CHAPTER 1

Computation with *MATLAB*

Numerical Methods in the Hydrological Sciences: AGU Lecture Notes 1
Copyright 2004 by the American Geophysical Union
doi: 10.1029/001LN03
(Version 2004-08-02)

1. *Computation with MATLAB*

This set of lecture notes describes numerical methods for solving a varietry of problems in the hydrological sciences. The computational engine used for the work is *MATLAB*, a comerically available software package. This first chapter introduces the basics of working with *MATLAB*.

1.1. Data in *MATLAB*

The basics: Matrices and vectors

MATLAB operates on ***matrices***, which are arrays of numbers. That is, a matrix is a "table" of numbers with, say, N rows and M columns. For example, if N=3 and M=2, a matrix A has dimension 3x2. Assignment of such a matrix in *MATLAB* can be accomplished as follows. Type in a "`[`" to let *MATLAB* know that you are going to input a matrix. Type the numbers for the first row of the matrix (say "`1`" and "`2`") and then type a "`;`" to let *MATLAB* know that you have reached the end of a row. Type the next two rows in the same way and end with a "`]`".

```
A=[1 2;3 4;5 6]
```

After typing this in, *MATLAB* echoes the contents of the matrix, A.

```
A =

     1     2

     3     4

     5     6
```

A ***vector*** of numbers is simply a matrix with one of the dimensions equal to one. For example, you might have data on air temperatures taken at noon every day over a week. These can be represented as a vector in *MATLAB*.

```
T=[12.1 13.6 9.5 8.2 10.4 11.7 11.9]
```

Typing this line into *MATLAB* results in:

```
T =

   12.1000   13.6000    9.5000    8.2000   10.4000   11.7000   11.9000
```

As defined, T is a "row vector". It can be converted to a column vector by taking its *transpose,* an action that is accomplished in *MATLAB* by placing a single quote after the matrix or vector. That is, to define the transpose of T, we type

```
T'
ans =
   12.1000
   13.6000
    9.5000
    8.2000
   10.4000
   11.7000
```

```
    11.9000
```

Setting vectors and matrices with internal MATLAB commands

Special vectors and matrices can be constructed with built-in *MATLAB* statements. For example, it often is useful to construct a vector that covers a certain range and has evenly-spaced entries. This would be the case where we want to examine a sequence of regularly spaced data where we did not explicitly have the independent variable recorded. If, for example, we had daily temperatures recorded for a year and wanted to plot the data versus a time variable, we can form such a variable as follows.

```
t=1:1:365;
```

Note that we put a semicolon at the end of this assignment statement. This tells *MATLAB* not to echo the 365-element vector of times.

Other *MATLAB* functions that are useful in setting vectors and matrices are `zeros` (sets a matrix with all zero elements), `ones` (sets a matrix with elements equal to one), and `rand` (sets a matrix with elements chosen using a pseudo-random-number generator for a uniform distribution on [0,1]). Try typing `a=rand(3,4)` and `b=ones(1,10)` to get the feel of these utilities. [The command `randn` generates numbers from a normal distribution.]

Simple functions and plots

MATLAB also has many built-in functions. Thus, we can form a sine function for a seasonal variable using the following *MATLAB* statement.

```
t=1:5:365;
temp=15*sin(2*pi*t/365)+12;
```

We can look at the result by using *MATLAB* graphics capabilities (Figure 1.1).

```
plot(t,temp,'+') ;
xlabel('Time, days');
ylabel('Smoothed temperature, degrees C')
```

The series of commands above actually produce a figure with default characteristics on line widths, font size, and so forth. Figure 1.1 and other figures in these lecture notes are modified to make them more easily readable. *MATLAB* commands allow specification of a variety of aspects of figures [Box 1.1].

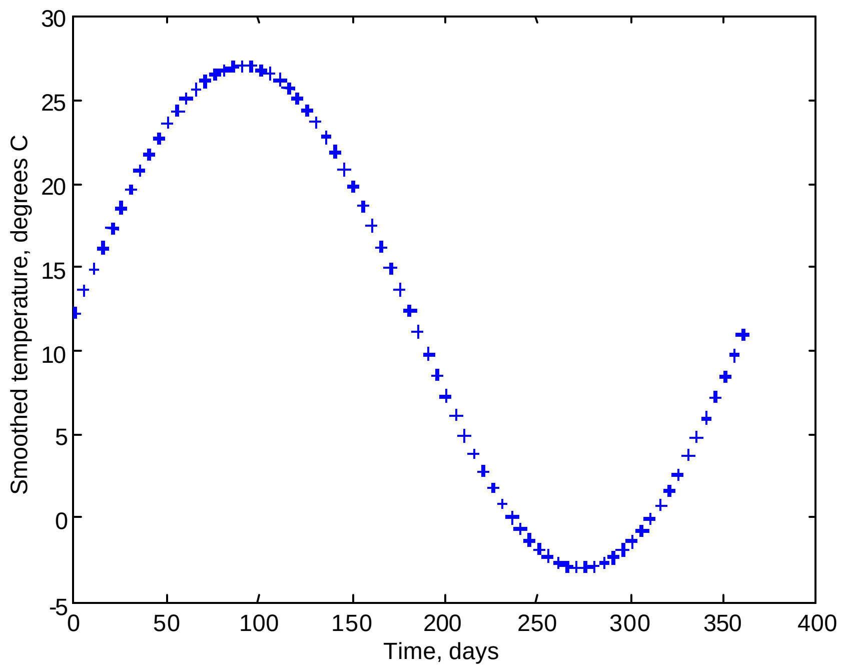


Figure 1.1. Example of a simple graph produced in *MATLAB*.

One of the most important things to note about *MATLAB* as you learn to use it is the extensive on-line HELP facility. Typing `help` by itself gives a series of choices from which you can refine your search. Typing `help *item*` gives help on the item chosen. For example to get help with the *MATLAB* plot utility[1],

```
help plot
 PLOT Plot vectors or matrices.
   PLOT(X,Y) plots vector X versus vector Y. If X or Y is a matrix,
   then the vector is plotted versus the rows or columns of the matrix,
   whichever line up.

   PLOT(Y) plots the columns of Y versus their index.
   If Y is complex, PLOT(Y) is equivalent to PLOT(real(Y),imag(Y)).
   In all other uses of PLOT, the imaginary part is ignored.

   Various line types, plot symbols and colors may be obtained with
   PLOT(X,Y,S) where S is a 1, 2 or 3 character string made from
   the following characters:

          y     yellow        .     point
          m     magenta       o     circle
          c     cyan          x     x-mark
          r     red           +     plus
          g     green         -     solid
          b     blue          *     star
```

[1] Reprinted with permission from The Mathworks, Inc.

```
          w     white         :     dotted
          k     black         -.    dashdot
                              --    dashed
   For example, PLOT(X,Y,'c+') plots a cyan plus at each data point.

   PLOT(X1,Y1,S1,X2,Y2,S2,X3,Y3,S3,...) combines the plots defined by
   the (X,Y,S) triples, where the X's and Y's are vectors or matrices
   and the S's are strings.

   For example, PLOT(X,Y,'y-',X,Y,'go') plots the data twice, with a
   solid yellow line interpolating green circles at the data points.

   The PLOT command, if no color is specified, makes automatic use of
   the colors specified by the axes ColorOrder property.  The default
   ColorOrder is listed in the table above for color systems where the
   default is yellow for one line, and for multiple lines, to cycle
   through the first six colors in the table.  For monochrome systems,
   PLOT cycles over the axes LineStyleOrder property.

   PLOT returns a column vector of handles to LINE objects, one
   handle per line.

   The X,Y pairs, or X,Y,S triples, can be followed by
   parameter/value pairs to specify additional properties
   of the lines.

   See also SEMILOGX, SEMILOGY, LOGLOG, GRID, CLF, CLC, TITLE,
   XLABEL, YLABEL, AXIS, AXES, HOLD, and SUBPLOT.
```

The `lookfor` command is often useful for finding what might be available on a given topic. For example, you might want to know what is available in *MATLAB* for interpolation[2].

```
lookfor interpolation


ICUBIC 1-D cubic Interpolation.
INTERP1 1-D interpolation (table lookup).
INTERP1Q Quick 1-D linear interpolation..
INTERP2 2-D interpolation (table lookup).
INTERP3 3-D interpolation (table lookup).
INTERP4 2-D bilinear data interpolation.
INTERP5 2-D bicubic data interpolation.
INTERP6 2-D Nearest neighbor interpolation.
INTERPFT 1-D interpolation using FFT method.
INTERPN N-D interpolation (table lookup).
SPLINE Cubic spline interpolation.
NTRP113 Interpolation helper function for ODE113.
NTRP15S Interpolation helper function for ODE15S.
NTRP23 Interpolation helper function for ODE23.
NTRP23S Interpolation helper function for ODE23S.
NTRP45 Interpolation helper function for ODE45.
NDGRID Generation of arrays for N-D functions and interpolation.
```

[2] Reprinted with permission from The Mathworks, Inc.

```
PDEARCL Interpolation between parametric representation and arc length.
```

After finding the items available, you can use the `help` command to get more information (e.g., `help INTERP1`).

Reading data into MATLAB

How about getting data into *MATLAB* without typing it directly from the keyboard? Suppose you have data from a laboratory experiment in which water flows through a column of sand. A chemical tracer (for example sodium chloride) is introduced as a pulse into the inflow end of the column and the breakthrough curve, chloride concentration as a function of time at the outflow end of the column, is observed. The result is a data file listing number of pore volumes that have been eluted (dimensionless surrogate for time) and relative concentration (measured concentration divided by the concentration of the pulse input) of the effluent water. The data look like this

```
0.0700   0.02
0.1400   0.02
0.2100   0.02
0.2700   0.02
0.3400   0.02
0.4100   0.02
0.4800   0.02
0.5500   0.02
0.6200   0.02
0.6800   0.000
0.7500   0.0000
0.8200   0.0000
0.8900   0.02
0.9600   0.1
1.0300   0.44
1.0900   0.78
1.1600   0.66
1.2300   0.3
1.3000   0.14
1.3700   0.06
1.4400   0.02
1.5000   0.02
```

Suppose these data are in a file called "bt.dat". Use the `load` command to get the data into *MATLAB*.

```
load bt.dat
```

Note that the files to be loaded into *MATLAB must* have an extension unless the extension is "mat". For example, a file named "bt.dat" can be loaded by `load bt.dat`; similarly, "bt.q" or "bt.xxx" can be loaded by typing the full name. If the file is named "bt.mat", the command `load bt` will work. Now we can set "pv" (for number of pore volumes) equal to the first column of the file and "rc" (for relative concentration) equal to the second column of the file by typing

```
pv=bt(:,1);
```

Note that the colon notation is used to select entire columns of the matrix. That is `bt(:,1)` means "bt(all elements, column 1)". In the same way, we set "rc" (for relative concentration) equal to the second column in the data file "bt".

```
rc=bt(:,2);
```

To look at the breakthrough curve (Figure 1.2):

```
plot(pv,rc,'or',pv,rc,'b')
xlabel('Dimensionless time, pore volumes')
ylabel('Reduced concentration, c/c_0')
```

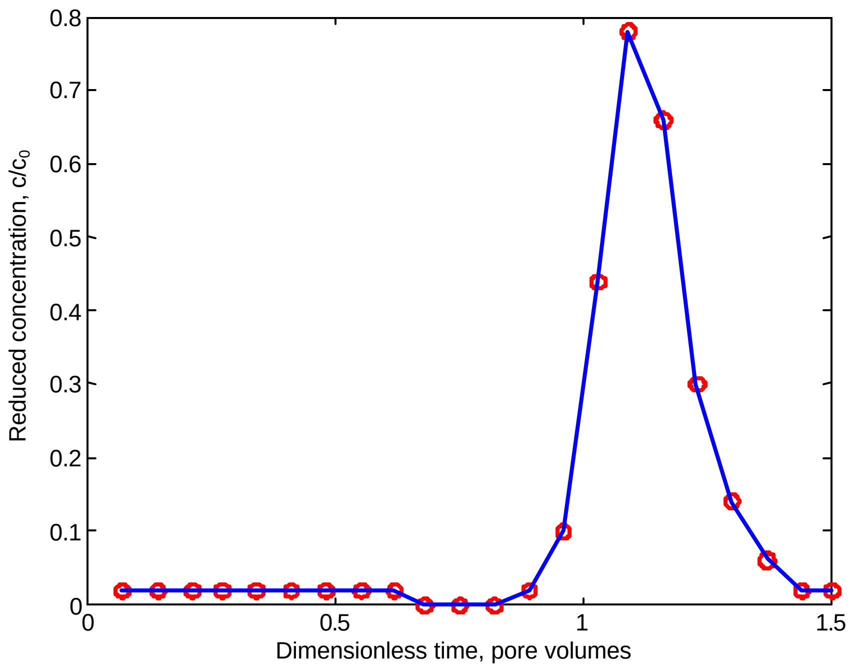


Figure 1.2. Plot of data from a laboratory column experiment.

You can load data into *MATLAB* as long as the data are in an array containing only numbers (e.g., no table headings) and containing no blank spaces (e.g., zeroes must be typed explicitly in the file, not left as blanks). (See the *MATLAB* functions `TBLREAD`, `TDFREAD`, and `XLSREAD` for other options to get data into the program.)

1.2. Mathematical operations with matrices

MATLAB is a powerful engine for data analysis and modelling primarily because of its design for performing matrix calculations. All standard mathematical operations are supported. You just have to remember that matrices on which you are operating must be "conformable" for that particular operation.

Elementary operations: Addition, subtraction, and so forth

Matrices (vectors as a special case) are conformable for addition (and subtraction) as long as they are the same size. Thus, if

```
A=[1 2 3; 4 5 6]

A =
     1     2     3
     4     5     6
```

and

```
B=[7 8 9; 10 11 12]

B =
     7     8     9
    10    11    12
```

then the sum is what you would expect:

```
A+B

ans =
     8    10    12
    14    16    18
```

as is the difference:

```
A-B

ans =
    -6    -6    -6
    -6    -6    -6
```

Multiplication by a scalar is also straightforward.

```
2*A

ans =
     2     4     6
     8    10    12
```

All of the functions available in *MATLAB* can be applied to matrices. These include sin, cos, exp, log, and others. For example,

```
sqrt(A)

ans =
    1.0000    1.4142    1.7321
    2.0000    2.2361    2.4495
```

Note that these functions operate on each element of the matrix.

Matrix multiplication

Two matrices, say C of dimension n by m and D of dimension k by j , are conformable for multiplication if m=k. Matrices A and B above are *not* conformable for multiplication because A is

2x3 and B is also 2x3. (That is, "m" is 3 and "k" is 2.) If you try to multiply the two, *MATLAB* gives an error message.

```
A*B

??? Error using ==> *
Inner matrix dimensions must agree.
```

Now the *transpose* of B is a 3x2 matrix. (In *MATLAB* the transpose is formed by placing a single quotation mark (') after the matrix.)

```
BT=B'


BT =
     7    10
     8    11
     9    12
```

By the rules of matrix multiplication, A and BT are conformable for multiplication.

```
A*BT


ans =
    50    68
   122   167
```

The case of multiplication of a matrix and a vector is of particular importance as systems of equations are represented in this form. Note that in this case, the vector length must equal the number of rows in the matrix. The form of a set of equations is Ax=b where A is a matrix of (known) coefficients, x is the unknown vector, and b is a vector of known quantities. The system of equations

$$3x + 3y = 9$$

$$2x - 2y = -2$$

is represented in matrix notation as

$$\begin{bmatrix} 3 & 3 \\ 2 & -2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 9 \\ -2 \end{bmatrix}$$

Our main interest in such matrix equations is in solving them -- determining values for x and y that satisfy the equations. We will explore this topic in some detail in a later lecture. For now, just accept that the "backslash" operator in *MATLAB* does the trick.

```
A=[3 3;2 -2];
b=[9 -2]';
xy=A\b


xy =
     1
     2
```

Multiplication on an element-by-element basis

Often we need to operate on a vector element by element. *MATLAB* uses a period at the end of a vector to indicate operation on an element-by-element basis. For example, suppose we want to take the vector of concentrations that we defined above and square each element. Note that the command `c2=c*c` will not work because the "`*`" implies matrix multiplication and the only time a matrix is conformable with itself for multiplication is when it is square. (Note that, even in the case of a square matrix, `A*A` is *not* equivalent to squaring each element! Try it. Define a square matrix by typing, say, `A=rand(3,3)` and examine `A` and `A*A`.) To have *MATLAB* square the elements of c, we type `c.*c` noting that the dot after the first c specifies calculation element by element. Alternatively, we can type `c.^2`, using *MATLAB*'s carat notation for exponentiation.

```
c2=c(12:18).*c(12:18)


c2 =
         0
    0.0004
    0.0100
    0.1936
    0.6084
    0.4356
    0.0900
```

Note that we looked at the result for only elements 12 to 18 of the vector by indicating the range 12 to 18 in parentheses after `c`. This is standard notation in *MATLAB*; `A(n1:n2,m1:m2)` indicates the submatrix with rows n1 to n2 and columns m1 to m2 of the original matrix.

Another example of the use of element-by element computation is finding the reciprocal of the elements of a vector. Take the vector `pv` from the breakthrough-curve data. Suppose we want to find $1/pv_i$ for i=12 to 18. As it turns out, we can't use `1/pv` because division for matrices is not defined. Also, `inv(pv)` -- read "inverse of pv"-- won't work because the inverse of a matrix is <u>not</u> the same as inverting element by element. So again, we use the "dot" at the end of a vector to denote the element-by-element operation.

```
oneoverpv=(ones(size(pv(12:18)))./pv(12:18))'


oneoverpv =
    1.2195    1.1236    1.0417    0.9709    0.9174    0.8621    0.8130
```

Note that we used the *MATLAB* `size` function to specify a vector of ones with the same length as the set of "pv's" that we want to invert. We also placed a dot at the end of the vector of ones to indicate that we wanted to take the inverse of each element of the vector.

As a final example, consider calculating the difference between successive values of `rc` of the breakthrough data that we loaded earlier (i.e., Δrc) divided by the difference in successive `pv` values (Δpv). The *MATLAB* `diff` function calculates these differences. For example,

```
diff(rc)'


ans =
  Columns 1 through 7
         0         0         0         0         0         0         0
  Columns 8 through 14
         0   -0.0200         0         0    0.0200    0.0800    0.3400
  Columns 15 through 21
    0.3400   -0.1200   -0.3600   -0.1600   -0.0800   -0.0400         0
```

We use the dot at the end of the numerator to calculate Δrc/Δpv on a term by term basis:

```
(diff(rc)./diff(pv))'


ans =
  Columns 1 through 7
         0         0         0         0         0         0         0
  Columns 8 through 14
         0   -0.3333         0         0    0.2857    1.1429    4.8571
  Columns 15 through 21
    5.6667   -1.7143   -5.1429   -2.2857   -1.1429   -0.5714         0
```

1.3. Symbolic math toolbox

The student version of *MATLAB* includes the symbolic mathematics toolbox. This tool performs mathematical operations – differentiation, integration, equation solving, etc. – on symbols. That is, the toolbox operates on mathematical symbols (the "x" of algebra!) and not on numbers. Although we concentrate on *numerical* methods in this series of lectures, access to the symbolic math toolbox will be useful. For example, the symbolic math toolbox can be used to obtain the exact analytical solution to some of the problems we will attack with numerical methods, allowing evaluation of errors in the numerical (by definition, approximate) solution. Our concentration, however, will be very strongly on numerical computation. We present just a few basics here.

To manipulate symbols, *MATLAB* first must be informed about what variables are to be treated symbolically. Such variables are declared using `syms`. For example, suppose we want to work with the Manning equation,

$$Q = \frac{1}{n} S^{1/2} \left(\frac{A}{p} \right)^{2/3} A .$$

We would type the declaration

```
syms Q n S A p
```

to make all of the variables in the equation symbolic. Now suppose that we want to solve this equation for the wetted perimeter, p. We can use the `solve` command:

```
p=solve('Q=(1/n)*sqrt(S)*(A/p)^(2/3)*A',p)
```

which results in the following.

```
p =

[  1/Q^2/n^2*(Q*n*S^(3/2)*A)^(1/2)*A^2]
[ -1/Q^2/n^2*(Q*n*S^(3/2)*A)^(1/2)*A^2]
```

Note that both the positive and negative solutions are given. To make the solution clearer (the positive solution, which is the one of interest), we use the `pretty` command:

```
pretty(p(1)).
```

Try this example to see how the solution looks in a more traditionally readable form.

The *MATLAB* toolbox can also be used to solve differential equations. Consider steady, one-dimensional flow of groundwater between two drains. The head remains constant at 5 m in one drain and at 10 m in the other. The aquifer, with transmissivity T, is being recharged at a constant rate, w, along its length. The (linearized) equation describing the case is:

$$\frac{d^2h}{dx^2} = -\frac{w}{T}$$
$$h(0) = 5$$
$$h(100) = 10.$$

The solution to the equation can be gotten with the following commands.

```
syms h T w
h=dsolve('D2h=-w/T','h(0)=5','h(100)=10','x')
```

This command results in the following response.

```
h =

-1/2*w/T*x^2+1/20*(1000*w+T)/T*x+5
```

Again, the `pretty` command puts the solution into a more standard symbolic format. To determine the solution for w/T=0.002 per meter, simply insert the numerical value at the appropriate place.

```
h=dsolve('D2h=-0.002','h(0)=5','h(100)=10','x')

h =

-1/1000*x^2+3/20*x+5
```

The command: `ezplot(h,[0 100])` shows graphically how head in the aquifer varies with distance (Figure 1.3).

