Chapter 4. Solution of a Single Nonlinear Algebraic Equation

4.1. Introduction

Life, my friends, is nonlinear. As such, in our roles as problem-solvers, we will be called upon time and again to tackle nonlinear problems. As luck would have it, life hasn't changed much in its nonlinearity from what it was for previous generations. Consequently, the mathematicians and computer scientists that have gone before us have left us with a plethora of tools for subduing all manner of nonlinear beasts. Therefore as we embrace what is undoubtedly a daunting challenge, we do so with the knowledge that, armed with the appropriate tools, the solution of nonlinear algebraic equations too is a task we can methodically add to our repertoire.

The solution of nonlinear algebraic equations is frequently called root-finding, since our goal in this chapter is to find the value of x such that $f(x)=0$ no matter how pernicious the function may appear. The roots of an equation are those values of x that satisfy $f(x)=0$.

There are a myriad of way to find roots.The organization of this chapter is to provide a general understanding of the approach to solving nonlinear algebraic equations. We then examine two intuitive approaches that we shall find are deeply flawed. We then examine other approaches that offer some improvements.

4.2. Iterative Solutions and Convergence

Our goal is to find the value of x that satisfies the following equation.

$$f(x) = 0 \tag{4.1}$$

where $f(x)$ is some nonlinear algebraic equation.

All root-finding techniques are iterative, meaning we make a guess and we keep updating our guess based on the value of $f(x)$. We stop updating our guess when we meet some specified convergence criterion. The convergence criteria can be on *x* or on *f*(*x*). One example of a convergence criteria is the absolute value of the function in question, *f*(*x*).

$$err_{f,i} = \left| f(x_i) \right| \tag{4.2}$$

Since *f*(*x*) goes to zero at the root, the absolute value of *f*(*x*) is an indication of how close we are to the root, at least in the neighborhood of the root. The problem with this error is that it has units of *f*(*x*). If the function is an energy balance, the units of the equation might be kJ/mole or it might be J/mole, which would make the value of *f*(*x*) one thousand times larger. If one possesses sufficient familiarity with the problem that one has an absolute measure of tolerance, then one can proceed with this convergence criterion.

Alternatively we can specify a convergence criteria on *x* itself. In this case, there are two types of convergence criteria, absolute and relative. The absolute error is given by

$$err_{x,i} = \left| x_i - x_{i-1} \right| \tag{4.3}$$

Again, this error has units of x. If we want a relative error, which is dimensionless, then we use:

$$err_{x,i} = \left| \frac{x_i - x_{i-1}}{x_i} \right| \tag{4.4}$$

This gives us a percent error based on our current value of *x*. The advantage of this error is that if we pick a tolerance of 10^{-n}, then our answer will always have *n* good significant digits. For example, if we use this relative error on *x* and set our acceptable tolerance to 10^{-6}, then our answer will always have 6 good significant digits. For this reason, the relative error on x is the preferred error to use. However, this error has one drawback that the others do not have. Namely, this error will diverge if the root is located at zero.

Regardless of what choice of error we use, we have to specify a tolerance. The tolerance tells us the maximum allowable error. We stop the iterations when the error is less than the tolerance.

4.3. Successive Approximations

A primitive technique to solve for the roots of *f*(*x*) is called successive approximation. In successive approximation, we rearrange the equation so that we isolate *x* on left-hand side. So for the example *f*(*x*) given below

$$f(x) = x - \exp(-x) = 0 \tag{4.5.a}$$

we rearrange as

$$x = \exp(-x) \tag{4.5.b}$$

We then make an initial guess for *x*, plug it into the right hand side and see if it equals our guess. If it does not, we take the new value of the right-hand side of the equation and use that for *x*. We continue until our guess gives us the same answer.

Example 4.1. Successive Approximations

Let's find the root to the nonlinear algebraic equation given in equation (4.5.b) using successive approximations. We will use an initial guess of 0.5. We will use a relative error on *x* as the criterion for convergence and we will set our tolerance at 10^{-6}.

```
iteration  x                    exp(-x)          relative error
   1       0.5000000            0.6065307        -
   2       0.6065307            0.5452392        1.1241E-01
   3       0.5452392            0.5797031        5.9451E-02
   4       0.5797031            0.5600646        3.5065E-02
   5       0.5600646            0.5711721        1.9447E-02
   6       0.5711721            0.5648629        1.1169E-02
   7       0.5648629            0.5684380        6.2893E-03
   8       0.5684380            0.5664095        3.5815E-03
   9       0.5664095            0.5675596        2.0265E-03
   10      0.5675596            0.5669072        1.1508E-03
   11      0.5669072            0.5672772        6.5221E-04
   12      0.5672772            0.5670674        3.7005E-04
   13      0.5670674            0.5671864        2.0982E-04
   14      0.5671864            0.5671189        1.1902E-04
   15      0.5671189            0.5671571        6.7494E-05
   16      0.5671571            0.5671354        3.8280E-05
   17      0.5671354            0.5671477        2.1710E-05
   18      0.5671477            0.5671408        1.2313E-05
   19      0.5671408            0.5671447        6.9830E-06
   20      0.5671447            0.5671425        3.9604E-06
   21      0.5671425            0.5671438        2.2461E-06
   22      0.5671438            0.5671430        1.2739E-06
   23      0.5671430            0.5671434        7.2246E-07
```

In this example it took 23 iterations, or evaluations of the function in order to converge (obtain an error less than our specified tolerance). So, we now have converged to a final answer of $x = 0.567143$.

