

## CHAPTER 2

# Solving Nonlinear Equations

- 2.1. Nonlinear algebraic equations
- 2.2. Bisection
- 2.3. Newton's method and secant method
  - Newton's method
  - Secant method
- 2.4. *MATLAB* methods for finding roots
- 2.5. Example: Leonardo of Pisa's polynomial
- 2.6. Iterative equations
- 2.7. *MATLAB* methods for solving iterative equations
- 2.8. Example: Wavelength of surface gravity waves
- 2.9. Systems of linear equations
- 2.10. *MATLAB* methods for solving sets of linear equations
- 2.11. Gaussian elimination
- 2.12. Example: Gaussian elimination
- 2.13. Problems
- 2.14. References

## 2. Solving Nonlinear Equations and Sets of Linear Equations

### 2.1. Nonlinear algebraic equations

Nonlinear algebraic equations and large systems of linear equations that are not easily solved by hand arise commonly in the hydrological sciences. The solution of such equations is the subject of this chapter. The techniques covered also are employed in many numerical solutions of differential equations.

Some equations can be solved easily. For example, the solution to the linear equation

$$y = mx + b$$

is just

$$x = (y - b) / m$$

The solution is the root of the equation  $0 = mx + b - y$ . Another common example is a quadratic equation

$$ax^2 + bx + c = 0$$

which has the well-known solution

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

Higher order polynomials and functions involving trigonometric functions or other transcendental functions can be difficult to solve in an explicit, analytical form. Examples of nonlinear equations you might encounter in hydrology are:

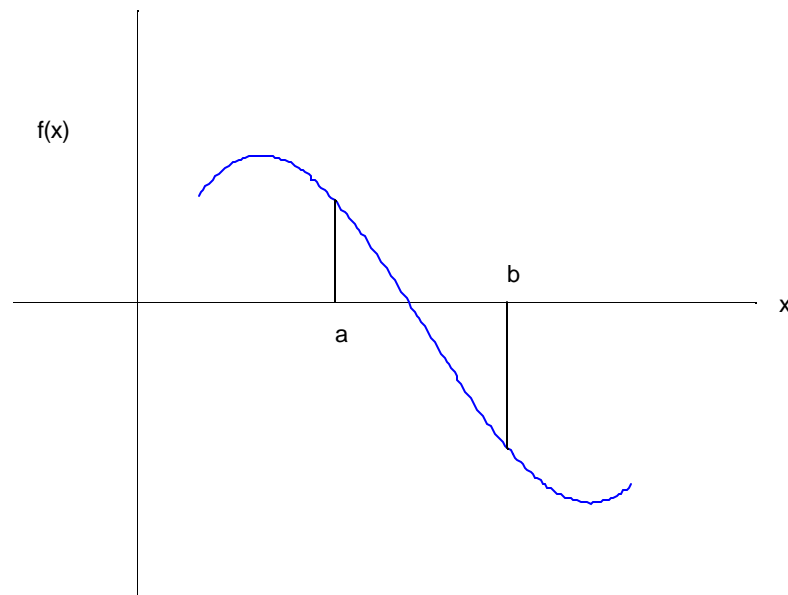
1. Specific energy equation [Box 2.1]:  $h^3 - Eh^2 + \frac{q^2}{2g} = 0$
2. Manning's equation, when used to solve for depth,  $h$  [Box 2.2]:

$$Q = \frac{1}{n} \left( \frac{wh}{w + 2h} \right)^{2/3} S^{1/2} wh \quad \text{or} \quad h = Qn \left[ \left( \frac{wh}{w + 2h} \right)^{2/3} S^{1/2} w \right]^{-1}$$

The specific energy equation is an example of a cubic equation (third-order polynomial) and Manning's equation for depth is an example of an equation of the form  $x = g(x)$ . There are many numerical methods available to solve equations such as these. A few of the important, common techniques are bisection, Newton's method, and iteration. The first two of these methods are cast in terms of finding the roots of an equation. Algebraic equations of the form  $y = f(x)$  can be rewritten as  $0 = f(x) - y$ . Their root(s) are the values of  $x$  satisfying this equation.

## 2.2. Bisection

Bisection is a very straightforward method for finding roots of a continuous function. Say you know the values of a function  $f(x)$  at  $x=a$  and  $x=b$  and that  $f(x=a)*f(x=b)<0$  so that  $[a,b]$  brackets a root of  $f(x)$  (Figure 2.1). In the bisection method, the interval around the root is successively halved until the value of  $f(x)$  is as close to 0 as desired (within tolerance). With each halving, the root is kept between the bracketing values by pairing the new value of  $x$  with the previous value that gives  $f(x)$  of the opposite sign. The error will be less than  $|b-a|/2^n$ , where  $n$  is the number of iterations. The method is very robust and, given initial values that bracket a root, the root will be found. It has the disadvantage of being slower to converge than many of the other methods. It is often used to find an approximate root to use as an initial value in some of the other more efficient techniques, such as Newton's method.



**Figure 2.1.** Values of  $f$  at  $x=a$  and  $x=b$  bracket a root of the function.

## 2.3. Newton's method and secant method

Newton's method and the secant method are more efficient at finding roots than bisection. Both are based on the idea that any function can be approximated by a straight line over some small interval – a fact that is taken advantage of over and over again in numerical solution techniques. These methods begin with an initial estimate  $x_0$  that is close to the real root.

