Numerical Methods in the Hydrological Sciences

George Hornberger
University of Virginia
Department of Environmental Sciences

Patricia Wiberg
University of Virginia
Department of Environmental Sciences

Published by the American Geophysical Union

Published under the aegis of the AGU Books Board

Jean-Louis Bougeret, Chair; Gray E. Bebout, Carl T. Friedrichs, James L. Horwitz, Lisa A. Levin, W. Berry Lyons, Kenneth R. Minschwaner, Andy Nyblade, Darrell Strobel, and William R. Young, members.

Library of Congress Cataloging-in-Publication Data

Hornberger, George M.
Numerical methods in the hydrological sciences / George Hornberger, Patricia Wiberg.
p. cm. (Lecture Notes; 01)
Includes bibliographical references and index.
1. Hydrology--Mathematical models. I. Wiberg, P. L. (Patricia L.) II. Title.

GB656.2.M33H67 2004
551.48'01'5118--dc22
2004024304

ISBN 0-87590-726-1

Preface

This electronic book provides students, instructors and professionals in the hydrological sciences the tools they need to pursue study and research using numerical methods. It will enable them to write programs to solve fairly complex problems, to explore and understand the current literature in which numerical methods are used, and to have the confidence to delve into texts on numerical methods to extend their knowledge to solve new problems.

The book combines an introduction to a suite of useful numerical methods with examples that illustrate their application to a range of hydrological problems. We have adopted *MATLAB*®[1] as the computational framework for the notes because it provides the reader with a set of mathematical and graphical tools that makes it possible to gain solid experience in writing code while avoiding the details of particular programming languages.

There are thirteen weeks in the typical semester and we have provided a chapter for each week. The first chapter presents an introduction to using *MATLAB* for computation, including the basics of command-line operation and a description of how script and function files are used to create code to solve problems. Subsequent chapters cover the solution of nonlinear equations (Chapter 2), numerical differentiation and integration (Chapter 3), numerical solution of ordinary differential equations (Chapters 4 and 5), numerical solution of partial differential equations using finite differences (Chapters 6, 8, and 9), iterative solution of systems of linear equations (Chapter 7), numerical solution of partial differential equations using the finite element method (Chapters 10 and 11), Fourier analysis of time series (Chapter 12), and interpolation of spatial data (Chapter 13). Each chapter uses one or more hydrological examples to illustrate methods as they are developed, and each of the chapter presents a set of homework problems, so readers can test their understanding of how to create code to solve problems.

The book derives from a set of lecture notes for a one-semester graduate course on applications of numerical methods in the hydrological sciences. The notes focus on the basic concepts, algorithms, and skills required for numerical computations and their application. Some background material and mathematical detail are presented in separate boxes; more often readers are directed to texts and papers where details can be found if needed. We recommend that the notes be used in conjunction with a "standard" numerical methods textbook to provide a more complete discussion of the mathematical underpinnings and considerations of the various methods.

The structure and length of the book are well suited to a semester-long course. Instructors can use it as the core of a course or as supplementary material in a broader course on numerical methods. Similarly, hydrology students might use it as a course text or as a supplement to a course with a less hydrological focus. Professionals in hydrology and other fields who want to learn numerical methods (or refresh their knowledge) via a self-study route will find that the notes provide a broad introduction to numerical methods with many examples and problems to illustrate and

[1] *MATLAB* is a registered trademark of The MathWorks, Inc.

provide experience with such methods. *MATLAB* codes for all example problems are provided; solutions to homework problems at the conclusion of each chapter are available in an appendix to the main text.

We thank the many students who took our course over the past decade for the comments that they made to help us improve the lecture notes. Our students tell us that the notes have proven very useful to them. We hope that they will prove to be as useful to others. Special thanks go to Jeffrey Raffensperger who pioneered the course with us in the 1990s.

George Hornberger
University of Virginia
Department of Environmental Sciences

Patricia Wiberg
University of Virginia
Department of Environmental Sciences

Numerical Methods in the Hydrological Sciences
Special Publication Series 57

doi:10.1029/057SP03

Introduction

In 1917, L.F. Richardson was attached to a French infantry division on the western front. As World War I raged about him, however, he took upon himself a scientific project of great complexity: to predict the weather over a section of western Europe for a six-hour period given measurements of winds, pressures, temperatures, etc. as a starting point. His numerical computation employed an approximation to the equations describing atmospheric dynamics and was done with the help of a slide rule, the only computational aid available at the time. The computation took six months to complete and was a spectacular failure. (See Hayes, 2001 for a delightful exposition of the story.) Failure or not, Richardson's basic plan was sound, as successful numerical weather prediction proved decades later, with the advent of digital computers.

Computation has been a staple of science, including hydrological science, from the very beginning. *Numerical* methods, whereby solutions to problems are obtained through approximations, are also venerable, with contributions from many famous mathematicians from about the 17th century onward. As was the case for numerical weather prediction, however, the widespread use and refinement of a variety of numerical methods applied to hydrological problems came with the development of digital computers.

Today quite sophisticated software is available commercially that allows hydrologists to solve problems and visualize results on a personal computer. Using such software often does not require much knowledge of how the codes work. However, familiarity with numerical methods (solution techniques and their implementation) can allow a person to modify existing software or to solve new problems that may arise in the course of a professional career. It also provides a solid conceptual basis for evaluating commercial software. Our view is that some experience with numerical methods, even for hydrologists who are "experimentalists" and not "theoreticians", is extremely valuable. For the more theoretically inclined, numerical methods are likely to figure prominently in their approach to solving hydrological problems.

The term "numerical methods" covers a very broad spectrum of topics. Some texts on numerical methods in hydrology focus exclusively on the solution of the partial differential equations governing flow of groundwater (e.g., Wang and Anderson 1982; Bear and Verruijt 1987). Other texts focus on models involving ordinary differential equations (e.g., Walker 1991). Data analysis, especially including analysis of time series and spatial data, is another area where a text on numerical methods in hydrology could be focused (e.g., Middleton 2000). And, of course, texts can "mix and match" (e.g., Carr 1995). We have chosen the latter course and cover in this book a series of topics that we deem of greatest importance given the constraints of covering them in a single semester.

When we first discussed how to structure the course from which these notes derive, we debated how much we could cover given that our students typically had limited programming experience. We finally decided to use *MATLAB* as the computational engine because we were convinced then that we could cover many more applications than we could if we used more standard programming languages while still giving students solid experience in writing code. We outlined a set of topics with which we thought all students should be acquainted.

The result was an ambitious undertaking for a one-semester course. But although students find our course challenging, they generally acknowledge at the completion of the term that they have met with success.

References

Bear, J. and A. Verruijt, *Modeling Groundwater Flow and Pollution*, 414pp., Dordrecht, Boston, 1987.

Carr, J.R., *Numerical Analysis for the Geological Sciences*, 592pp., Prentice-Hall, Englewood Cliffs NJ, 1995.

Hayes, B. The Weatherman. *American Scientist 89*: 10-14, 2001.

Middleton, G.V., *Data Analysis in the Earth Sciences using MATLAB*, 260pp., Prentice Hall, Saddle River NJ, 2000.

Walker, J.C.G., *Numerical Adventures With Geochemical Cycles*, 192pp., Oxford University Press, New York, 1991.

Wang, H.F. and M.P. Anderson, *Introduction to Groundwater Modeling: Finite Difference and Finite Element Methods*, 237pp., W.H. Freeman and Co., San Francisco, 1982.

George Hornberger
University of Virginia
Department of Environmental Sciences

Patricia Wiberg
University of Virginia
Department of Environmental Sciences

Numerical Methods in the Hydrological Sciences
Special Publication Series 57

doi:10.1029/057SP02

About the Authors

George M. Hornberger is Ernest H. Ern Professor of Environmental Sciences at the University of Virginia, where he has taught since 1970. His research focuses on understanding how hydrological processes affect the transport of dissolved and suspended constituents through catchments and aquifers. Throughout his career, Professor Hornberger has employed numerical analyses in his research and in his teaching. Hornberger received his B.S. and M.S. degrees from Drexel University in 1965 and 1967, respectively. In 1970, he received a Ph.D. in hydrology from Stanford University. He is co-author of two textbooks, including *Numerical Methods in Subsurface Hydrology* (John Wiley & Sons).

Patricia L. Wiberg is Professor of Environmental Sciences at the University of Virginia, where she has taught since 1990. Her research focuses on the mechanics of sediment erosion, transport and deposition, as well as associated evolution of sediment bed properties and morphology. Professor Wiberg's research frequently involves development and application of process-based numerical models and analysis of time series and other data. Wiberg received her B.A. in mathematics from Brown University in 1976 and an M.S. and Ph.D. in oceanography from the University of Washington in 1983 and 1987, respectively. Wiberg and Hornberger, along with Jeffrey Raffensperger and Keith Eshleman, are co-authors of *Elements of Physical Hydrology* (Johns Hopkins University Press).

Table of Contents

CHAPTER 1

Computation with *MATLAB*

Numerical Methods in the Hydrological Sciences
Special Publication Series 57

doi:10.1029/057SP06

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].

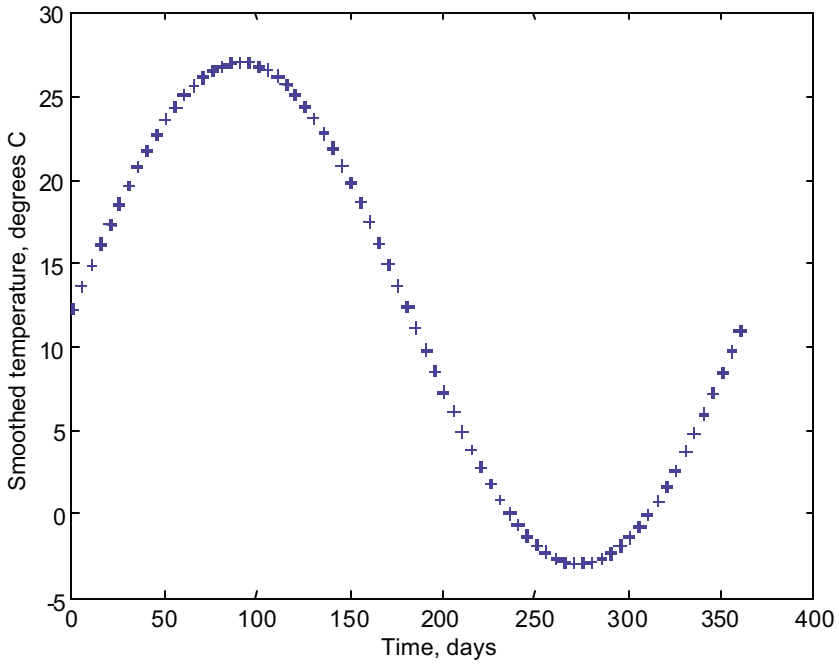


Figure 1.1. Example of a simple graph produced in *MATLAB*.