Example 4.2. Successive Approximations

Now let's try to solve an analogous problem. Equation (4.6) has the same root as equation (4.5).

$$f(x) = x + \ln(x) = 0 \tag{4.6.a}$$

We rearrange the function so as to isolate x on the left-hand side as

$$x = -\ln(x) \tag{4.6.b}$$

We perform the iterative successive approximation procedure as before. We use the same initial guess of 0.5.

```
iteration     x                   -ln(x)           relative error
   1          0.5000000           0.6931472        -
   2          0.6931472           0.3665129        8.9119E-01
   3          0.3665129           1.0037220        6.3485E-01
   4          1.0037220           -0.0037146       2.7121E+02
   5          -0.0037146       Does Not Exist
```

By iteration 5, we see that we are trying to take the natural log of a negative number, which does not exist. The program crashes. Taken together, these two examples illustrate several key points about successive approximations, which are summarized in the table below.

Successive Approximation	
Advantages	• simple to understand and use
Disadvantages	• no guarantee of convergence • very slow convergence • need a good initial guess for convergence

My advice is to never use successive approximations. As a root-finding method it is completely unreliable. The only reason it is presented here is to try to convince you that you should not use it, no matter how simple it looks.

A MATLAB code which implements successive approximation is provided later in this chapter.

4.4. Bisection Method of Rootfinding

Another method for finding roots is called the bisection method. In the bisection method we still want to find the root to $f(x) = 0$. We do so by finding a value of *x*, namely x_+, where the function is positive, $f(x) > 0$, and a second value of *x*, namely x_-, where the function is negative, $f(x) < 0$. These two values of x are called brackets. If we have brackets and our function is continuous, then we know that at least one value of *x* for which $f(x) = 0$ lies somewhere between the two brackets.

In the bisection method, we initiate the procedure by finding the brackets. The bisection method does not provide brackets. Rather it requires them as inputs. Perhaps, we plot the function and visually identify points where the function is positive and negative. After we have the brackets, we then find the value of x midway between the brackets.

$$x_{mid} = \frac{x_+ + x_-}{2} \tag{4.7}$$

At each iteration, we evaluate the function at the current midpoint. If $f(x_{mid}) > 0$, then we replace x_+ with x_{mid}, namely $x_+ = x_{mid}$. The other possibility is that $f(x_{mid}) < 0$, in which case $x_- = x_{mid}$. With our new brackets, we find the new midpoint and continue the iterative procedure until we have reached the desired tolerance.

Example 4.3. Bisection Method

Let's solve the problem that the successive approximations problem could not solve.

$$f(x) = x + \ln(x) = 0 \tag{4.6.a}$$

We will take as our brackets,

$$\begin{aligned} x_- &= 0.1 \text{ where } f(x_-) = -2.203 < 0 \\ x_+ &= 1.0 \text{ where } f(x_+) = 1.0 > 0 \end{aligned}$$

How did we find these brackets? It was either by trial and error or we plotted *f*(*x*) vs *x* to obtain some idea where the function was positive and negative.

We will again use a relative error on *x* as the criterion for convergence and we will set our tolerance at 10^{-6}.

	x_-	x_+	$f(x_-)$	$f(x_+)$	error
1	5.500000E-01	1.000000E+00	-4.783700E-02	1.000000E+00	4.500000E-01
2	5.500000E-01	7.750000E-01	-4.783700E-02	5.201078E-01	2.903226E-01
3	5.500000E-01	6.625000E-01	-4.783700E-02	2.507653E-01	1.698113E-01
4	5.500000E-01	6.062500E-01	-4.783700E-02	1.057872E-01	9.278351E-02
5	5.500000E-01	5.781250E-01	-4.783700E-02	3.015983E-02	4.864865E-02
6	5.640625E-01	5.781250E-01	-8.527718E-03	3.015983E-02	2.432432E-02
7	5.640625E-01	5.710938E-01	-8.527718E-03	1.089185E-02	1.231190E-02
8	5.640625E-01	5.675781E-01	-8.527718E-03	1.201251E-03	6.194081E-03
9	5.658203E-01	5.675781E-01	-3.658408E-03	1.201251E-03	3.097041E-03
10	5.666992E-01	5.675781E-01	-1.227376E-03	1.201251E-03	1.548520E-03
11	5.671387E-01	5.675781E-01	-1.276207E-05	1.201251E-03	7.742602E-04

```
12   5.671387E-01     5.673584E-01     -1.276207E-05    5.943195E-04     3.872800E-04
13   5.671387E-01     5.672485E-01     -1.276207E-05    2.907975E-04     1.936775E-04
14   5.671387E-01     5.671936E-01     -1.276207E-05    1.390224E-04     9.684813E-05
15   5.671387E-01     5.671661E-01     -1.276207E-05    6.313133E-05     4.842641E-05
16   5.671387E-01     5.671524E-01     -1.276207E-05    2.518492E-05     2.421379E-05
17   5.671387E-01     5.671455E-01     -1.276207E-05    6.211497E-06     1.210704E-05
18   5.671421E-01     5.671455E-01     -3.275270E-06    6.211497E-06     6.053521E-06
19   5.671421E-01     5.671438E-01     -3.275270E-06    1.468118E-06     3.026770E-06
20   5.671430E-01     5.671438E-01     -9.035750E-07    1.468118E-06     1.513385E-06
21   5.671430E-01     5.671434E-01     -9.035750E-07    2.822717E-07     7.566930E-07
```