### *Newton's method (Figure 2.2)*

In Newton's method, the tangent to  $f(x_0)$  is found and extrapolated to  $f(x) = 0$  to obtain an improved estimate of the root of  $f(x)$ ,  $x_1$ . [If  $f(x)$  were a line,  $x_1$  would be the root.] From calculus, the tangent to a function at a given point is given by the first derivative of the function at that point. Thus  $\tan \theta_0 = f'(x_0)$ . From trigonometry, we also know that

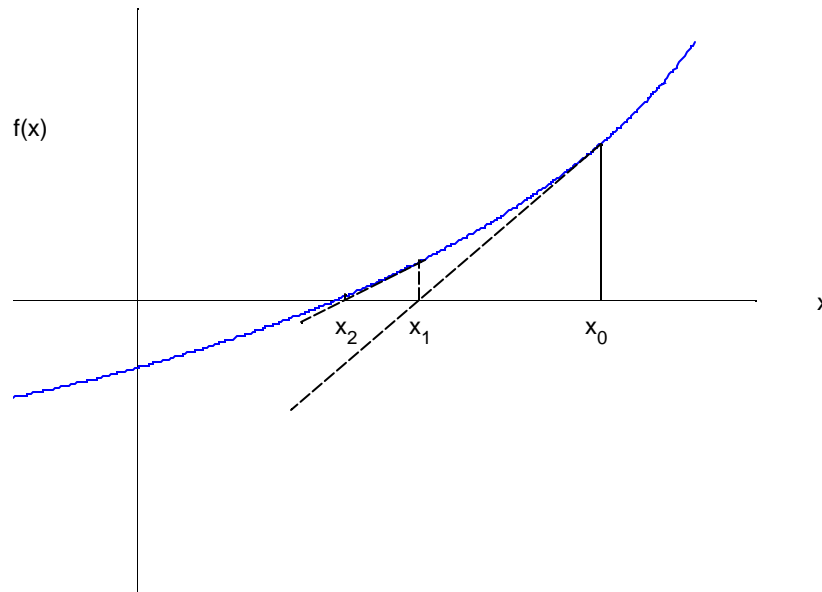
$$\tan \theta_0 = f'(x_0) / (x_0 - x_1).$$

Combining these, we get

$$f'(x_0) = \frac{f(x_0)}{x_0 - x_1} \quad \text{or} \quad x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

Repeating the process at  $x = x_1$  gives  $x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}$  or, in general,

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \quad \text{for } n=0,1,2,\dots \quad (2.1)$$

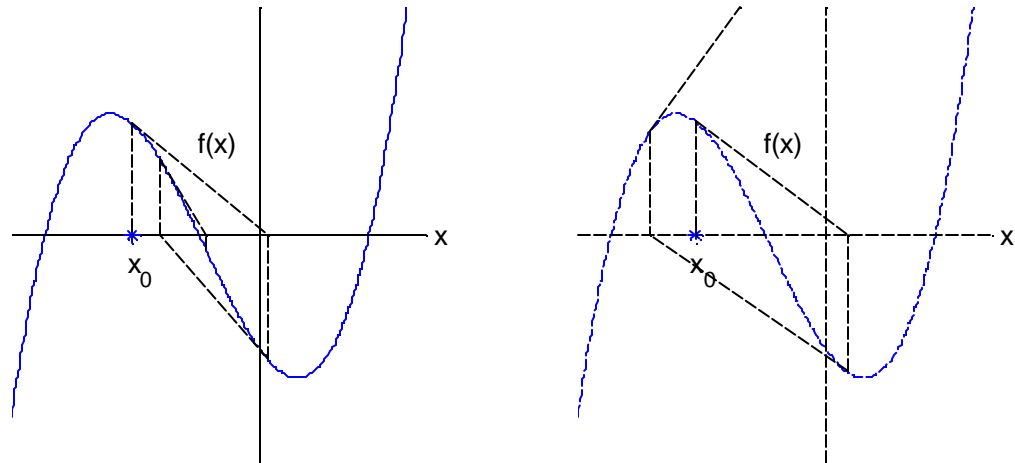


**Figure 2.2.** Schematic of Newton's method.



Provided the successive approximations are convergent, the procedure is carried out until  $f(x_n)$  is as close to 0 as desired or  $|x_{n+1} - x_n|$  is less than some small number. Newton's method generally works well for cases in which the derivative is known in advance, such as polynomials or other functions with straightforward derivatives.

The closer the initial guess  $x_0$  is to the root, the faster and more certain the convergence. What is close enough depends on the function. For example, consider the cubic equation plotted in Figure 2.3. For the choice of  $x_0$  in the left panel (indicated by \*), Newton's method converges relatively quickly. The initial value of  $x_0$  in the right panel differs by just a fraction, but in this case the iterations diverge and the root is not found.



**Figure 2.3.** Newton's method may (left) or may not (right) converge to a root depending on the shape of the function and the proximity of the starting value to the root.