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')
```

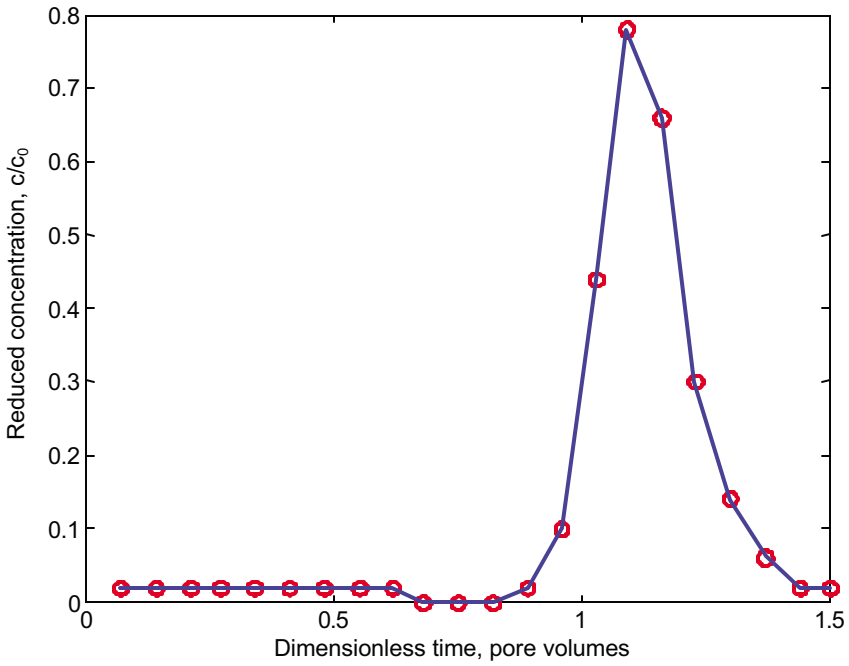


Figure 1.2. Plot of data from a laboratory column experiment.

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).

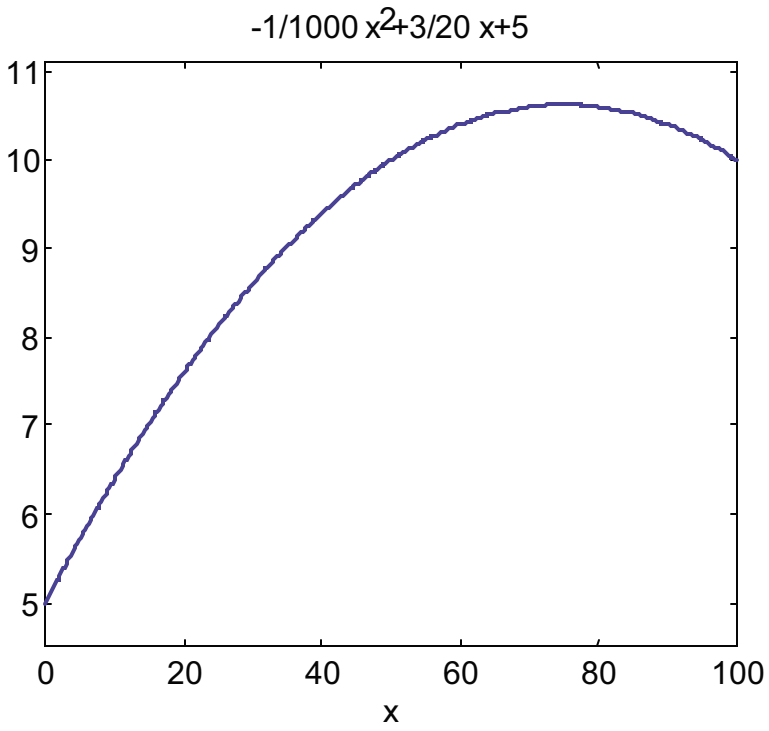


Figure 1.3. Solution to differential equation from the symbolic math toolbox.

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 $\sum c_i t_i / \sum c_i$. 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).

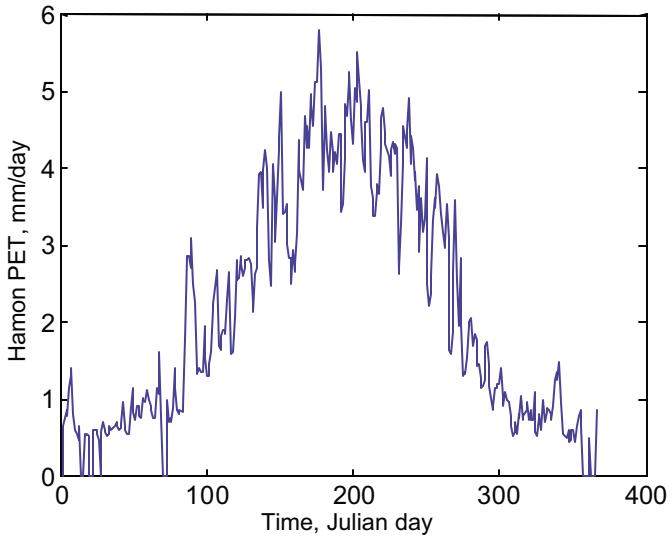


Figure 1.4. Potential evapotranspiration estimated using Hamon's method.

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):

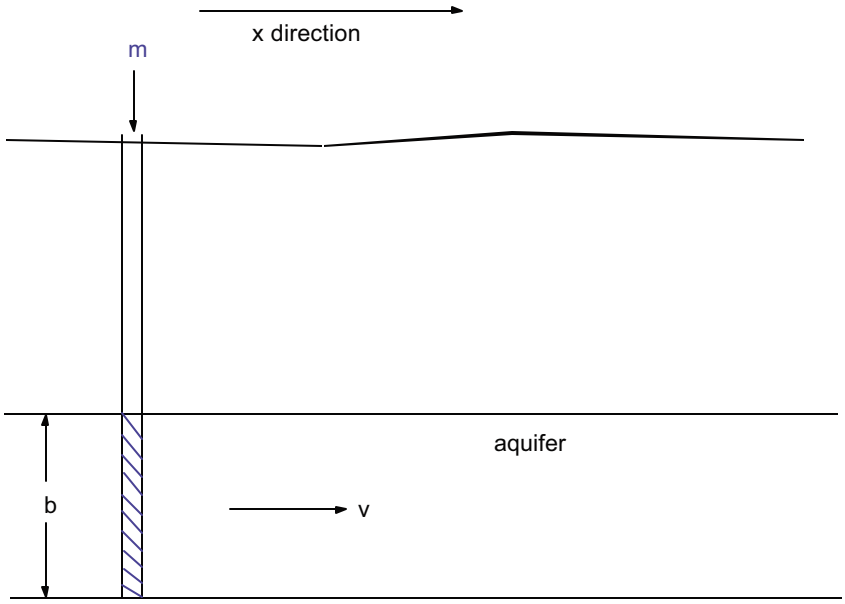


Figure 1.5. Schematic of contaminant pulse injection into an aquifer.

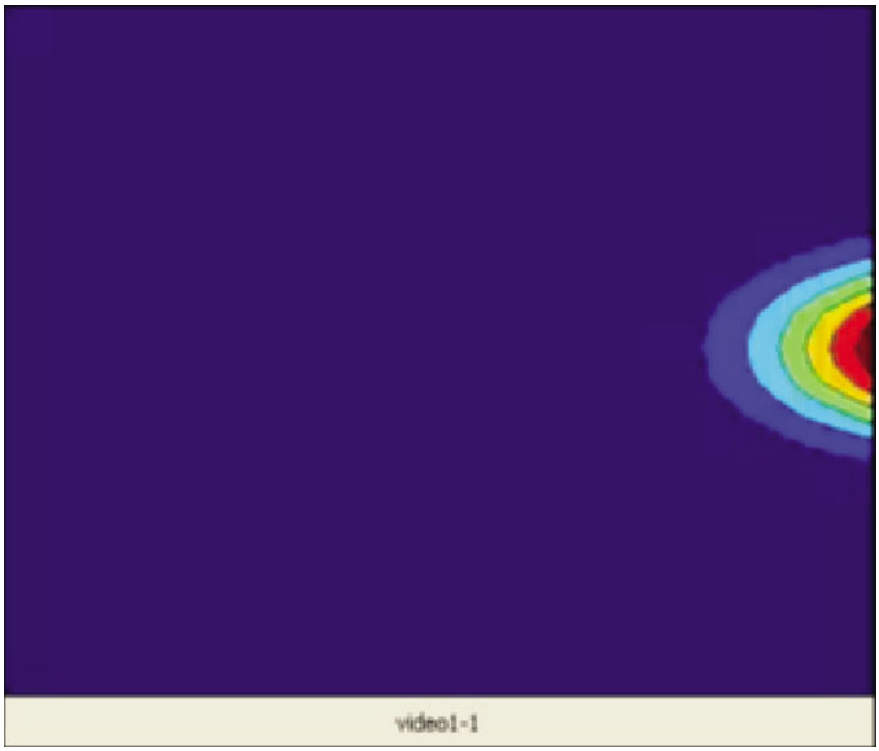
$$\text{Equation: } \frac{\partial c}{\partial t} = D_x \frac{\partial^2 c}{\partial x^2} + D_y \frac{\partial^2 c}{\partial y^2} - v\frac{\partial c}{\partial x}$$

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

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: $e_s = 0.2749 x 10^8 \exp\left[\frac{-4278.6}{T_t + 242.8}\right]$. H_t can be calculated by using the maximum number of daylight hours on day t, N_t, which is equal to $\frac{24\,\omega_s}{\pi}$, where ω_s is the sunset hour angle of day t, $\omega_s = \arccos(-\tan\phi \tan\delta)$, where φ is the latitude and δ is the solar declination given by $\delta = 0.4093 \sin(\frac{2\pi}{365} J - 1.405)$. On days when $T_t \leq 0$, Haith and Shoemaker set E=0.

CHAPTER 2

Solving Nonlinear Equations

Numerical Methods in the Hydrological Sciences
Special Publication Series 57

doi:10.1029/057SP07

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 $0 = mx + b\text{-}y$. Another common example is a quadratic equation

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

which has the well-known solution

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

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

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

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

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

2.2. Bisection

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

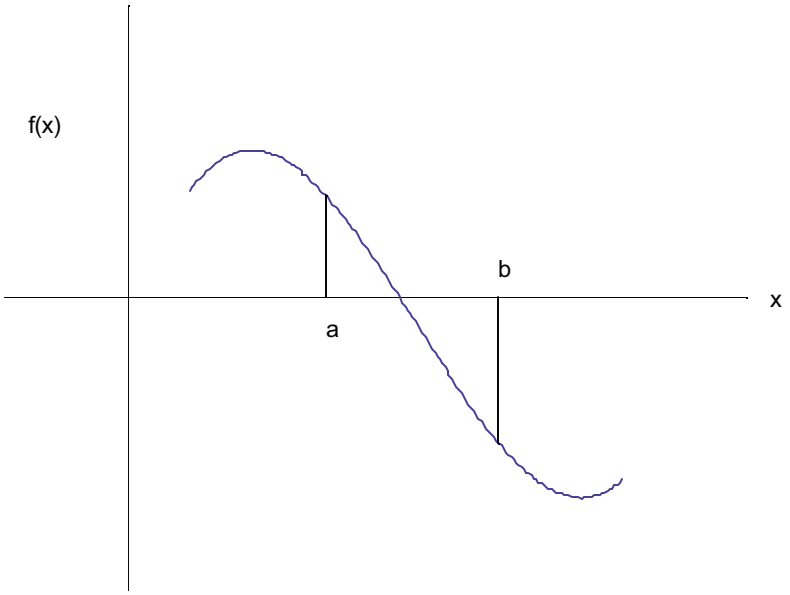


Figure 2.1. Values of f at $x=a$ and $x=b$ bracket a root of the function.

2.3. Newton's method and secant method

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

Newton's method (*Figure 2.2*)

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

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

Combining these, we get

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

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

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

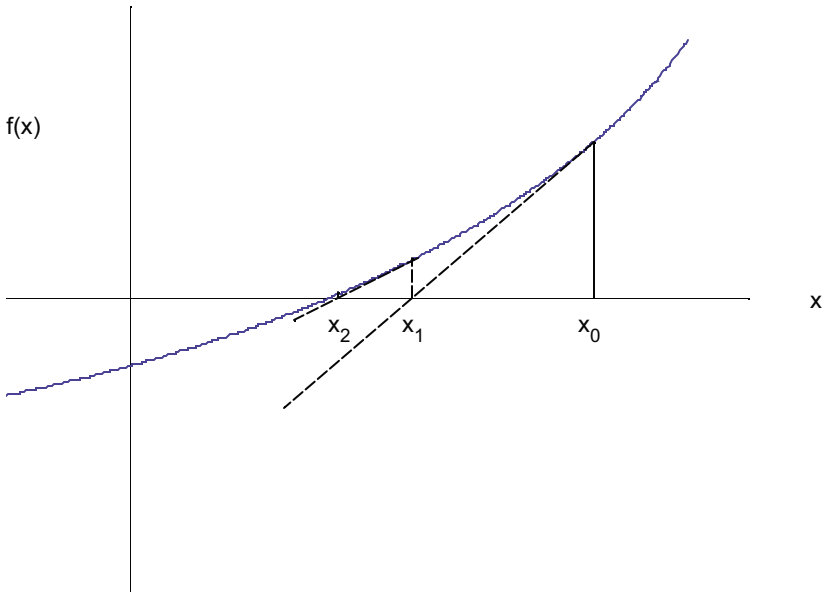


Figure 2.2. Schematic of Newton's method.

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.

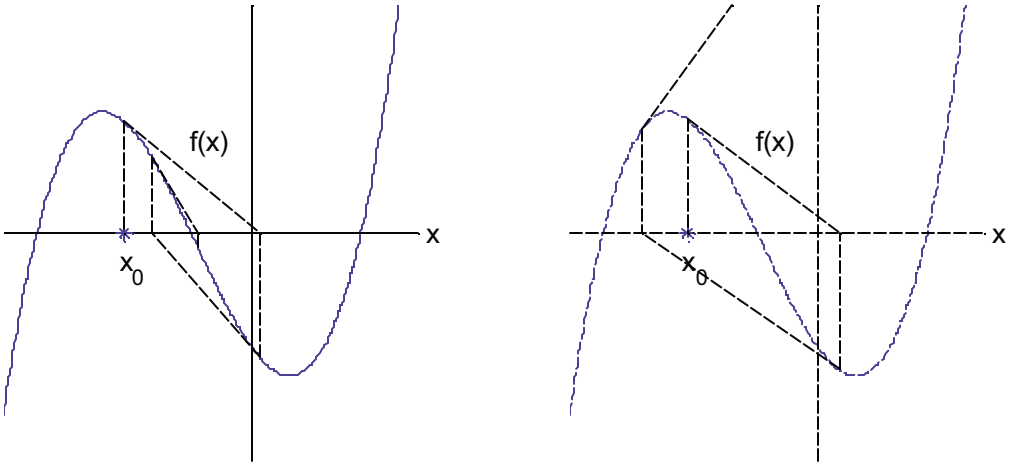


Figure 2.3. Newton's method may (left) or may not (right) converge to a root depending on the shape of the function and the proximity of the starting value to the root.

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.

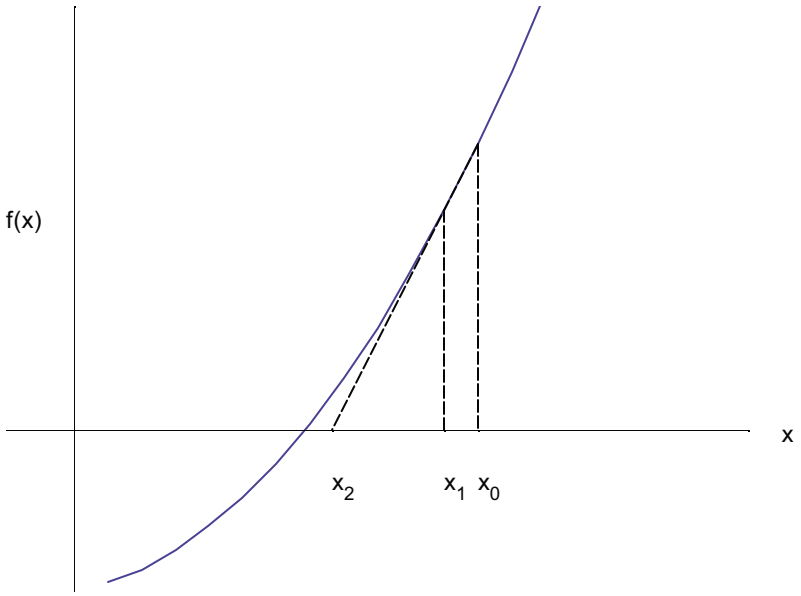


Figure 2.4. Schematic of the secant method.

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` (=2.2×10^{-16})). Including `trace` gives information for each iteration. `'function'` can be a *MATLAB* built-in function (e.g. `sin`) or you can define a function f by writing an m-file `f.m` including

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

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

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

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

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

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

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

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

2.5. Example: Leonardo of Pisa's polynomial

Consider the polynomial

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

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

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