After 21 iterations, we have converged to a final answer of x = 0.567143. The bisection method converged even for the form of the equation where successive approximations would not. In fact, the bisection method is guaranteed to converge, if you can first find brackets. However, the bisection method was still pretty slow; it took a lot of iterations.

This example illustrates several key points about the bisection method:

Bisection Method	
Advantages	• simple to understand and use • guaranteed convergence, if you can find brackets
Disadvantages	• must first find brackets (i.e., you need a good initial guess of where the solution is) • very slow convergence

A MATLAB code which implements the bisection method is provided later in this chapter.

4.5. Single Variable Newton-Raphson

One of the most useful root-finding techniques is called the Newton-Raphson method. Like all the methods in this chapter, the Newton-Raphson technique allows you to find solutions to a general non-linear algebraic equation, $f(x) = 0$.

The advantage of the Newton-Raphson method lies in the fact that we include information about the derivative in the iterative procedure. We can approximate the derivative of $f(x)$ at a point x_1 numerically through a finite difference formula,

$$f'(x_1) = \left.\frac{df}{dx}\right|_{x_1} \approx \frac{f(x_1) - f(x_2)}{x_1 - x_2} \tag{4.8}$$

In the Newton-Raphson procedure, we make an initial guess of the root, say x_1. Since we are looking for a root to $f(x)$, let's say that we want x_2 to be a solution to $f(x)=0$. Let's rearrange the equation to solve for x_2.

$$x_2 = x_1 - \frac{f(x_1) - f(x_2)}{f'(x_1)} \tag{4.9}$$

Now, if x_2 is a solution to $f(x)=0$, then $f(x_2)=0$ and the equation becomes:

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} \tag{4.10}$$

This is the Newton-Raphson Method. Based on the value of the function, $f(x_1)$, and its derivative, $f'(x_1)$, at x_1 we estimate the root to be at x_2. Of course, this is just an estimate. The root will not actually be at x_2 (unless the problem is linear). Therefore, we can implement the Newton-Raphson Method as an iterative procedure

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{4.11}$$

until the difference between x_{i+1} and x_i is small enough to satisfy us.

The Newton-Raphson method requires you to calculate the first derivative of the equation, $f'(x)$. Sometimes this may be problematic. Additionally, we see from the equation above that when the derivative is zero, the Newton-Raphson method fails, because we divide by the derivative. This is a weakness of the method.

However because we go to the trouble to give the Newton-Raphson method the extra information about the function contained in the derivative, it will converge must faster than the previous methods. We will see this demonstrated in the following example.

Finally, as with any root-finding method, the

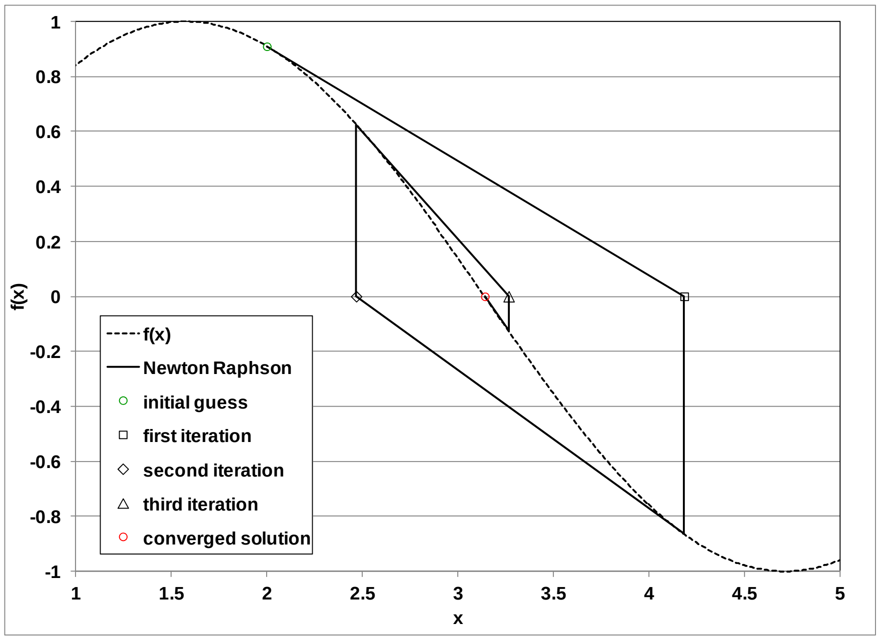

Figure 4.1. Graphical illustration of the Newton-Raphson method.