### ***Secant method (Figure 2.4)***

The secant method also uses a linear approximation to a function near a root to make successively improved estimates of the value of the root. The secant method turns out to be equivalent to Newton's method when  $f'(x)$  in equation 2.1 is approximated using finite differences. The simplest approximation to a derivative is just

$$\frac{f(x_1) - f(x_0)}{x_1 - x_0} = \frac{\Delta f}{\Delta x}$$

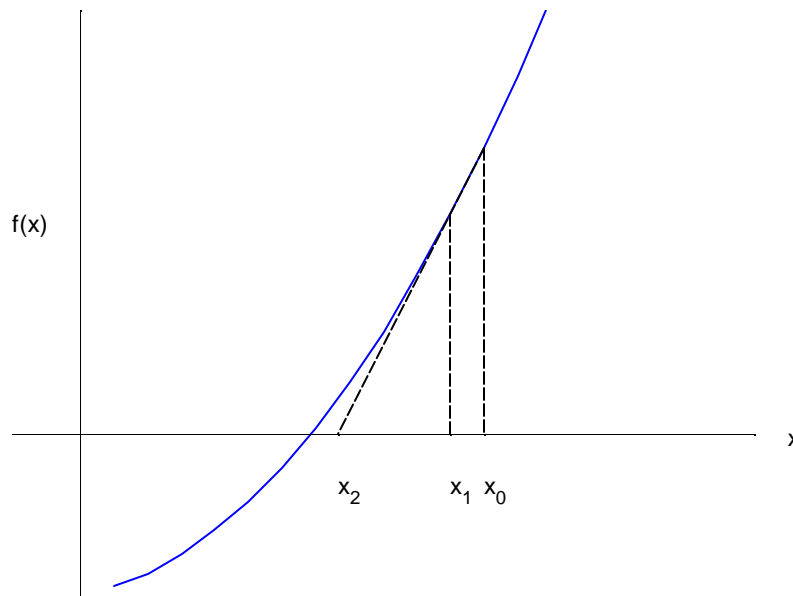
Given initial estimates of the root  $x_0$  and  $x_1$ , this gives the slope of a line connecting  $f(x_0)$  and  $f(x_1)$ , the secant. Extending this line to  $f(x)=0$  gives an improved estimate of the root,  $x_2$ ,

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} \cong x_1 - f(x_1) \frac{(x_1 - x_0)}{f(x_1) - f(x_0)}$$

The same procedure is used successively

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

until  $f(x_{n+1})$  is as close to 0 as necessary or the difference between  $x_n$  and  $x_{n+1}$  is acceptably small. Two initial estimates close to a root,  $x_0$  and  $x_1$ , are needed to begin the method. As is true of Newton's method, the secant method can fail to converge. A "good" initial estimate often is needed.



**Figure 2.4.** Schematic of the secant method.

## 2.4. MATLAB methods for finding roots

**fzero:** `fzero` uses a combination of bisection, secant, and inverse quadratic interpolation methods to find the root of a function of one variable near an initial value  $x_0$ .

```
z = fzero('function',x0,[tol],[trace])
```

Including `tol` returns a value of  $z$  accurate to within a relative error of `tol` (default `tol` = `eps` ( $=2.2 \times 10^{-16}$ )). Including `trace` gives information for each iteration. `'function'` can be a *MATLAB* built-in function (e.g. `sin`) or you can define a function `f` by writing an m-file `f.m` including

```
function y=f(x)
y = [whatever function of x you want to define]
```

**roots:** `roots` uses an entirely different method to find the roots of polynomials based on matrix algebra.

```
r = roots(p)
```

where `p` is a row vector containing the coefficients of the polynomial in descending order. This method returns all of the roots of the polynomial, including complex values.

**solve:** `solve` is a function within the symbolic math toolbox of *MATLAB* that finds the solution of the equations (the roots of the expressions) specified in the command

```
solve('eqn1','eqn2',... 'eqnN','var1','var2',... 'varN')
```

where `'eqn'` is a string containing the equation to be solved and `'var'` specifies the unknown variables; if not specified, the unknowns are determined as part of the solution.

The function `solve` will return an analytical solution if found, otherwise it will provide a numerical solution.

## 2.5. Example: Leonardo of Pisa's polynomial

Consider the polynomial

$$f(x) = x^3 + 2x^2 + 10x - 20 = 0$$

The remarkable thing about this polynomial is that in 1225, Leonardo of Pisa (*aka* Fibonacci) published an approximate solution to this equation,  $x=1.368\ 808\ 107\ 5$ , that he calculated by some unknown means. We can use the various techniques discussed above to see how many iterations it takes to find the roots to this accuracy. The `fprintf` command can be used to print the root  $x$  to the desired accuracy. For example, to print  $x$  to 10 decimal places on the screen,

```
fprintf(1, '%13.10f', x)
```

Beginning with the methods available in *MATLAB*, `roots` gives, for  $p = [1\ 2\ 10\ -20]$

```
ans =
    -1.6844 + 3.4313i
    -1.6844 - 3.4313i
     1.3688
```

Using `fprintf` to express the real root to greater precision, gives 1.368 808 107 8.

To use the *MATLAB* function `fzero`, first we must write an m-file, `f.m`

```
function y=f(p,x)
p=[1 2 10 -20];
y=polyval(p,x);
```

and then use the command `fzero('f', x0)`. For an initial guess of  $x_0=1.5$ , *MATLAB* returns  $z=1.368\ 808\ 107\ 8$ . The function `roots` is a little faster than `fzero` and finds all of the roots rather than the one closest to a given initial guess  $x_0$ . The *MATLAB* symbolic math function

```
s=solve('x^3+2*x^2+10*x-20')
```

returns

```
s =
[ 1/3*(352+6*3930^(1/2))^(1/3)-26/3/(352+6*3930^(1/2))^(1/3)-2/3;
 -1/6*(352+6*3930^(1/2))^(1/3)+13/3/(352+6*3930^(1/2))^(1/3)-2/3+
  1/2*i*3^(1/2)*(1/3*(352+6*3930^(1/2))^(1/3)+
  26/3/(352+6*3930^(1/2))^(1/3));
 -1/6*(352+6*3930^(1/2))^(1/3)+13/3/(352+6*3930^(1/2))^(1/3)-2/3-
  1/2*i*3^(1/2)*(1/3*(352+6*3930^(1/2))^(1/3)+
  26/3/(352+6*3930^(1/2))^(1/3))]
```

When evaluated, these expressions give the same answer as we obtained using `roots`.