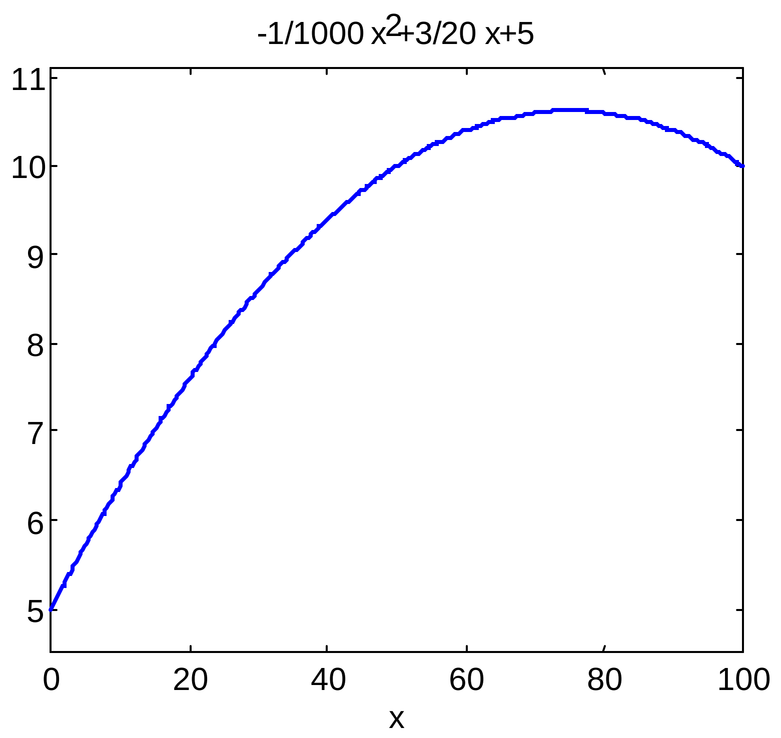


Figure 1.3. Solution to differential equation from the symbolic math toolbox.

1.4. Programming in *MATLAB*

To utilize the full power of *MATLAB*, you will need to learn to write "m-files". These files are programs to execute a sequence of calculations and operations. For example, you might want to have a program to take a set of data and perform some statistical calculation. The files can contain any legitimate *MATLAB* command. The files can have any legal name with the extension "m". *MATLAB* looks for files with an "m" extension when the prefix is typed, interprets the commands, and executes them in sequence.

Script files

The first type of m-file that you will use is a straight script file. Script files are, literally, a list of commands that are interpreted and executed. To illustrate, suppose you want to create a program to load a set of concentration-time data (defining a breakthrough curve) and calculate the mean travel time as $\Sigma c_i t_i / \Sigma c_i$. Using the *MATLAB* editor/debugger (or your favorite word processor), you type in the following:

```
% Calculate mean travel time
%
load bt.dat
t=bt(:,1);c=bt(:,2);
tbar=sum(t.*c)/sum(c)
```

and save it in file tbar.m. (Note that lines in the file preceded by a "%" are comments; *MATLAB* ignores them.) Then, from within *MATLAB*, type `tbar` and return.

```
tbar


tbar =
    1.0796
```

The 'tbar' code is a particularly simple m-file, but you should be able to appreciate how easy it is to program more complicated tasks in *MATLAB*. For example, the script below calculates daily estimates of potential evapotranspiration using the Hamon (1961) method [Box 1.2].

```
% Hamon calculation of PET in mm/day
% inputs are (1) latitude (degrees) and (2) a file name with data:
%     Julian day
%     Temperature (Celsius)
% output is daily values of PET

fname=input('name of file listing Julian day and T in Celsius? ');
fid=fopen(fname,'r');
data=fscanf(fid,'%g');
data=reshape(data,2,length(data)/2);data=data';
status=fclose(fid);
J=data(:,1);T=data(:,2);   % data are J, Julian day, and T, temperature

phi=input('latitude in degrees? ');    % latitude
delta=0.4093*sin((2*pi/365)*J-1.405);  % solar declination
omega_s=acos(-tan(2*pi*phi/360).*tan(delta));   % sunset hour angle
Nt=24*omega_s/pi;              % hours in day
a=0.6108;b=17.27;c=237.3;
es=a*exp(b*T./(T+c));             % saturation vapor pressure
E=(2.1*(Nt.^2).*es)./(T+273.3);     % Hamon PET
i_cold=find(T<=0);E(i_cold)=0;

plot(J,E)
xlabel('Time, Julian day')
ylabel('Hamon PET, mm/day')
```

Executing the code with one year of data for a site in central Virginia shows the annual cycle of potential evapotranspiration with day-to-day variability introduced by temperature fluctuations (Figure 1.4).

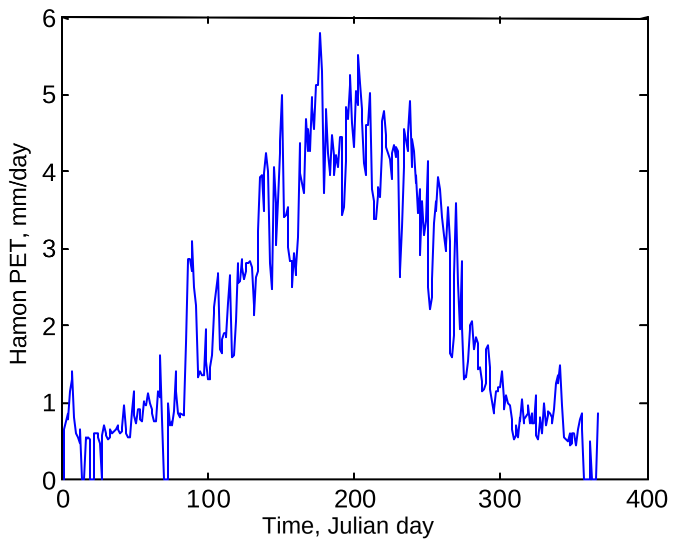


Figure 1.4. Potential evapotranspiration estimated using Hamon's method.

Function files

The other type of m-file, which is used more frequently than a script file, is a function file. A function file operates on vectors or matrices specified in the call and returns variables defined in the function statement. The *MATLAB* help on this topic illustrates the concept nicely[3].

```
help function

 FUNCTION Function M-files.
   New functions may be added to MATLAB's vocabulary if they
   are expressed in terms of other existing functions. The
   commands and functions that comprise the new function must
   be put in a file whose name defines the name of the new
   function, with a filename extension of '.m'. At the top of
   the file must be a line that contains the syntax definition
   for the new function. For example, the existence of a file
   on disk called STAT.M with:

           function [mean,stdev] = stat(x)
           n = length(x);
           mean = sum(x) / n;
           stdev = sqrt(sum((x - mean).^2)/n);

   defines a new function called STAT that calculates the
   mean and standard deviation of a vector. The variables
   within the body of the function are all local variables.
   See SCRIPT for procedures that work globally on the work-
   space.
   See also ECHO, SCRIPT.
```

[3] Reprinted with permission from The Mathworks, Inc.

Function files are especially useful for doing computations to be applied to different data sets or for different values of parameters in an equation. As an example of a case where we want a function to operate on various data sets, consider the code below for doing simple linear regression on a set of x-y data.

```
function [m,b,sm,sb,r2]=linreg(x,y)
%
% Simple linear regression of variable y on variable x
%   y=mx+b
% syntax: [m,b,sm,sb,r2]=linreg(x,y)
% m is estimated slope; sm is standard error of estimated slope
% b is estimated intercept; sb is standard error of estimated intercept
% r2 is the coefficient of variation
% Solution to normal equations using unweighted least squares:
%   m=(n*sum(x*y)-sum(x)*sum(y))/(n*sum(x^2)-(sum(x))^2)
%   b=mean(y)-m*mean(x)
%
[k1,k2]=size(x);
n=max(k1,k2);
m=(n*sum(x.*y)-sum(x)*sum(y))/(n*sum(x.^2)-sum(x)^2);
b=mean(y)-m*mean(x);                   % regression parameters
r=(y-b-m*x);                           % residuals
s2=sum(r.*r)/(n-2);                    % mean squared error
sm=sqrt(s2*(1/n+mean(x)^2)/sum((x-mean(x).^2))); % standard error
sb=sqrt(s2/sum((x-mean(x).^2)));       % standard error
numerator=sum(x.*y)-(sum(x)*sum(y)/n)).^2;
denominator=(sum(x.^2)-(sum(x)^2/n))*(sum(y.^2)-(sum(y).^2/n));
r2=numerator/denominator;              % r-squared
y_hat=m*x+b;                           % estimated y from regression
%
% 95% prediction confidence intervals based on large values of n
factor= sqrt(1+1/n+(x-mean(x)).^2/sum((x-mean(x)).^2));
y_upper=y_hat+2.306*sqrt(s2)*factor;
y_lower=y_hat-2.306*sqrt(s2)*factor;
d=max(x)-min(x);
xi=min(x):d/50:max(x);
yh=m*xi+b;
yyl=spline(x,y_lower,xi);
yyu=spline(x,y_upper,xi);
plot(x,y,'+r',xi,yh,xi,yyu,'g',xi,yyl,'g','MarkerSize',8)
xlabel('x')
ylabel('y')
text(0.2*mean(x),0.8*max(y),['r^2=' num2str(r2)])
```

As an example of a function file to compute results with different model parameters, consider the case of dispersion of a non-reactive contaminant in a homogeneous aquifer with average pore velocity v. A mass m of contaminant is assumed to be injected instantaneously into an extensive aquifer of thickness b (Figure 1.5). The equation governing the flow and dispersion (determined by the dispersion coefficients, D_x and D_y, of the contaminant and the analytical solution to the equation for this case are (Wilson and Miller 1978):

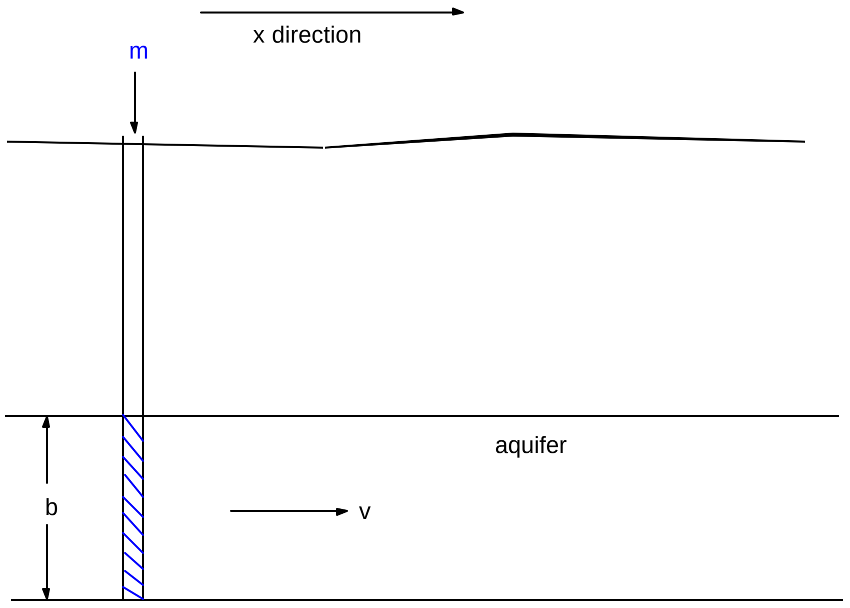


Figure 1.5. Schematic of contaminant pulse injection into an aquifer.

Equation: $$\frac{\partial c}{\partial t} = D_x \frac{\partial^2 c}{\partial x^2} + D_y \frac{\partial^2 c}{\partial y^2} - v\frac{\partial c}{\partial x}$$

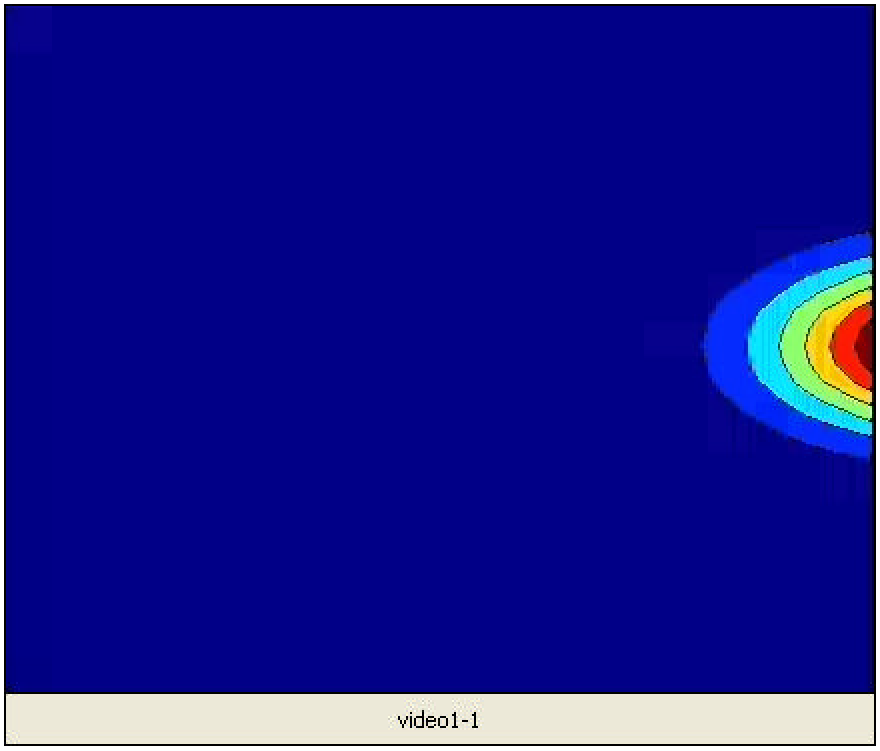
Solution: $$c(x,y,t) = \frac{m/b}{4pt\sqrt{D_x D_y}} \exp\left(-\frac{(x-vt)^2}{4D_x t} - \frac{y^2}{4D_y t}\right)$$

Note that m,b,v,D_x, and D_y are passed to the function file as parameters. This solution is sometimes referred to as a "Gaussian puff" because the cloud spreads out in the form of a Gaussian distribution. The *MATLAB* function file below computes the solution and also demonstrates one of the visualization tools available in *MATLAB*. The solution is contoured at a series of times and each "frame" is saved using `getframe`. After executing `G_puff` the command `movie(M)`can be used to display the sequence of frames (Video 1.1).

```
function M=G_puff(m,Dx,Dy,v,b)
% Gaussian puff over a 100-meter length
% syntax: M=G_puff(m,Dx,Dy,v,b)
% m=mass; v=average pore velocity; b=aquifer thickness
% Dx=axial dispersion coeff.; Dy=transverse dispersion coeff.;

[X, Y]=meshgrid(-5:2.5:100, -50:2.5:50);
tmax=110/v;
t=tmax/20:tmax/20:tmax;
n=length(t);
```

```
axis tight
set(gca,'nextplot','replacechildren');
for k=1:n
e=exp(-(X-v*t(k)).^2/(4*Dx*t(k))-Y.^2/(4*Dy*t(k)));
c=(m/b)/(4*pi*t(k)*sqrt(Dx*Dy))*e;
contourf(c)
set(gca,'Visible','off');
M(k)=getframe;
end
```



Video 1.1. Transport of a contaminant in an aquifer.

1.5. Summary

Throughout the series of lectures presented in these notes, *MATLAB* is used for computation and for programming. You should work to familiarize yourself with this software. Read these notes, read the documentation (available with the student edition of *MATLAB*, which is recommended strongly), use the on-line help facility in the program liberally, and practice working with *MATLAB* by writing m files.

1.6. Problems

The problems below are designed to introduce work with *MATLAB* and to do some basic data manipulation and simple programming. The data below are precipitation in mm for the Southern Piedmont of the U.S. (Karl et al. 1994). The rows represent years from 1960 through 1989.

JAN	FEB	MAR	APR	MAY	JUN	JUL	AUG	SEP	OCT	NOV	DEC
133.8	157.6	120.9	84.5	87.5	71.3	130.0	112.7	100.1	55.2	28.0	48.1
70.4	153.6	115.7	136.7	91.7	143.1	96.4	159.4	35.6	44.8	54.2	127.1
139.2	100.0	124.7	94.4	63.3	136.0	96.4	77.0	121.5	35.6	130.1	73.1
92.0	78.5	127.1	69.3	80.2	106.5	87.5	53.9	114.0	4.7	135.7	70.3
142.7	125.5	129.5	118.6	51.9	90.1	170.1	166.0	84.4	178.4	49.6	111.8
48.9	104.7	159.1	81.7	46.6	157.4	151.6	89.5	68.2	64.3	44.1	15.8
130.5	109.0	69.8	67.7	116.2	75.6	77.5	94.7	130.9	74.9	40.1	78.5
62.5	86.9	67.3	55.3	127.3	73.8	115.7	176.8	64.2	40.4	68.0	123.0
111.9	19.4	79.4	68.4	107.9	125.4	138.9	69.0	34.3	92.6	107.1	69.3
64.8	84.8	103.6	112.5	84.8	107.2	120.5	123.6	105.2	37.6	37.1	114.4
58.5	79.4	121.5	66.6	75.9	67.9	136.5	140.7	50.1	125.8	63.6	75.3
100.1	116.5	133.8	73.8	153.0	87.5	147.7	137.7	108.9	171.9	69.5	43.7
107.8	108.1	75.3	58.0	145.9	181.2	117.4	86.1	84.0	89.2	125.7	140.2
97.9	114.1	144.2	126.6	109.0	172.1	85.7	101.5	72.9	60.5	27.9	146.6
122.4	90.2	78.6	66.4	126.9	91.1	91.4	139.6	121.0	14.4	61.5	132.5
137.9	108.9	185.7	66.0	141.0	90.3	213.7	71.7	182.5	58.7	71.6	94.5
84.0	36.5	98.0	24.2	146.3	144.2	85.4	66.2	116.3	184.6	77.2	107.6
68.3	31.9	145.1	52.5	60.7	84.0	54.8	114.9	111.8	134.6	103.4	89.7
199.0	22.2	111.0	101.5	116.5	95.6	140.7	116.0	54.9	28.9	75.8	76.3
148.8	157.4	84.4	134.4	116.3	131.2	95.9	93.8	208.8	77.9	109.4	35.6
120.7	42.6	211.3	68.3	86.3	80.3	82.5	53.5	128.2	88.4	72.6	29.9
17.1	98.0	58.5	45.2	75.9	102.4	146.0	123.2	72.5	82.8	24.0	139.6
120.3	131.1	57.3	114.4	93.7	170.0	118.4	93.9	67.0	88.2	74.0	106.4
68.4	126.5	176.7	147.2	81.6	79.2	51.5	79.7	77.2	98.0	119.4	172.8
95.1	137.5	148.7	119.5	142.1	62.9	207.8	113.0	19.7	57.0	52.4	49.2
92.4	124.4	26.5	28.9	104.3	85.5	158.1	166.1	17.3	115.3	213.2	31.9
33.8	50.3	67.4	28.7	67.8	36.4	83.5	192.1	35.7	85.7	118.3	101.6
153.5	98.3	117.9	107.2	56.3	116.3	73.7	75.8	193.0	32.1	98.2	70.5

85.7	44.5	60.7	71.7	86.7	73.2	102.7	109.1	109.3	74.1	96.7	22.2
53.6	112.3	131.4	102.9	135.7	172.8	178.8	99.6	131.1	121.9	71.1	80.3

1. Read the data into *MATLAB*.
2. Calculate the total annual ppt for each year and plot these data versus year. (Hint: use the *MATLAB* help facility to check on the **sum** command. Recall that you can perform operations on the transpose of a matrix.)
3. Calculate the mean monthly ppt for each month and plot the values using a bar chart. (Hint: check on the **mean** and **bar** commands.) Plot the monthly means on a regular plot along with "error bars" showing the standard deviations of the monthly means. (Hint: check the **errorbar** command and the **std** command.) How different are the means and medians? Hint: check the **median** command.)
4. Plot all monthly precipitation values consecutively. (You may want to examine the **reshape** command.) Is a seasonal pattern evident in the data?
5. Write an m-file program to: (a) query for a year (see the **input** command); (b) for the selected year, calculate and display the minimum, maximum, and mean monthy precipitation (see the **min** and **max** commands); (c) make a stem plot of the data (see the **stem** command), labelling the axes and placing an appropriate title (see the **xlabel**, **ylabel** and **title** commands).
6. Write a function file that accepts an N × M matrix (such as the precipitation file) as an argument, finds the maximum correlation coefficient (absolute value) between any two columns of the data, reports that value as output, and presents a scatter plot of the two columns of data with maximal correlation. Apply the code to the precipitation file and briefly comment on any interpretation of the result. (You may want to use **corrcoef** and **eye**.)
7. Use the symbolic math toolbox with Manning's equation to solve for channel width given the following information: S=0.036, n=0.02, Q=2 m^3s^{-1}, channel width=w (the "unknown"), and water depth=w/2.5.

1.7. References

Haith, D. A. and L. L. Shoemaker, Generalized watershed loading functions for stream flow nutrients. *Water Res. Bull., 23*: 471-478, 1987.

Hamon, W.R., Estimating Potential Evapotranspiration, *J. Hydraul. Div., ASCE, 87*: 107-120, 1961.

Karl, T.R., D.R. Easterling, and P.Ya.Groisman, United States historical climatology network - National and regional estimates of monthly and annual precipitation. pp. 830-905. *In:* T.A. Boden, D.P. Kaiser, R.J. Sepanski, and F.W. Stoss (eds.) *Trends '93: A Compendium of Data on Global Change*, ORNL/CDIAC-65. Carbon Dioxide Information Analysis Center, Oak Ridge National Laboratory, Oak Ridge, Tenn., USA, 1994.

Wilson, J. and P. Miller, Two-dimensional plume in uniform groundwater flow, *J. Hydraul. Div., ASCE, 104*: 503-514, 1978.

Box 1.1. Controlling attributes of a plot

Properties of graphs can be changed in *MATLAB* using the editor in the Figure window itself. Clicking the "edit plot" icon invokes the editing mode from which various Figure properties can be changed. Alternatively, plots can be edited from the command line by setting properties. Many properties can be set within the basic graphics commands. For example, the commands listed below produce larger symbol sizes and larger fonts relative to the defaults.

```
plot(t,temp,'+','MarkerSize',8);
xlabel('Time, days','FontName','helvetica','FontSize',14);
ylabel('Temperature, degrees C','FontName','helvetica','FontSize',14)
```

Use the "Help Desk" facility in *MATLAB* to learn more about how to edit Figure properties.