Newton-Raphson method requires a good initial guess of the root.

In Figure 4.1., a graphical illustration of the Newton-Raphson method is provided to find the root of $f(x) = \sin(x)$ at $x=\pi$. from an initial guess of $x=2$. One follows the slope at $x=2$ until one reaches, $f(x) = 0$, which specifies the new iterated value of x. The new iteration begins at the point $(x, f(x))$ and this procedure continues until the solution converges within an acceptable tolerance.

Example. 4.4. The Newton-Raphson Method

Let's again solve the problem that the successive approximations problem could not solve.

$$f(x) = x + \ln(x) = 0 \qquad (4.6.a)$$

The derivative is

$$f'(x) = 1 + \frac{1}{x} \qquad (4.12)$$

We will use the same initial guess of 0.5. We will again use a relative error on x as the criterion for convergence and we will set our tolerance at 10^{-6}. At each iteration we must evaluate both the function and its derivative.

	x_{old}	$f(x_{old})$	$f'(x_{old})$	x_{new}	error
1	5.000000E-01	-1.931472E-01	3.000000E+00	5.643824E-01	-
2	5.643824E-01	-7.640861E-03	2.771848E+00	5.671390E-01	4.860527E-03
3	5.671390E-01	-1.188933E-05	2.763236E+00	5.671433E-01	7.586591E-06
4	5.671433E-01	-2.877842E-11	2.763223E+00	5.671433E-01	1.836358E-11

We see in this example that the Newton-Raphson method converged to a root with 11 good significant figures in only four iterations. We observe that for the last three iterations, the error dropped quadratically. By that we mean

$$err_{i+1} \approx err_i^2 \text{ or } \frac{err_{i+1}}{err_i^2} \approx 1 \qquad (4.13)$$

Quadratic convergence is a rule of thumb for the Newton-Raphson method. The ratio ought to be on the order of 1. In this example, we have

$$\frac{err_3}{err_2^2} = \frac{7.6 \cdot 10^{-6}}{\left(4.9 \cdot 10^{-3}\right)^2} = 0.32 \qquad \text{and} \qquad \frac{err_4}{err_3^2} = \frac{1.83 \cdot 10^{-11}}{\left(7.6 \cdot 10^{-6}\right)^2} = 0.32$$

We only obtain quadratic convergence near the root. How near do we have to be? The answer to this question depends upon the equation we are solving. It is worth noting that just because the Newton-Raphson method converges to a root from an initial guess, it may not converge to the same root (or converge at all) from all initial guesses closer to the root.

This example illustrates several key points about successive approximations:

Newton-Raphson Method	
Advantages	• simple to understand and use • quadratic (fast) convergence, near the root
Disadvantages	• have to calculate analytical form of derivative • blows up when derivative is zero. • need a good initial guess for convergence

A MATLAB code which implements the Newton-Raphson method is provided later in this chapter.

4.6. Newton-Raphson with Numerical Derivatives

For whatever reason, people don’t like to take derivatives. Therefore, they don’t want to use the Newton-Raphson method, since it requires both the function and its derivative. However, we can avoid analytical differentiation of the function through the use of numerical differentiation. For example, we might choose to approximate the derivative at x_i using the second-order centered finite difference formula, as provided in equation (3.7)

$$f'(x_i) = \frac{f(x_i + h) - f(x_i - h)}{2h} \tag{3.7}$$

where h is some small number. Generally I define h according to a rule of thumb

$$h = \min(0.01 \cdot x_i, 0.01) \tag{4.14}$$

This is just a rule of thumb that I made up that seems to work 95% of the time. More sophisticated rules for estimated the increment size certainly exist. Using this rule of thumb, we execute the Newton-Raphson algorithm in precisely the same way, except we never to have to evaluate the derivative analytically.

Example 4.5. Newton-Raphson with Numerical Derivatives

Let’s again solve the problem that the successive approximations problem could not solve.

$$f(x) = x + \ln(x) = 0 \tag{4.6.a}$$

We will use an initial guess of 0.5. We will use a relative error on *x* as the criterion for convergence and we will set our tolerance at 10^{-6}.

```
   xold            f(xold)         f'(xold)        xnew            error
1  5.000000E-01    -1.931472E-01   3.000067E+00    5.643810E-01    1.000000E+02
2  5.643810E-01    -7.644827E-03   2.771912E+00    5.671389E-01    4.862939E-03
3  5.671389E-01    -1.206407E-05   2.763295E+00    5.671433E-01    7.697929E-06
4  5.671433E-01    -2.862443E-10   2.763282E+00    5.671433E-01    1.826496E-10
```

So we converged to 0.5671433 in only four iterations, just as it did in the rigorous Newton-Raphson method. This example illustrates several key points about successive approximations:

Newton-Raphson with Numerical Derivatives	
Advantages	• simple to understand and use • quadratic (fast) convergence, near the root
Disadvantages	• blows up when derivative is zero. • need a good initial guess for convergence

A MATLAB code which implements the Newton-Raphson method with numerical derivatives is provided later in this chapter.