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

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

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

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

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

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

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

returns

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

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

For initial values x_1=1.5 and x_2=1.0, the following bisection m-file gives the root x_3 = 1.368 835 449 to within an error of 1×10^{-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 $k = 2\pi/L$, $k_d = 2\pi/L_d$, and $L_d = gT^2/(2\pi)$. This equation is already in the form $x = g(x)$. If we plot $k_d h \tanh(kh)$ against kh (Figure 2.5), we get a nice graphical illustration of how fixed-point iteration works. A wave period of 12 s, yielding a deep water wavelength of 225 m ($k = 0.028$ 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.

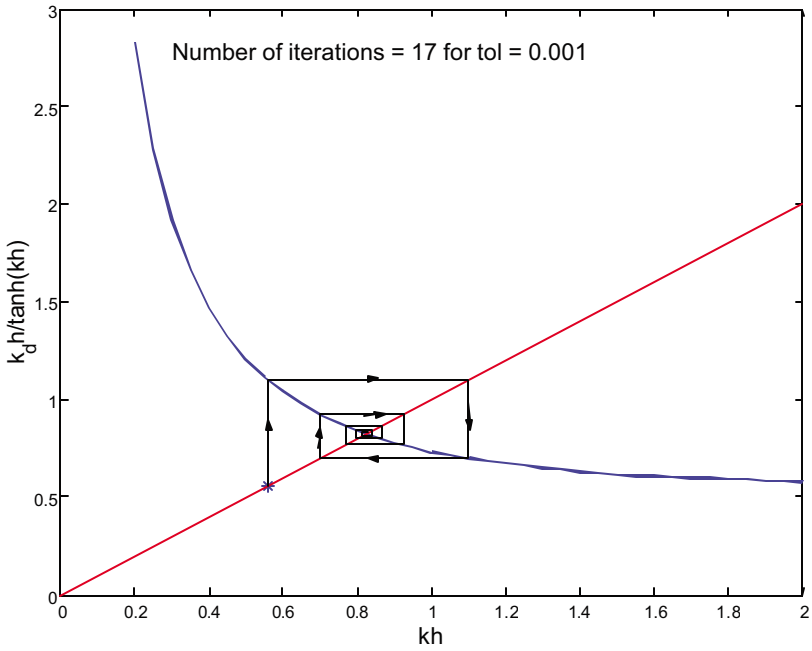


Figure 2.5. Graphical illustration of an iterative solution to $kh = k_d h \tanh(kh)$.

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.

$$ax + by = c$$
$$dx + ey = f$$

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

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

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

$$13x_1 - 8x_2 - 3x_3 = 20$$
$$-8x_1 + 10x_2 - x_3 = -5$$
$$-3x_1 - x_2 + 11x_3 = 0$$

For this set of equations

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

and

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

The augmented matrix is

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

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

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

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

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

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

$$x_1 - 0.61x_2 - 0.23x_3 = 1.54$$
$$x_2 - 0.56x_3 = 1.43$$
$$x_3 = 1$$

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

In *MATLAB*, we can set `A = [13 -8 -3;-8 10 -1;-3 -1 11]` and `b = [20;-5;0]`. Solving this with the *MATLAB* command `x=A\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 $4a$ and a carrying capacity of 1. Sequential calculations give the population evolution through time. For iterations that converge, the model represents a stable population. If the iteration diverges, the indication is that the modeled population is unstable.

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

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

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

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

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

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

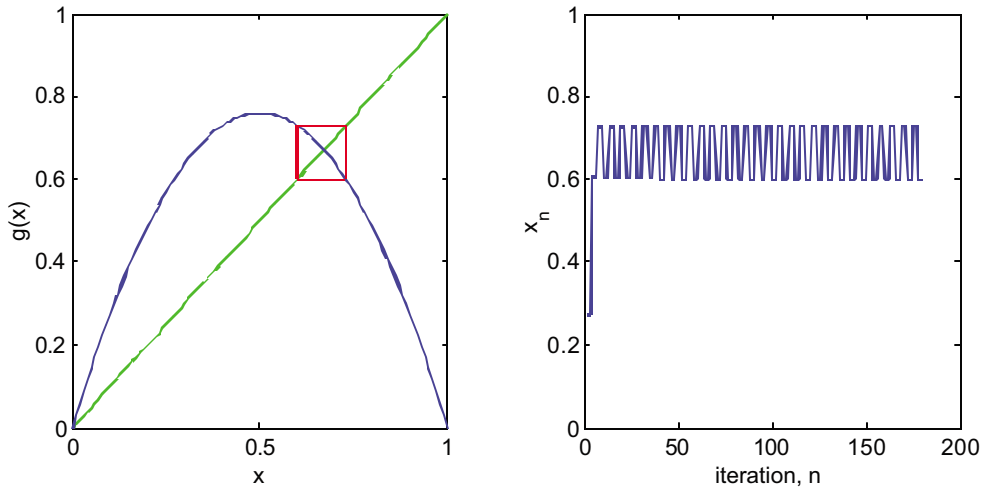


Figure B2.3.1. Results of function `hump.m` with $a=0.76$ illustrating cyclic behavior.

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

CHAPTER 3

Numerical Differentiation and Integration

Numerical Methods in the Hydrological Sciences
Special Publication Series 57

doi:10.1029/057SP08

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.

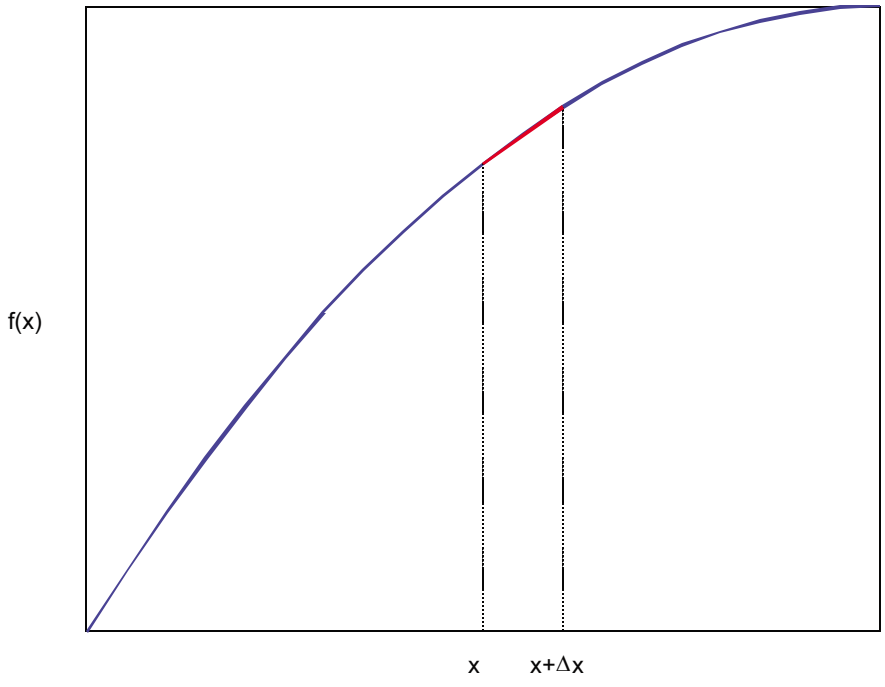


Figure 3.1. Finite difference approximation to a first derivative.

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

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

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 $f^{(n+1)}(\xi)$, 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 $O(\Delta x)$.

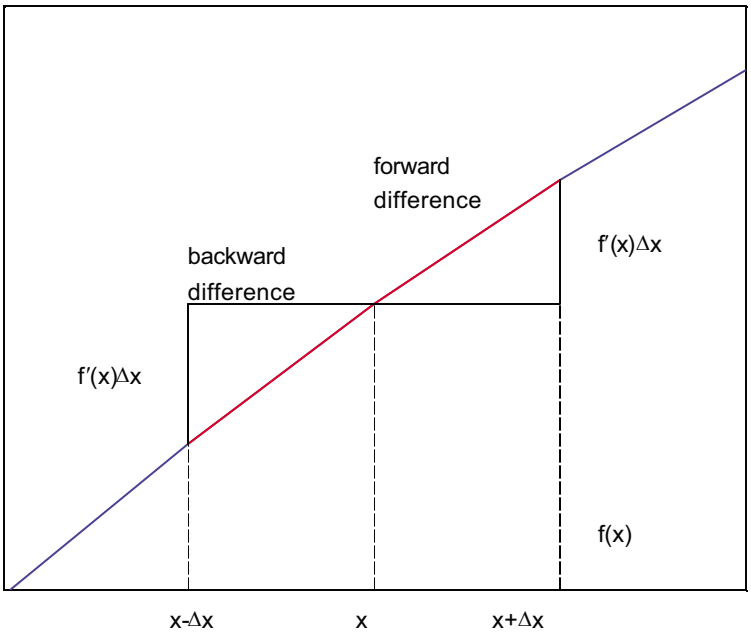


Figure 3.2. Forward and backward difference approximations to a first derivative.

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

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

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

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

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):

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

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')
```