For initial values  $x_1=1.5$  and  $x_2=1.0$ , the following bisection m-file gives the root  $x_3 = 1.368\ 835\ 449$  to within an error of  $1 \times 10^{-4}$  (1E-4) after 13 halvings. Accuracy comparable to the *MATLAB* `roots` function can be achieved by setting `tol=eps` (2.2E-16). With this value of the tolerance it takes 51 halvings to get the solution.



```
%bisect.m
```

```
tol=1e-4; %tol=eps; %tol can be set to any desired values
p=[1 2 10 -20]; %p are the coefficients of the cubic polynomial
%polyval evaluates a polynomial with coefficients p at the points
% specified in the vector x.
x=-5:0.1:5;
plot(x,polyval(p,x)); grid

%input initial values of x1 and x2 such that f(x1) has opposite sign from
% f(x2)
x1=input('input x1: ')
x2=input('input x2 such that f(x2) has opposite sign from f(x1): ')
%test to see if f(x1) and f(x2) have opposite signs
if polyval(p,x1)*polyval(p,x2)>0,
    x2=input('input x2 such that f(x2) has opposite sign from f(x1)!: ');
end;

%successively halve interval until approximations < tol apart
iter=0;
while abs(x1-x2)>tol
    x3=(x1+x2)./2;
    iter=iter+1;
    if polyval(p,x3)*polyval(p,x1)<0; , x2=x3;
    else x1=x3; end;
end;
fprintf(1,'root = %13.10f\n',x3)
fprintf(1,'iterations = %6.1f\n',iter)
```

A simple m-file for the secant method written using the algorithm below yields the value to eps precision after 8 iterations.

#### *Outline of algorithm for secant method*

```
If |f(x0)| < |f(x1)|, switch x0 and x1.
Do until |f(x2)| < specified tolerance,
    x2=x0-f(x0)*(x0-x1)/[f(x0)-f(x1)]
    x0=x1
    x1=x2
End
```

## 2.6. Iterative equations

A different approach to solving nonlinear equations is to approximate the solution iteratively using what is called 'fixed point' iteration. To use this technique, an equation of the form  $f(x) = 0$  (a form in which any equation can be expressed) is rewritten as  $x - g(x) = 0$ , or

$$x = g(x)$$

A value of  $x$  that satisfies  $x = g(x)$  will be a root of  $f(x)$ . Given an initial guess,  $x_0$ , the iteration formula is simply:  $x_1 = g(x_0)$ ,  $x_2 = g(x_1)$ , ...,  $x_{n+1} = g(x_n)$ . The function  $g$  is evaluated with successive values of  $x$  until  $x_{n+1} - x_n$  is as small as desired. There is generally more than one way in which a function can be written in the form  $x = g(x)$ . For example, the polynomial



$$ax^3 + bx^2 + cx + d = 0$$

can be rewritten as

$$x = -(ax^3 + bx^2 + d) / c$$

or

$$x = \sqrt{-(ax^3 + cx + d) / b}$$

Often different ways of writing the equation will yield different roots. This method can be used to solve many problems for which an explicit solution cannot be obtained, e.g., Manning's equation for  $h$  (Section 2.1, Box 2.2) or the equation describing the transformation of wavelength  $L$  as surface gravity waves propagate into shallow water (Section 2.8).

## 2.7. MATLAB methods for solving iterative equations

*MATLAB* has no built-in function for solving iterative equations, but the method is very straightforward and easy to code using the following algorithm:

*Outline of algorithm for solving iterative equations*

```
Rearrange equation into form  $x=g(x)$ 
Select starting value,  $x_1$ 
Do until  $|x_1-x_0| < \text{specified tolerance}$ ,
     $x_0=x_1$ 
     $x_1=g(x_0)$ 
End
```

This algorithm can be even further streamlined using *MATLAB*'s `eval` command. To use this, first assign a string containing the expression for  $g(x)$  to the variable `g` (i.e., `g='eqn'`) and then replacing the 'do loop' with

```
Select starting value,  $x$ 
Do until  $|x_1-x| < \text{specified tolerance}$ ,
     $x_1=x$ 
     $x=\text{eval}(g)$ 
End
```

As noted above, different arrangements of the equation can lead to different roots. As with all of these methods, the closer the initial value is to the solution, the faster and more certain the convergence. It turns out that if  $g(x)$  and  $g'(x)$  are continuous within an interval around a root of the equation  $x=g(x)$  and if  $|g'(x)| < 1$  for all  $x$  in the interval, then this method will converge to the root provided the initial value of  $x$  is within the interval (Gerald and Wheatley, 1999). A problem one can run into with iterative solutions is finding a form of  $g(x)$  that will converge to any particular root. Not only can solutions diverge, they can also display more complex behavior including cycling between values and even chaos [Box 2.3].