4.7. Solution in MATLAB

MATLAB has an intrinsic function to find the root of a single nonlinear algebraic equation. The routine is called *fzero.m*. You can access help on it by typing *help fzero* at the MATLAB command line prompt. You can also access the fzero.m file itself and examine the code line by line. The routine uses a procedure that is classified as a "search and interpolation" technique. The first part of the algorithm requires an initial guess. From this guess, the code searches until it finds two brackets, just as in the bisection case. The function is positive at one bracket and negative at the other. Once the brackets have been obtained, rather than testing the midpoint, as in the bisection method, this routine performs a linear interpolation between the two brackets.

$$x_{new} = x_- - \frac{f(x_-)}{f(x_-) - f(x_+)}\left(x_- - x_+\right) \tag{4.15}$$

One of the brackets is replaced with x_{new} , based on the sign of $f(x_{new})$. This procedure is iterated until convergence to the desired tolerance. The actual MATLAB code is a little more sophisticated but we now understand the gist behind a "search and interpolate" method.

The simplest syntax for using the fzero.m code is to type at the command line prompt:

```
>> x = fzero('f(x)',x0,tol,trace)
```

where *f*(*x*) is the function we want the roots of, x_0 is the initial guess, and *tol* is the relative tolerance on *x*., and a non-zero value of *trace* requests iteration information.

Example. 4.6. fzero.m

Let's again solve the problem that the successive approximations problem could not solve.

$$f(x) = x + \ln(x) = 0 \tag{4.6.a}$$

The command at the MATLAB prompt is

```
» x = fzero('x+log(x)',0.5,1.e-6,1)
```

The code output is given below.

```
 Func evals      x           f(x)          Procedure
    1            0.5     -0.193147        initial
    2       0.485858     -0.235981        search
    3       0.514142     -0.151113        search
    4           0.48     -0.253969        search
    5           0.52     -0.133926        search
    6       0.471716     -0.279663        search
    7       0.528284     -0.109836        search
    8           0.46     -0.316529        search
    9           0.54    -0.0761861        search
   10       0.443431     -0.369781        search
   11       0.556569    -0.0293964        search
   12           0.42     -0.447501        search
   13           0.58     0.0352728        search

   Looking for a zero in the interval [0.42, 0.58]

   14        0.56831    0.00322159        interpolation
   15       0.567143  8.92955e-008        interpolation
   16       0.567141 -5.43716e-006        interpolation

x =   0.56714332272548
```

MATLAB's fzero.m (search and interpolate)	
Advantages	• comes with MATLAB • slow convergence

Disadvantages	• has to find brackets before it can begin converging • need a good initial guess for convergence • somewhat difficult to use for more complex problems.

4.8. Existence and Uniqueness of Solutions

When dealing with linear algebraic equations, we could determine how many roots were possible. There were only three choices. Either there was 0, 1, or an infinite number of solutions. When dealing with nonlinear equations, we have no such theory. A nonlinear equation can have 0, 1, 2… up through an infinite number of roots. There is no sure way to tell except by plotting it out. In Figure 4.2., four examples of nonlinear equations with 0, 1, 2 and an infinite number of roots are plotted.

It is important to remember that when you use any of the numerical root-finding techniques described above, you will only find one root at a time. Which root you locate depends upon your choice of method and the initial guess.

