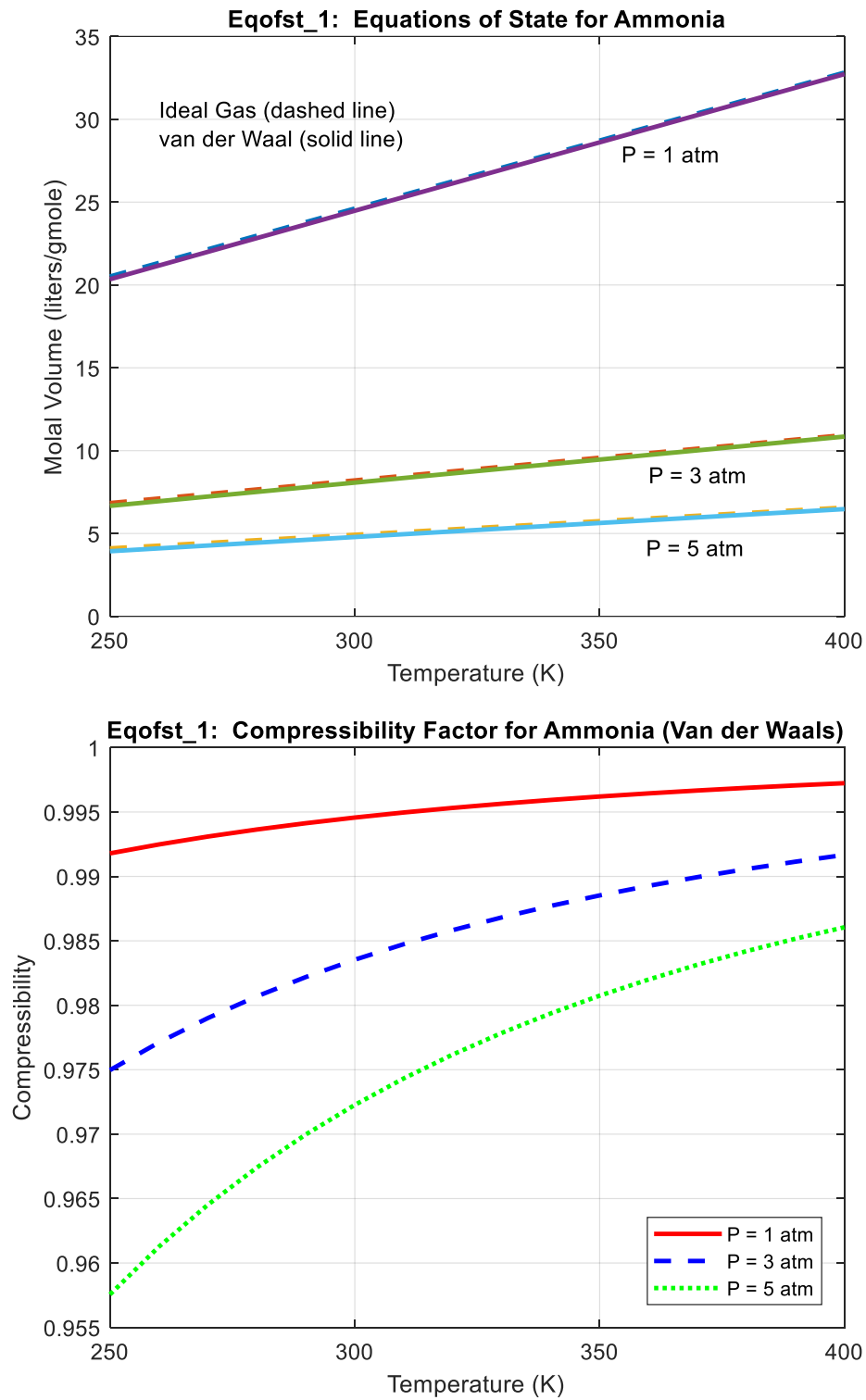


Fig. 3 Graphical representation for the equation of state for ammonia for Problem 2.