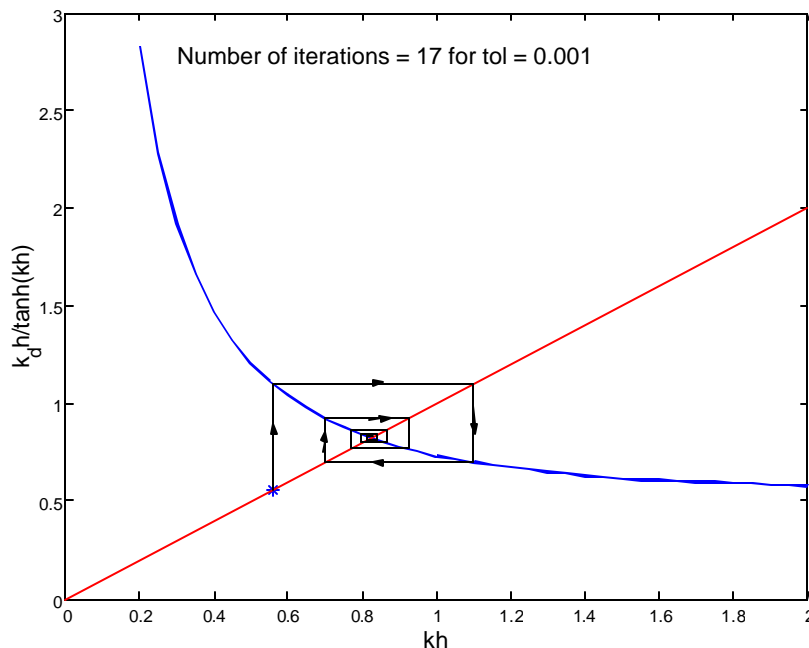
## 2.8. Example: Wavelength of surface gravity waves

Small-amplitude surface gravity waves move at a speed  $c=L/T$  where  $L$  is the wavelength and  $T$  is the period of the wave. In deep water, the wavelength is related to the period so that the speed is entirely dependent on the wave period. In shallow water, the wavelength becomes shorter and the speed becomes entirely dependent on water depth,  $h$ . What qualifies

as 'deep' and 'shallow' depends on the ratio of water depth to wavelength ( $h/L > 2$  is 'deep';  $h/L < 2$  is 'shallow'). The dependency of wavelength on depth and period is expressed through the nonlinear equation

$$kh = k_d h \tanh(kh)$$

where  $k = 2\pi/L$ ,  $k_d = 2\pi/L_d$ , and  $L_d = gT^2/(2\pi)$ . This equation is already in the form  $x = g(x)$ . If we plot  $k_d h \tanh(kh)$  against  $kh$  (Figure 2.5), we get a nice graphical illustration of how fixed-point iteration works. A wave period of 12 s, yielding a deep water wavelength of 225 m ( $k = 0.028 \text{ m}^{-1}$ ), and a water depth of 20 m was chosen for this example. Because  $20 < h/L < 2$ , the wavelength is expected to be between the 'deep' and 'shallow' water values. The curve in Figure 2.5 is the function  $g(x) = k_d h \tanh(kh)$  and the straight line is the relationship  $g(x) = x$ . The intersection of the curve and the line is the solution being sought. The iterations were begun with the initial value  $x = k_d h (= 0.559$  in this example). To solve: 1) start on the straight line in Figure 2.5 at  $k_d h = 0.559$ ; 2) evaluate  $g(0.559)$ , which is equivalent to moving vertically upward on the figure to the curved line representing  $g(x)$ ; 3) this gives a new value of  $kh$  [remember the iteration formula in this case is  $(kh)_{n+1} = k_d h \tanh(kh)_n$ ], which is equivalent to moving horizontally on the figure from the curve to the straight line. The graphical representation of this iterative solution continues with a vertical move downward to the  $g(x)$  curve, and so forth. It took 17 iterations to achieve the specified tolerance level of 0.001. The solution is  $kh = 0.824$ , or  $L = 152$  m.



**Figure 2.5.** Graphical illustration of an iterative solution to  $kh = k_d h \tanh(kh)$ .



## 2.9. Systems of linear equations

Solving sets of linear equations is another common application of numerical methods to algebraic equations. Consider the following simple set of linear equations.

$$\begin{aligned} ax + by &= c \\ dx + ey &= f \end{aligned}$$

This is a set of two equations for two unknowns,  $x$  and  $y$ . Simple systems can be solved easily by hand or by calculator. To solve the equations, multiply each equation by a constant to make the first terms identical. Then, subtract the equations to obtain equations for the remaining variables. Continuing this process finally yields one variable and its value. Then by back-substitution, values for all the variables can be obtained.

Although it is impractical to solve large systems of equations by hand, it turns out that the method of elimination and back substitution forms the basis for many of the methods used to solve large systems of equations by computer. These large systems are most easily expressed in terms of matrices.

A general set of  $n$  equations with  $n$  unknowns has the form

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \dots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \dots + a_{2n}x_n &= b_2 \\ \vdots & \\ a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \dots + a_{nn}x_n &= b_n \end{aligned}$$

$a_{ij}$  are the coefficients (known) of the equations,  $b_i$  are the right hand sides (known), and  $x_j$  are the unknowns. The set of equations can be written more compactly as

$$\sum_{j=1}^n a_{ij}x_j = b_i \quad \text{for } i=1,2,\dots,n$$

The  $a_{ij}$ 's represent a  $n \times n$  (square) matrix  $A$ , and  $b_i$  and  $x_j$  are vectors of length  $n$ . The system of linear equations can be written even more compactly in matrix notation as

$$Ax = b$$

The solution of this matrix equation is equally simple to write, but can be difficult to compute,

$$x = A^{-1}b$$

where  $A^{-1}$  is the inverse of the matrix  $A$ . Determining the inverse of a matrix is notoriously difficult and it is not the preferred method for obtaining the solution to a set of equations. Thus, the notation in the equation above is more schematic than utilitarian.