Box 1.2. Estimating potential evapotranspiration (PET) using temperature data

One of the simplest estimates of potential evaporation is presented by Hamon (1961). Following Haith and Shoemaker (1987), Hamon's estimate of potential evaporation is:

$$E_t = \frac{2.1 H_t^2 e_s}{T_t + 273.3}$$

E_t=evaporation on day t [mm day^{-1}]

H_t = average number of daylight hours per day during the month in which day t falls

e_s = saturated vapor pressure at temperature T [kPa]

T_t= temperature on day t [° C]

e_s can be calculated from temperature: $e_s = 0.2749x10^8 \exp\left[\frac{-4278.6}{T_t + 242.8}\right]$. H_t can be calculated by using the maximum number of daylight hours on day t, N_t, which is equal to $\frac{24 w_s}{p}$, where w_s is the sunset hour angle of day t, $w_s = \arccos(-\tan f \tan d)$, where j is the latitude and d is the solar declination given by $d = 0.4093 \sin(\frac{2p}{365} J - 1.405)$. On days when $T_t \leq 0$, Haith and Shoemaker set E=0.

CHAPTER 2

Solving Nonlinear Equations

Numerical Methods in the Hydrological Sciences: AGU Lecture Notes 1
Copyright 2004 by the American Geophysical Union
doi: 10.1029/001LN04
(Version 2004-08-02)

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

$$x = \left(y - b\right) / m$$

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.

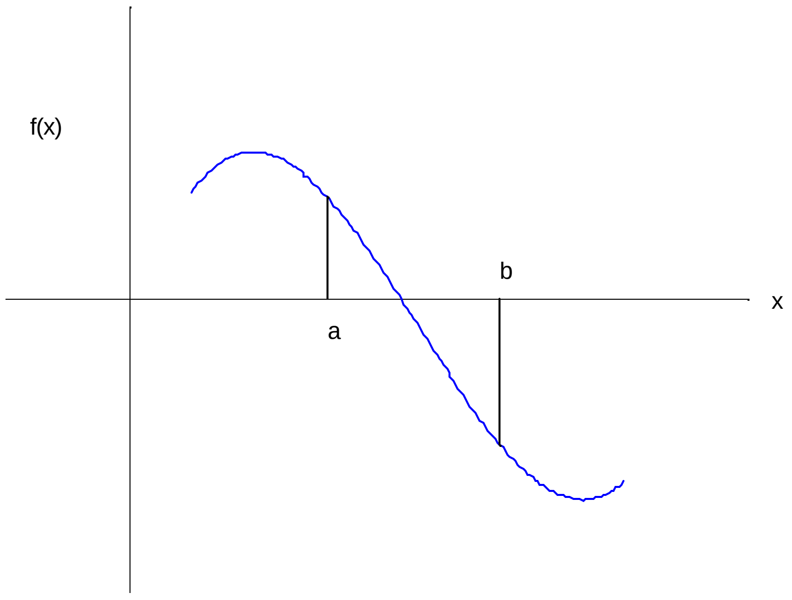


Figure 2.1. Values of f at $x=a$ and $x=b$ bracket a root of the function.

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 $\tan q_0 = f'(x_0)$. From trigonometry, we also know that

$$\tan q_0 = f(x_0)/(x_0 - x_1).$$

Combining these, we get

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

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

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

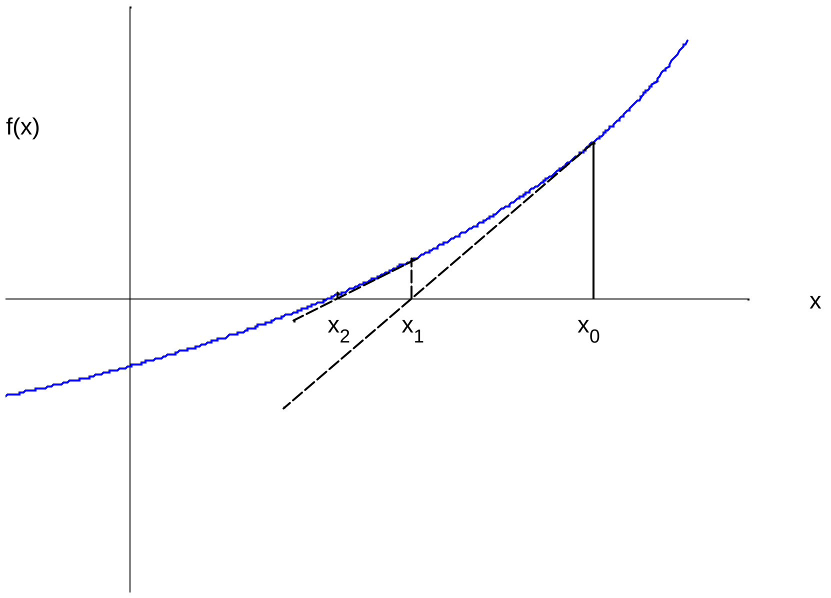


Figure 2.2. Schematic of Newton's method.

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.

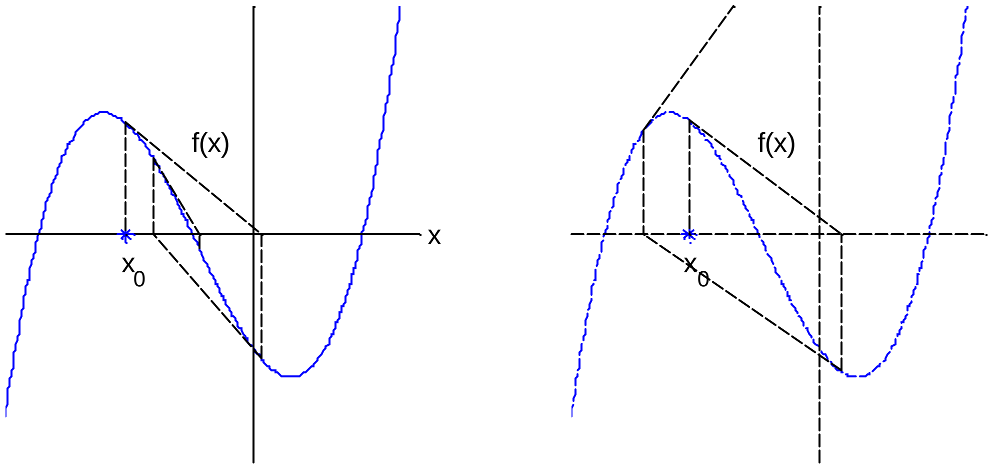


Figure 2.3. Newton's method may (left) or may not (right) converge to a root depending on the shape of the function and the proximity of the starting value to the root.

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.

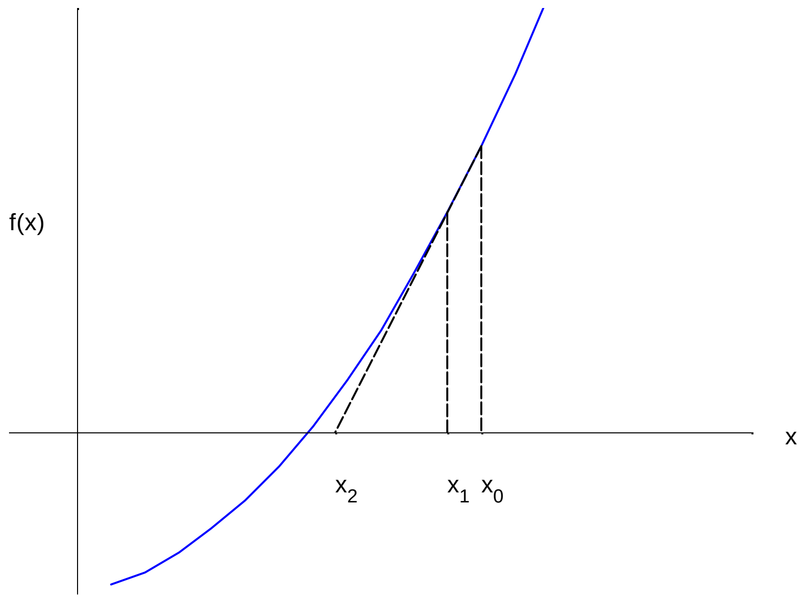


Figure 2.4. Schematic of the secant method.

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` $(=2.2\times10^{-16})$). 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 1×10^{-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: $x_1 = g(x_1)$, $x_2 = g(x_2)$, …, $x_{n+1} = g(x_{n+1})$. 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

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

or

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

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, x1
Do until |x1-x0| < specified tolerance,
      x0=x1
   x1=g(x0)
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 |x1-x| < specified tolerance,
   x1=x
   x=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*<20 is 'shallow'). The dependency of wavelength on depth and period is expressed through the nonlinear equation

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

where $k = 2p/L$, $k_d = 2p/L_d$, and $L_d = gT^2/(2p)$. 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 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.

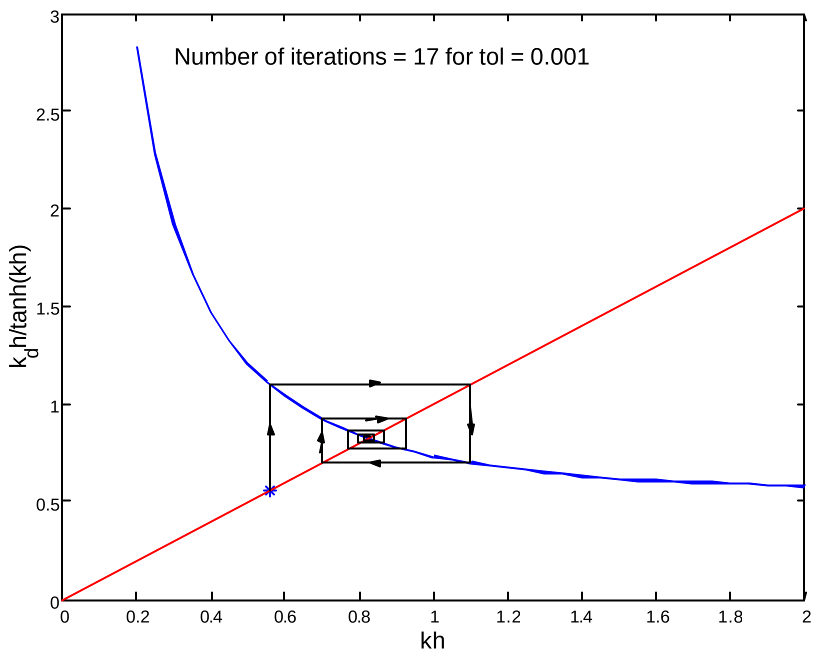


Figure 2.5. Graphical illustration of an iterative solution to $kh = k_d h \tanh(kh)$.

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

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

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

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

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\b
```

The backslash operator "\" solves the system of equations using Gaussian elimination, without calculating the matrix inverse, and is preferable to "`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, $A \mid b$. 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

$$\begin{aligned} 13x_1 - 8x_2 - 3x_3 &= 20 \\ -8x_1 + 10x_2 - x_3 &= -5 \\ -3x_1 - x_2 + 11x_3 &= 0 \end{aligned}$$

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

$$\begin{matrix} 1 & -0.61 & -0.23 & 1.54 \\ 0 & 5.08 & -2.84 & 7.31 \\ 0 & -2.84 & 10.3 & 4.62 \end{matrix}$$

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

$$\begin{matrix} 1 & -0.61 & -0.23 & 1.54 \\ 0 & 1 & -0.56 & 1.43 \\ 0 & 0 & 8.71 & 8.71 \end{matrix}$$

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

$$\begin{aligned} x_1 - 0.61x_2 - 0.23x_3 &= 1.54 \\ x_2 - 0.56x_3 &= 1.43 \\ x_3 &= 1 \end{aligned}$$

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\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^3, find the roots of the equation for h using the following methods:

   a) Plot the function and roughly estimate the roots (they are all real).
   b) Use the *MATLAB* command `roots` to obtain the roots.
   c) 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].

   a) Write an m-file to iteratively solve for h given Q = 2.0 m^{3}s^{-1}, n = 0.03, S = 0.0005, and w = 5.0 m. For a reasonable first estimate, assume $w >> h$, so that $wh/(w+2h) \approx h$, in which case $h = \left(Qn/(w\sqrt{S})\right)^{3/5}$.
   b) 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*, 6th 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, $\mathbf{F} = U/\sqrt{gh}$, is subcritical ($\mathbf{F}<1$) or supercritical ($\mathbf{F}>1$). (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>>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

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

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 4*a* 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.)

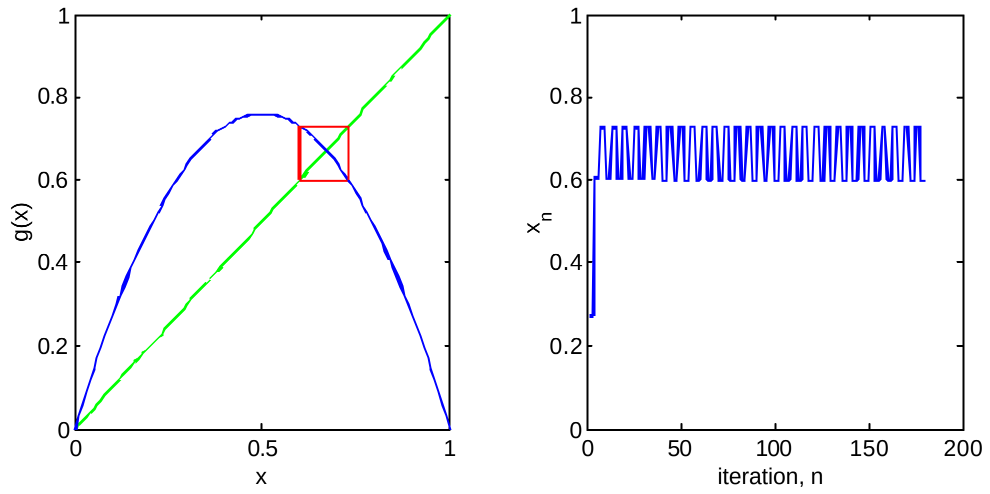


Figure B2.3.1. Results of function `hump.m` with *a*=0.76 illustrating cyclic behavior.

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

CHAPTER 3

Numerical Differentiation and Integration

Numerical Methods in the Hydrological Sciences: AGU Lecture Notes 1
Copyright 2004 by the American Geophysical Union
doi: 10.1029/001LN05
(Version 2004-08-02)

3. Numerical Differentiation and Integration

There are many situations in the hydrological sciences in which the need to perform a numerical differentiation or integration arises. Examples include integration of functions that are difficult or impossible to solve analytically and differentiation or integration of data having an unknown functional form. Numerical differentiation is also central to the development of numerical techniques to solve differential equations.

3.1. Finite differences

The definition of a derivative is

$$f'(x) = \lim_{\Delta x \to 0} \frac{f(x+\Delta x) - f(x)}{\Delta x}$$

In numerical differentiation, instead of taking the limit as Δx approaches zero, Δx is allowed to have some small but finite value. The simplest form of a finite difference approximation of a derivative follows from the definition above:

$$f'(x) \cong \frac{f(x+\Delta x) - f(x)}{\Delta x}$$

Thought of geometrically, this estimate of a derivative is the slope of a linear approximation to the function f over the interval Δx (Figure 3-1). How well a linear approximation works depends on the shape of the function f and the size of the interval Δx.

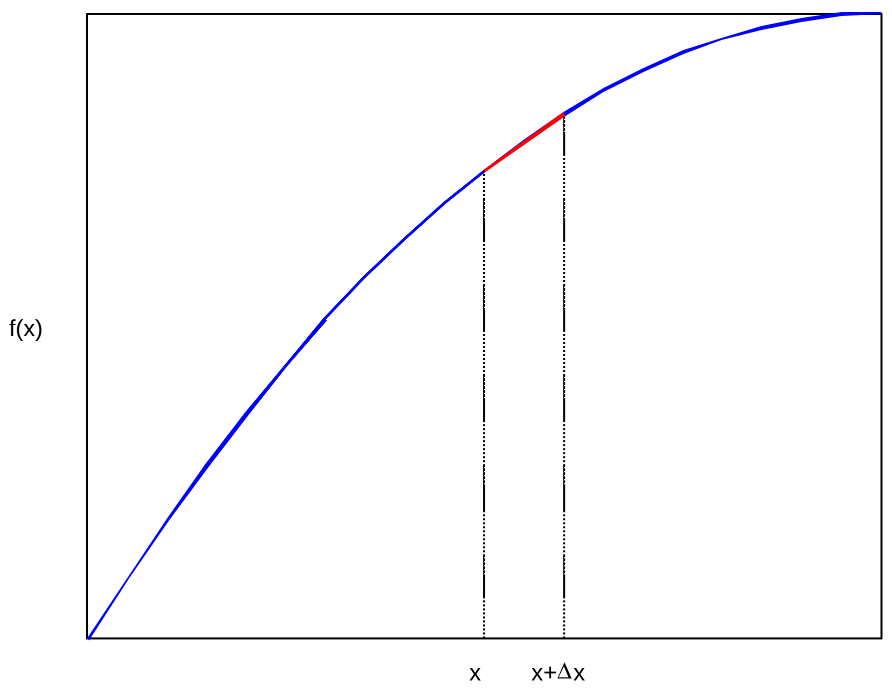


Figure 3.1. Finite difference approximation to a first derivative.

3.2. Taylor series approach

It is important that we have a way of determining the error associated with a finite difference approximation to a derivative. One straightforward way of determining the error is to use a Taylor series expansion to express the value of a function $f(x+\Delta x)$ in terms of the function and its derivatives at a nearby point x:

$$f(x+\Delta x) = f(x) + f'(x)\Delta x + \frac{f''(x)}{2}(\Delta x)^2 + \ldots + \frac{f^{(n)}(x)}{n!}(\Delta x)^n + R_{n+1} \qquad (3.1)$$

where the remainder, R_{n+1} is equal to

$$R_{n+1} = \frac{f^{(n+1)}(\chi)}{(n+1)!}(\Delta x)^{n+1} \qquad \text{for} \quad x < \chi < x+\Delta x$$

The error produced by truncating the Taylor series after the $f^{(n)}$-th term is given by R_{n+1} and is said to be of order $(\Delta x)^{n+1}$ or $O((\Delta x)^{n+1})$. Although in general we don't know the value of $f^{(n+1)}(\chi)$, it is evident that the smaller Δx, the smaller the error [but see Box 3.1 for additional information].

If we truncate the Taylor series after the first-derivative term,

$$f(x \pm \Delta x) = f(x) \pm f'(x)\Delta x + O(\Delta x)^2$$

and rearrange the expression to give an equation for $f'(x)$, we obtain

$$f'(x) = [f(x+\Delta x) - f(x)]/\Delta x + O(\Delta x) \qquad (3.2)$$

or

$$f'(x) = [f(x) - f(x+\Delta x)]/\Delta x + O(\Delta x) \qquad (3.3)$$

The first of these is called a *forward difference* and the second, a *backward difference* (Figure 3.2). Both expressions have a truncation error of $O(\Delta x)$.

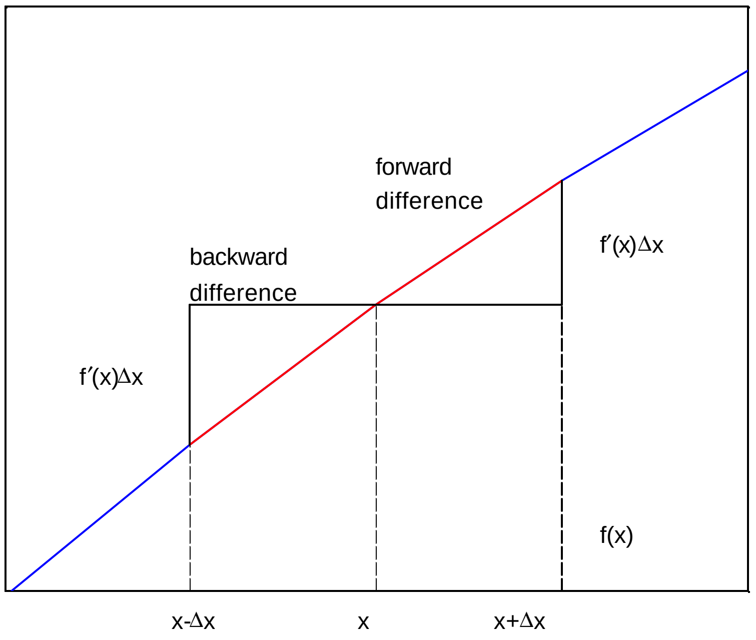


Figure 3.2. Forward and backward difference approximations to a first derivative.

An expression for $f'(x)$ with a smaller error (i.e., a higher order approximation) can be obtained using the first three terms of Taylor series (up to f'' in Eq. 3.1) for $f(x+\Delta x)$ and $f(x-\Delta x)$ and subtracting to cancel the f'' terms.

$$f(x+\Delta x) = f(x) + \Delta x f'(x) + \frac{(\Delta x)^2}{2} f''(x) + O((\Delta x)^3)$$
$$f(x-\Delta x) = f(x) - \Delta x f'(x) + \frac{(\Delta x)^2}{2} f''(x) + O((\Delta x)^3)$$

Subtracting these leaves,

$$f(x+\Delta x) - f(x-\Delta x) = 2\Delta x f'(x) + O((\Delta x)^3)$$

Rearranging to give an expression for f' gives

$$f'(x) = \frac{f(x+\Delta x) - f(x-\Delta x)}{(\Delta x)^2} + O((\Delta x)^2) \qquad (3.4)$$

This is referred to as a *central difference* approximation. The error in this case is of order $(\Delta x)^2$, compared to an error of $O(\Delta x)$ for forward or backward differencing. The higher order approximation will generally yield a more accurate estimate of the derivative [see Box 3.1]. It can be applied everywhere except at the boundaries where a forward or backward estimate is necessary.

An $O((\Delta x)^2)$ approximation to $f''(x)$, the second derivative of $f(x)$, can be obtained by adding the $O((\Delta x)^3)$ expressions for $f(x+\Delta x)$ and $f(x-\Delta x)$, producing

$$f''(x) = \frac{f(x-\Delta x) - 2f(x) + f(x+\Delta x)}{(\Delta x)^2} + O((\Delta x)^2) \tag{3.5}$$

3.3. Differentiation using interpolating polynomials

Another way to derive expressions for numerical differentiation is to use interpolating polynomials. The resulting $O(\Delta x)$ and $O((\Delta x)^2)$ approximations to f' are the same, but the approach provides some insight into these approximations and into how to generate expressions for higher order derivatives.

An n-th degree polynomial $P_n(x)$ can be fit through any n+1 points in such a way that the polynomial is equal to the known function values $f(x)$ at the n+1 points $x_1, x_2, \ldots x_{n+1}$. The derivative of this polynomial then provides an approximation to the derivative of the function $f(x)$. Consider the 2nd-order polynomial,

$$P_2(x) = a_i + (x - x_i)a_{i+1} + (x - x_i)(x - x_{i+1})a_{i+2}$$

We assume that the values of $f(x)$ are known at 3 values of x. Our aim is to choose the values a_i so that $P_2(x) = f(x)$ at each x_i. The resulting polynomial $P_2(x)$ will provide a smooth interpolation between the known values of $f(x)$.

To find the coefficients of $P_2(x)$, we evaluate the function at $x = x_i$, $x = x_{i+1}$, and $x = x_{i+2}$. For $x = x_i$, this just gives $P_2(x_i) = a_i = f_i$ (all other terms are 0). For $x = x_{i+1}$, $P_2(x_{i+1}) = a_i + (x_{i+1}-x_i)\, a_{i+1} = f_{i+1}$. It is not hard to determine that this is satisfied if $a_{i+1} = (f_{i+1} - f_i)/(x_{i+1}-x_i)$. This ratio is referred to as the first divided difference $f[x_i, x_{i+1}]$. By extension, $a_{i+2} = f[x_i, x_{i+1}, x_{i+2}]$, the second divided difference, which is given by

$$f[x_i, x_{i+1}, x_{i+2}] = \frac{f[x_{i+1}, x_{i+2}] - f[x_i, x_{i+1}]}{x_{i+2} - x_i} = \frac{1}{x_{i+2} - x_i}\left(\frac{f_{i+2} - f_{i+1}}{x_{i+2} - x_{i+1}} - \frac{f_{i+1} - f_i}{x_{i+1} - x_i}\right)$$

If the points are evenly spaced, the expression for $P_2(x)$ can be simplified. In that case, we can let $\Delta x = x_{i+1}-x_i$ and denote $f_{i+1} - f_i$ as Δf_i. With these modifications, the expression for $P_2(x)$ becomes

$$P_2(x) = f_i + \frac{(x - x_i)}{\Delta x}\Delta f_i + \frac{(x - x_i)(x - x_{i+1})}{2(\Delta x)^2}\Delta^2 f_i$$

where $\Delta^2 f_i = \Delta(\Delta f_i) = \Delta(f_{i+1} - f_i) = \Delta f_{i+1} - \Delta f_i = f_{i+2} - 2f_{i+1} + f_i$. Letting $s=(x-x_i)/\Delta x$, the equation takes a still simpler form,

$$P_2(x) = f_i + s\Delta f_i + \frac{s(s-1)}{2}\Delta^2 f_i$$

If $P_n(x)$ is a good approximation to $f(x)$ over some range of x, then $P_n'(x)$ should approximate $f¢(x)$ in that range. Thus we can write, $f'(x) = P_n'(x) + error$. For example, computing the derivative of $P_2(x)$, assuming evenly spaced points and noting that $df/dx = df/ds\,(ds/dx) = (1/\Delta x)\, df/ds$, gives

$$f'(x) = \frac{1}{\Delta x}\left(\Delta f_i + (2s-1)\frac{\Delta^2 f_i}{2}\right) + \mathit{error\ of}\ O((\Delta x)^2)$$

If we restrict ourselves to evaluating the derivative at one of the given points, say $x=x_{i+1}$, then s=l and the derivative reduces to

$$f'(x_{i+1}) = \frac{1}{\Delta x}\left(\Delta f_i + \frac{\Delta^2 f_i}{2}\right) + O((\Delta x)^2)$$
$$= \frac{1}{\Delta x}\left[(f_{i+1} - f_i) + \frac{f_{i+2} - 2f_{i+1} + f_i}{2}\right] + O((\Delta x)^2)$$
$$= \frac{f_{i+2} - f_i}{2\Delta x} + O((\Delta x)^2)$$

This is the same result we obtained using the Taylor series.

The table below summarizes some of the common formulas for computing numerical derivatives.

Table 3.1. Formulas for numerical differentiation

First derivatives	$f'(x_0) = \frac{f_1 - f_0}{\Delta x} + O(\Delta x)$	1st order, forward* difference
	$f'(x_0) = \frac{f_1 - f_{-1}}{2\Delta x} + O((\Delta x)^2)$	2nd order, central difference
	$f'(x_0) = \frac{-f_2 + 4f_1 - 3f_0}{2\Delta x} + O((\Delta x)^2)$	2nd order, forward* difference
Second derivatives	$f''(x_0) = \frac{f_1 - 2f_0 + f_{-1}}{(\Delta x)^2} + O((\Delta x)^2)$	2nd order, central difference
	$f''(x_0) = \frac{-f_3 + 4f_2 - 5f_1 + 2f_0}{(\Delta x)^2} + O((\Delta x)^2)$	2nd order, forward§ difference
Third derivatives	$f'''(x_0) = \frac{f_2 - 2f_1 + 2f_{-1} - f_{-2}}{2(\Delta x)^3} + O((\Delta x)^2)$	2nd order, central difference
	$f'''(x_0) = \frac{-3f_4 + 14f_3 - 24f_2 + 12f_1 - 5f_0}{(\Delta x)^3} + O((\Delta x)^2)$	2nd order, forward* difference

* To obtain the backward difference approximation for odd-order derivatives, multiply the forward difference equation by –1 and make all non-zero subscripts negative; e.g., the 1st-order backward difference approximation to the first derivative is $f'(x_0) = (f_0 - f_{-1})/\Delta x$

§ To obtain the backward difference approximation for even-order derivatives, make all non-zero subscripts negative.

3.4. *MATLAB* methods for finding derivatives

MATLAB has no built-in derivative functions except `diff`, which differences a vector; `diff(f)` is equivalent to Δf. A simple forward-difference estimate of the derivative is given by `diff(f)./diff(x)`, where f are the function values at the n points x. Note that if x has length n, `diff` returns a vector of length n-1. `diff` can be nested, so that $\Delta^2 f$ = `diff(diff(f))`. A centered difference estimate of a derivative can be obtained using:

```
x=x(:)'; %makes x a row vector to begin
xd=[x-h;x+h];
yd=f(xd); %define a function f or substitute the function for f
dfdx=diff(yd)./diff(xd);
```

MATLAB's symbolic math toolbox offers another option with `diff`. If we set `f='eqn'`, and `x` is the independent variable in `f`, then

```
dfdx=diff(f,'x')
```

returns the analytical expression for the derivative which can then be evaluated at any desired values of x.

For equally spaced points, the following expression provides an estimate for $f¢(x_i)$ where x_i are values of x where f is known (see Gerald and Wheatley, 1999, or other texts on numerical analysis for details):

$$f'(x_i) = \frac{1}{\Delta x}\left[\Delta f_i - \frac{\Delta^2 f_i}{2} + \frac{\Delta^3 f_i}{3} - \dots \frac{\Delta^n f_i}{n}\right]_{x=x_i}$$

This can be implemented in *MATLAB* to examine the improvement in the estimation of $f¢(x)$ with additional terms and/or smaller values of Δx. The following m-file `deriv.m` does this for the function f=sin(x) and $0<x<2\pi$. The results are compared in Figure 3.3.

```
%deriv.m -- difference formulae for derivatives

dx=0.1;
x=0:dx:2.*pi;
f=sin(x);
n=length(x);

%first difference
df1=diff(f)./dx;
mean(abs(cos(x(1:n-1))-df1))
plot(x(1:n-1),cos(x(1:n-1))-df1,'-b')
hold on;

%second difference
df2=(diff(f(1:n-1))-diff(diff(f))./2)./dx;
mean(abs(cos(x(1:n-2))-df2))
plot(x(1:n-2),cos(x(1:n-2))-df2,'--r')

%third difference
df3=(diff(f(1:n-2))-diff(diff(f(1:n-1)))./2+diff(diff(diff(f)))./3)./dx;
mean(abs(cos(x(1:n-3))-df3))
plot(x(1:n-3),cos(x(1:n-3))-df3,'-g')
```

```
%smaller dx
dx2=0.01; x2=0:dx2:2.*pi;
f2=sin(x2); n2=length(x2);
df1=diff(f2)./dx2;  %first difference
plot(x2(1:n2-1),cos(x2(1:n2-1))-df1,'--c')
mean(abs(cos(x2(1:n2-1))-df1))

hold off;
legend('1st diff','2nd diff','3rd diff','1st diff, 0.1*dx')
xlabel ('x'); ylabel ('error in derivative estimate')
```

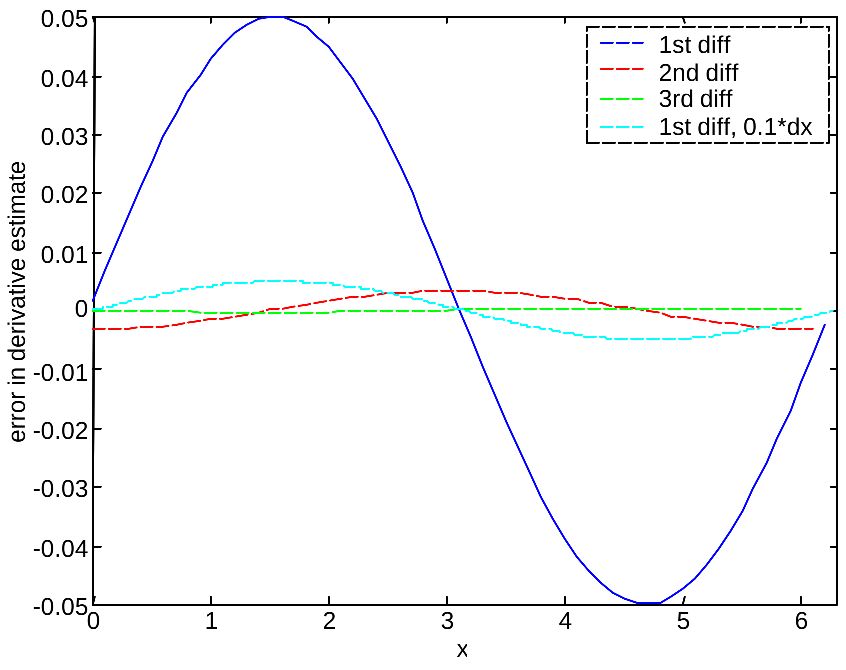


Figure 3.3. Accuracy of finite difference estimates of the derivative of sin(*x*).

3.5. Numerical integration

We will consider three methods of numerical integration: the trapezoidal rule, Simpson's rule(s), and Gaussian quadrature. *MATLAB* has functions `trapz`, `quad`, and `quadl` that perform numerical integration. In addition, the symbolic math function `int` finds integrals.

3.6. Trapezoidal rule

The simplest approach to numerically integrating a function *f* over the interval [*a*,*b*] is to divide the interval into *n* subdivisions of equal width, $\Delta x=(b-a)/n$ and approximate *f* in each interval either by the value of *f* at the midpoint of the interval, which gives the rectangular rule, or by the average of the function over the interval, $\frac{1}{2}(f_{i+1} + f_i)$, which gives the *trapezoidal rule*. The rectangular rule approximates the function over each subinterval as a constant, while the trapezoidal rule makes a linear approximation to the function (Figure 3.4). Over each subinterval, the trapezoidal rule gives

$$\int_{x_i}^{x_{i+1}} f(x)dx \cong \frac{f(x_i)+f(x_{i+1})}{2}\Delta x = \frac{\Delta x}{2}(f_i + f_{i+1})$$

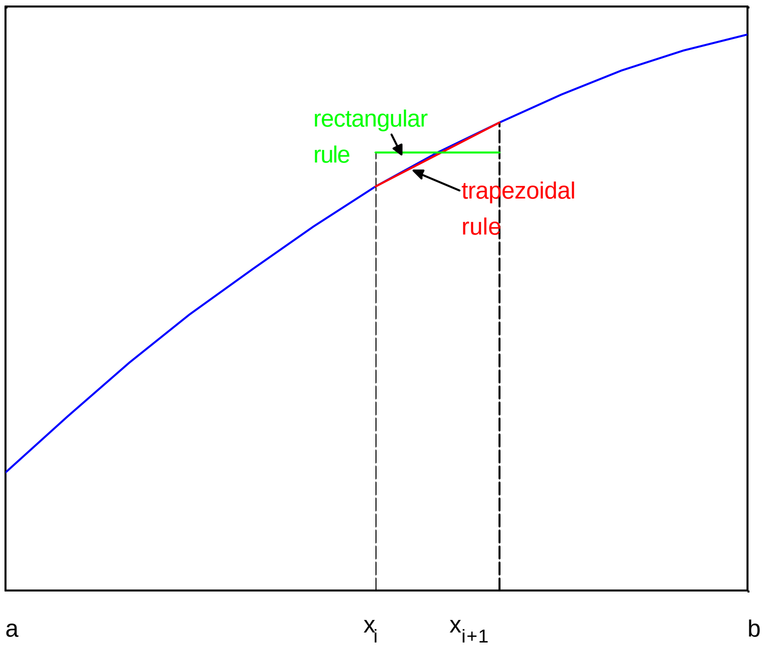


Figure 3.4. Trapezoidal and rectangular approximations of an integral.

If the interval $[a,b]$ is subdivided into n subintervals of size Δx, then, over the whole interval, the trapezoidal rule gives

$$\int_a^b f(x)dx = \frac{\Delta x}{2}(f_a + 2f_2 + 2f_3 + \ldots + 2f_n + f_b)$$

This is called the composite trapezoidal rule. However, it is not necessary for the subintervals to be equally spaced when applying the trapezoidal rule. This and its simplicity make it a valuable technique.

As we would expect, the errors associated with the trapezoidal rule depend on the step size. We need to consider two errors in this case. The first is the *local error* for each step, which is $O((\Delta x)^3)$. Generally, the trapezoidal rule is applied over an interval comprising n equal steps. The total error, or *global error* is given by the sum of the local errors, and can be shown to be $O((\Delta x)^2)$ [Box 3.1].

The trapezoidal rule is simple, but uses only a linear approximation between successive points to estimate the integral. We could expect better accuracy if we instead approximated the function over two adjacent subintervals of equal width using a quadratic. This can be done with interpolating polynomials.

First, consider the interpolating polynomial $P_1(x) = f_i + s\Delta f_i$. If we integrate $P_1(x)$ between x_0 and x_1, noting that $dx=\Delta x\ ds$, we get

$$\int_{x_0}^{x_1} f(x)dx \cong \Delta x \int_{s=0}^{s=1} (f_0 + s\Delta f_0)ds = \Delta x f_0 s\,|_0^1 + \Delta x \Delta f_0 \frac{s^2}{2}\,|_0^1 = \Delta x(f_0 + \frac{1}{2}\Delta f_0)$$
$$= \frac{\Delta x}{2}[2f_0 + (f_1 - f_0)] = \frac{\Delta x}{2}(f_0 + f_1)$$

This is the trapezoidal rule. The same approach can be used to develop higher order methods such as Simpson's rules.

3.7. Simpson's rules

Following the same procedure as above with the polynomial $P_2(x)$ gives us

$$\int_{x_0}^{x_2} f(x)dx \cong \Delta x \int_0^2 (f_0 + s\Delta f_0 + \frac{s(s-1)}{2}\Delta^2 f_0)ds = \Delta x(2f_0 + 2\Delta f_0 + \frac{1}{3}\Delta^2 f_0)$$
$$= \frac{\Delta x}{3}(f_0 + 4f_1 + f_2)$$

The local error term is $O((\Delta x)^5)$. If we did the same thing using a cubic approximation over three adjacent subintervals, we would obtain

$$\int_{x_0}^{x_3} f(x)dx = \int_{x_0}^{x_3} P_3(x)dx = \frac{3\Delta x}{8}(f_0 + 3f_1 + 3f_2 + f_3)$$

with a local error $O((\Delta x)^5)$; this is Simpson's 3/8 rule. Notice that adding the extra point into the formula does not increase the order of accuracy of the approximation.

The first of these equations (based on a quadratic) is called Simpson's 1/3 rule. The corresponding composite formula for the integral over the interval $[a,b]$ is

$$\int_a^b f(x)dx = \frac{\Delta x}{3}(f_a + 4f_1 + 2f_2 + 4f_3 + 2f_4 + \ldots + 4f_{n-1} + f_b)$$

with a global error term of $O((\Delta x)^4)$. Because the method uses pairs of panels, the number of panels (subintervals) must be even. If the number of panels is uneven, another rule, e.g. Simpson's 3/8 rule, could be used at one end and the 1/3 rule over the remaining even number of panels. Alternatively, the size of the panels can be adjusted to accommodate an even number of panels.

The trapezoidal rule and Simpson's rules are examples of the general family of Newton-Cotes integration formulas. The general form is

$$\int_a^b f(x)dx \cong \int_a^b P_n(x)dx$$

with an error of $O((\Delta x)^{n+1})$. For interpolating polynomials of order 1, 2, and 3, the Newton-Cotes formulas give the trapezoidal rule, Simpson's 1/3 rule, and Simpson's 3/8 rule, respectively. If the degree of the interpolating polynomial is too high, errors due to round-off and local irregularities can cause a problem [Box 3.1]. This is why usually only the lower-degree Newton-Cotes formulas are used.

3.8. Gaussian quadrature

Another common method for numerical integration is Gaussian quadrature. Although this method is most easily derived using the method of undetermined coefficients (e.g., see Gerald and Wheatley, 1999), we can gain an appreciation for its origin by recognizing that the methods we've described so far all have the form

$$\int_a^b f(x) \cong \sum_{i=1}^{n} w_i \; f(x_i)$$

where w_i are weights assigned to values of $f(x_i)$ in the integration formulas. The Newton-Cotes methods are based on evenly spaced values of x. However, we could let the x's be free parameters in our attempt to fit a polynomial through the function. This requires that $f(x)$ is known explicitly so it can be evaluated at any desired value of x. Using this approach, we can improve the accuracy of, e.g., a two-point method over that attainable with the trapezoidal rule. With two points, x_1 and x_2, and two weights, a and b, we can fit exactly polynomials of order 0, 1, 2, and 3. To simplify the calculations, we will evaluate the integrals over the interval [-1 1]. A transformation can be used to change these limits to those for any other bounded region.

$$\int_{-1}^{1} f(x)dx = w_1 f(x_1) + w_2 \; f(x_2)$$

We require this formula to be exact for polynomials of order three or less, including $f(x)=x^3$, $f(x)=x^2$, $f(x)=x$, and $f(x)=1$. Substituting these into the equation above gives,

$$\int_{-1}^{1} x^3 dt = 0 = w_1 x_1{}^3 + w_2 x_2{}^3$$

$$\int_{-1}^{1} x^2 dx = 2/3 = w_1 x_1{}^2 + w_2 x_2{}^2$$

$$\int_{-1}^{1} x dx = 0 = w_1 x_1 + w_2 x_2$$

$$\int_{-1}^{1} dx = 2 = w_1 + w_2$$

We now have four equations for the four unknown parameters, w_1, w_2, x_1, and x_2. Solving these gives $w_1 = w_2 = 1$, and $x_2 = -x_1 = \sqrt{1/3} = 0.5773$. Substituting these values back into $\int f = w_1 x_1 + w_2 x_2$ results in

$$\int_{-1}^{1} f(x)dx \cong f\left(\frac{-1}{\sqrt{3}}\right) + f\left(\frac{1}{\sqrt{3}}\right)$$

This sum gives the exact integral of any cubic over the interval from -1 to 1. If the limits are [a,b] rather than [-1,1], then it is necessary to use the linear transformation t = [(b-a)x + b+a]/2, dt = [(b-a)/2]dx, which gives

$$\int_a^b f(t)dt = \frac{(b-a)}{2}\int_{-1}^{1} f\left(\frac{(b-a)\,x + b + a}{2}\right)dx$$

Gaussian quadrature can be extended to include more than two points. The general expression has the form

$$\int_{-1}^{1} f(x)dx \cong \sum_{i=1}^{n} w_i f_i(x_i)$$

and is exact for functions that are polynomials of degree 2*n*-1 or less. A method for determining the weights w_i and the x_i's uses Legendre polynomials (see Gerald and Wheatley, 1999, or other texts on numerical analysis for details).

3.9. *MATLAB* methods

As noted previously, *MATLAB* has built-in functions to integrate using the trapezoidal rule and variants of Simpson's rule and another higher-order approximation.

trapz: `z=trapz(x,y)` or `z=trapz(y)` computes the integral of *y* with respect to *x* using trapezoidal integration. `trapz(y)` assumes unit spacing between data points. For other spacings, multiply *z* by the actual interval width. `trapz(x,y)` can be used for unequally spaced grids.

quad: `a=quad('fname',a,b,[tol],[trace])` approximates the integral of a function over the interval [*a*,*b*] using quadrature. The default tolerance is 1E-3. The function `fname` must return a vector of output values when given a vector of input values. `quad` uses a "recursive adaptive, Simpson's rule"; `quadl` uses high order recursive, adaptive Lobatto quadrature. Neither can integrate over singularities (essentially, places where the function is undefined).

3.10. Problems

1. Use forward, backward, and central difference approximations to numerically differentiate $\sqrt{x}$ over the range $0<x<2$. Compare the answers to the exact solutions at x=0.2, 0.5, 2 for Δx=0.05 and 0.02. How are the errors affected by the method and step size?

2. The Gaussian error function *erf*(x) is

$$erf(x) = \frac{2}{\sqrt{p}} \int_0^x e^{-t^2} dt$$

   a) Write an m-file implementing Simpson's 1/3 rule.

   b) Use the m-file to evaluate *erf*(x) between 0 and 5 (inclusive) with $\Delta x = 0.1$.

   c) How small does Δx have to be for the answer to be correct to 4 decimal places (error<0.00005) at $x = 1$? Note that *MATLAB* has a built-in function `erf` that you can use to check the accuracy of your answer. [*MATLAB*'s `erf` function also uses a numerical solution, but it is accurate to 1E-16.]

   d) Compare your answer to the results of `trapz` for the same Δx and `quad` or `quadl` for the same level of error.

3. The velocity distribution at the centerline of a steady, uniform channel flow is approximately given by the equation,

$$u(z) = \frac{u_*}{k}\left(\ln(z) - \ln(z_0)\right)$$

   where the shear velocity $u_* = \sqrt{ghS}$, S is channel slope, $k = 0.41$, z is level above the bottom, h is flow depth, and z_0 is the level close to the bed where the velocity u = 0. Use this velocity equation to generate values of velocity at 0.1h intervals for a flow with h = 0.7 m, $S = 2.5\times10^{-4}$, $z_0 = 1\times10^{-4}$ m. We will consider these to be our "data".

   a) Depth-averaged velocity $<u>$ is defined as $h^{-1}\int_0^h u(z)dz$. For the logarithmic velocity profile indicated above, the exact integral $<u> = u_* k^{-1}[\ln(0.367h) - \ln(z_0)]$. For wide, rectangular channels, this is a good approximation to the mean velocity in the channel. Numerically integrate the velocity "data" generated above to estimate the depth-averaged velocity using `trapz.` Compare the results to the exact integral.

   b) The exact derivative of the velocity profile is $u_*/(kz)$. Differentiate the profile "data" using centered differences and compare to the exact value.

   c) Add some noise to the data to represent measurement/instrument error. This can be done using the *MATLAB* command `un=u+a*randn(size(z))` where `a` is the amplitude of the noise and `z` is the vector of vertical positions; let `a` = 0.1. Now, repeat the integration and differentiation of parts a) and b) to see how much the addition of this noise affects the results.

d) The most efficient way to get a good numerical estimate of $\int u\,dz$ for a logarithmic velocity profile is to perform the integration on a logarithmically spaced grid for which the grid spacing close to the bottom is smaller than that near the surface. Compare the results of integrating the velocity profile (using `trapz`) over 20 equally spaced points and 20 logarithmically spaced points to the exact solution.

3.11. References

Gerald C.F. and P.O. Wheatley, *Applied Numerical Analysis*, 319 pp., Addison Wesley, Reading, MA, 1999.

Box 3.1. Errors in numerical methods

There are two principal sources of error in numerical computation. One is due to the fact that an *approximation* is being made (e.g., a derivative is being approximated by a finite difference). These errors are called *truncation errors*. The second is due to the fact that computers have limited precision, i.e., they can store only a finite number of decimal places. These errors are called *roundoff errors*.

Truncation errors arise when a function is approximated using a finite number of terms in an infinite series. For example, truncated Taylor series are the basis of finite difference approximations to derivatives (Chapter 3.2). The error in a finite difference approximation to a derivative is a direct result of the number of terms retained in the Taylor series (i.e., where the series is truncated). Truncation error is also present in other numerical approximations. In numerical integration, for example, when each increment of area under a curve is calculated using a polynomial approximation to the true function (Chapters 3.6-3.7), truncation errors arise that are related to the order of the approximating polynomial. For example an n^{th}-order polynomial approximation to a function results in an error in the integral over an increment Δx of $O(\Delta x)^{n+2}$ (*local* error). When the integrals over each increment are summed to approximate the integral over some domain $a \le x \le b$, the local errors sum to give a *global* error of $O(\Delta x)^{n+1}$. Truncation errors decrease as step size (Δx) is decreased – the finite difference approximation to a derivative is better (has lower truncation error) when Δx is "small" relative to when Δx is "large."

Roundoff errors stem from the fact that computers have a maximum number of digits that can be used to express a number. This means that the machine value given to fractional numbers without finite digit representations, for example, 1/3 = 0.33333333…, will be rounded or chopped at the precision of the computer. It also means that there is a limit to how large or small a number a computer can represent in floating point form. For many computations, the small changes in values resulting from roundoff are insignificant. However, roundoff errors can become important. For example, subtraction of two nearly identical numbers (as occurs when computing finite differences with very small values of Δx) can lead to relatively large roundoff error depending on the number of significant digits retained in the calculation. Interestingly, this means that approximations to derivatives will be improved by reducing Δx to some level because truncation errors are reduced, but that further decreases in Δx will make the estimate *worse* because roundoff error becomes large and dominates for very small values of Δx. Roundoff error can also complicate some logical operations that depend on establishing equality between two values if one or both are the result of computations that involved chopping or rounding.

Finally, in the hydrological sciences, measured data are often used in a calculation. For example, one might want to find the derivative of water velocity with respect to height above a streambed using numerical differentiation of data measured using a flowmeter. Such data are subject to *measurement errors*, which are then inserted into any numerical computation in which they are used.

CHAPTER 4

Numerical Methods for Solving First-Order Ordinary Differential Equations

Numerical Methods in the Hydrological Sciences: AGU Lecture Notes 1
Copyright 2004 by the American Geophysical Union
doi: 10.1029/001LN6
(Version 2004-08-02)

4. Numerical Methods for Solving First-Order Ordinary Differential Equations

4.1. Ordinary differential equations in hydrology

Ordinary differential equations (ODEs) arise naturally in the hydrological sciences in cases where we know something about rates of change of variables and about their relationship to other quantities. Several simple examples illustrate.

1. The Green-Ampt equation is used to describe the progress of a wetting front into an initially dry soil (e.g., see Hornberger et al., 1998). The depth of the wetting front L_f is the variable to be determined as a function of time, t. The differential equation involves several parameters – the saturated hydraulic conductivity, K_s, the difference between the saturated moisture content and the initial moisture content, Δq, and the capillary pressure head at the wetting-front, y_f. The equation describing progress of the infiltration front is:
$$\frac{dL_f}{dt} = \frac{K_s}{\Delta q}\left(\frac{-y_f + L_f}{L_f}\right)$$
2. The hydration of carbon dioxide in water, $CO_2\,(aq)+H_2O(l) \Rightarrow H_2CO_3\,(aq)$, occurs at a rate proportional to the concentration of CO_2 (e.g., see Lasaga and Kirkpatrick, 1981):
$\frac{dm_{CO_2}(aq)}{dt} = -km_{CO_2}(aq)$. The rate constant, k, is 2×10^{-3} s^{-1} at 0°C.

In many instances, a *system* of equations arises. We may be interested in the cycling of nutrients through different soil "pools" or in the progress of a geochemical reaction involving several interacting species, for example. In such cases, we expect to write a "rate-of-change" equation for each pool or chemical species. Because of interdependencies among the variables (e.g., uptake of nutrients by trees affects the concentration in soil as well as concentration in the tree), the equations are linked. This is what is meant by a system of equations.

Below we examine several methods for solving first-order ordinary differential equations (including systems of equations).

4.2. Euler and Taylor-series methods

A first-order differential equation has the form

$$\frac{dy}{dx} = f(x,y)\,.$$

Often what we want are values of y for any value of x given that $y=y_0$ at $x=0$. That is, we want a solution to the initial-value problem.

If we approximate the derivative dy/dx as a finite difference, $(y_{i+1}-y_i)/(x_{i+1}-x_i)$ and consider evaluation of the function at $[x_i, y_i]$, we obtain, after rearranging the equation

$$y_{i+1} = y_i + \Delta x\, f(x_i, y_i) \tag{4.1}$$

where $\Delta x = (x_{i+1} - x_i)$. Equation (4-1) is known as the Euler method for solving a first-order ODE numerically. Given an initial value for y ($y = y_0$ at $x = 0$) and a step size Δx, the equation is successively applied to find $y_1, y_2, y_3, \ldots, y_n$ at the corresponding values of $x_1, x_2, x_3, \ldots, x_n$.

Intuitively, we can guess that the accuracy of the results from the Euler method (and other numerical methods) depends on the size of the step, Δx. We can obtain insight about the error by deriving the Euler method using a Taylor series expansion of y,

$$y(x+\Delta x) = y(x) + y'(x)\Delta x + y''(x)\frac{(\Delta x)^2}{2!} + y'''\frac{(\Delta x)^3}{3!} + \cdots$$

or, using the notation in equation (4.1), we can write the series expansion about the point $x = x_i$ as,

$$y_{i+1} = y_i + y_i'\,\Delta x + y_i''\frac{(\Delta x)^2}{2} + \cdots \tag{4.2}$$

If the series is truncated after the second term, the Euler method is recovered. [Note that $y_i' = f(x_i, y_i)$.] The truncation, or local, error is $O(\Delta x^2)$, i.e., is given by $y_i''(\Delta x)^2/2$. The global error, the accumulated error over the whole range of the solution, is $O(\Delta x)$. This can be appreciated by noting that the Euler method for approximation of the derivative is obtained by rearranging equation (4.2):

$$\left.\frac{dy}{dx}\right|_{x=x_i} = y_i' = \frac{y_{i+1} - y_i}{\Delta x} + y_i''\frac{\Delta x}{2} + \cdots$$

Because we divide through by Δx, the error in the Euler method is $O(\Delta x)$. The step size for the Euler method generally must be very small to get good accuracy.

As an example, consider the equation

$$\frac{dy}{dx} = y - x \tag{4.3}$$

with initial condition $y_0 = 2$. The exact solution[1] for this problem is $y = e^x + x + 1$. A *MATLAB* code for the Euler solution for this problem is given below. For $\Delta x = 0.1$ the difference between the known exact solution and the Euler solution can be seen to be growing after only about ten steps (Figure 4.1).

[1] This solution can be obtained using the *MATLAB* symbolic toolbox: `y=Dsolve('Dy=y-x','y(0)=2','x')`.

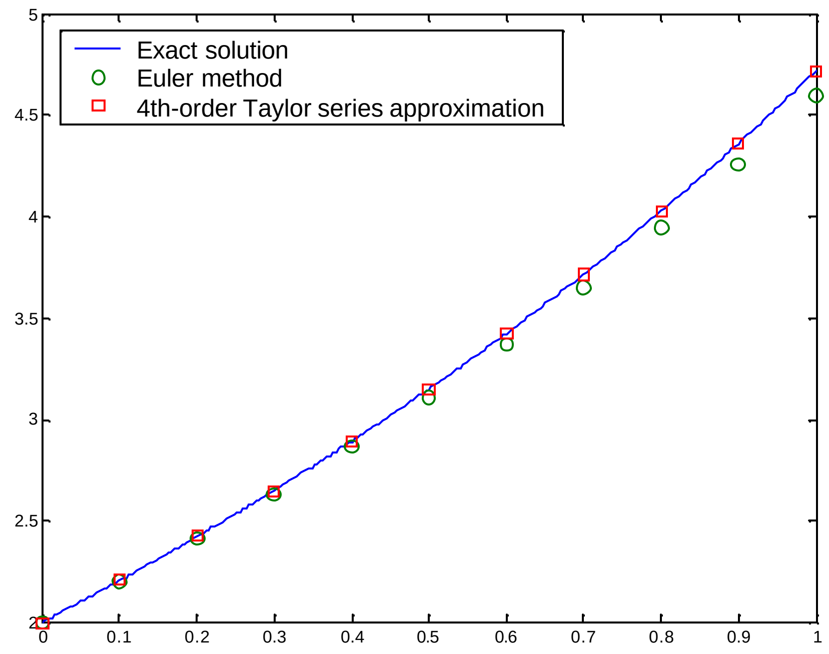


Figure 4.1. Example solution with the Euler and Taylor series methods for Δx=0.1.