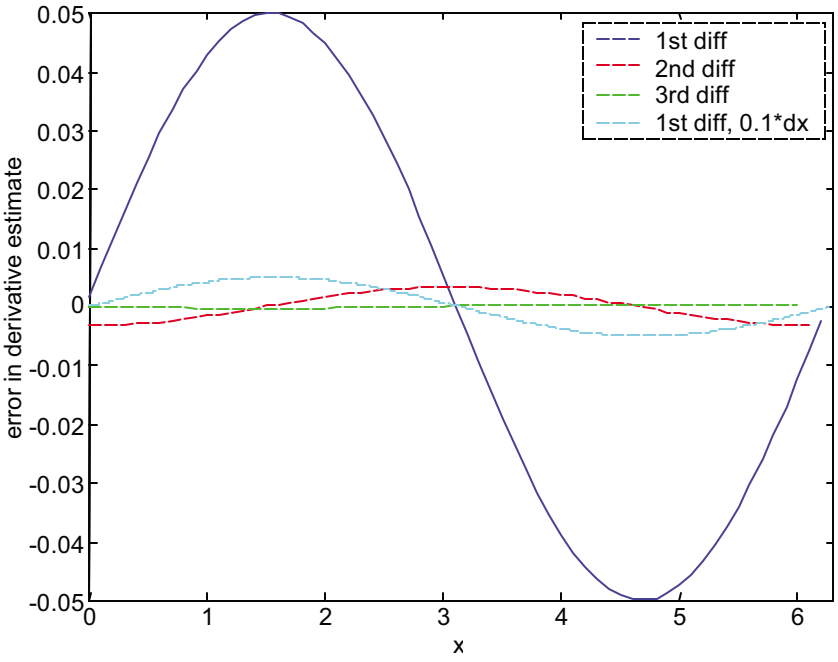


Figure 3.3. Accuracy of finite difference estimates of the derivative of sin(x).

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

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

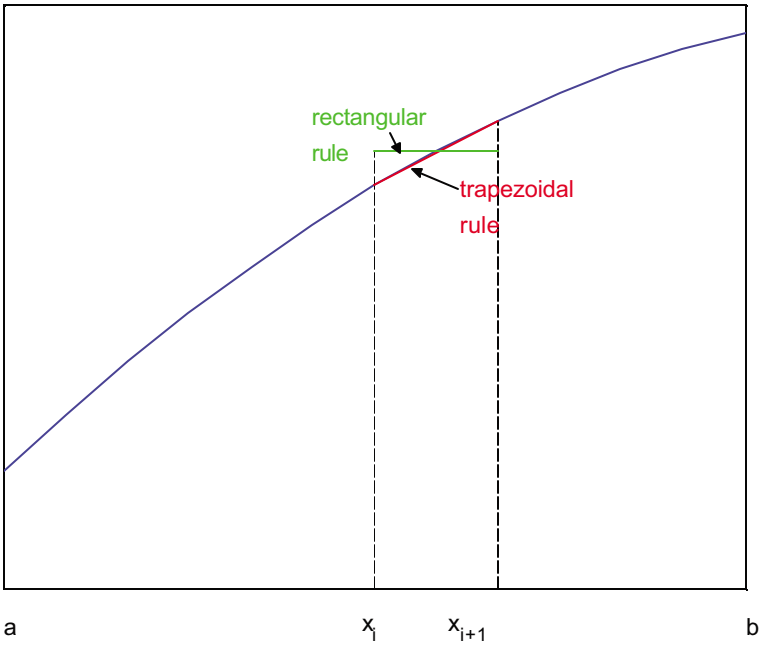


Figure 3.4. Trapezoidal and rectangular approximations of an integral.

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 $2n$-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

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

   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 Δx = 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,

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

   where the shear velocity $u_* = \sqrt{ghS}$, S is channel slope, $\kappa = 0.41$, 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 $< u >= u_*\kappa^{-1}[\ln(0.367h) - \ln(z_0)]$. 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 $u_*/(\kappa z)$. 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 $\int udz$ 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
Special Publication Series 57

doi:10.1029/057SP09

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, $\Delta\theta$, and the capillary pressure head at the wetting-front, ψ_f. The equation describing progress of the infiltration front is: $\frac{dL_f}{dt} = \frac{K_s}{\Delta\theta}\left(\frac{-\psi_f + L_f}{L_f}\right)$.
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, $\left(y_{i+1}-y_i\right)/\left(x_{i+1}-x_i\right)$ 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')`.