Table 4 Listing of the programs to solve Problem 2.

```

%
% EQOFST_1.M      Plot equation of state for ammonia over selected range
%                  using van der Waal's equation
%
% This demo evaluates and plots the equation of state for ammonia over a
% range of pressures and temperatures.  In this case, the ideal gas law is
% compared to van der Waal's equation, which is given by
%  $(P + a/v^2)(v - b) = RT$ 
% where a and b are constants for a particular gas.  For ammonia, given some
% value of P and T, the goal is to find the value of molar volume (v in liters/
% gmole) that satisfies the above equation.  Note that an anonymous function is
% used for the implicit form of van der Waal's equation,  $f(v) = 0$ 
%
% The compressibility factor,  $Z = Pv/RT$ , can be used to compare how close van
% der Waal's equation is to the ideal gas law.  Note that, for the ideal gas
% law, Z is always unity, but other (more accurate) forms for the equation of
% state will deviate somewhat from unity.
%
% File prepared by J. R. White, UMass-Lowell (last update: Nov. 2017)
%

clear all, close all, nfig = 0;

%
% setup range of variables
press = [1 3 5]; % pressure (atm)
temp = linspace(250,400,16); % temperature (K)
R = .08205; % universal gas constant (liters-atm/gmole-K)
Np = length(press); Nt = length(temp);

%
% evaluate ideal gas equation (Pv = RT)
v1 = zeros(Nt,Np);
for i = 1:Np
    v1(:,i) = R*temp/press(i);
end

%
% evaluate van der Waals equation
% (need a little extra work here because of implicit form)
v2 = zeros(Nt,Np); Z = zeros(Nt,Np);
a = 4.19; b = .0372; % constants for ammonia
for i = 1:Np
    P = press(i);
    for j = 1:Nt
        T = temp(j);
        fv = @(v) (P + a/(v*v))*(v - b) - R*T; % anonymous function used in fzero
        v2(j,i) = fzero(fv,v1(j,i));
        Z(j,i) = P*v2(j,i)/(R*T);
    end
end

%
% plot molal volume - ideal (dash), van der Waals (solid)
nfig = nfig+1; figure(nfig)
plot(temp,v1,'--',temp,v2,'-','LineWidth',2)
title('Eqofst_1: Equations of State for Ammonia')
xlabel('Temperature (K)'),ylabel('Molal Volume (liters/gmole)'),grid
for i = 1:Np, gtext(['P = ',num2str(press(i)),' atm']), end
v = axis; xt = temp(1)+10; yt = v(3) + 0.85*(v(4)-v(3));
txt1 = ['Ideal Gas (dashed line) '; 'van der Waal (solid line)'];
text(xt,yt,txt1)

%
% plot compressibility factor
nfig = nfig+1; figure(nfig)
plot(temp,Z(:,1),'r-',temp,Z(:,2),'b--',temp,Z(:,3),'g:', 'LineWidth',2)
title('Eqofst_1: Compressibility Factor for Ammonia (Van der Waals)')
xlabel('Temperature (K)'),ylabel('Compressibility'),grid
for i = 1: Np, txt2(i) = {'P = ',num2str(press(i)),' atm'}; end
legend(txt2,'Location','SouthEast')

%
% end of problem

```

As apparent from the plots, the ideal gas approximation for ammonia for the temperatures and pressures given here is not too bad. Even at the highest pressure (5 atm) and lowest temperature (250 K), the ideal gas law only deviates from the more accurate van der Waal equation by a little over 4%. This is probably acceptable for most applications -- but if higher accuracy is needed, we can always use *fzero* to help evaluate a more complicated form of the equation of state!!!

Problem 3 Solution

The goal here was simply to plot the solution of a particular IVP, where the solution, given by eqn. (9), is written in implicit form, $u(x,y) = 0$. However, for a given value of the independent variable, x , this becomes a classical root finding problem -- that is, “What is the value of y such that $f(y) = 0$?”. As such, we simply need to define a vector of x values and then call *fzero* for each value. Also, for a small x increment, we expect the current y value, y_i , for point x_i , to be close to the previous value, y_{i-1} . Thus, we can use y_{i-1} as an estimate of the root for the current value, y_i . This relatively simple solution scheme was implemented within Matlab file **analytical_ivp1.m** as seen in Table 5 and the resultant plot of $y(x)$ vs. x is given in Fig. 4.