## 2.10. MATLAB methods for solving sets of linear equations

If a system of linear equations is expressed in terms of a square matrix of coefficients,  $A$ , the vector  $b$ , and the unknowns  $x$  such that  $Ax = b$ , then  $x$  can be found using either of two *MATLAB* commands:

```
x=inv(A)*b
```

or

$x=A \backslash b$

The backslash operator “ $\backslash$ ” solves the system of equations using Gaussian elimination, without calculating the matrix inverse, and is preferable to “ $\text{inv}$ ” because it is more efficient (in terms of computer time and memory) and has better error detection properties.

## 2.11. Gaussian elimination

This is a simple but effective method for solving systems of equations. Begin with the matrix of coefficients  $A$ , augmented by the vector  $b$ ,  $A / b$ . The steps are as follows:

**Step 1:** write the coefficients as an  $n \times n+1$  array and reduce the elements of the first column to a 1 in the first row and 0's in the remaining rows:

- (i) divide row 1 by  $a_{11}$
- (ii) multiply row 1 by  $a_{21}$  and subtract from row 2
- (iii) perform similar operation for row 3, etc;

It is important that the calculations be carried out with as much precision as possible.

**Step 2:** make the coefficient of the second column, second row 1 and the elements below the second row in the second column 0 using operations similar to those in Step 1.

**Step 3:** follow the same steps for each successive column to place 1's on the main diagonal of the coefficient matrix and 0's below. The operations for the last column will leave a simple equation of the form  $x_n = \hat{b}_n$ , where  $\hat{b}$  represents the terms on the right-hand sides of the modified equations.

**Step 4:** back substitute  $x_n$  into the equation corresponding to row  $n-1$  to find  $x_{n-1}$ . Continue until all of the  $x_i$ 's have been found.

This process results in a coefficient matrix that has 0's in the lower left portion of the matrix. This is called an upper triangular matrix. Upper triangular matrices have some nice properties from the point of view of matrix calculations. A matrix with 0's in the upper right side is called a lower triangular matrix. The upper triangular matrix  $U$  resulting from Gaussian elimination on the coefficient matrix  $A$  has a corresponding lower triangular matrix  $L$  such that  $LU=A$ . *MATLAB* has a command to perform what is called an LU decomposition of a matrix:

```
[L,U]=lu(X)
[L,U,P]=lu(X)
```

where  $P$  is a permutation matrix indicating rows that have been switched or pivoted in the process of computing  $L$  and  $U$  (which is done largely by Gaussian elimination). Pivoting is a process in which rows are rearranged so as to place the coefficients with the largest magnitudes on the diagonal at each step. This avoids winding up with a 0 on the diagonal.

### 2.12. Example: Gaussian elimination

Consider the set of equations

$$13x_1 - 8x_2 - 3x_3 = 20$$

$$-8x_1 + 10x_2 - x_3 = -5$$

$$-3x_1 - x_2 + 11x_3 = 0$$

For this set of equations

$$A = \begin{bmatrix} 13 & -8 & -3 \\ -8 & 10 & -1 \\ -3 & -1 & -11 \end{bmatrix}$$

and

$$b = \begin{bmatrix} 20 \\ -5 \\ 0 \end{bmatrix}$$

The augmented matrix is

$$A|b = \begin{bmatrix} 13 & -8 & -3 & 20 \\ -8 & 10 & -1 & -5 \\ -3 & -1 & -11 & 0 \end{bmatrix}$$

The first step in solving by Gaussian elimination is to divide the first row of the augmented matrix by  $a_{11} = 13$  and multiply it by  $a_{21} = -8$  and subtract it from row 2. Doing the same operations for row 3 gives

$$1 \quad -0.61 \quad -0.23 \quad 1.54$$

$$0 \quad 5.08 \quad -2.84 \quad 7.31$$

$$0 \quad -2.84 \quad 10.3 \quad 4.62$$

The second step is to make  $a_{22} = 1$  and  $a_{32} = 0$ , giving

$$1 \quad -0.61 \quad -0.23 \quad 1.54$$

$$0 \quad 1 \quad -0.56 \quad 1.43$$

$$0 \quad 0 \quad 8.71 \quad 8.71$$

The third step is to make  $a_{33} = 1$ , giving the set of equations

$$x_1 - 0.61x_2 - 0.23x_3 = 1.54$$

$$x_2 - 0.56x_3 = 1.43$$

$$x_3 = 1$$

Thus,  $x_3 = 1$ ;  $x_2 - 0.56 = 1.43$  or  $x_2 = 2$  when the calculation is carried out to full precision; and  $x_1 = 1.54 + 0.61(2) + 0.23$  or  $x_1 = 3$ .

In *MATLAB*, we can set  $A = [13 \ -8 \ -3; -8 \ 10 \ -1; -3 \ -1 \ 11]$  and  $b = [20; -5; 0]$ . Solving this with the *MATLAB* command  $x=A \backslash b$  produces (in much less time than it takes to do it by hand!)