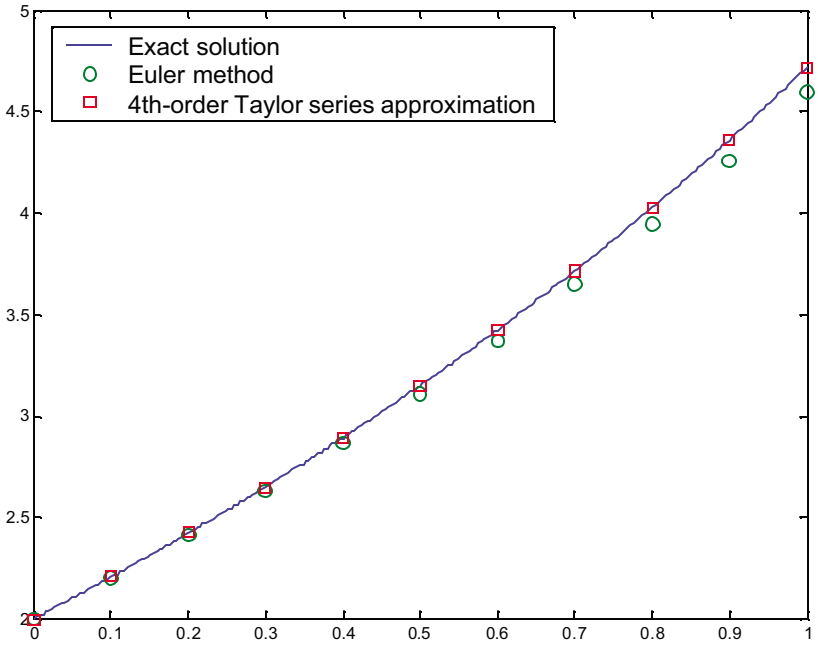


Figure 4.1. Example solution with the Euler and Taylor series methods for Δx=0.1.

```
% 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:

$$y' = y - x; \quad y'' = y' - 1; \quad y''' = y'' = y' - 1.$$

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.

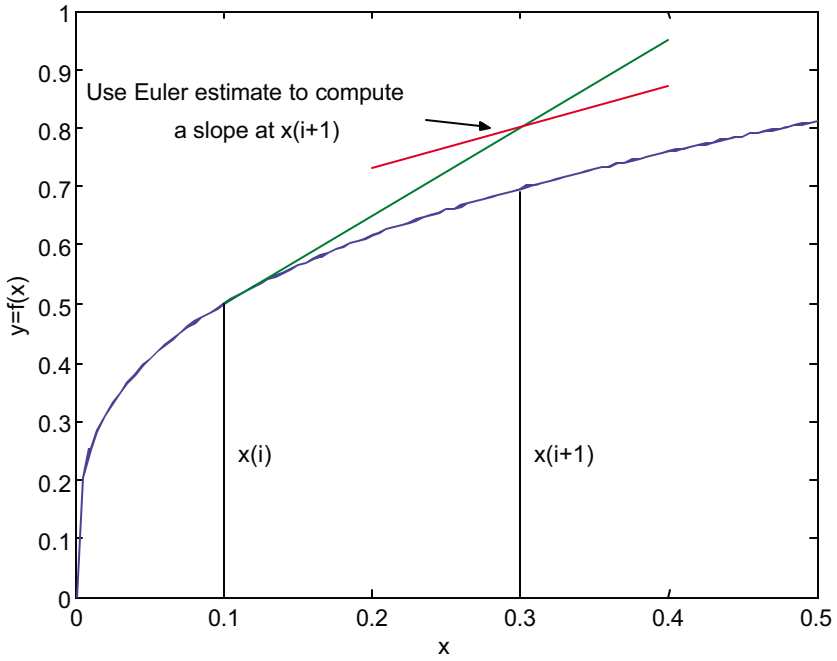


Figure 4.2. Schematic for modified Euler method.

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

$$k_1 = f(x_i, y_i)\Delta x$$
$$k_2 = f(x_i + \alpha\Delta x, y_i + \beta k_1)\Delta x$$

and a, b, α and β 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 α=β=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, α and β 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

$$f(x+\Delta x, y+\Delta y) = f(x,y) + f_x(x,y)\Delta x + f_y(x,y)\Delta y + \tfrac{1}{2}\left[f_{xx}(\Delta x)^2 + 2f_{xy}(\Delta x\Delta y) + f_{yy}(\Delta y)^2 \right] + \ldots$$

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

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

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

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

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'' 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, αb = 1/2, and βb = 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, β =

1 and α = 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

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

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

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

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 ρ, 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 ($\mathbf{R}_D = wD\rho/\mu < 0.5$), where D is sphere diameter and μ is fluid viscosity), $C_D = 24/\mathbf{R}_D$ and the terminal velocity, w_s, is given by Stokes law:

$$w_s = \frac{(\rho_s - \rho)\, gD^2}{18\mu}$$

where ρ_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), $\rho_s = 2650$ kg m^{-3}, $\rho = 1000$ kg m^{-3}, and $\mu = 0.001$ 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,w,'-b',t2,w2,'-r',t3,w3,'-g',max(t),ws,'*')
xlabel('t (s)'); ylabel('w (m/s)')
legend('ode23','ode45','ode23, rtol=1e-6','exact')
```

```
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.

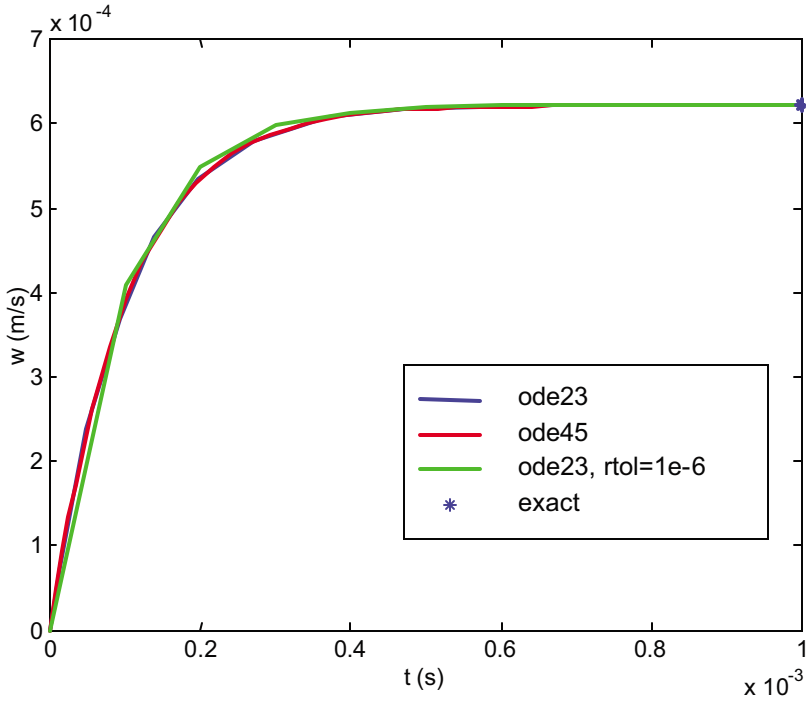


Figure 4.3. Solution to the example problem using *MATLAB* ode functions.

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 $\Delta x \rightarrow 0$. 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.

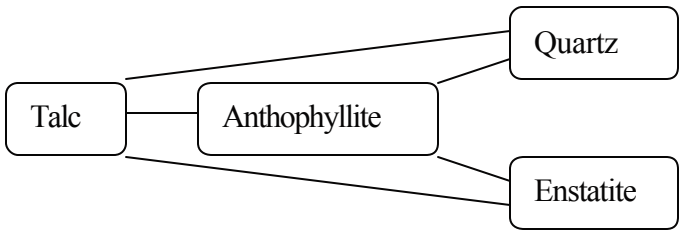
4.9. Systems of equations

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

$$y_1' = (1 - \alpha\, y_2)\, y_1$$
$$y_2' = (-1 + \beta\, y_1)\, y_2$$

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): $a_{11} = -18.0\times10^{-4}$ min^{-1}, $a_{21} = 11.0\times10^{-4}$ min^{-1}, $a_{22} = -5.9\times10^{-4}$ min^{-1}, $a_{31} = 3.9\times10^{-4}$ min^{-1}, $a_{32} = 4.8\times10^{-4}$ min^{-1}, $a_{41} = 2.1\times10^{-4}$ min^{-1}, $a_{42} = 0.5\times10^{-4}$ 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].

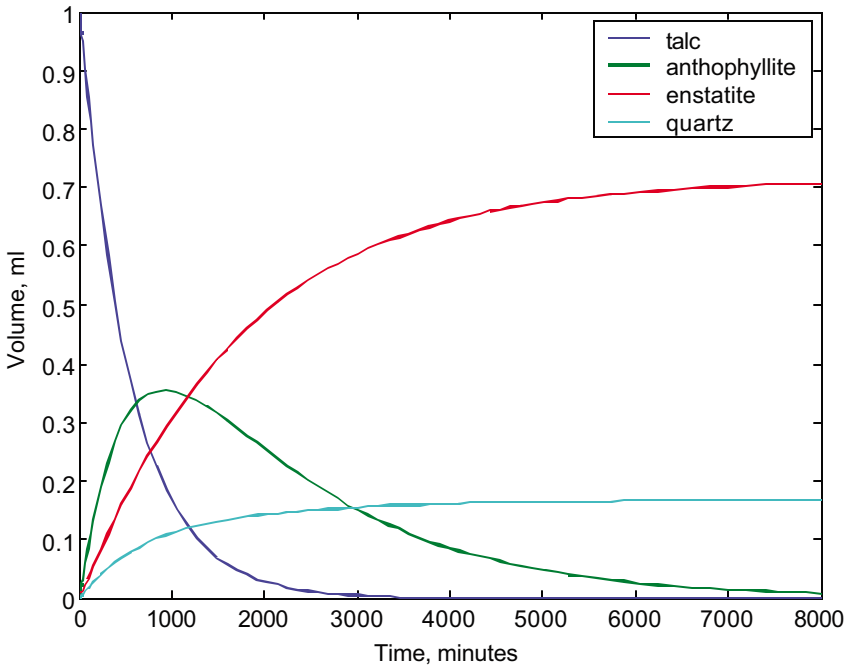


Figure 4.4. Results from *MATLAB* integration of talc equations.

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

$$\frac{dL_f}{dt} = \frac{K_s}{\Delta\theta}\left(\frac{-\psi_f + L_f}{L_f}\right).$$

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}, $\Delta\theta = 0.2$, and $\psi_f = -0.2$ m. Compare the accuracy of the methods. The analytical solution is:

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

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:

$$\frac{dP}{dt} = P - \alpha\frac{P^2}{H}$$
$$\frac{dH}{dt} = H - \beta PH$$

where P represents the population of parasites and H represents the population of hosts. Let α=2.5 and β=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

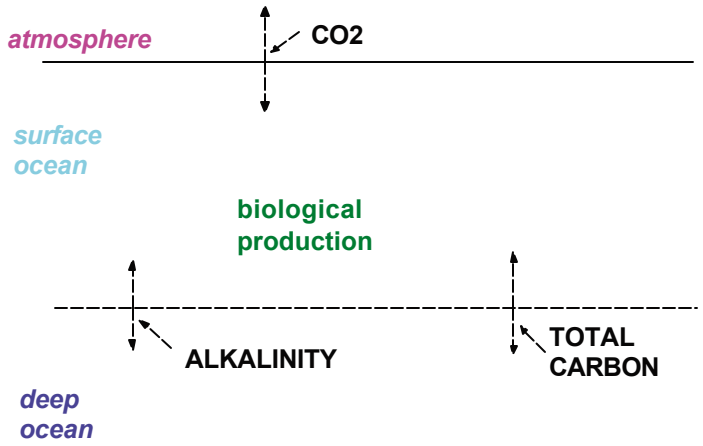
point discharge into a river with steady uniform flow is envisioned, a typical pattern of DO, called a "sag" is observed. Below the waste outfall, DO drops in response to the BOD. As the decomposition of the waste proceeds, the BOD is decreased and atmospheric reaeration replenishes the DO. DO decreases 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: $K_C=K_C(20)\times(1.024)^{(T-20)}$, $K_N=K_N(20)\times(1.024)^{(T-20)}$, and $K_a=K_a(20)\times(1.047)^{(T-20)}$. 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):

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

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

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

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

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

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

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 4.95×10^{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

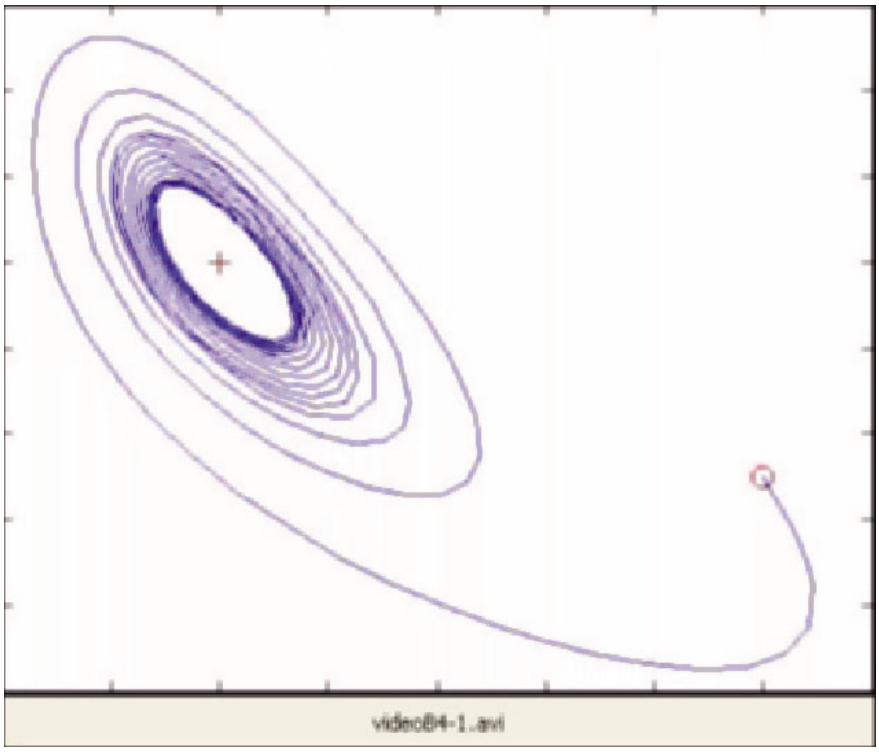
The following equations, defining what has been called the "brusselator" (Nicolis and Prigogine, 1977), serve as a model for certain self-catalyzing biochemical reactions.

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

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

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 $a \le x \le b$, 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 5

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

Numerical Methods in the Hydrological Sciences
Special Publication Series 57

doi:10.1029/057SP10

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 n^{th}-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 n^{th} 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

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

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

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

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

$$y'' = -\frac{1}{x} y' - \frac{(x^2 - \nu^2)}{x^2} y \cdot \qquad (5.3)$$

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 ν. For this example, we'll take $\nu = 0$, 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.

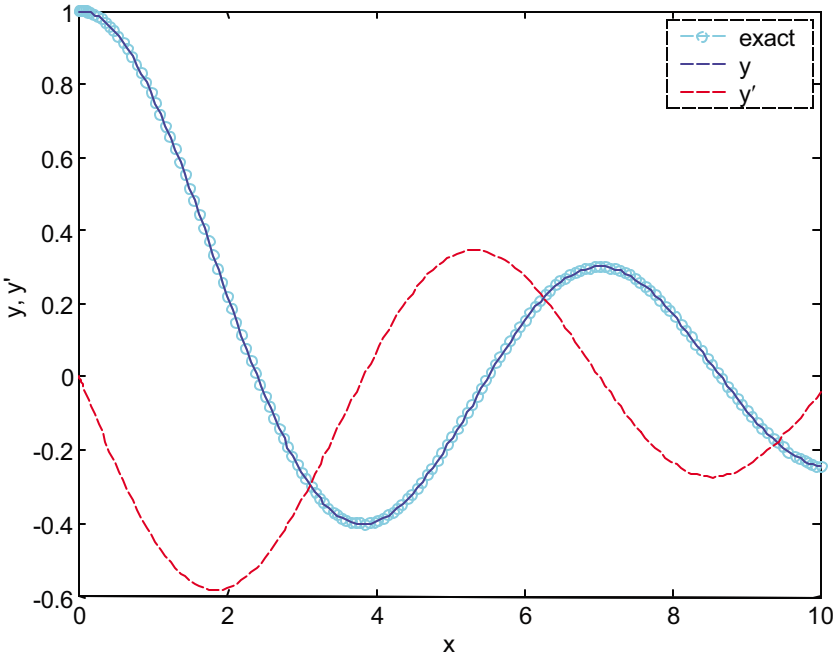


Figure 5.1. Solution to the Bessel equation of order 0 with specified initial values.

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.

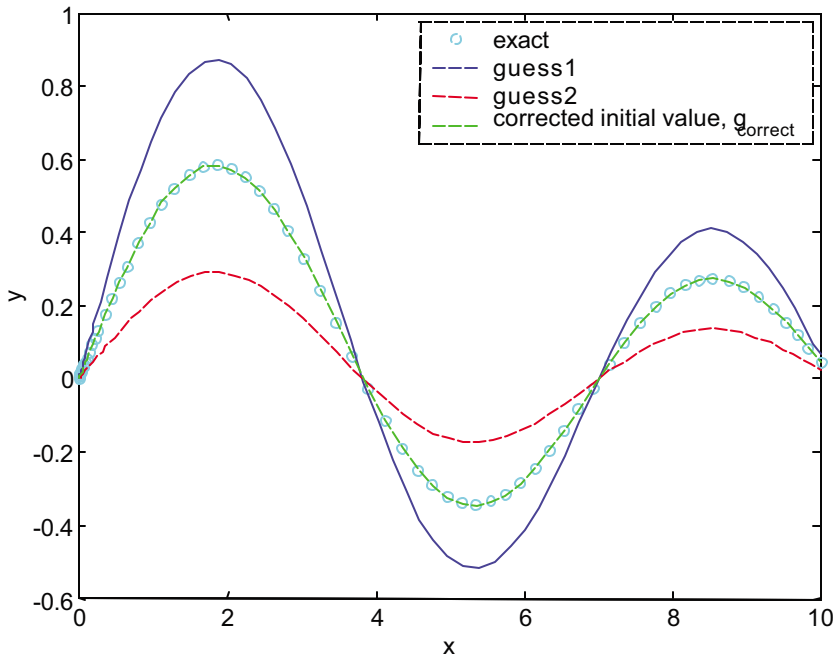


Figure 5.2. Solution to Bessel equation boundary value problem using the shooting method.

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+1 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, $\mathbf{A}y = b$, where $\mathbf{A}$ 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:

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

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

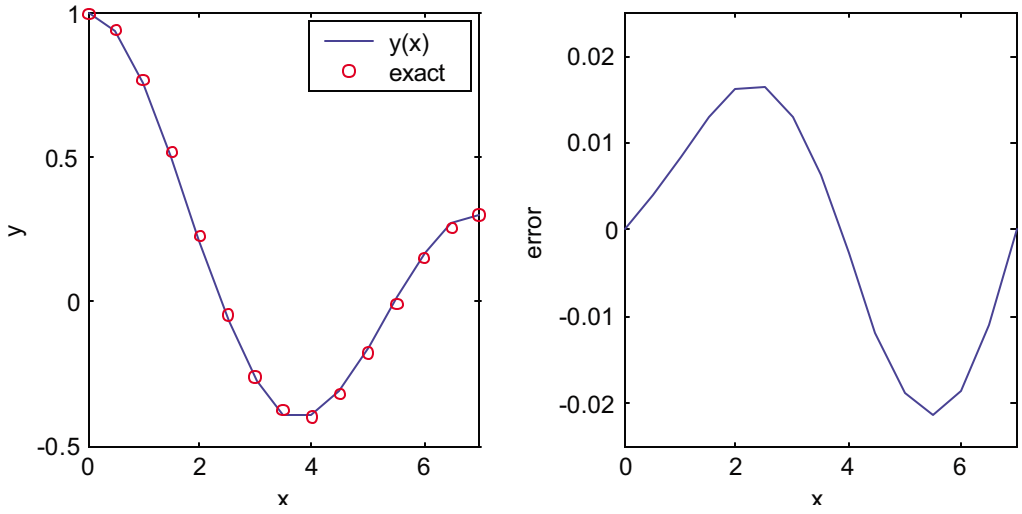


Figure 5.3. Finite difference solution and error for Δx=0.5.

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');
```