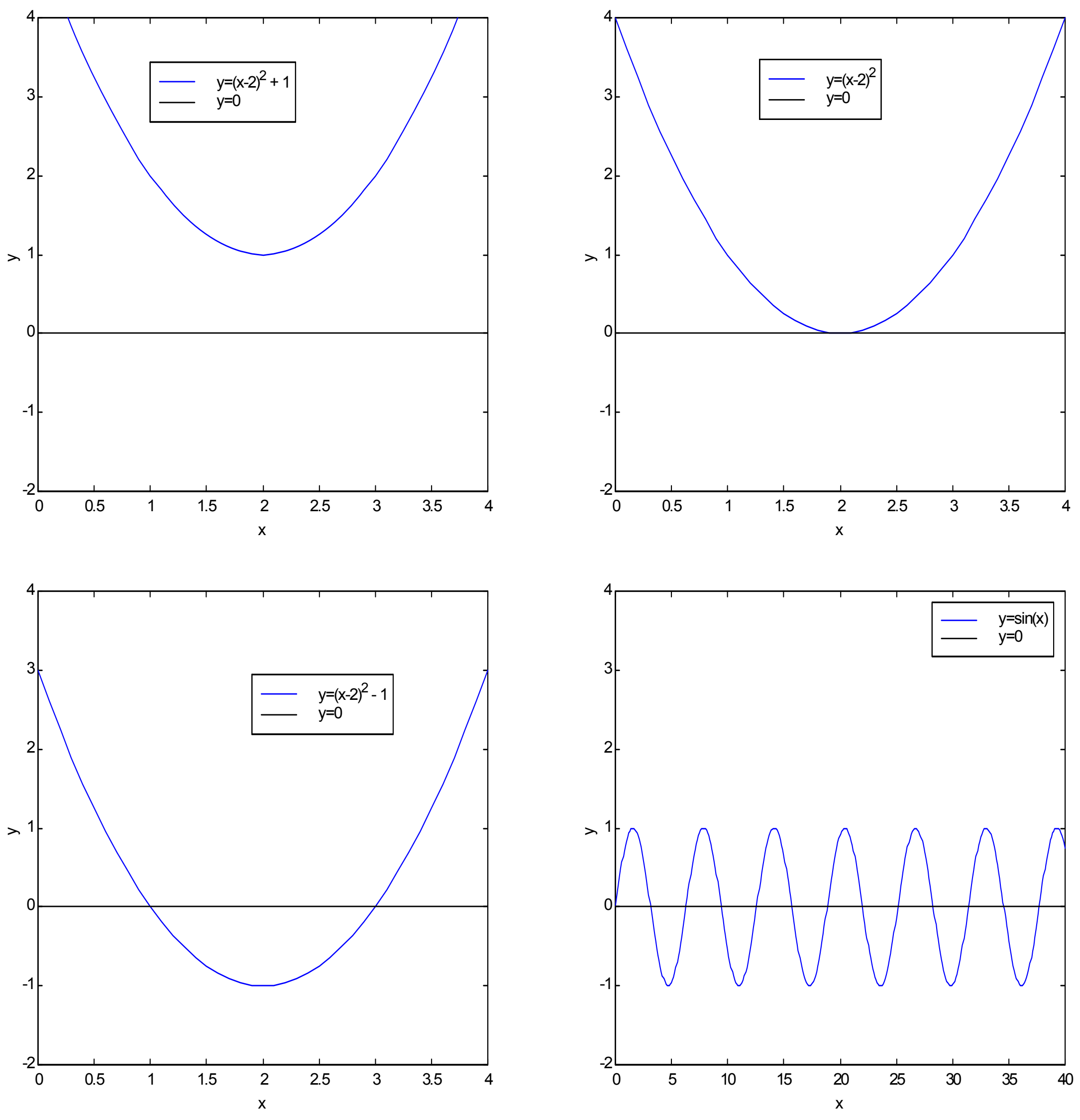

Figure 4.2. Examples of nonlinear equations with zero (top left), one (top right), two (bottom left) and infinite (bottom right) real roots.

Example. 4.7. van der Waals equation of state

When one is interested in finding the several roots of an equation, one must provide multiple different guesses to find each root. To illustrate this problem, we examine the van der Waals equation of state (EOS), which relates pressure, *p*, to molar volume, *V*, and temperature, *T*, via

$$p = \frac{RT}{V-b} - \frac{a}{V^2} \tag{4.16}$$

where *R* is the gas constant and *a* and *b* are species-dependent van der Waals constants. In truth the van der Waals EOS is a cubic equation of state and can be expressed as a cubic polynomial, which can be exploited to reveal the roots through numerical polynomial solvers. (Interestingly, some software solve for the roots of polynomials by constructing a matrix whose characteristic equation corresponds to the polynomial and then using a routine intended to determine eigenvalues in order to determine the roots of the polynomial.) Here, however, we will not take advantage of this fact but will deal with the EOS in the form presented in equation (4.16).

At temperature below the critical temperature, the van der Waals EOS predicts vapor-liquid equilibrium. Our task is to find the molar volumes corresponding to the liquid phase, the vapor phase and a third intermediate molar volume that is useful for identifying the vapor pressure at a given temperature. Thus our task is to find all three roots corresponding to a given temperature and pressure.

We rearrange the equation into the familiar form corresponding to $f(x)=0$

$$f(V)=\frac{RT}{V-b}-\frac{a}{V^2}-p=0 \tag{4.17}$$

We accept that we are more likely to find relevant roots if we provide initial guesses close to those roots. Therefore it is to our advantage to plot the equation to get a general idea of where the roots lie. For our purposes, let us define a state at *T*=98 K and *p*=101325 Pa. The van der Waals constants for argon are a=0.1381 m^6/mol^2 and b=3.184x10^{-5} m^3/mol. The gas constant is R=8.314 J/mol/K. The plot is shown in Figure 4.3. Note that the since the three roots occur at different orders of magnitude, the x-axis is plotted on a logarithmic scale. If it were plotted on a linear scale, you wouldn't be able to see the roots. Note also that no values of x are plotted below *b*. It requires some knowledge of the thermodynamics to realize that the molar volume cannot be less that the van der Waals parameter *b*. There may be additional mathematical roots that lie below *b* but they are of no interest to us.