$x =$

```
3.0000
2.0000
1.0000
```

$[L,U,P]=lu(A)$  gives

$L =$

```
1.0000      0      0
-0.6154    1.0000      0
-0.2308   -0.5606    1.0000
```

$U =$

```
13.0000   -8.0000   -3.0000
      0    5.0769   -2.8462
      0      0    8.7121
```

$P =$

```
1      0      0
0      1      0
0      0      1
```

### 2.13. Problems

1. The specific energy equation is 
$$h^3 - Eh^2 + \frac{q^2}{2g} = 0$$

where  $h$  is water depth,  $E$  is specific energy,  $q$  is specific discharge ( $Q/w$ ) and  $g$  is gravitational acceleration [Box 2.1]. For  $E=1$  m and  $q^2 / (2g) = 0.05$  m<sup>3</sup>, find the roots of the equation for  $h$  using the following methods:

- Plot the function and roughly estimate the roots (they are all real).
- Use the *MATLAB* command `roots` to obtain the roots.
- Write an m-file to find the roots using the secant method. Determine the number of iterations required for two different levels of error tolerance.

2. Manning's discharge equation, rearranged to solve for flow depth in a channel, is

$$h = Qn \left[ \left( \frac{wh}{w+2h} \right)^{2/3} S^{1/2} w \right]^{-1}$$

where  $Q$  = channel discharge,  $n$  = Manning roughness,  $w$  = channel width, and  $S$  = channel slope [Box 2.2].

- Write an m-file to iteratively solve for  $h$  given  $Q = 2.0$  m<sup>3</sup>s<sup>-1</sup>,  $n = 0.03$ ,  $S = 0.0005$ , and  $w = 5.0$  m. For a reasonable first estimate, assume  $w \gg h$ , so that  $wh/(w+2h) \approx h$ , in which case  $h = (Qn/(w\sqrt{S}))^{3/5}$ .
- Show your solution graphically. Begin with a plot of  $g(h)$  (right-hand side of equation for depth) vs.  $h$  and the straight line  $h=h$ . Starting with your initial guess for  $h$ , plot your solution as in Figure 2.5.

3. Solve the set of linear equations

$$\begin{aligned} x_1 + 2x_2 + x_3 &= 2 \\ 3x_1 + x_2 + 2x_3 &= 1 \\ -2x_2 + 4x_3 &= 1 \end{aligned}$$

using the *MATLAB* command `x=A\b` and by hand to 2 or 3 decimal places of accuracy. Compare the answers.

### 2.14. References

Gerald C.F. and P.O. Wheatley, *Applied Numerical Analysis*, 6<sup>th</sup> Edition, 698 pp., Addison Wesley, Reading, MA, 1999.

Hornberger, G.M., Raffensperger, J.P., Wiberg, P.L., and K. Eshleman, *Elements of Physical Hydrology*, 302 pp., Johns Hopkins Press, Baltimore, 1998.

May, R., Simple mathematical models with very complicated dynamics, *Nature*, 261: 459-467, 1976.

### Box 2.1. Specific energy

Specific energy,  $E$ , is the energy per unit weight of water flowing through a channel relative to the channel bottom,  $U^2/(2g) + h$ , where  $U$  is channel mean velocity,  $g$  is gravitational acceleration, and  $h$  is flow depth. If the flow is steady and uniform and the channel cross section is rectangular, then  $U = Q/(wh) = q_w/h$ , where  $Q$  is channel discharge,  $w$  is channel width, and  $q_w$  is discharge per unit width or specific discharge. Combining these equations gives the specific energy equation

$$E = \frac{q_w^2}{2gh^2} + h \quad \text{or} \quad h^3 - Eh^2 + \frac{q^2}{2g} = 0$$

Of the three roots to the specific energy equation, two are positive and one is negative. The negative root has no physical meaning. The two positive roots, called *alternate depths*, are both possible, depending on whether the Froude number,  $F = U / \sqrt{gh}$ , is subcritical ( $F < 1$ ) or supercritical ( $F > 1$ ). (See Hornberger et al. (1998) for the more information about specific energy and alternate depths.)



**Box 2.2. Manning equation**

The Manning equation is an equation commonly used to calculate the mean velocity  $U$  in a channel:

$$U = \frac{1}{n} R_H^{2/3} S^{1/2}$$

where  $n$  is the Manning roughness coefficient,  $R_H$  is hydraulic radius, and  $S$  is channel slope. Hydraulic radius is the ratio of the cross-sectional area,  $A$ , of flow in a channel to the length of the wetted perimeter,  $P$ . For a rectangular channel,  $R_H = wh/(w + 2h)$ ; if the channel is wide ( $w \gg h$ ),  $R_H \cong h$ . Values of  $n$  range from 0.025 for relatively smooth, straight streams to 0.075 for coarse bedded, overgrown channels. For steady, uniform flow, the channel bed slope  $S$  is equal to the water surface (friction) slope  $S_f$ . For non uniform flow, Manning's equation can still be used if  $S$  is replaced by  $S_f$ . The Manning equation can be combined with the discharge relationship  $Q=UA$  to give an expression for discharge

$$Q = \frac{1}{n} R_H^{2/3} S^{1/2} wh .$$

### Box 2.3. Complex behavior of iterative solutions

Fixed-point iteration can converge to a solution, as in Figure 2.5, or it can diverge. Fixed-point iteration can also produce more complicated patterns. Consider the equation


$$4ax^2 - (4a - 1)x = 0$$

One iteration formula for this equation is

$$x_{n+1} = 4ax_n(1 - x_n) \quad (\text{B2.3.1})$$

Other interpretations of equation (B2.3.1) are possible. For example, ecologists use equations like (B2.3.1) to model population dynamics. In this interpretation, the " $n$ " subscripts refer to generation (or time). Equation (B2.3.1) would then indicate a population (normalized so that values are between 0 and 1) with an intrinsic growth rate equal to  $4a$  and a carrying capacity of 1. Sequential calculations give the population evolution through time. For iterations that converge, the model represents a stable population. If the iteration diverges, the indication is that the modeled population is unstable.