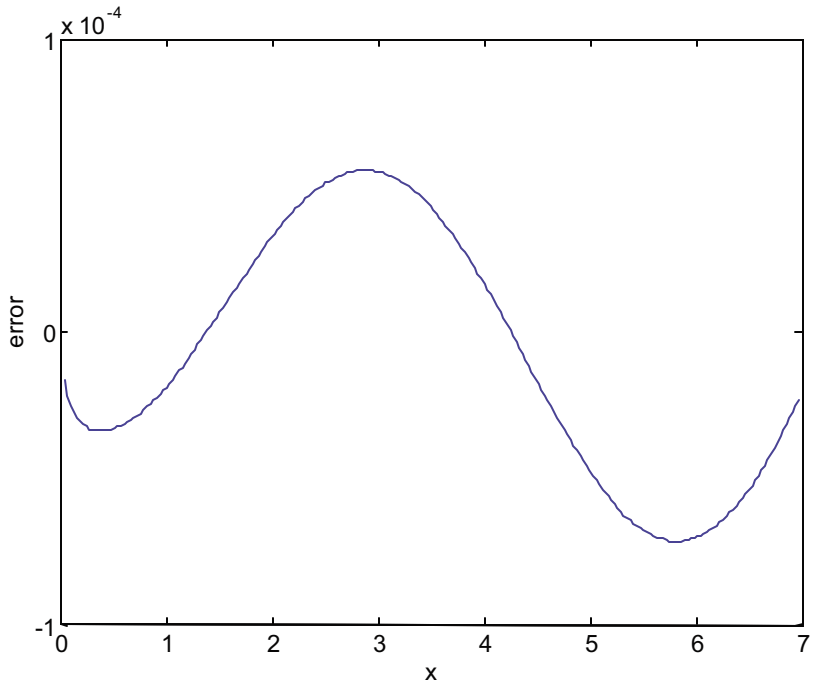


Figure 5.4. Error in finite difference approximation when Δx is reduced to 0.03.

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 $1/3^{rd}$ 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.

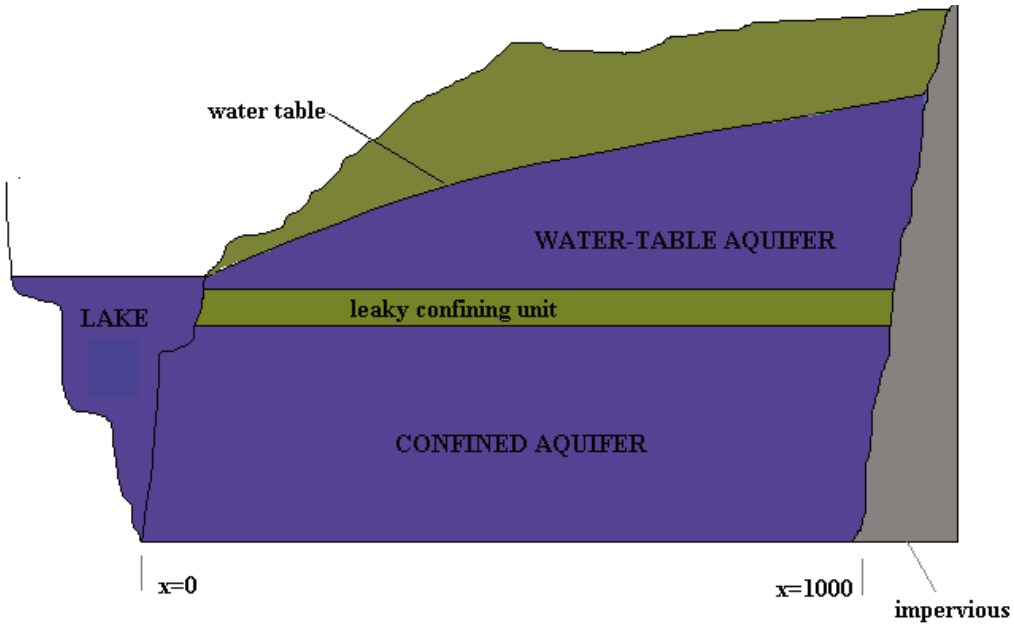
5.7. Problems

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 = l000m, head is maintained at 100m by a lake (see figure below). The other end of the aquifer (x = l000) 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.

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

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

   where $F_D = \frac{1}{2} C_D \rho v^2 A$, $v = \sqrt{v_x^2 + v_y^2}$, and θ is the angle between the x axis and the velocity vector. [Note that $\cos\theta$ and $\sin\theta$ 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 ρ 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

CHAPTER 6

Introduction to Finite Difference Methods for Partial Differential Equations

Numerical Methods in the Hydrological Sciences
Special Publication Series 57

doi:10.1029/057SP11

6. Introduction to Finite Difference Methods for Partial Differential Equations

6.1. Classification of partial differential equations

There is a "standard" classification of partial differential equations (PDE's). There is nothing sacred about this classification, and there is actually nothing essential in it that we will use. Nevertheless, the terminology is used widely so it pays to have a passing knowledge of the taxonomic jargon. Consider the following to be your introduction to some terminology.

Consider the second-order PDE with constant coefficients A, B, and C. ("D" represents some arbitrary function and is irrelevant to the classification.)

$$A\frac{\partial^2 u}{\partial x^2} + B\frac{\partial^2 u}{\partial x \partial y} + C\frac{\partial^2 u}{\partial y^2} + D\left(x, y, u, \frac{\partial u}{\partial x}, \frac{\partial u}{\partial y}\right) = 0.$$

The equation is:

elliptic, if $B^2-4AC<0$;

parabolic, if $B^2-4AC=0$;

hyperbolic, if $B^2-4AC>0$.

Three "general" equations from mechanics that often appear in the hydrological sciences are 1) Laplace's equation, 2) the heat equation, and 3) the wave equation. These equations are, respectively, elliptic, parabolic, and hyperbolic. Consider the definition of the classes above and convince yourself that the classifications are as indicated by completing Table 6.1. Also think about the one-dimensional forms of the Navier-Stokes and the advection dispersion equations listed in Table 6.1. The latter equations are presented using dimensionless variables. Note that the classification isn't particularly informative about these equations unless the second-order terms (the ones involving the Reynolds number, **R**, and Peclet number, **Pe**) dominate. That is, the classification "works" only for low Reynolds numbers and low Peclet numbers. (If you are unfamiliar with these equations or the dimensionless parameters, consult a text on fluid dynamics.)

6.2. Finite difference approximations for derivatives

The method known as the "finite-difference method" uses approximations to the partial derivatives in equations to reduce PDE's to a set of algebraic equations. Various approximations to derivatives are listed in Table 3.1. In particular, recall the approximations to first and second derivatives listed below (cf. equations 3.2 through 3.5).

Table 6.1. *Some equations that arise frequently in hydrological problems.*

EQUATION	A	B	C	B^2-4AC	CLASSIFICATION
Laplace's equation: $\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}=0$	1	0	1	-4	Elliptic
Heat equation: $\frac{\partial u}{\partial t}=\frac{\partial^2 u}{\partial x^2}$					
Wave equation: $\frac{\partial^2 u}{\partial t^2}=\frac{\partial^2 u}{\partial x^2}$					
Navier-Stokes: $\frac{\partial u}{\partial T}+u\frac{\partial u}{\partial X}=-\frac{1}{\mathbf{F}}-\frac{P_0}{\rho U_0^2}\frac{\partial p}{\partial X}+\frac{1}{\mathbf{R}}\frac{\partial^2 u}{\partial X^2}$					
Advection-dispersion: $\frac{\partial c}{\partial T}+\frac{\partial c}{\partial X}=\frac{1}{\mathbf{Pe}}\frac{\partial^2 c}{\partial X^2}$					

The forward-difference approximation to a first partial derivative with respect to x at point x_i, y_j:

$$\frac{\partial f(x_i,y_j)}{\partial x}=\frac{f(x_i+\Delta x,y_j)-f(x_i,y_j)}{\Delta x}+O(\Delta x)$$
$$=\frac{f_{i+1,j}-f_{i,j}}{\Delta x}+O(\Delta x)$$

The backward-difference approximation to a first derivative:

$$\frac{\partial f(x_i,y_j)}{\partial x}=\frac{f(x_i,y_j)-f(x_i-\Delta x,y_j)}{\Delta x}+O(\Delta x)$$
$$=\frac{f_{i,j}-f_{i-1,j}}{\Delta x}+O(\Delta x)$$

The central-difference approximation to a first derivative:

$$\frac{\partial f(x_i,y_j)}{\partial x}=\frac{f(x_i+\Delta x,y_j)-f(x_i-\Delta x,y_j)}{\Delta x}+O(\Delta x)^2$$
$$=\frac{f_{i+1,j}-f_{i-1,j}}{2\Delta x}+O(\Delta x)^2$$

A central-difference approximation to the second derivative:

$$\frac{\partial^2 f}{\partial x^2} = \frac{f(x_i + \Delta x, y_j) - 2f(x_i, y_j) + f(x_i - \Delta x, y_j)}{(\Delta x)^2} + O(\Delta x)^2$$
$$= \frac{f_{i+1,j} - 2f_{i,j} + f_{i-1,j}}{(\Delta x)^2} + O(\Delta x)^2$$

Similar equations give the approximations for partial derivatives with respect to y.

Substitution of these approximations for the derivatives in a PDE leads to a set of algebraic equations. The numerical solution to the PDE is recovered by solving the algebraic equations.

6.3. Example: The Laplace equation

Steady-state flow of groundwater in a homogeneous, isotropic, constant-thickness, horizontal aquifer is governed by the Laplace equation [Box 6.1]. The equation, in terms of groundwater head, h, is

$$\frac{\partial^2 h}{\partial x^2} + \frac{\partial^2 h}{\partial y^2} = 0$$

Consider the Laplace equation applied to a rectangular section of aquifer with Dirichlet boundary conditions (which just means that the heads on the boundaries are specified). Let the aquifer be 400m long in the x direction and 200m long in the y direction. Apply a grid with Δx=50m and Δy=50m and use i and j to index the position on the grid superimposed on the domain. The heads on the boundary are fixed at 100m along the right side of the boundary and 0 elsewhere (Figure 6.1).