```
% euler_ex.m
delta_x=input('delta_x= ')
x0=0;
xf=1;
x=(x0:delta_x:xf)'; %domain of solution
n=length(x);
y=zeros(size(x));
y(1)=2;   %initial condition
for i=1:n-1
   y(i+1)=y(i)+delta_x*yprime(x(i),y(i));
end
xx=x0:delta_x/25:xf;
exact=exp(xx)+xx+1;   %exact solution
plot(xx,exact,x,y,'o')
title('Euler solution to dy/dx=x-y')

function yp=yprime(x,y)
yp=y-x
```

By retaining more terms in the Taylor series, we can derive methods that have higher-order accuracy (i.e., methods with global errors of $O(\Delta x^2)$, $O(\Delta x^3)$ or as high as we wish) and for which larger values of Δx can be used and still retain reasonable accuracy. For example, consider a second-order Taylor-series method derived by retaining terms up to and including y'' in the expansion.

$$y_{i+1} = y_i + y_i' \Delta x + y_i'' \frac{(\Delta x)^2}{2!} + y_i''' \frac{(\Delta x)^3}{3!} + \cdots \tag{4.4}$$

Consider the example we used in the previous section, $y' = y - x$. The derivatives can be evaluated directly from the function:

$$y' = y - x; \quad y'' = y' - 1; \quad y''' = y'' = y' \text{-} 1.$$

Substituting for these derivatives in equation (4.4) gives

$$y_{i+1} = y_i + (y_i - x_i)\Delta x + (y_i - x_i - 1)\frac{(\Delta x)^2}{2!} + O\left((\Delta x)^3\right) \tag{4.5}$$

The initial condition is $y(x=0) = 2$. Let $\Delta x = 0.1$. Starting with $i = 0$ in equation (4.5), we get

$$y_1 = 2 + (2)0.1 + (2-1)).005 = 2.2050$$

Using this value of y_1 at $x_1 = 0.1$, we obtain $y_2 = 2.4210$. This stepping can be repeated to obtain an estimate of y at successive values of x_i. The *MATLAB* m-file `taylor_ex.m` listed below solves the example equation for several orders of Taylor approximations. The results from the program show how the accuracy of the solution increases as additional terms in the Taylor series approximation are retained.

```
%taylor_ex.m
%Calculates 4 solutions to y'=y-x with initial condition y(x=0)=2
%based on 2, 3, 4 and 5 terms of the Taylor series.
delta_x=input ('delta_x = ' )
x0=0;
xf=1.;
x= (x0:delta_x:xf)';
n=length (x);
y=zeros(size(x));
yl(1)=2; y2(1)=2; y3(1)=2; y4(1)=2;  %initial conditions
for i=1:n-1,
    ypl=yprime(x(i),yl(i));
    yl(i+1)=yl(i)+delta_x.*ypl;  %two terms – simple Euler method
    yp2=yprime(x(i),y2(i));
    y2(i+1)=y2(i)+delta_x.*yp2+(delta_x.^2)./2.*(yp2-1);  %three terms
    yp3=yprime(x(i),y3(i));
    y3(i+1)=y3(i)+delta_x*yp3+(delta_x.^2)./2.*(yp3-1)...
      +(delta_x.^3)./6.*(yp3-1);    %four terms
    yp4=yprime(x(i),y4(i));
    y4(i+1)=y4(i)+delta_x.*yp4+(delta_x.^2)./2.*(yp4-1)...
      +(delta_x.^3)./6.*(yp4-1)+(delta_x.^4)./24.*(yp4-1); %five terms
end;
exact=exp(x)+x+1;[x exact yl' y2' y3' y4' (y4'-exact).*1e4]
```

The 5-term ($O(\Delta x)^4$) solution is shown in Figure 4.1. The disadvantages of the higher-order Taylor-series methods for obtaining higher accuracy are that they are cumbersome and involve many computations (i.e., they are slow).

4.3. Modified Euler method or Euler predictor-corrector method

In the Euler method, we estimate each value of y based on a linear extrapolation from the last computed value. The more nearly linear the function or the smaller the step size, the better this estimate is likely to be. We could expect that our accuracy would be improved, however, if we could make a better estimate of the average slope of the function over each interval as we step through the solution. If we knew in advance the value of y' at x and $x+\Delta x$, we could use the mean of these slopes to improve our estimate:

$$y_{i+1} = y_i + \Delta x \frac{y'_i + y'_{i+1}}{2} \tag{4.6}$$

We don't know y_{i+1} or $y¢_{i+1}$ in advance, but once we compute y_{i+1} using the Euler method (Figure 4.2), we can use the computed value to estimate $y¢_{i+1}$. This value can then be used to get an improved, or "corrected" value of y_{i+1}. The difference between the two estimates of y_{i+1} provides an estimate of the accuracy of the approximation.

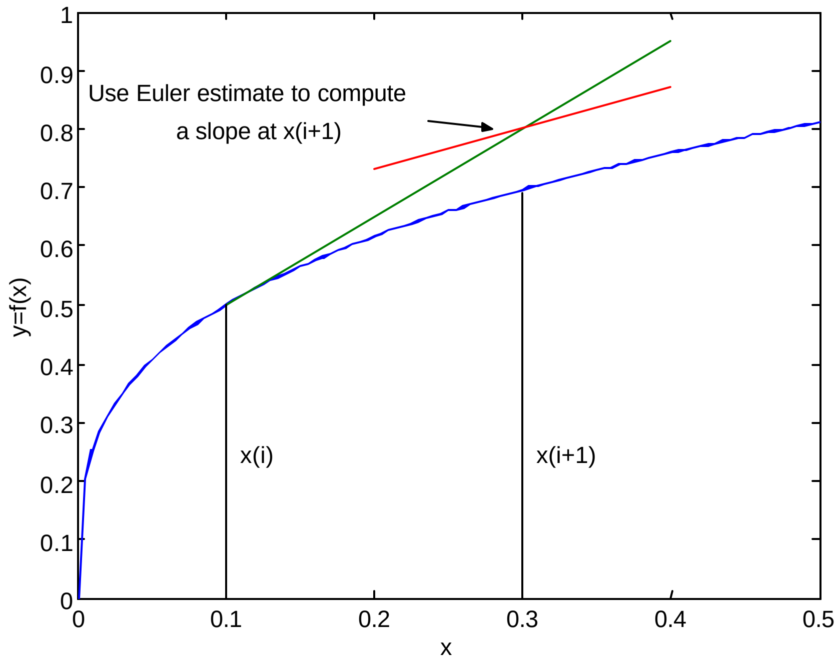


Figure 4.2. Schematic for modified Euler method.

The modified method then consists of two steps. First, the Euler method (equation 4.1) is used to *predict* (compute an estimate of) y_{i+1}. Next this predicted value is *corrected* (improved) by using equation (4.6). The local error for this method is $O(\Delta x)^3$, while the global error is $O(\Delta x)^2$ (e.g., see Gerald and Wheatley, 1999) . This is one order better than the simple Euler method.

4.4. Runge-Kutta methods

The idea of using information about the function $y(x)$ at points other than the initial point of an interval can be generalized to produce more efficient and more accurate schemes for solving ordinary differential equations. Probably the most widely used of these are the methods of Runge and Kutta.

The second-order Runge-Kutta method is a good illustration of the approach. The general method is represented by:

$$y_{i+1} = y_i + ak_1 + bk_2 \tag{4.7}$$

where

$$\begin{aligned} k_1 &= f(x_i, y_i)\Delta x \\ k_2 &= f(x_i + a\Delta x, y_i + b\,k_1)\Delta x \end{aligned}$$

and a, b, a and b are constants to be selected. If we set a=l and b=0, we get the simple Euler method. If a=b=1/2 and a=b=1, we recover the modified Euler method.

The second-order Runge-Kutta formula is obtained by setting the values of the four parameters a, b, a and b to make the expression for y_{i+1} agree with the Taylor series through the second-order in Δx. The Runge-Kutta methods are derived by rewriting k_2 in terms of the function f at $[x_i,y_i]$. This can be done using a Taylor series expansion of $f(x_i,y_i)$. The general form of a Taylor series for a function of two variables is

$$\begin{aligned} f(x+\Delta x, y+\Delta y) &= f(x,y) + f_x(x,y)\Delta x + f_y(x,y)\Delta y \\ &\quad + ½\left[f_{xx}(\Delta x)^2 + 2 f_{xy}(\Delta x \Delta y) + f_{yy}(\Delta y)^2 \right] + \ldots \end{aligned}$$

where the x and y subscripts stand for partial derivatives. This allows us to approximate k_2 as

$$k_2 \cong \Delta x\left[f(x_i, y_i) + (f_x a\,\Delta x + f_y b\, f\Delta x)_i \right]$$

with truncation error $O((\Delta x)^2)$. Substituting this expression for k_2 into equation (4.7) gives

$$\begin{aligned} y_{i+1} &= y_i + af\Delta x + b\,\Delta x\,(f + \Delta x a\, f_x + \Delta x b\, f_y f) \\ &= y_i + \Delta x f(a+b) + (\Delta x)^2\,(ba\, f_x + bb\, f_y f) \end{aligned} \tag{4.8}$$

where f and its derivatives are evaluated at x_i, y_i. We want to match the terms of equation (4.8) to the first 3 terms of the Taylor series for y_{i+1},

$$y_{i+1} = y_i + y_i'\Delta x + \frac{(\Delta x)^2}{2} y_i'' + \ldots = y_i + f\Delta x + \frac{(\Delta x)^2}{2}(f_x + f_y f) + \ldots \tag{4.9}$$

The substitution for y^2 after the second equal sign comes directly from $dy/dx = f(x,y)$ by taking the derivative of the right-hand side. For the terms of equations (4.8) and (4.9) to match, we must take $a+b$ = l, ab = 1/2, and bb = 1/2. These relationships do not uniquely specify the four parameters, but once we choose one, the other three are set. If we pick a = 1/2, then b = 1/2, b =

1 and a = l; as we noted above, this combination of values gives the modified Euler method.

The fourth-order Runge-Kutta method is a commonly used algorithm. The approach is the same as that for the second-order method, but now we define

$$
\begin{aligned}
y_{i+1} &= y_i + \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right) \\
k_1 &= \Delta x\left[f(x_i, y_i)\right] \\
k_2 &= \Delta x\left[f(x_i + \Delta x/2, y_i + k_1/2)\right] \\
k_3 &= \Delta x\left[f(x_i + \Delta x/2, y_i + k_2/2)\right] \\
k_4 &= \Delta x\left[f(x_i + \Delta x, y_i + k_3)\right]
\end{aligned}
$$

where the coefficients have been set to their commonly used values. Matching the equation for y_{i+1} to the first 5 terms of the Taylor series yields 11 equations for 13 unknown coefficients, of which two may be chosen arbitrarily. The local error term for the fourth-order Runge Kutta method is $O((\Delta x)^5)$ because we matched terms up to $O((\Delta x)^4)$ in setting the coefficients. The global error is $O((\Delta x)^4)$. The method is more efficient than the Euler method because the step size can be much larger even though there are more function evaluations per step.

4.5. Runge-Kutta-Fehlberg method

A goal of most numerical solutions to differential equations is to attain the desired accuracy with the fewest computations, which, for any given method means running it with the largest step size that satisfies the accuracy conditions. How can we determine what that step size is, particularly inasmuch as we generally don't know the solution in advance? One strategy would be to halve the step size and compare the answers with two step sizes. Because smaller step sizes generally lead to more accurate results, a significant difference between the two answers would suggest that the step size is too big. This process can be continued until an acceptably small difference between the two estimates is obtained. Although this approach is straightforward, it can be computationally intensive.

An approach that requires fewer function evaluations is to compare the results using two Runge-Kutta methods of different order, e.g. a second and third-order method or a fourth and fifth-order method. This is called the Runge-Kutta-Fehlberg method. The resulting values of y_{i+1} are compared, and Δx is adjusted until the difference between values computed using the two methods is as small as desired. By making use of the same *k's* in both methods, the number of function evaluations can be minimized - only six are required for the fourth- and fifth-order Runge-Kutta formulas. Forms for the *k's* and values of the coefficients can be found in texts on numerical analysis (e.g., see Gerald and Wheatley, 1999).

4.6. *MATLAB* methods

MATLAB has several functions to solve ordinary differential equations, including `ode23` and `ode45`. They are both implementations of the Runge-Kutta-Fehlberg method, using second-third-order and fourth-fifth-order formulas, respectively. The syntax of both commands is similar. From the *MATLAB* documentation[2]:

```
ODE45  Solve non-stiff differential equations, medium order method.
[T,Y] = ODE45(ODEFUN,TSPAN,Y0) with TSPAN = [T0 TFINAL] integrates the
system of differential equations y'=f(t,y) from time T0 to TFINAL with
initial conditions Y0. Function ODEFUN(T,Y) must return a column vector
corresponding to f(t,y). Each row in the solution array Y corresponds
to a time returned in column vector T. To obtain solutions at specific
times T0, T1, ..., TFINAL (all increasing or all decreasing), use
TSPAN = [T0 T1 ... TFINAL].

[T,Y] = ODE45(ODEFUN,TSPAN,Y0,OPTIONS) solves as above with default
integration properties replaced by values in OPTIONS, an argument
created with the ODESET function.  See ODESET for details.  Commonly
used options are scalar relative error tolerance 'RelTol' (1e-3 by
default) and vector of absolute error tolerances 'AbsTol' (all components
1e-6 by default).

[T,Y] = ODE45(ODEFUN,TSPAN,Y0,OPTIONS,P1,P2,...) passes the additional
parameters P1,P2,... to the ODE file as ODEFUN(T,Y,P1,P2,...) and to all
functions specified in OPTIONS. Use OPTIONS = [] as a place holder if no
options are set.
```

The ode commands in *MATLAB* require a function m-file defining the differential equation, e.g. `f.m`. The default accuracy for `ode23` is 1×10^{-3} (1E-3); and for `ode45` is 1×10^{-6} (1E-6). The default accuracy can be changed using "odeset" to adjust the options (see *MATLAB* help for details).

Example

The equation

$$m_s \frac{dw}{dt} = (m_s - rV)g - \tfrac{1}{2} r C_D A w^2$$

describes the acceleration to terminal velocity of a spherical object of mass m_s and volume V released in a non-moving fluid of density r, where w is the vertical velocity of the sphere, g is gravitational acceleration, C_D is the drag coefficient, and A is the cross-sectional area of the sphere. For a small sphere ($\mathbf{R}_D = wDr/m < 0.5$), where D is sphere diameter and m is fluid viscosity), C_D = 24/ $\mathbf{R}_D$ and the terminal velocity, w_s, is given by Stokes law:

$$w_s = \frac{(r_s - r)\, gD^2}{18m}$$

where r_s is the density of the sphere. We can check our solution by comparing it to w_s after w reaches a constant value (i.e. its terminal velocity, also referred to as settling velocity or fall velocity). A *MATLAB* code to solve this equation is given below for an example in which $D = 3\times10^{-5}$ m (30 μm), r_s = 2650 kg m^{-3}, r = 1000 kg m^{-3}, and m = 0.001 Pa-s.

[2] Reprinted with permission from MathWorks, Inc.

```
%ode_ex.m

[t,w]=ode23('wprime',[0 0.001],0);
[t2,w2]=ode45('wprime',[0 0.001],0);
opt=odeset('RelTol',1e-6);   %reset tol value
[t3,w3]=ode23('wprime',[0 0.001],0,opt);

%calculate Stokes settling velocity
rho=1000; rhos=2650; mu=1.3e-3; g=9.81; D=.3e-4;
ws=(rhos-rho)*g*D.^2/(18*mu);
%check if particle Reynolds number < 0.5 as required for Stokes equation
Rd=ws*D*rho/mu;
if Rd>0.5, fprintf('Stokes eqn invalid'), end;
%check that particle Reynolds number < 800 as required
% for the Schiller et al. (1933) drag coefficient formula.
if Rd>800, fprintf('Drag coefficient invalid'), end;

plot(t,y,'-b',t2,y2,'-r',t3,y3,'-g',max(t),ws,'*')
xlabel('t (s)'); ylabel('w (m/s)')
legend('ode23','ode45','ode23, rtol=1e-6','exact')

xlabel('x')
ylabel('error')
legend(['ode23, n=', num2str(n)],['ode45, n=', num2str(n2)],...
      ['ode23, rtol=1e-6, n=', num2str(n3)])

function wp=wprime(t,w)

rho=1000; rhos=2650; mu=1.3e-3; g=9.81;
D=.3e-4; V=pi/6*D.^3; A=pi/4*D.^2; ms=rhos*V;
Rd=rho*D*w/mu; Cd=0;
if w~=0, Cd=24/Rd*(1+0.150*Rd^0.687); end;  %let Cd=0 if w=0
a=g*(ms-rho*V)/ms; b=0.5*rho*Cd*A ./ms;
wp=a-b*w.^2;
```

The solutions are shown in Figure 4.3. For this example, $\mathbf{R}_D < 0.5$, so comparing the calculated terminal velocity to Stokes equation is appropriate. For larger $\mathbf{R}_D$, the drag coefficient for spheres takes a different form, one that cannot be represented by a simple algebraic expression. For values of $\mathbf{R}_D$ up to 800, the approximate formula of Schiller et al. (1933; see Graf, 1971) can be used, $C_D = \left(24/\mathbf{R}_D\right)\left(1+0.15\mathbf{R}_D^{0.687}\right)$, as in the m-file above. Note that the acceleration time is very short for small particles like silt and sand falling through water and generally can be (and is) neglected in calculations of particle settling.

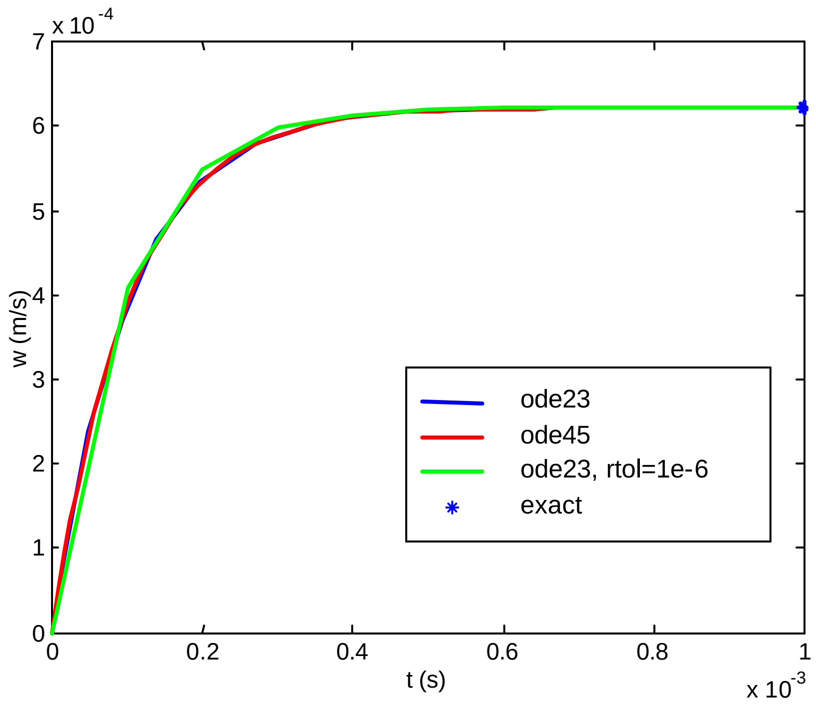


Figure 4.3. Solution to the example problem using *MATLAB* ode functions.

4.7. Multistep methods

All of the methods described above are examples of single-step methods, that is, they use information about the solution at just one location, x_i, to advance the solution to x_{i+1}. Single step methods have a number of advantages, including being self-starting and having the flexibility to change step size from one step to the next. Another approach is to use information about the solution at several previous locations, thus more fully utilizing what we know about the function. This is the idea behind the multistep methods. Most methods use equally spaced points to facilitate the calculations. This limits flexibility in terms of changing step size, but the methods can be very efficient.

Most multistep methods use past (already computed) values to construct a polynomial that approximates the derivative of the function and uses that to extrapolate to the next point. The number of past points used determines the order of the polynomial we can fit, and therefore the truncation error. Using two previous points would allow a quadratic approximation, and would result in a $O((\Delta x)^3)$ global error. Because in general we don't initially know values for y at the first three points, this method is not self-starting. It is necessary to use other methods, e.g. a Runge-Kutta method, to get values at, say, the first three points, after which a three-point multistep method can be used. See a text on numerical analysis, e.g. Gerald and Wheatley (1999), for details on multistep methods.

4.8. Sources of error, convergence, and stability

It is important to keep in mind that there are three possible sources of error in numerical calculations, such as solutions to ordinary differential equations. The one we have discussed the most is truncation error, the error associated with the number of terms in a series, e.g. Taylor series. Other sources of error are round-off errors, which will always be present even if our method is exact, and original data errors, associated with not knowing the initial conditions or boundary conditions exactly [see Box 3.1]. Errors at each step propagate through the solution, so the global error is generally about an order larger than the local error.

A method is convergent if the approximate solution tends toward the true solution as $\Delta x \to 0$. All of the widely used methods for solving ordinary differential equations (in particular the ones we have discussed) are convergent. Stability refers to the growth of errors as the solution proceeds. A stable method is one in which the global error does not grow in an unbounded manner. For many ODE's, the step size required to obtain an accurate solution is much smaller than the step size required for stability. As a result, stability is often not a major concern. When a solution is unstable, it is usually obvious. A smaller step size will generally rectify the problem, although there are cases that are unconditionally unstable for some methods. Changing methods is the best approach in such cases.

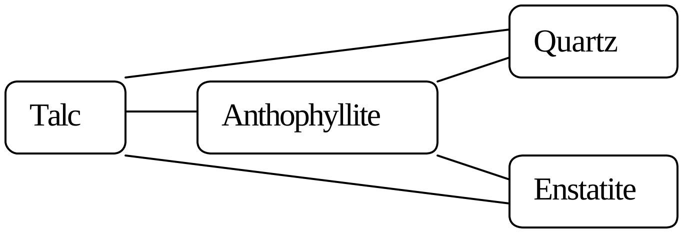
4.9. Systems of equations

The differential equations we have considered so far are simple first-order ordinary differential equations. There are other problems described by several coupled, first-order differential equations. The predator-prey equations (e.g., see Kot 2001) used in population ecology are an example. This problem is defined by the two equations

$$\begin{aligned} y_1' &= (1 - a\, y_2)\, y_1 \\ y_2' &= (-1 + b\, y_1)\, y_2 \end{aligned}$$

The size of the prey population is given by y_1, and the predator population by y_2. The prey population, y_1, decreases for large values of y_2, which in turn decreases the predator population, which increases the number of prey, and so forth. The *MATLAB* ode routines can solve a set of equations such as this by defining y and y' as column vectors; the initial conditions are also given by a column vector. (The m-file defining the equations must return the y' values as a vector.) The solution is a matrix in which the first column is y_1 and the second column is y_2.

As an example of solving a system of equations, consider the chemical reaction defining the dehydration of talc,



The rates of the reactions are well described by first-order kinetics (rate proportional to amount) so the differential equations are:

$$\frac{dy_1}{dt} = a_{11}y_1$$
$$\frac{dy_2}{dt} = a_{21}y_1 + a_{22}y_2$$
$$\frac{dy_3}{dt} = a_{31}y_1 + a_{32}y_2$$
$$\frac{dy_4}{dt} = a_{41}y_1 + a_{42}y_2$$

where y_1 refers to talc, y_2 refers to anthophyllite, y_3 refers to enstatite, and y_4 refers to quartz. Expressing the constituents in terms of volume and for a temperature of 830°C and a pressure of 1000 bars, the coefficients are (see Greenwood, 1963, but be advised that some tables are mislabeled): a_{11} = -18.0×10^{-4} min^{-1}, a_{21} = 11.0×10^{-4} min^{-1}, a_{22} =- 5.9×10^{-4} min^{-1}, a_{31}= 3.9×10^{-4} min^{-1}, a_{32} = 4.8×10^{-4} min^{-1}, a_{41} = 2.1×10^{-4} min^{-1}, a_{42} = 0.5×10^{-4} min^{-1}. The set of equations can be rewritten in matrix form as

$$\frac{d\mathbf{y}}{dt} = \begin{pmatrix} a_{11} & 0 & 0 & 0 \\ a_{21} & a_{22} & 0 & 0 \\ a_{31} & a_{32} & 0 & 0 \\ a_{41} & a_{42} & 0 & 0 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix}$$

We can write a function file to define the equations. Call this `rates.m`.

```
function ydot=rates(t,y)
% function for the dehydration of talc
a=zeros(4,4);
a(1,1)=-18.0e-4; % all a's are min^-1
a(2,1)=11.0e-4; a(2,2)=-5.9e-4; a(3,1)=3.9e-4;
a(3,2)=4.8e-4; a(4,1)=2.1e-4; a(4,2)=0.5e-4;
ydot=a*y;
```

If we start at time zero with a unit volume of talc and consider the evolution over 8000 minutes, we can obtain the solution with the *MATLAB* command

```
[t,y]=ode45('rates',[0 8000], [1 0 0 0]');
```

The results can be plotted (Figure 4.4) with the commands