Now consider (B2.3.1) graphically by plotting the sequence of calculations on a figure showing the line  $x=g(x)$  and the curve  $g(x)$  vs.  $x$ . The convergence (or divergence) of a calculation can also be visualized by plotting  $x_n$  vs.  $n$ . The following m-file uses both types of plots with (B2.3.1).



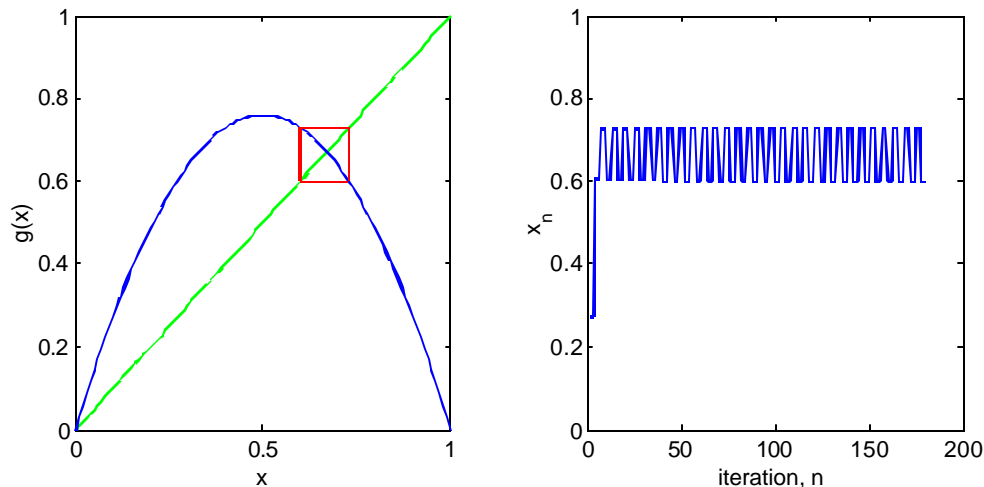
```
function b=hump(a)
%simple iteration leading to period doubling and chaos

x0=0.9;
xx=0:0.01:1;
yy=4*a*xx.*(1-xx);
subplot(121)
plot(xx,xx,xx,yy); hold on;
for i=1:3:180,
    x(i)=4*a*x0*(1-x0); x(i+1)=x(i); x(i+2)=x(i);
    if i<11,
        subplot(121),
        plot([x0 x0],[x0 x(i)],[x0 x(i)],[x(i) x(i)]); drawnow; pause(1);
    end;
    if i>11 & i<75,
        hold off; plot(xx,xx,xx,yy); hold on;
        plot([x(i-12) x(i-12)],[x(i-12) x(i-9)],[x(i-12) x(i-9)],...
            [x(i-9) x(i-9)]); drawnow;
        plot([x(i-9) x(i-9)],[x(i-9) x(i-6)],[x(i-9) x(i-6)],...
            [x(i-6) x(i-6)]);
        plot([x(i-6) x(i-6)],[x(i-6) x(i-3)],[x(i-6) x(i-3)],...
            [x(i-3) x(i-3)]);
        plot([x(i-3) x(i-3)],[x(i-3) x(i)],[x(i-3) x(i)],[x(i) x(i)]);
        drawnow; pause(0.6);
    end;
    x0=x(i);
end; hold off; xlabel('x'); ylabel('g(x)')
subplot(122)
i=1:180;
comet(i,x,0.2)
xlabel('iteration'); ylabel('x_n')
```

The function file `hump.m` does calculations with selected values of the parameter  $a$ , which should be greater than 0 and less than 1. For small values of  $a$ , the iteration converges to the root at 0 – or in terms of the population model, the intrinsic growth rate is too small to sustain the population and it "crashes." (Try running the program for values of  $a < 0.25$ .) For values of  $a$  in an intermediate range, the iteration converges to the positive root (where the function crosses the  $g(x)$  vs.  $x$  line). In terms of the population model, the population reaches a stable equilibrium. (Try running the program for  $a = 0.5$  or so.)

Something different happens when  $a > 0.75$ . The iteration does not diverge, but it does not converge to a root of the equation, either. Rather, the iteration "cycles" around the positive root. In terms of the population model, the population fluctuates between two levels forever (e.g., see the Figure 2.6 below for  $a = 0.76$ ).

As the  $a$  parameter is increased, other interesting things happen. At around  $a = 0.85$ , the period of the oscillation doubles. That is, the iteration cycles around the root, but in two distinct "loops." (Try running the program for  $a = 0.88$ .) As  $a$  is increased, the period doubles again (4 loops), and then again (8 loops). At about  $a = 0.95$ , the iteration becomes "chaotic," meaning that the calculations cycle around the root in never-repeating loops – the population varies all over the place in a pattern that is indistinguishable from random! (Try running the program for a value of  $a = 0.98$ .)



**Figure B2.3.1.** Results of function `hump.m` with  $a = 0.76$  illustrating cyclic behavior.

There are lots of places to read more about this. One classic paper (relative to the population model interpretation) is by May (1976).