The substitution of central-difference approximations to the derivatives in the Laplace equation for a general node (i,j) leads to:

$$\frac{h_{i+1,j} - 2h_{ij} + h_{i-1,j}}{(\Delta x)^2} + \frac{h_{i,j+1} - 2h_{ij} + h_{i,j-1}}{(\Delta y)^2} = 0 \qquad (6.1)$$

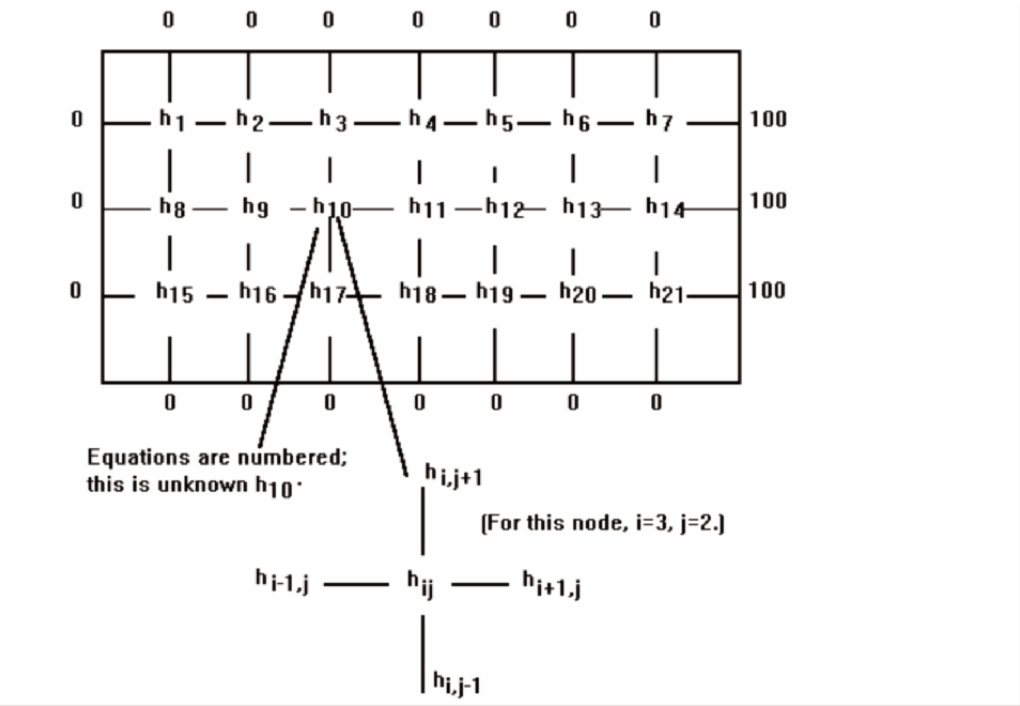


Figure 6.1. Finite difference grid for the Laplace equation.

Let Δd represent both Δx and Δy, which we specified were equal at 50m. Equation (6.1) can then be written:

$$\left(\frac{1}{(\Delta d)^2}\right)\left(h_{i+1,j} + h_{i-1,j} + h_{i,j+1} + h_{i,j-1} - 4h_{ij}\right) = 0 \qquad (6.2)$$

Now consider the equation for node 1 of Figure 6.1. It is clear from the problem that we have 21 unknowns (7 "i" nodes times 3 "j" nodes). We number the nodes and use a single subscript on h to designate which unknown we are considering (Figure 6.1). Equation (6.2) for node 1 indicates

$$\frac{1}{2500}(h_2 + 0 + h_8 + 0 - 4h_1) = 0. \qquad (6.3)$$

We can write an equation like this for each of the 21 nodes to get a system of 21 equations in 21 unknowns. The first of these equations (for h_1) shows a -4 as the coefficient on the first unknown and ones for the second and eighth unknown. Write the equation for the second node. Your result should show a -4 coefficient for the second unknown and ones for the first, third, and ninth. If we write all of the equations and put them in matrix-vector form, the result is a set of simultaneous linear equations.

Solving the example problem in MATLAB

We have already seen how to solve systems of equations in *MATLAB* (Chapter 2.9). All that must be done is to specify the matrix and right-hand vector. One way that comes immediately to mind is to write an m-file that loops through the 21x21 elements of the matrix. Such a file might look like this. (If you haven't already used all of the *MATLAB* logical operations, "= =" indicates "equal to", "~=" indicates "not equal to".)

```
A=zeros(21,21);          %preallocate space for the matrix
for k=1:21               % 21 rows
      for n=1:21         % 21 columns
            A(k,n)=0;
            if k==n A(k,n)=-4;end
            if k==n-1 & n~=8 & n~=15 A(k,n)=1;end
            if k==n+1 & k~=8 & k~=15 A(k,n)=1;end
            if k==n-7 A(k,n)=1;end
            if k==n+7 A(k,n)=1;end
      end
end  ;
```

The result in *MATLAB* is (for the first 9 rows and columns)

```
A(1:9,1:9)

ans =
    -4     1     0     0     0     0     0     1     0
     1    -4     1     0     0     0     0     0     1
     0     1    -4     1     0     0     0     0     0
     0     0     1    -4     1     0     0     0     0
     0     0     0     1    -4     1     0     0     0
     0     0     0     0     1    -4     1     0     0
     0     0     0     0     0     1    -4     0     0
     1     0     0     0     0     0     0    -4     1
     0     1     0     0     0     0     0     1    -4
```

which is the desired result.

There are very good reasons to avoid this "brute-force" method for setting up finite-difference matrices. First, *MATLAB* is very efficient at dealing with vector operations and not very efficient at dealing with "for loops". (If you do want to perform loop calculations such as those shown above, make sure to preallocate space for the matrix, i.e., to use `A=zeros(21,21)` before starting the loop.) More importantly, the matrices can become large rather quickly. For the simple example above, we had 21 unknowns leading to a matrix with 21 rows and 21 columns, designated a 21x21 matrix. In this case the matrix has 441 entries. But if we want to halve the grid spacing in the example problem, we wind up with 15x7=105 unknowns and our matrix is 105x105, and has 11025 entries. And this isn't a large problem at all. How do we recover from the difficulty of working with large numbers of matrix entries? If we look at the matrix for our example problem, we get a clue. *Most of the entries in the matrix are zero*. Matrices that arise in finite-difference (and finite-element) methods are *sparse;* they have many zero elements. If we can take advantage of the

sparseness and store only non-zero entries, we save tremendously. *MATLAB* does this through a series of sparse matrix commands as indicated below[1].

```
help sparse

 SPARSE     Build sparse matrix from nonzeros and indices.

      S = SPARSE(...) is the built-in function which generates matrices
      in MATLAB's sparse storage organization.  It can be called with
      1, 2, 3, 5 or 6 arguments.

      S = SPARSE(X) converts a sparse or full matrix to sparse form by
      squeezing out any zero elements.

      S = SPARSE(i,j,s,m,n,nzmax) uses the rows of [i,j,s] to generate
      an m-by-n sparse matrix with space allocated for nzmax nonzeros.
      The two integer index vectors, i and j, and the real or complex
      entries vector, s, all have the same length, nnz, which is the
      number of nonzeros in the resulting sparse matrix S .

      There are several simplifications of this six argument call.

      S = SPARSE(i,j,s,m,n) uses nzmax = length(s).

      S = SPARSE(i,j,s) uses m = max(i) and n = max(j).

      S = SPARSE(m,n) abbreviates SPARSE([],[],[],m,n,0).  This
      generates the ultimate sparse matrix, an m-by-n all zero matrix.

      The argument s and one of the arguments i or j may be scalars,
      in which case they are expanded so that the first three arguments
      all have the same length.

      For example, this dissects and then reassembles a sparse matrix:

                 [i,j,s] = find(S);
                 [m,n] = size(S);
                 S = sparse(i,j,s,m,n);

      So does this, if the last row and column have nonzero entries:

                 [i,j,s] = find(S);
                       S = sparse(i,j,s);

      All of MATLAB's built-in arithmetic, logical and indexing operations
      can be applied to sparse matrices, or to mixtures of sparse and
      full matrices.  Operations on sparse matrices return sparse matrices
      and operations on full matrices return full matrices.  In most cases,
      operations on mixtures of sparse and full matrices return full
      matrices.  The exceptions include situations where the result of
      a mixed operation is structurally sparse, eg.  A .* S is at least
      as sparse as S .  Some operations, such as S >= 0, generate
```

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

```
    "Big Sparse", or "BS", matrices -- matrices with sparse storage
    organization but few zero elements.

    See also FULL, FIND and the sparfun directory.
```

Let's use the sparse matrix commands to build the matrix for the example problem. First we want to have a 21x21 square matrix with values of –4 on the main diagonal (the entries running from the top left to the bottom right of the matrix).

```
M_diag=sparse(1:21,1:21,-4,21,21);
```

Next we build the subdiagonal, the line of entries directly below the main diagonal.

```
L1_diag=sparse(2:21,1:20,1,21,21);
```

Finally, we build the diagonal that is 7 elements down from the main diagonal. These are the entries that arise from the line of nodes below the one being considered, e.g., h_8 occurs in the first equation (equation 6.3 and Figure 6.1).

```
L2_diag=sparse(8:21,1:14,1,21,21);
```

Note that we do not have to build the super diagonals, the ones above the main diagonal, explicitly --- they are just the transpose of the subdiagonals. The matrix that we want (actually, we will have to make a minor modification; see below) is just the sum of the appropriate diagonal matrices.

```
A=M_diag+L1_diag+L2_diag+L1_diag'+L2_diag';
```

How big is our sparse matrix? `size(A)`

```
ans =

          21    21
```

What does the matrix look like? `A(1:9,1:2)`

```
ans =
   (1,1)       -4
   (2,1)        1
   (8,1)        1
   (1,2)        1
   (2,2)       -4
   (3,2)        1
   (9,2)        1
```

You see that the only entries listed are non zero. The *MATLAB* "full" command can be used to convert a sparse matrix to the regular form.

```
full(A(1:9,1:9))

ans =
    -4     1     0     0     0     0     0     1     0
     1    -4     1     0     0     0     0     0     1
     0     1    -4     1     0     0     0     0     0
     0     0     1    -4     1     0     0     0     0
```

```
     0     0     0     1    -4     1     0     0     0
     0     0     0     0     1    -4     1     0     0
     0     0     0     0     0     1    -4     1     0
     1     0     0     0     0     0     1    -4     1
     0     1     0     0     0     0     0     1    -4
```

The matrix is not quite right yet. The matrix should be "blocked" with the limits of the "blocks" corresponding with the ends of the rows of the finite-difference grid. (Figure 6.2 shows that the matrix is composed of three 7x7 "blocks" that are identical.) The blocks arise because there is no "one" in the position to the right of the rightmost point on the row (e.g., variable 8 does not appear in the equation for node 7). Likewise, there is no "one" in the position to the left of the leftmost position in a row (e.g., variable 14 does not appear in the equation for node 15). Thus, we need to set several elements in our sparse matrix to zero.

```
A(7,8)=0;A(8,7)=0;A(14,15)=0;A(15,14)=0;
```

We can use the *MATLAB* "spy" command to look at the structure of matrix A graphically (Figure 6.2).

```
spy(A)
```

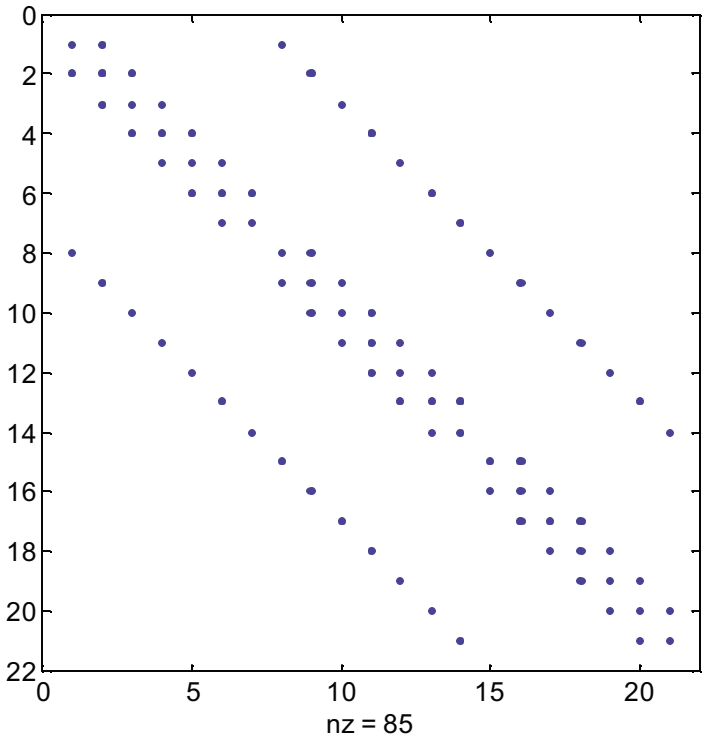


Figure 6.2. The sturcture of matrix A. The number of non-zero entries in the matrix is nz.