This solution is actually quite interesting because of the discontinuity that is observed at $x = 0.3256$ (this value was obtained to 4 significant digits by using the zoom feature within the Matlab figure window). We will see later (in Lesson 8) that the numerical solution of the given IVP has a problem at this point because the slope goes to infinity. In fact, the IVP cannot be numerically integrated past this singular point without knowledge of the analytical solution (so that we can cheat a little by jumping over the singularity). However, for now, our focus is on root finding, and we see that the use of *fzero* allows us to easily plot the exact implicit solution (even with a discontinuity within the range of interest)!

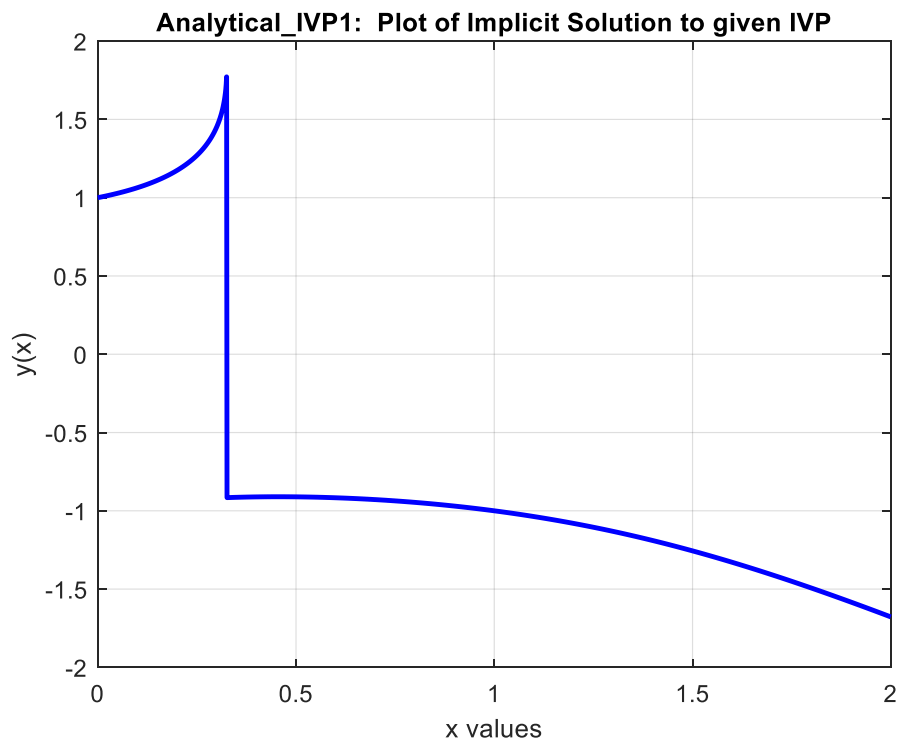


Fig. 4 Plot of implicit solution to the IVP given in Problem 3.