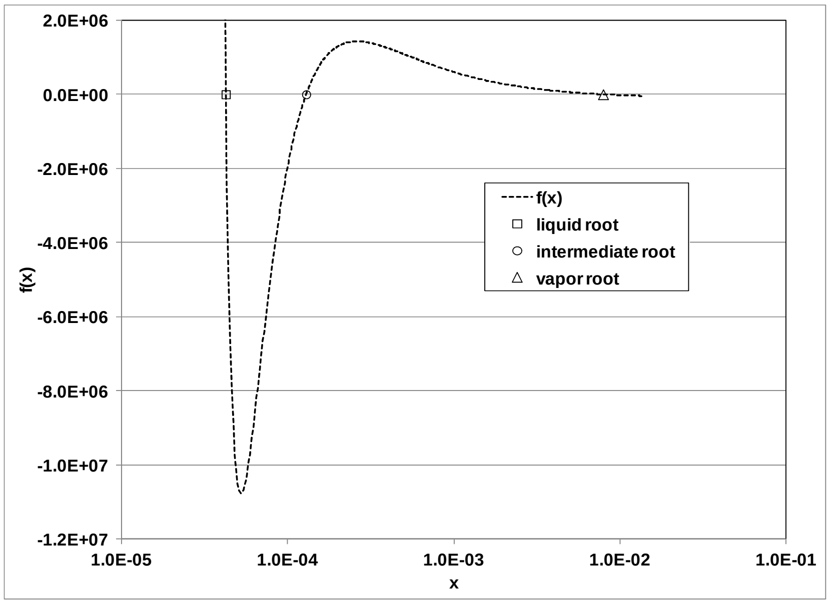

Figure 4.3. van der Waals equation of state (equation (4.17)) plotted showing roots. Note that the x-axis is logarithmic.

We require good initial guesses in order to find each root. Perhaps the ideal gas law provides an initial guess for the vapor root. The ideal gas law, $pV = RT$, provides an initial guess of $V = 8.041x10^{-3}$ m^3/mol. If we put this in the Newton-Raphson method, we converge in a few iterations to $V_{vapor} = 7.901x10^{-3}$ m^3/mol. So our vapor guess was pretty good and we have found a root. Presumably it is the vapor root, which must have the largest molar volume.

Knowing that the van der Waals parameter *b* provides a lower limit for the molar volume suggests that we guess something a little larger than *b* as an initial guess for the liquid root. If we make a guess of 1.1*b* ($3.50x10^{-5}$ m^3/mol), we converge in nine iterations to $V_{liquid} = 4.25x10^{-5}$ m^3/mol.

The intermediate root must lie between these other two roots. Some initial guesses between them will lead to the vapor root and some will lead to the liquid root. There is a range of initial guesses that will lead to the intermediate root. This range is generally defined by the values of x, where the slope points toward the intermediate root. Looking at Figure 4.3 suggests this range may be fairly narrow. If one investigates a range of initial guesses, one finds that initial guesses of $9x10^{-3} < V < 1.9x10^{-4}$ lead to an intermediate root of $V_{\text{int}er} = 1.29x10^{-4}$ m^3/mol. Guesses outside that range lead to other roots including unphysical roots less than *b* or else the procedure diverges entirely.

This example illustrates a key point in the solution of nonlinear algebraic equations. **Finding an initial guess is often the most important and most difficult part of the problem.** As shown in this example, the good initial guesses came from our understanding of the physical system—that the constant *b* provided a lower asymptote for the liquid molar volume and that the ideal gas law provided a good estimate of the vapor molar volume.

People who do not rely on numerical methods to solve problems have in the past voiced the criticism that “numerical methods” deprive the student of developing the capability to make reasonable back of the envelope calculations. This example offers the contrary point of view. The ability to solve this problem required reasonable estimates of the roots to initiate the iterative procedure.

4.9. Rootfinding Subroutines

In this section, we provide short routines for implementing four of these root-finding techniques in MATLAB. Note that these codes correspond to the theory and notation exactly as laid out in this book. These codes do not contain extensive error checking, which would complicate the coding and defeat their purpose as learning tools. That said, these codes work and can be used to solve problems.

As before, on the course website, two entirely equivalent versions of this code are provided and are titled *code.m* and *code_short.m*. The short version is presented here. The longer version, containing instructions and serving more as a learning tool, is not presented here. The numerical mechanics of the two versions of the code are identical.

Code 4.1. Successive Approximation (succapp_short)

```
function [x0,err] = succapp_short(x0);
maxit = 100;
tol = 1.0e-6;
err = 100.0;
icount = 0;
xold = x0;
while (err > tol & icount <= maxit)
   icount = icount + 1;
   xnew = funkeval(xold);
   if (icount > 1)
      err = abs((xnew - xold)/xnew);
   end
   fprintf(1,'i = %i xnew = %e xold = %e err = %e \n',icount, xnew, xold, err);
   xold = xnew;
end
x0 = xnew;
if (icount >= maxit)
   fprintf(1,'Sorry.  You did not converge in %i iterations.\n',maxit);
   fprintf(1,'The final value of x was %e \n', x0);
end

function x = funkeval(x0)
x = exp(-x0);
```

An example of using succapp_short is given below.

```
»  [x0,err] = succapp_short(0.5)
i = 1 xnew = 6.065307e-001 xold = 5.000000e-001 err = 1.000000e+002
…
i = 23 xnew = 5.671434e-001 xold = 5.671430e-001 err = 7.224647e-007

x0 =    0.5671
err =  7.2246e-007
```

The root is at 0.5671 and the error is less than the tolerance of 10^{-6}..