```
plot(t,y)
xlabel('Time, minutes')
ylabel('Volume, ml')
legend('talc','anthophyllite','enstatite','quartz')
```

In cases where there are several dependent variables (e.g., with a system of equations rather than just a single equation), it sometimes is useful to plot the dependent variables against each other and not simply view each variable as a function of time [Box 4.1].

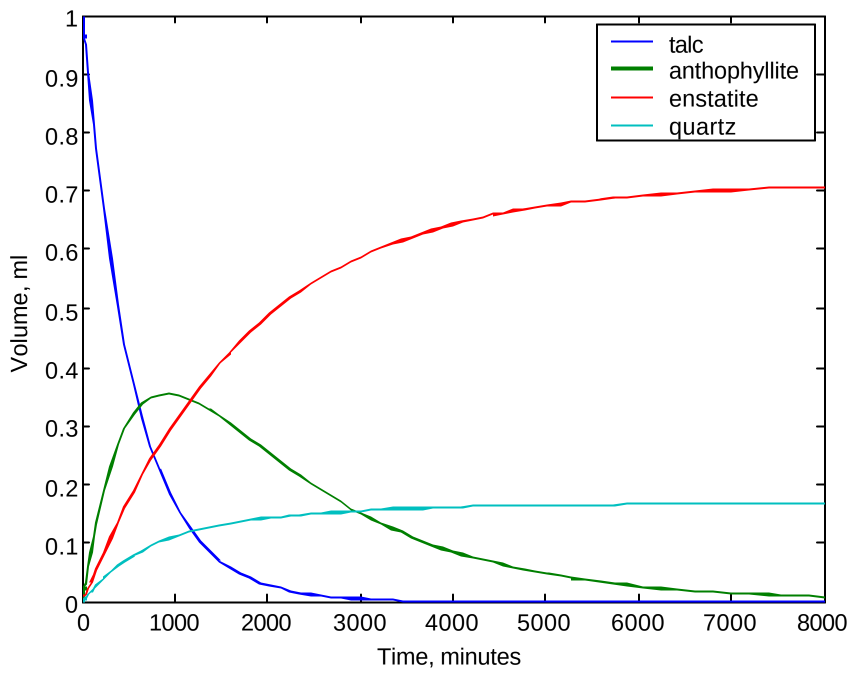


Figure 4.4. Results from *MATLAB* integration of talc equations.

4.10. Problems

1. As mentioned in Section 4.1, the Green-Ampt equation for the movement of a wetting front into a dry soil is

$$\frac{dL_f}{dt} = \frac{K_s}{\Delta q}\left(\frac{-y_f + L_f}{L_f}\right)$$

Solve this equation using the *MATLAB* functions `ode23`, `ode45`, a standard fourth-order Runge-Kutta method, and the Euler method. Take K_s=3×10^{-6} m s^{-1}, $\Delta q = 0.2$, and $y_f = -0.2$ m. Compare the accuracy of the methods. The analytical solution is:

$$t = \frac{L_f \Delta q}{K_s} + \frac{y_f \Delta q}{K_s}\log_e\left(1 + \frac{L_f}{-y_f}\right).$$

2. When open channel flow encounters a change in bed slope, there is an adjustment in flow depth (see, e.g., Henderson 1970). Assuming water depth and bed elevation vary gradually, the change of depth with downstream distance is given by

$$\frac{dh}{dx} = \frac{S_b - S_f}{1 - \mathbf{F}^2}$$

where S_f is energy slope, S_b is channel bed slope, h is flow depth, **F** is Froude number, $U/\sqrt{gh}$, U is channel mean velocity = Q/A, A is channel cross-sectional area and g is gravitational acceleration (9.81 m s^{-2}). S_f is given by Manning's equation (as a function of h in a rectangular channel) for a given Q, n, and channel width, w [see Box 2.2]. Let Q=20 m^3 s^{-1}, S_b=0.0008, n=0.015, and w=15 m. Use `ode45` to solve for $h(x)$ from x=0 m to x=150 m for an initial flow depth of h=0.8 m at x=0. Plot the result.

3. The interaction between hosts and parasites can be described by a pair of coupled first-order ordinary differential equations:

$$\frac{dP}{dt} = P - a\frac{P^2}{H}$$
$$\frac{dH}{dt} = H - b\,PH$$

where P represents the population of parasites and H represents the population of hosts. Let a=2.5 and b=0.1. At t=0, P=6 and H=15. Use the *MATLAB* ode functions to solve for P and H from t=0 to t=20. Plot P and H against t and plot P against H.

4. When an organic waste is discharged into a stream, a biochemical oxygen demand (BOD) is exerted; that is, bacterial decomposition of the organic waste uses oxygen and thus depletes dissolved oxygen in the water (e.g., Hemond and Fechner, 1994). Dissolved oxygen (DO) is replenished through atmospheric reaeration (by diffusion through the air-water interface). If a

point discharge into a river with steady uniform flow is envisioned, a typical pattern of DO, called a "sag" is observed. Below the waste outfall, DO drops in response to the BOD. As the decomposition of the waste proceeds, the BOD is decreased and atmospheric reaeration replenishes the DO. DO decreases downstream to a "critical value" (a minimum) and then gradually returns to ambient conditions. The process can be modelled using a form of the Streeter-Phelps equations (Streeter and Phelps, 1925):

$$\frac{dL_C}{dx} = -(K_C / U)L_C$$

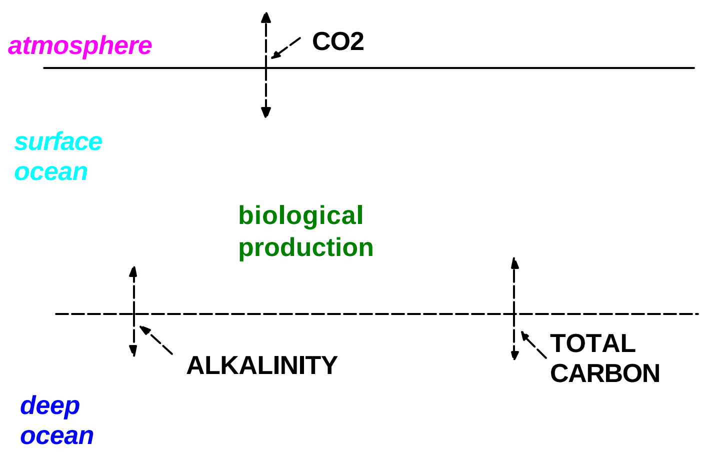
$$\frac{dL_N}{dx} = -(K_N / U)L_N$$

$$\frac{dc}{dx} = -(K_C / U)L_C - (K_N / U)L_N + (K_a / U)(c_s - c)$$

where L_c=carbonaceous BOD, L_N=nitrogenous BOD, c=DO concentration, c_s=saturation DO concentration, U is mean velocity, and the K's are rate constants.

Solve the Streeter-Phelps equations and plot the results for K_C=1.2 day^{-1}, K_N=0.8 day^{-1}, K_a=1.9 day^{-1}, $L_C(0)$=5 mg L^{-1}, $L_N(0)$=8 mg L^{-1}, river velocity=10 km day^{-1}; all rate constants are for T=20°C. Temperature adjustments for the rate constants are: $K_C=K_C(20)\times(1.024)^{(T\text{-}20)}$, $K_N=K_N(20)\times(1.024)^{(T\text{-}20)}$, and $K_a=K_a(20)\times(1.047)^{(T\text{-}20)}$. Saturation DO concentration is a function of T as well: $c_s=14.652-0.41022T+0.007991T^2-0.00077774T^3$. Consider results for T=5°C and T=20°C.

5. An issue of current concern is the fate of carbon dioxide being added to the atmosphere by the burning of fossil fuel. Over the long term, oceanographers view the problem as one of partitioning carbon among the atmosphere, the surface ocean, and the deep ocean. Given this simplified view, we can construct a simple model for the atmosphere-ocean coupling with regard to carbon balance (Walker, 1991). The model must keep track of CO_2 in the atmosphere, total carbon in the surface and deep ocean, and the speciation of inorganic carbon in the surface ocean because the exchange with the atmosphere is a diffusive process depending on the partial pressure of CO_2 in the surface ocean.



Schematic of the ocean-atmosphere system for carbon

The equations are (see Walker 1991):

$$\frac{d(CO_2)_{ATM}}{dt} = \frac{[(CO_2)_{SO} - (CO_2)_{ATM}]}{T} + input \Big/ 4.95\cdot 10^{16}$$

$$\frac{dC_{SO}}{dt} = \left\{ -\frac{[(CO_2)_{SO} - (CO_2)_{ATM}]4.95\cdot 10^{16}}{T} - 1.25P + (C_{DO} - C_{SO})w \right\} \Big/ V_{SO}$$

$$\frac{dALK_{SO}}{dt} = \left[\left(ALK_{DO} - ALK_{SO}\right)w - 0.35P\right] \Big/ V_{SO}$$

$$\frac{dC_{DO}}{dt} = \left[1.25P - (C_{DO} - C_{SO})w\right] \Big/ V_{DO}$$

$$\frac{dALK_{DO}}{dt} = \left[-\left(ALK_{DO} - ALK_{SO}\right)w + 0.35P\right] \Big/ V_{DO}$$

$$(CO_2)_{SO} = 0.054\frac{(2C_{SO} - ALK_{SO})^2}{(ALK_{SO} - C_{SO})}$$

where the subscripts refer to the atmosphere, the surface ocean, and the deep ocean. CO_2 is mass of carbon dioxide (expressed in relative units, i.e., unity for present day), C is total carbon concentration, ALK is alkalinity. The constant 4.95x10^{16} in the equations is a conversion from relative CO_2 to actual moles of CO_2 in the atmosphere. T is a time constant, w is a flux (exchange) from surface to deep ocean, P is biological production of organic (and associated inorganic) carbon that "rains" into the deep ocean, and V represents the volume of the ocean compartments. Finally, *input* represents the rate of addition of carbon to the atmosphere by fossil-fuel burning. Appropriate values for the computation are (with initial conditions at 1850): T = 8.64 y; $V_{SO} = 0.12\times 10^{18}$ m^3; $V_{DO} = 1.23\times 10^{18}$ m^3; $P = 0.000175\times 10^{18}$ mole y^{-1}; w =

0.001×10^{18} m^3; and, at time "zero", $(CO_2)_{ATM} = 1$ (relative units, 4.95×10^{16} mole actual units), $C_{SO} = 2.01$ mole m^{-3}, $C_{DO} = 2.23$ mole m^{-3}, $ALK_{SO} = 2.2$ mole m^{-3}, $ALK_{DO} = 2.26$ mole m^{-3}. Assuming the following time course for *input* (in 10^{14} mole y^{-1}), with approximately linear changes between specific times and with all inputs zero after 2500, write a code to compute the time course of atmospheric and oceanic carbon over (a) 1850-2600 and (b) 1850-6000.

Year	1850	1950	1980	2000	2050	2080	2100	2120	2150	2225	2300	2500
Input	0	1	4	5	8	10	10.5	10	8	3.5	2	0

4.11. References

Fetter, C.W. Jr., *Applied Hydrogeology*, 598 pp., Prentice-Hall, Upper Saddle River, NJ, 2001.

Gerald C.F. and P.O. Wheatley, *Applied Numerical Analysis*, 319 pp., Addison Wesley, Reading, MA, 1999.

Graf. W.H., *Hydraulics of Sediment Transport*, 513 pp., McGraw-Hill, New York, 1971.

Greenwood, H.J., The synthesis and stability of anthophyllite. *J. Petrology, 4*: 317-351, 1963.

Hemond, H.F. and E.J. Fechner, *Chemical fate and transport in the environment*, 433pp., Academic Press, San Diego, 1994.

Henderson, F.M., *Open Channel Flow*, 522 pp., Macmillan, New York, 1970.

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

Kot, M., *Elements of Mathematical Ecology*, 453 pp., Cambridge University Press, New York, 2001.

Lasaga, A.C. and R.J. Kirkpatrick (eds.), *Kinetics of Geochemical Processes*, 398 pp., Mineralogical Society of America, Washington, DC, 1981.

Nicolis, G. and I. Prigogine, *Self-organization in Non-equilibrium Systems: From Dissipative Structures to Order through Fluctuations*, 512 pp., Wiley, New York, 1977.

Streeter, H.W. and E.B. Phelps, A study of the pollution and natural purification of the Ohio River, III, Factors concerned in the phenomena of oxidation and reaeration. U.S. Public Health Service, Pub. Health Bull. No. 146, 75 pp., 1925.

Walker, J.C.G., *Numerical adventures with geochemical cycles*, 192 pp., Oxford University Press, New York, 1991.

Box 4.1. The brusselator

The following equations, defining what has been called the "brusselator" (Nicolis and Prigogine, 1977), serve as a model for certain self-catalyzing biochemical reactions.

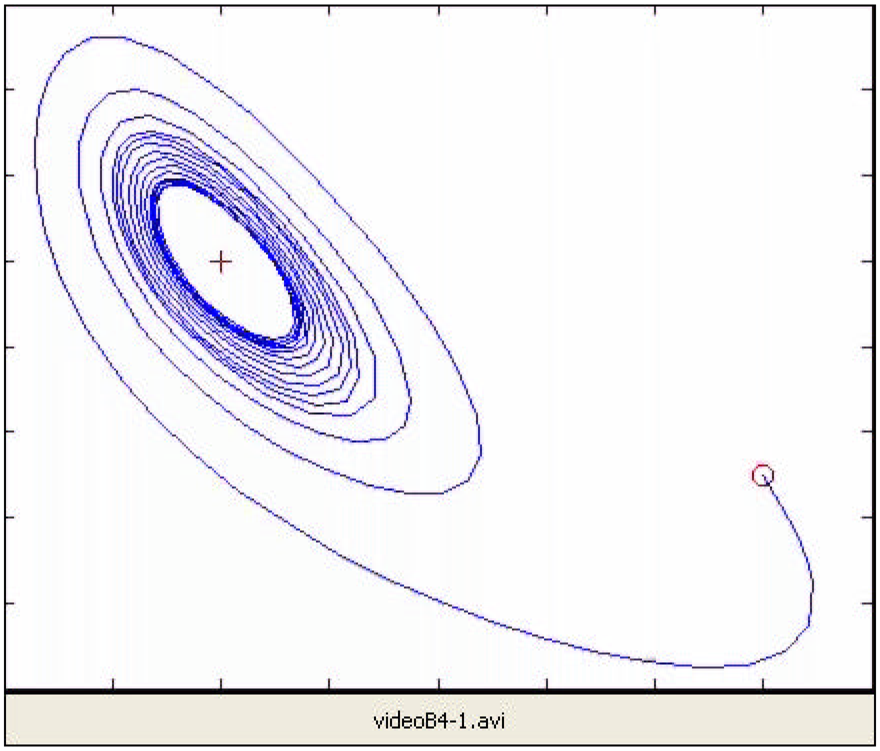
$$\frac{dx}{dt} = A - Bx + x^2 y - x$$

$$\frac{dy}{dt} = Bx - x^2 y$$

The code for solving the equations and for plotting the "phase plane", i.e. *x* versus *y*, is shown below as are the results for *A*=1 and *B*=2 (Video B4.1). The "o" marks the initial condition and the "+" marks the equilibrium point where the time derivatives are zero. For these values of the parameters *A* and *B*, the solution is converging toward the equilibrium point. For larger values of *B* the solution to the equations is a limit cycle in which the values of *x* and *y* cycle around the unstable equilibrium point.

```
% brus_mov.m
% make a brusselator movie
% z0 is the initial condition
% par contains parameters A and B
%
z0=[2;1.5];par=[1 2];
[t, z]=ode45(@brusselator,tspan,z0,[],par);
A=par(1);B=par(2);
x=z(:,1);
y=z(:,2);
plot(x,y,'w');
hold on
plot(A,B/A,'+r','MarkerSize',8);
plot(z0(1),z0(2),'or','MarkerSize',8);
k=1;
M(k)=getframe;
for i=1:10:length(x)-10
    k=k+1;
    plot(x(i:i+10),y(i:i+10),'b','LineWidth',1.2)
    M(k)=getframe;
end
hold off

function ydot=brusselator(t,z,params)
%
% brusselator equations, parameters A and B
% ydot=brusselator(x,y,A,B)
%
A=params(1);B=params(2);
x=z(1,:);y=z(2,:);
dxdt=A+x.^2*y-B*x-x;
dydt=B*x-x.^2*y;
ydot=[dxdt; dydt];
```



Video 4.1. Brusselator results.

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>>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 3.1. Errors in numerical methods

There are two principal sources of error in numerical computation. One is due to the fact that an *approximation* is being made (e.g., a derivative is being approximated by a finite difference). These errors are called *truncation errors*. The second is due to the fact that computers have limited precision, i.e., they can store only a finite number of decimal places. These errors are called *roundoff errors*.

Truncation errors arise when a function is approximated using a finite number of terms in an infinite series. For example, truncated Taylor series are the basis of finite difference approximations to derivatives (Chapter 3.2). The error in a finite difference approximation to a derivative is a direct result of the number of terms retained in the Taylor series (i.e., where the series is truncated). Truncation error is also present in other numerical approximations. In numerical integration, for example, when each increment of area under a curve is calculated using a polynomial approximation to the true function (Chapters 3.6-3.7), truncation errors arise that are related to the order of the approximating polynomial. For example an n^{th}-order polynomial approximation to a function results in an error in the integral over an increment Δx of $O(\Delta x)^{n+2}$ (*local* error). When the integrals over each increment are summed to approximate the integral over some domain $a \leq x \leq b$, the local errors sum to give a *global* error of $O(\Delta x)^{n+1}$. Truncation errors decrease as step size (Δx) is decreased – the finite difference approximation to a derivative is better (has lower truncation error) when Δx is "small" relative to when Δx is "large."

Roundoff errors stem from the fact that computers have a maximum number of digits that can be used to express a number. This means that the machine value given to fractional numbers without finite digit representations, for example, 1/3 = 0.33333333…, will be rounded or chopped at the precision of the computer. It also means that there is a limit to how large or small a number a computer can represent in floating point form. For many computations, the small changes in values resulting from roundoff are insignificant. However, roundoff errors can become important. For example, subtraction of two nearly identical numbers (as occurs when computing finite differences with very small values of Δx) can lead to relatively large roundoff error depending on the number of significant digits retained in the calculation. Interestingly, this means that approximations to derivatives will be improved by reducing Δx to some level because truncation errors are reduced, but that further decreases in Δx will make the estimate *worse* because roundoff error becomes large and dominates for very small values of Δx. Roundoff error can also complicate some logical operations that depend on establishing equality between two values if one or both are the result of computations that involved chopping or rounding.

Finally, in the hydrological sciences, measured data are often used in a calculation. For example, one might want to find the derivative of water velocity with respect to height above a streambed using numerical differentiation of data measured using a flowmeter. Such data are subject to *measurement errors*, which are then inserted into any numerical computation in which they are used.

CHAPTER 5

Numerical Methods for Solving Higher-Order and Boundary-Value Ordinary Differential Equations

Numerical Methods in the Hydrological Sciences: AGU Lecture Notes 1
Copyright 2004 by the American Geophysical Union
doi: 10.1029/001LN7
(Version 2004-08-02)

5. Numerical Methods for Solving Higher-Order and Boundary-Value Ordinary Differential Equations

5.1. Introduction

Many of the ordinary differential equations encountered in the hydrological sciences are second or higher-order differential equations, including ones for which boundary conditions at two boundaries are specified rather than at just one. Higher-order differential equations have the form

$$\frac{d^n y}{dx^n} = f(x, y, y', ..., y^{(n-1)}) \tag{5.1}$$

Hydrological examples include:

1. Steady, uniform flow in one dimension through an unconfined aquifer receiving recharge at rate w. In this case, conservation of mass, Darcy's law, and the Dupuit assumption result in a second-order differential equation for h, the height of the water table: $\frac{d}{dx}\left[Kh\frac{dh}{dx}\right] = -w.$
2. The Theim equation for the drawdown associated with a steady, radial flow to a well in a confined, homogeneous, isotropic aquifer: $\frac{d^2h}{dx^2} + \frac{1}{r}\frac{dh}{dx} = 0.$

To solve second-order equations like these, we must specify two initial or boundary conditions; an nth-order ordinary differential equation requires n initial or boundary conditions.

5.2. Solving higher-order initial value problems

To begin, we'll assume that the given conditions are initial conditions, i.e., they are all specified at the same endpoint of the domain. The initial conditions for a second-order equation, would have the form

$$y_0 = y(x = x_0) = a; \quad y_0' = y'(x = x_0) = b$$

For an n^{th}-order ODE, n-1 derivatives of y at x_0 would be specified.

There are several methods for solving second-order equations, including Taylor series and Runge-Kutta methods. One straightforward general method for solving second and higher-order initial value problems is to transform the equation into a system of simultaneous first-order ordinary differential equations. Then the methods covered in the previous chapter for solving first-order equations, including the *MATLAB* programs `ode23` and `ode45`, can be used to solve the system of equations.

The general approach to reducing an nth order ordinary differential equation to a set of n simultaneous first order equations is to let

$$
\begin{aligned}
&y_1 = y \\
&y_2 = y_1' = y' \\
&y_3 = y_2' = y'' \\
&\vdots \\
&y_n = y_{n-1}' = y^{(n-1)}
\end{aligned}
$$

Equation (5.1) can then be written

$$
\begin{aligned}
&y_n' = f(x, y_1, y_2, \ldots, y_n) \\
&y_{n-1}' = y_n \\
&\vdots \\
&y_2' = y_3 \\
&y_1' = y_2
\end{aligned}
\tag{5.2}
$$

with appropriate initial conditions. Recalling that the initial values of y and (n-1) of its derivatives are known at $x=x_0$, we set

$$y_1(x_0) = y(x_0),\ y_2(x_0) = y_0'(x_0), \ldots, y_n(x_0) = y^{(n-1)}(x_0).$$

Equations (5.2) are in a vector form that can be used in the *MATLAB* `ode` commands to solve for the y_i's, given the vector of initial conditions.

5.3. Example: Bessel equation

Consider the second-order ordinary differential equation

$$x^2 y'' + xy' + (x^2 - n^2)y = 0$$

which can be written equivalently in the form of equation (5.1) as

$$y'' = -\frac{1}{x}y' - \frac{(x^2 - n^2)}{x^2}y\,. \tag{5.3}$$

This is known as the Bessel equation. This equation arises commonly in physical problems, including problems in fluid flow, elasticity, and electrical field theory. A solution to the equation is used to generate the *MATLAB* logo. Solutions to equation (5.3) are known as Bessel functions of order n. For this example, we'll take $n = 0$, with initial conditions $y(x=0) = 1$and $y'(x=0) = 0$.

To rewrite the Bessel equation as a system of first order equations, let $y_1=y$ and $y_2=y'$. Then, the equation (5.3) can be written

$$
\begin{aligned}
&y_1' = y_2 \\
&y_2' = -y_2/x - y_1
\end{aligned}
$$

with initial conditions, $y_1(0) = 1$ and $y_2(0) = 0$. [Note that equation (5.3) for y'' is undefined at $x=0$; we will begin the calculation a small distance from 0, e.g., $x = 0.001$.] We can write a *MATLAB* function file `yprime0.m` to solve the problem. The analytical solution (a series solution) is given by the *MATLAB* function `besselj(nu,x)`with `nu=0`.

```
function yp=yprime0(x,y)
yp=[y(2);-y(2)/x-y(1)];
```

The *MATLAB* commands

```
[x,y]=ode45('yprime0',[0.001 1],[1 0]');
exact=besselj(0,x);
[x y exact]
plot(x,exact,'-k',x,y(:,1),'-b',x,y(:,2),'-r')
```

result in (partial list)

```
ans =
    0.0010    1.0000         0    1.0000
    0.2776    0.9808   -0.1375    0.9808
    0.6330    0.9023   -0.3009    0.9023
    1.2225    0.6598   -0.5039    0.6599
    1.9189    0.2705   -0.5807    0.2708
    2.6368   -0.1140   -0.4602   -0.1139
    3.2905   -0.3420   -0.2244   -0.3422
    3.9011   -0.4016    0.0277   -0.4018
    4.4645   -0.3283    0.2213   -0.3286
    5.0260   -0.1689    0.3302   -0.1691
    5.6307    0.0372    0.3313    0.0372
    6.2212    0.2065    0.2276    0.2066
    6.8112    0.2936    0.0617    0.2938
    7.3941    0.2790   -0.1081    0.2792
    7.9630    0.1800   -0.2290    0.1802
    8.5550    0.0268   -0.2730    0.0269
    9.1537   -0.1265   -0.2244   -0.1265
    9.6663   -0.2176   -0.1242   -0.2177
   10.0000   -0.2457   -0.0433   -0.2459
```
(x)   (y)   (y′)   (exact)

The solution is illustrated in Figure 5.1. Note that the exact solution and our calculated solution lie on top of each other.

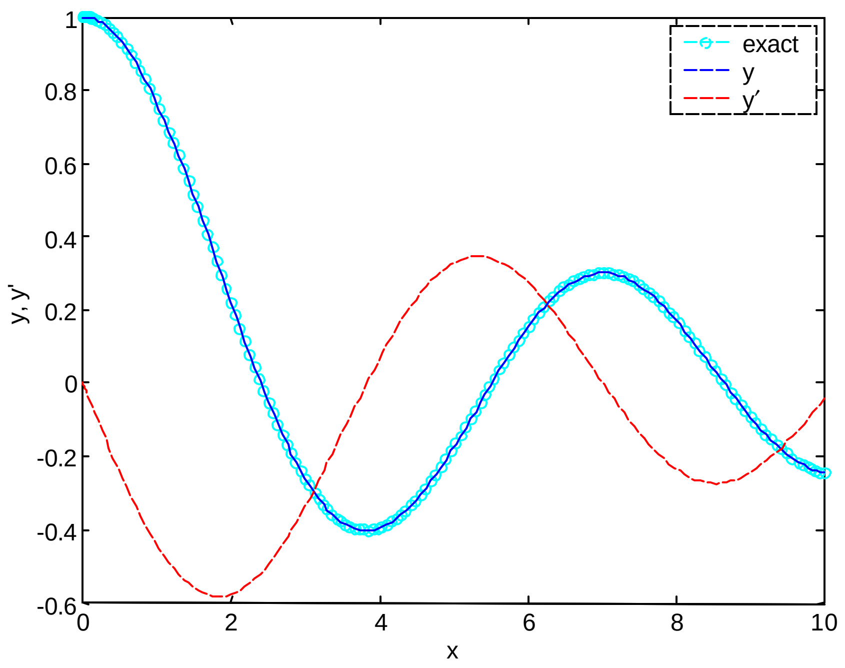


Figure 5.1. Solution to the Bessel equation of order 0 with specified initial values.

5.4. Solving boundary value problems using the shooting method

Boundary value ODEs are problems in which the known conditions are specified at the two end points of the domain. One approach to solving such an equation is to express it as a system of simultaneous first order equations as described above, with the known initial conditions specified and the others estimated. (This approach works best with only one or two unknown initial conditions.) The equation is solved, and the solution at the far boundary compared to the specified condition(s) on that boundary. Assuming the first guess is not correct, it can be successively adjusted until the value at the far boundary matches the boundary condition. This can be done by trial and adjustment, but there are several more efficient approaches for finding the proper condition.

The initial conditions necessary to match the far boundary condition can be found quite easily for linear equations, particularly second and third-order equations. A differential equation is linear if the dependent variable and all of its derivatives appear only as linear terms. Linear equations have the nice property that linear combinations of any two solutions are also solutions. This means that if we make two estimates of the unknown initial condition, g_1 and g_2, which result in the two values of y at the far boundary, v_1 and v_2, then the value of the "guessed" initial condition that will give the desired boundary condition at the far boundary can be calculated as

$$g_{correct} = g_1 + \frac{g_2 - g_1}{v_2 - v_1}\left[BC_{desired} - v_1\right] \tag{5.4}$$

Consider the example equation used above, but with ν=1 and specified boundary conditions rather than initial conditions, y(0) = 0; y(10) = 0.0435 (corresponding to a Bessel equation of order 1).

If $y'(0) = g_1 = 0.5$, then $y(2) = v_1 = 0.0217$. If we repeat the calculation with $y'(0) = g_2 = 1.5$, then the calculation gives $y(2) = v_2 = 0.0650$. Based on these values we find from equation (5.4) that $g_{correct} = 1.0032$.

Using this value in `ode45`:

```
[x,z]=ode45('yprime1',[0 10],[1 1.0032]);
exact=besselj(1,x);
[x y exact]
plot(x1,y1(:,1),'-b',x2,y2(:,2),'-r',x,z,'-g',x,exact,'-k')
```

The results, illustrated in Figure 5.2, show that the method works for second-order, linear differential equations like the Bessel equation.

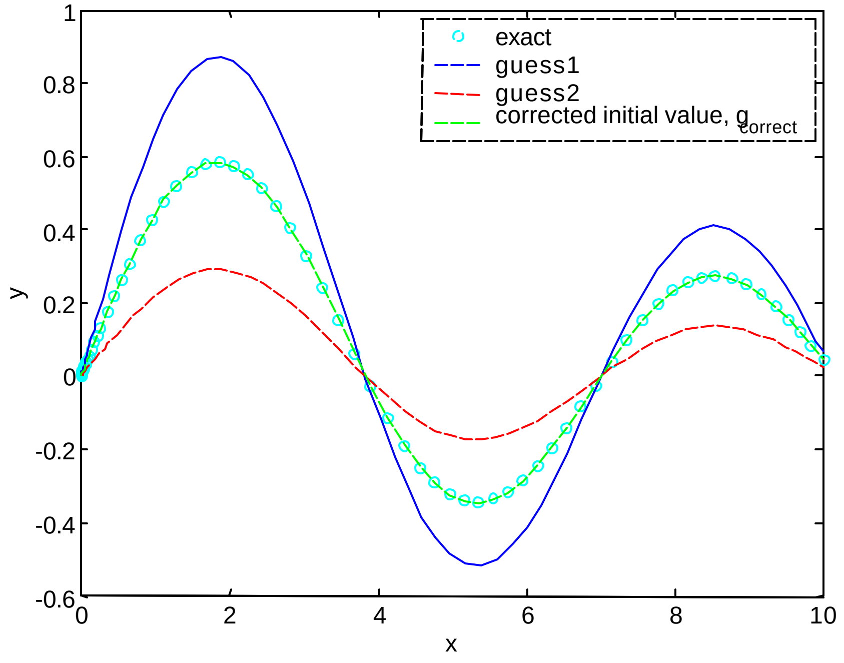


Figure 5.2. Solution to Bessel equation boundary value problem using the shooting method.

If the equation is a third-order, linear, boundary value ODE, then there will be two conditions on one boundary and one on the other. Only one condition must be "shot for" if we start the calculation at the boundary on which two of the three conditions are specified, although this may mean working in negative time or space steps. For fourth-order, linear, boundary value ODE's, two conditions may be given at each boundary. In that case, we could still find the solution by taking a linear combination, but we would need to use four solutions, and the algebra gets more complicated.

If the differential equation is not linear, then this method of combining solutions generally won't work unless the two estimates are quite close to the correct value. We can, however, think of the problem as one of finding the root of the equation given by the difference of the true boundary condition and the estimated values. Therefore, we could use something like the secant method

[Chapter 2.3] to find the initial condition that gives the desired boundary conditions at the far boundary.

5.5. Solving boundary value problems using finite differences

Another approach to solving a boundary value ODE is to express the equation in terms of finite differences. This leads to a set of algebraic equations that can be solved, for example, by Gaussian elimination [Chapter 2.11].

Previously we discussed several ways to estimate derivatives using finite differences [Chapter 3.2]. We found that forward and backward difference estimates of a first derivative have an error of $O(\Delta x)$, while central difference estimates have an error of $O(\Delta x^2)$; we also found estimates for the second derivative with errors $O(\Delta x^2)$. Using the $O(\Delta x^2)$ formulations, we can estimate the first and second derivatives of $y(x)$ as

$$\begin{aligned} y' &= \frac{y_{i+1} - y_{i-1}}{2\Delta x} + O((\Delta x)^2) \\ y'' &= \frac{y_{i+1} - 2y_i + y_{i-1}}{(\Delta x)^2} + O((\Delta x)^2) \end{aligned} \tag{5.5}$$

To illustrate the use of finite differences for solving ordinary differential equations, we will again use the Bessel equation of order 0, with $y(0) = 1.0000$ and $y(7) = 0.3001$. Substituting the finite difference approximations for y' and y'' into the differential equation (5.3), we obtain

$$x_i \frac{y_{i+1} - 2y_i + y_{i-1}}{(\Delta x)^2} + \frac{y_{i+1} - y_{i-1}}{2(\Delta x)} + x_i y_i = 0,$$

or

$$\left(x_i - \frac{\Delta x}{2}\right) y_{i-1} + \left(-2x_i + x_i (\Delta x)^2\right) y_i + \left(x_i + \frac{\Delta x}{2}\right) y_{i+1} = 0 \tag{5.6}$$

Equation (5.6) applies for each grid point x_i except the two end points, at which the values of y are known. If we divide the solution interval into n subintervals (n+l grid points), this procedure results in n-1 equations for n-1 unknowns (the values of y at each internal grid point). We can express this system of equations as a matrix equation, Ay = b, where A contains the coefficients of the y terms and b is a vector made up of the right-hand sides of the equations.

If we take $\Delta x = 0.5$ ($n = 14$), we obtain 13 equations:

$$\begin{aligned} &0.25y_0 - 0.875y_1 + 0.75y_2 = 0 \\ &0.75y_1 - 1.750y_2 + 1.25y_3 = 0 \\ &1.25y_2 - 2.625y_3 + 1.75y_4 = 0 \\ &\vdots \\ &6.25y_{12} - 11.375y_{13} + 6.75y_{14} = 0 \end{aligned}$$

The first equation above contains y_0 which is known to be 1.0000 from the specified boundary condition; likewise the last of the equations contains y_{14} which is known to be 0.3001. The set of equations can be written in matrix-vector form as:

$$\left|\begin{array}{cccccccc}
-0.875 & 0.750 & 0 & 0 & & & & \\
0.75 & -1.750 & 1.25 & 0 & & & & \\
0 & 1.25 & -2.625 & 1.75 & & & & \\
 & & & \bullet & & & & \\
 & & & & \bullet & & & \\
 & & & & & \bullet & & \\
 & & & & & 5.25 & -9.625 & 5.75 & 0 \\
 & & & & & 0 & 5.75 & -10.50 & 6.25 \\
 & & & & & 0 & 0 & 6.25 & -11.375
\end{array}\right| \left|\begin{array}{c} y_1 \\ y_2 \\ y_3 \\ \bullet \\ \bullet \\ \bullet \\ y_{11} \\ y_{12} \\ y_{13} \end{array}\right| = \left|\begin{array}{c} -0.250 \\ 0.000 \\ 0.000 \\ \bullet \\ \bullet \\ \bullet \\ 0.000 \\ 0.000 \\ -2.0257 \end{array}\right|$$

Letting A be the coefficient matrix, *y* be the vector of unknowns, and *b* be the right-hand-side vector, we can use the *MATLAB* command `y=A\b` to solve the set of equations. The result is (to 4 decimal places)

```
        x          y        exact      error
________________________________________________
        0     1.0000     1.0000          0
   0.5000     0.9345     0.9385     0.0040
   1.0000     0.7569     0.7652     0.0083
   1.5000     0.4989     0.5118     0.0129
   2.0000     0.2078     0.2239     0.0161
   2.5000    -0.0648    -0.0484     0.0164
   3.0000    -0.2732    -0.2601     0.0131
   3.5000    -0.3864    -0.3801     0.0063
   4.0000    -0.3944    -0.3971    -0.0028
   4.5000    -0.3086    -0.3205    -0.0119
   5.0000    -0.1588    -0.1776    -0.0188
   5.5000     0.0146    -0.0068    -0.0214
   6.0000     0.1694     0.1506    -0.0187
   6.5000     0.2711     0.2601    -0.0110
   7.0000     0.3001     0.3001    -0.0000
```

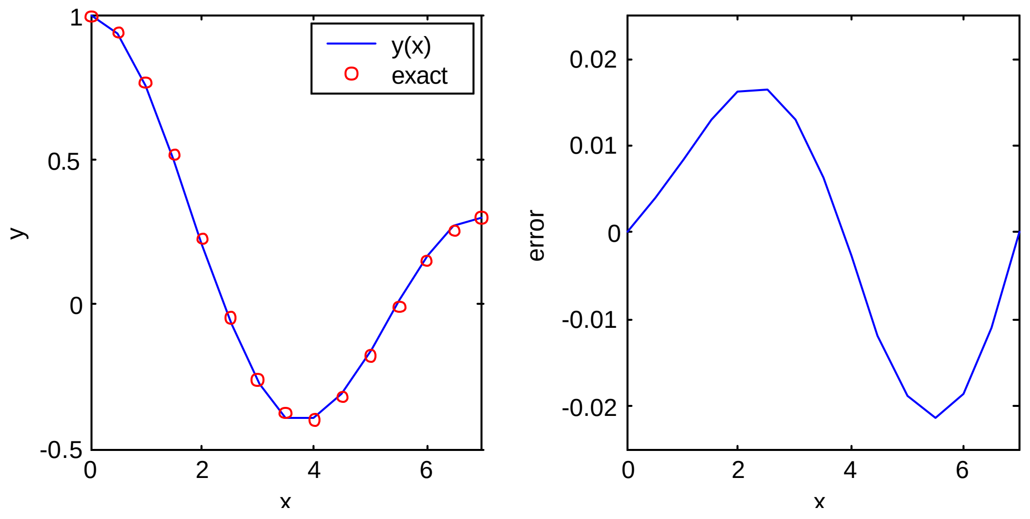


Figure 5.3. Finite difference solution and error for Δx=0.5.

We match the endpoints, but the accuracy is not good because of the coarse grid size. We can make Δx smaller to improve the accuracy, but in that case we probably don't want to type the whole of the matrix A in by hand because the size of the matrix becomes much larger. We can write an m-file to assign the values of A and solve the problem for arbitrary step size, as given below. As we would expect from the $O((\Delta x)^2)$ finite difference estimates we used to generate the finite difference equation, if we want accuracy to 3 decimal places, we need a step size of roughly $\sqrt{0.001} \cong 0.03$.

```
%besselfd.m
%finite difference solution to Bessel equation

dx=input('step size: ')
n=round((7/dx)-1);
dx=7/(n+1);  %adjust dx to give integer number of nodes
x=dx:dx:7-dx;
y1=0; y7=0.3001; %boundary conditions
A=zeros(n,n);
%form matrix A using MATLAB diag command; see 'help diag'
A=diag(-2*x+x*dx^2)+diag(x(2:n)-dx/2,-1)+diag(x(1:n-1)+dx/2,1);
b=zeros(n,1);  %right hand side
b(1)=-(x(1)-dx/2)*y1;   %left boundary condition
b(n)=-(x(n)+dx/2)*y7; %right boundary condition
y=A\b;      %solution
exact=besselj(0,x);
error=exact'-y;
plot(x,error)
xlabel('x'); ylabel('error');
```

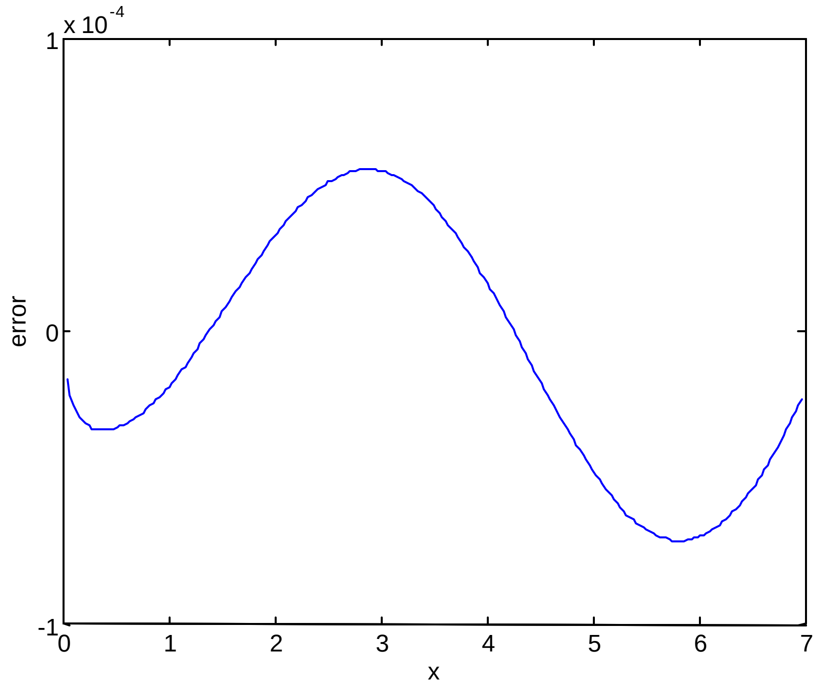


Figure 5.4. Error in finite difference approximation when Δx is reduced to 0.03.

An interesting note in regard to step size is that if we compute values of y for step sizes Δx and Δx /2, we can obtain an improved estimate using Richardson extrapolation. If we begin with $O((\Delta x)^2)$ estimates of the derivatives in our finite difference equation, we can improve the accuracy to $O((\Delta x)^4)$ by calculating a new estimate that is a weighted sum of the results for step Δx and step Δx /2, specifically the "better" value (i.e., done with Δx /2) plus 1/3rd of the difference between the "better" value and the "less good" (i.e., done with Δx). The resulting calculation has error $O((\Delta x)^4)$! Schematically, the calculation is:

$$y_{O((\Delta x)^4)} = y[\Delta x/2] + \frac{1}{3}\{y[\Delta x/2] - y[\Delta x]\}$$

where $y[\Delta x]$ and $y[\Delta x/2]$ refer to the values obtained for step sizes Δx and $\Delta x/2$, respectively.

5.6. Derivative boundary conditions

Derivative boundary conditions are relatively easily handled when the equation is solved as a system of simultaneous first-order equations. In fact, derivative initial conditions must be specified for second and higher order equations. When this method is being used to shoot for a derivative boundary condition at the far end of the domain, we can use the same procedures described above, but applied to the term in the vector y corresponding to that derivative, e.g., $y(2)$ for a condition specified in terms of y'.

The problem is less straightforward when a finite difference approximation is used to solve the equation. To specify a derivative boundary condition in this case, it is necessary to extend the domain beyond the specified interval and then to use finite difference forms of the boundary conditions to eliminate the fictitious points. This is probably most easily understood in the context of an example.

Consider the second-order equation we have been working with in these examples, but now with a derivative boundary condition specified at $x = 0$,

$$xy'' + y' + xy = 0 \qquad y(0) = 1, \quad y'(7) = 0.0046$$

We need a difference equation for each point at which y is unknown, which now includes $y(7)$. In this case all of the equations are the same as those given above except for the equations involving y_n. For example, with $\Delta x = 0.5$ we have as our equation at $x = 7$ (i.e., x_{14})

$$6.75y_{13} - 12.25y_{14} + 7.25y_{15} = 0$$

and for x_{13}

$$6.25y_{12} - 11.375y_{13} + 6.75y_{14} = 0$$

We had to add the equation at $x = 7$ because y at this location (y_{14}) is now an unknown. Doing this, however, introduces another unknown, y_{15}, into the difference equation; y_{15} indicates a value of y at $x = 7+\Delta x$, a point outside the domain of the problem. To eliminate y_{15} from the difference equation, we employ a finite difference approximation to the boundary condition, $y'(7) = 0.0046$.

$$y'(7) = \frac{y_{15} - y_{13}}{2\Delta x} + O((\Delta x)^2) = 0.0046$$

Thus, to $O((\Delta x)^2)$, we can write $y_{15} = y_{13} + 2\Delta x$*0.0046, and the difference equation for y_{14} becomes

$$14y_{13} - 12.25y_{14} = -0.0046(2\Delta x)(7.25)$$

Now we once again have as many equations as unknowns and can solve the system of equations as before. More details on derivative boundary conditions in finite difference equations are provided in the next chapter.

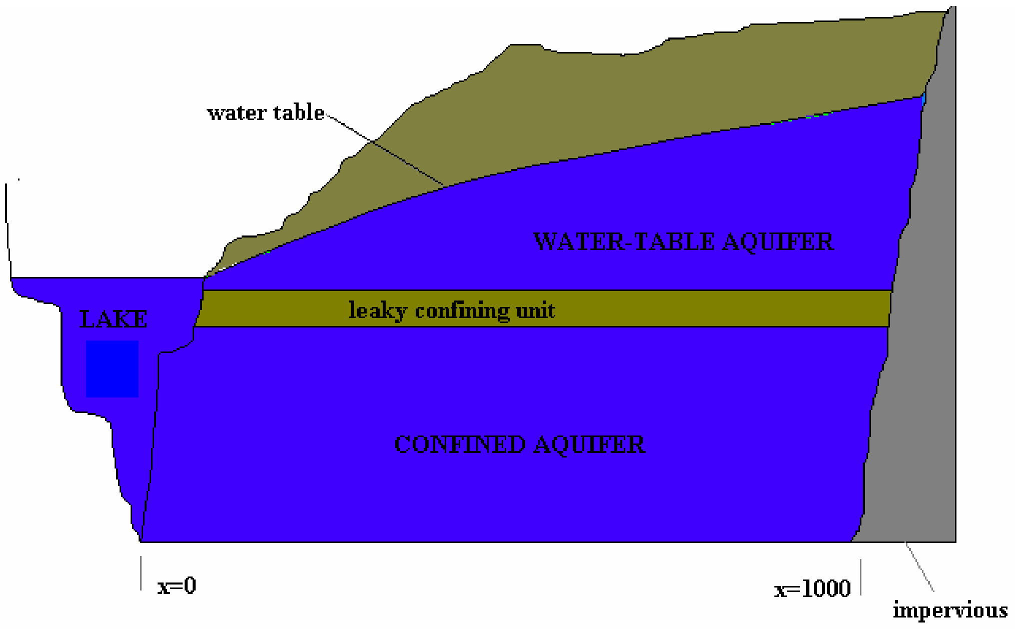
5.7. Problems

1. Solve the equation

$$ff''+2f'''=0$$

for the boundary conditions $f(z=0)=0$; $f'(z=0)=0$; $f'(z\to\infty)=1$. This is the Blasius equation for the velocity distribution in a laminar boundary layer on a flat plate. The velocity profile is given by f' (i.e., df/dz) as a function of z. To solve the equation numerically, the infinite value of z must be replaced by some large value z_{max}; z_{max}= 10 should do for this problem. Rewrite the 3rd-order ordinary differential equation as a system of first-order equations and use the shooting method to solve the problem using `ode45`. You can do this by trial and adjustment. For a challenge, try coding a more efficient method for finding the initial condition $f''(0)$ that gives back the desired boundary condition at z_{max} (i.e., $f'(z_{max})=1$.

2. At one end of a confined aquifer that extends from $x = 0$ to $x = 1000$m, head is maintained at 100m by a lake (see figure below). The other end of the aquifer ($x = 1000$) is intersected by an impervious unit, so that $dh/dx = 0$. The aquifer, with transmissivity 2.5x10^{-5} m^2 s^{-1}, is overlain by a 10m-thick, leaky confining bed with hydraulic conductivity 10^{-10} m s^{-1}. The overlying water-table aquifer has a water-table elevation given by $h_{wt}=100+0.06x-0.00003x^2$.



The equation describing the head distribution in the aquifer is

$$\frac{d^2h}{dx^2}=-\frac{C_{confining}}{T}\left(h-h_{wt}\right)$$

where T is aquifer transmissivity and $C_{confining}$ = hydraulic conductivity/thickness of the confining layer. Use the finite difference method to obtain the head distribution $h(x)$ in the aquifer. Examine the solution for different values of Δx.

3. The equations describing the trajectory of a projectile (e.g., an artillery shell) through the atmosphere are based on Newton's second law of motion. The problem is complicated by several aspects of the physical system: (1) drag force exerted on the projectile depends on the square of the velocity; (2) drag is dependent on air density which varies with altitude; (3) drag must be computed from experimental data rather than a theoretical relationship. These complexities dictate that a numerical rather than an analytical solution is necessary.

Solve the dynamic equations for a projectile in flight.

$$m\frac{d^2x}{dt^2} = -F_D \cos q$$

$$m\frac{d^2y}{dt^2} = -F_D \sin q - mg$$

where $F_D = \frac{1}{2}C_D r\, v^2 A,\ v = \sqrt{v_x^2 + v_y^2}$, and q is the angle between the x axis and the velocity vector. [Note that $\cos q$ and $\sin q$ can be expressed as v_x/v and v_y/v respectively.] The diameter of the projectile is 0.30 m and its mass is 7 kg. The initial velocity of the projectile is 670 m s^{-1} at an angle of 45°. Table 1 gives values of air density and the speed of sound as a function of altitude. Table 2 gives C_D as a function of Mach number, the ratio of projectile speed to the speed of sound at any given altitude. [You can use the *MATLAB* function `interp1` to obtain values of r and C_D for any altitude.]

Table 1.

Altitude (m)	*Air Density (kg m^{-3})*	*Speed of Sound (m s^{-1})*
0.00	1.225	340.16
2000.00	1.008	332.46
4000.00	0.821	324.46
6000.00	0.660	316.18
8000.00	0.526	307.85
10000.00	0.414	299.51
12000.00	0.311	295.24
14000.00	0.228	295.05
16000.00	0.167	295.05
18000.00	0.121	295.05
20000.00	0.088	295.16
22000.00	0.064	296.36
26000.00	0.034	299.06
28000.00	0.025	300.38
24000.00	0.047	297.56
30000.00	0.018	301.77

Table 2.

Mach Number	*Drag Coefficient*
0.0	0.50
0.4	0.52
0.8	0.66
1.2	0.93
1.6	1.03
2.0	1.01
2.4	0.99
2.8	0.97
3.2	0.95
3.6	0.93
4.0	0.92