Table 5 Listing of the programs to solve Problem 3.

```

%
% ANALYTICAL_IVP1.M   Plot Implicit Solution to a given IVP
%
% This problem illustrates how to evaluate and plot a function in implicit
% form. The function of interest here is the analytical solution to the IVP
% given by
%
%      -(2xy^3 + y^4)
%      y' = -----      with y(0) = 1
%      xy^3 - 2)
%
% The goal is to plot y(x) vs x for the implicit solution to the IVP given by:
%      u(x,y) = x^2 + xy + y^(-2) - 1 = 0
%
% This requires use of Matlab's fzero root finding routine with the implicit
% function evaluated within an anonymous function. To plot y versus x, we find
% the 'root' of this implicit equation for each value of x, and then plot the
% results.
%
% File prepared by J. R. White, UMass-Lowell (last update: Nov. 2017)
%
%
%      clear all, close all, nfig = 0;
%
% initial setup
%      xo = 0;   xf = 2;   yo = 1;   Nx = 2001;
%      xe = linspace(xo,xf,Nx);   ye = zeros(size(xe));
%
% now evaluate the implicit eqn at each point (uses previous y-value as guess)
%      xe(1)= xo;   ye(1) = yo;
%      for i = 2:Nx
%          x = xe(i);
%          fy = @(y) x^2 + x*y + y^(-2) - 1; % anonymous function for used in fzero
%          ye(i) = fzero(fy,ye(i-1));
%      end
%
% now plot results
%      nfig = nfig+1;   figure(nfig)
%      plot(xe,ye,'b-','LineWidth',2)
%      title('Analytical_IVP1: Plot of Implicit Solution to given IVP')
%      xlabel('x values'),ylabel('y(x)'), grid, hold on
%
% end of simulation

```

Problem 4 Solution

The solution to this problem involves finding the roots to the characteristic equation for a 4th order ODE. The polynomial of interest is given in eqn. (20), and we can easily find all four roots with Matlab's **roots** command, as follows:

```

>> a = [1 0 0 3 -4]; roots(a)
ans =
    -1.7430
    0.3715 + 1.4687i
    0.3715 - 1.4687i
    1.0000

```

where we should note that two zeros were needed in the coefficient representation of the polynomial to account for the missing r^3 and r^2 terms in eqn. (20).

Now, with the roots known, we can write the general solution to the ODE by substituting explicit values for the roots in eqn. (21), or

$$y(x) = c_1 e^{-1.7430x} + c_2 e^{(0.3715+1.4687i)x} + c_3 e^{(0.3715-1.4687i)x} + c_4 e^{1.0000x} \quad (36)$$

This form, however, is somewhat cumbersome because of the complex roots. This can be rewritten, however, using Euler's formulas for the complex exponentials (see Chapter 3 of the Differential Equations text by Edwards and Penny, for example) to give

$$y(x) = a_1 e^{-1.7430x} + e^{0.3715x} (a_2 \sin(1.4687x) + a_3 \cos(1.4687x)) + a_4 e^x \quad (37)$$

and this is the way I would write the general solution to the given ODE. Note that four unique conditions would be required to determine the arbitrary coefficients in this general solution to give a unique solution (but this is another problem -- see Lesson 6)...

Summary and Additional Applications

Well, this completes our brief overview of some root finding techniques and polynomial manipulations within Matlab. You should have a basic understanding of the various methods for finding real roots of nonlinear equations (i.e. using both bracketing and open methods) and be comfortable with using the hybrid algorithm within Matlab's *fzero* routine to solve a range of practical problems. In addition, you should also be able to solve problems that involve polynomial root finding, evaluation, and algebraic manipulation using a variety of built-in functions in Matlab (*roots*, *polyval*, *conv*, *deconv*, *poly*, etc.). I think you will find this capability to be very useful in a variety of areas...

To give some further experience with root finding applications, I have also included three additional worked-out examples that show how *fzero* can be used to help solve some interesting engineering problems in the areas of fluid flow, heat transfer, and fluid statics. These examples are available as separate pdf files, as follows:

pipe_friction_1.pdf -- Friction Effects in Pipe Flow

conducting_rod_1.pdf -- Energy Balance on a Conducting Rod

slanted_gate_1.pdf -- Comprehensive Analysis of a Slanted Gate

After you have finished your reading assignment for this lesson and after reviewing the above examples, you should be ready to do HW #5 (see **hw5xxx.pdf**). This homework usually involves several problems that require you to use a root finding routine (i.e. *fzero*) and possibly several of the polynomial-related functions in Matlab to solve a variety of practical problems. As before, I prefer that you collect the Matlab m-files, the resultant plots and/or tabular results, and a brief description of the results of each problem in a separate solution packet for each HW problem.

Well, good luck with HW #5 -- I think you will find the problems to be both interesting and challenging!!!