Code 4.2. Bisection (bisect_short)

```
function [x0,err] = bisect(xn,xp);
maxit = 100;
tol = 1.0e-6;
err = 100.0;
icount = 0;
fn = funkeval(xn);
fp = funkeval(xp);
```

```
while (err > tol & icount <= maxit)
   icount = icount + 1;
   xmid = (xn + xp)/2;
   fmid = funkeval(xmid);
   if (fmid > 0)
      fp = fmid;
      xp = xmid;
   else
      fn = fmid;
      xn = xmid;
   end
   err = abs((xp - xn)/xp);
   fprintf(1,'i = %i xn = %e xp = %e fn = %e fp = %e err = %e \n',icount, xn,
xp, fn, fp, err);
end
x0 = xmid;
if (icount >= maxit)
   fprintf(1,'Sorry.  You did not converge in %i iterations.\n',maxit);
   fprintf(1,'The final value of x was %e \n', x0);
end

function f = funkeval(x)
f = x + log(x);
```

An example of using bisect_short is given below.

```
» [x0,err] = bisect_short(0.1,1.0)
i = 1 xn = 5.500000e-001 xp = 1.000000e+000 fn = -4.783700e-002 fp =
1.000000e+000 err = 4.500000e-001
…
i = 21 xn = 5.671430e-001 xp = 5.671434e-001 fn = -9.035750e-007 fp =
2.822717e-007 err = 7.566930e-007

x0 = 0.5671
err = 7.5669e-007
```

The root is at 0.5671 and the error is less than the tolerance of 10^{-6}..

Code 4.3. Newton-Raphson (newraph1_short)

```
function [x0,err] = newraph1_short(x0);
maxit = 100;
tol = 1.0e-6;
err = 100.0;
icount = 0;
xold =x0;
while (err > tol & icount <= maxit)
   icount = icount + 1;
   f = funkeval(xold);
   df = dfunkeval(xold);
   xnew = xold - f/df;
   if (icount > 1)
      err = abs((xnew - xold)/xnew);
   end
   fprintf(1,'icount = %i xold = %e f = %e df = %e xnew = %e  err = %e 
\n',icount, xold, f, df, xnew, err);
   xold = xnew;
end
x0 = xnew;
if (icount >= maxit)
   % you ran out of iterations
   fprintf(1,'Sorry.  You did not converge in %i iterations.\n',maxit);
   fprintf(1,'The final value of x was %e \n', x0);
end

function f = funkeval(x)
f = x + log(x);

function df = dfunkeval(x)
df = 1 + 1/x;
```

An example of using newraph1_short is given below.

```
»  [x0,err] = newraph1_short(0.5)
icount = 1 xold = 5.000000e-001 f = -1.931472e-001 df = 3.000000e+000 xnew = 
5.643824e-001  err = 1.000000e+002
…
icount = 4 xold = 5.671433e-001 f = -2.877842e-011 df = 2.763223e+000 xnew = 
5.671433e-001  err = 1.836358e-011

x0 = 0.5671
err = 1.8364e-011
```

The root is at 0.5671 and the error is less than the tolerance of 10^{-6}.

Code 4.4. Newton-Raphson with Numerical derivatives (newraph_short)

```
function [x0,err] = nrnd1(x0);
maxit = 100;
tol = 1.0e-6;
err = 100.0;
icount = 0;
xold =x0;
while (err > tol & icount <= maxit)
   icount = icount + 1;
   f = funkeval(xold);
   h = min(0.01*xold,0.01);
   df = dfunkeval(xold,h);
   xnew = xold - f/df;
   if (icount > 1)
      err = abs((xnew - xold)/xnew);
   end
   fprintf(1,'icount = %i xold = %e f = %e df = %e xnew = %e  err = %e
\n',icount, xold, f, df, xnew, err);
   xold = xnew;
end
x0 = xnew;
if (icount >= maxit)
   fprintf(1,'Sorry.  You did not converge in %i iterations.\n',maxit);
   fprintf(1,'The final value of x was %e \n', x0);
end

function f = funkeval(x)
f = x + log(x);

function df = dfunkeval(x,h)
fp = funkeval(x+h);
fn = funkeval(x-h);
df = (fp - fn)/(2*h);
```

An example of using nrnd1_short is given below.

```
» [x0,err] = nrnd1_short(0.5)
icount = 1 xold = 5.000000e-001 f = -1.931472e-001 df = 3.000067e+000 xnew =
5.643810e-001  err = 1.000000e+002
…
icount = 4 xold = 5.671433e-001 f = -2.862443e-010 df = 2.763282e+000 xnew =
5.671433e-001  err = 1.826496e-010

x0 = 0.5671
err = 1.8265e-010
```

The root is at 0.5671 and the error is less than the tolerance of 10^{-6}.

4.10. Problems

Problems are located on course website.