The final step, solving the equations, can be done using methods presented in Chapter 2. For example, we can set the right-hand vector using known quantities (the heads on the boundaries) and obtain the solution easily using the *MATLAB* backslash command

```
b=zeros(21,1);b(7)=-100;b(14)=-100;b(21)=-100 ;
```

```
h=A\b;
output=[h(1:7) h(8:14) h(15:21)]

output =
    0.3530    0.4989    0.3530
    0.9132    1.2894    0.9132
    2.0103    2.8324    2.0103
    4.2957    6.0194    4.2957
    9.1532   12.6538    9.1532
   19.6632   26.2894   19.6632
   43.2101   53.1774   43.2101
```

You might want to prepare an m-file, generalizing the statements above, to solve the sample problem for user-specified grid spacing and look at the effect of decreasing grid spacing on the answers.

6.4. Example problem: The "Tóth" problem

As an illustration of how to generalize the approach outlined above, consider the m-file below to solve the Laplace equation for a two-dimensional, vertical slice of an aquifer with a top boundary specified by undulating values of ground-water head. The undulations are to represent the effects of topography. Tóth (1963) investigated the problem in a now classic paper.

The boundary conditions for the left and right sides and for the bottom of the domain are not fixed heads. Rather, we assume that there is no flow across these boundaries. In this case the derivative of h with respect to the spatial coordinate is zero. We handle the condition by using the central difference approximation to the derivative at the boundary node. Consider the approximation for a boundary node "i" in a grid (Figure 6.3).

Equation (3.4), the central-difference approximation for the first derivative, requires values at "i-1" as well as at "i+1". That is,

$$\frac{\partial h}{\partial x} \approx \frac{\left(h_{i+1} - h_{i-1}\right)}{2\Delta x}$$

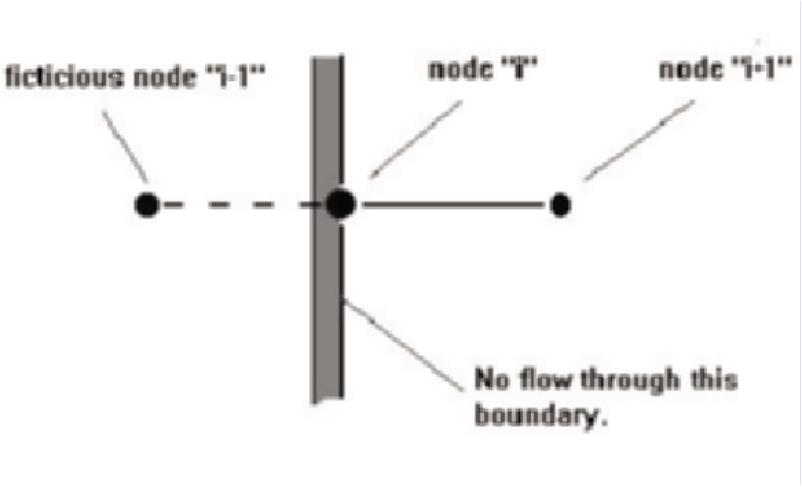


Figure 6.3. Nodes in the vicinity of a no-flow boundary.

In reality, we don't have a node at "i-1", but we insert a fictitious node just to allow us to treat the boundary condition. For no flow, the derivative above must be zero and this implies that h_{i-1} must equal h_{i+1}. Consequently, when h_{i-1} occurs in the finite-difference equations, we set it to h_{i+1}. In effect, this means that the coefficients on the "h_{i+1}'s" in the Tóth problem must be set to 2 instead of 1. (The best way for you to digest this is to study the code and think about the appropriate boundary conditions.)

```
% This is a finite-difference solution for Toth's problem.
% Code by George Hornberger and Jeff Raffensperger
%
% First set the grid size.
%
J=input('Number of nodes in the x direction? (Try 75 if in doubt.)')
K=input('Number of nodes in the y direction? (Try 25.) ')
W=input('Number of hill-valley waves? (Try 3.) ')
amp=input('Amplitude of hill-valley waves (0-10)? (Try 3.) ')
%
% The number of nodal points is J*K.
%
n=J*K;
%
% Finite-difference equations generate SPARSE matrices;
% that is, most entries are zero.
% MATLAB takes advantage of this sparsity by NOT storing the entire matrix.
%
% Set the coefficient matrix for the Laplace equation.
%
M_DIAG=sparse(1:n,1:n,-4,n,n);
L1_DIAG=sparse(2:n,1:n-1,1,n,n);
L2_DIAG=sparse(J+1:n,1:n-J,1,n,n);
A=L2_DIAG+L1_DIAG+M_DIAG+L1_DIAG'+L2_DIAG';
%
% Next set the vector of "knowns" and modify the coefficient matrix
```

```
% to account for the boundary conditions.
% The heads at the top are set to grade linearly from 20 to 0.2,
% with W sine waves of amplitude A superimposed.
%
dh=20/J;
hT=20:-dh:0.2;
%
for i=1:J
      hT(i)=hT(i)+amp*sin(2.*pi*((i-1)/(J/W)));
end
%
% Let's look at the topography of the water table...
%
dJ=0:1:J-1;
dK=0:1:K-1;
figure(1)
plot(dJ,hT)
%
% Next set the right-hand vector, adjust the coefficients on the no-flow
boundaries, and fix the matrix entries at the edges of the "blocks".
%
rhs=zeros(n,1);
for j=1:J                       % Bottom and top boundaries
    rhs((K-1)*J+j)=-hT(j);
    A(j,j+J)=2;
end
for k=1:K                       % Right-hand boundary
    A(k*J,k*J-1)=2;
end
for j=1:J:K*J                   % Left-hand boundary
    A(j,j+1)=2;
    if j>1 & j<K*J              % Block the matrix
        A(j,j-1)=0;
        A(j-1,j)=0;
    end
end
%
% The finite difference equations are in the form A*h=rhs, where
% "A" is the coefficient matrix, "h" is the vector of unknown heads,
% and "rhs" is the vector of known quantities.
%
% The MATLAB "\" function solves the system of equations.
%
h=A\rhs;
%
% The unknown heads can be put back into the gridded form to view
% in our x-y orientation.
%
hh=reshape(h,J,K);
%
% Add in the known heads along the top boundary and rotate the grid
% into the upright position.
%
hh=[hh hT'];
```

```
hh=hh';
%
% Next we can contour the heads.
figure(2)
contour(hh,12)
axis equal
%
% The MATLAB gradient function, allows us to calculate the flow
% directions.
%
[px,py]=gradient(hh);
%
% We can plot the flow vectors on the contour plot.
% (Note: the "1" in the "quiver" command regulates the size of the
% arrows.)
%
hold on;quiver(-px,-py,1);hold off;
```

6.5. Solving the two-dimensional Poisson equation

One problem that arises very frequently in practice involves the solution of a two-dimensional ground-water flow problem through a horizontal aquifer. Heads are averaged over the vertical and the equation to be solved is the Poisson equation ([Box 6.1]; also see a hydrogeology text – Fetter, 2001, for example – to review the derivation of the equation and the physical meaning of the terms.) The pertinent equation is:

$$\frac{\partial}{\partial x}\left(T\frac{\partial h}{\partial x}\right)+\frac{\partial}{\partial y}\left(T\frac{\partial h}{\partial y}\right)=-w \qquad (6.4)$$

where w is a source term such as recharge (dimensions of length per time); a negative w would indicate a sink such as evaporation or pumping. The transmissivity, T, can vary with x and y.

A block-centered grid is often very useful for solving ground-water problems, especially for grids with unequal grid spacing (varying Δx and Δy) and for spatially varying transmissivities, T [Box 6.2]. We picture the nodes at the center of blocks that define the transmissivities and the finite-difference steps in x and y directions. For example, consider the 3x5 set of blocks shown schematically below. Values of head, transmissivity, and grid spacing for each block are indicated within the block.

$h_1, T_1, \Delta x_1, \Delta y_1$	$h_2, T_2, \Delta x_2, \Delta y_2$	$h_3, T_3, \Delta x_3, \Delta y_3$		
$h_6, T_6, \Delta x_6, \Delta y_6$	$h_7, T_7, \Delta x_7, \Delta y_7$			
$h_{11}, T_{11}, \Delta x_{11}, \Delta y_{11}$				

A finite-difference approximation for the second derivative with respect to x in the second block above can be written

$$\frac{\left[\dfrac{2T_2T_3}{\Delta x_3T_2+\Delta x_2T_3}\left(h_3-h_2\right)-\dfrac{2T_1T_2}{\Delta x_2T_1+\Delta x_1T_2}\left(h_2-h_1\right)\right]}{\left(\dfrac{\Delta x_1+2\Delta x_2+\Delta x_3}{2}\right)}$$

These types of equations are the basis for many of the computer codes used to solve ground-water problems (e.g., see Bredehoeft 1990 and [Box 6.3]). What is important from our standpoint is that we again obtain a set of equations that must be solved and all we need do is determine how to set up the appropriate matrix and right-hand vector if we want to use *MATLAB* to obtain a solution.

Following Bredehoeft (1990) we define

$$\frac{T_{i-1/2}}{\Delta x_{i-1/2}}=\frac{2T_iT_{i-1}}{\Delta x_{i-1}T_i+\Delta x_iT_{i-1}},\quad c^x=\frac{T_{i-1/2}}{\Delta x_{i-1/2}\Delta x},\quad c^y=\frac{T_{j-1/2}}{\Delta y_{j-1/2}\Delta y},$$

$$c^x_{i+1}=\frac{T_{i+1/2}}{\Delta x_{i+1/2}\Delta x},\quad \text{and}\quad c^y_{j+1}=\frac{T_{j+1/2}}{\Delta y_{j+1/2}\Delta y},$$

where Δx is the width of the central (i,j) block, and Δy is the height of the central block. With these definitions, we can derive the following as the finite-difference representation of the Poisson equation.

$$c^xh_{i-1}+c^x_{i+1}h_{i+1}+c^yh_{j-1}+c^y_{j+1}h_{j+1}-\left(c^x+c^x_{i+1}+c^y+c^y_{j+1}\right)h_i=-w_i$$

Note that the finite-difference matrix for a problem will have the form (for a grid with 9 nodes in the x direction):

$$\begin{pmatrix}
a_1 & b_1 & & & & & & d_1 & \\
c_2 & a_2 & b_2 & & & & & & d_2 \\
 & c_3 & a_3 & b_3 & & & & & \\
 & & & \cdot & & & & & \\
 & & & & \cdot & & & & \\
 & & & & & \cdot & & & \\
 & & & & & & \cdot & & \\
 & & & & & & & \cdot & \\
e_9 & & & & & & & a_9 & b_9 \\
 & e_{10} & & & & & & & a_{10}
\end{pmatrix}$$

where the "a's" are the coefficients on h_i in the Bredehoeft equation, the "b's" are the coefficients on h_{i+1}, the "c's" are the coefficients on h_{i-1}, the "d's" are the coefficients on h_{j+1}, and the "e's" are the coefficients on h_{j-1}.

6.6. An example problem

A waste-disposal pond is designed to lose water to evaporation and to "trap" the contaminants in the sludge at the bottom of the pond (Figure 6.4). Unfortunately, the pond, which is about 50 m on a side, leaks (as do all such ponds!) and contaminated water is recharged to the underlying aquifer at a rate of 4 m y^{-1} (~1.25 m s^{-1}). Recharge over the rest of the area in this semi-arid environment is negligible. Streams that dissect the area bound the aquifer to the north and south. The stream to the north, 75 m from the north end of the pond, is topographically higher and, on average, is a "losing stream", with water infiltrating into the aquifer. The boundary can be approximated as having a constant head of 0.2 m above datum. The boundary to the south, some 100 m from the south end of the pond, also can be approximated by a constant-head boundary with a head of 0m. Because the undisturbed flow in this stream-aquifer system tends to be directly from north to south, we can choose east and west boundaries (away from the pond) to be "no-flow" boundaries. The west no-flow boundary is 80 m from the edge of the pond and the east no-flow boundary is 50 m from the pond. The transmissivity of the aquifer is estimated to be 0.0015 m^2 s^{-1}, except in the southwestern part of the aquifer where the transmissivity is thought to be half of that value.

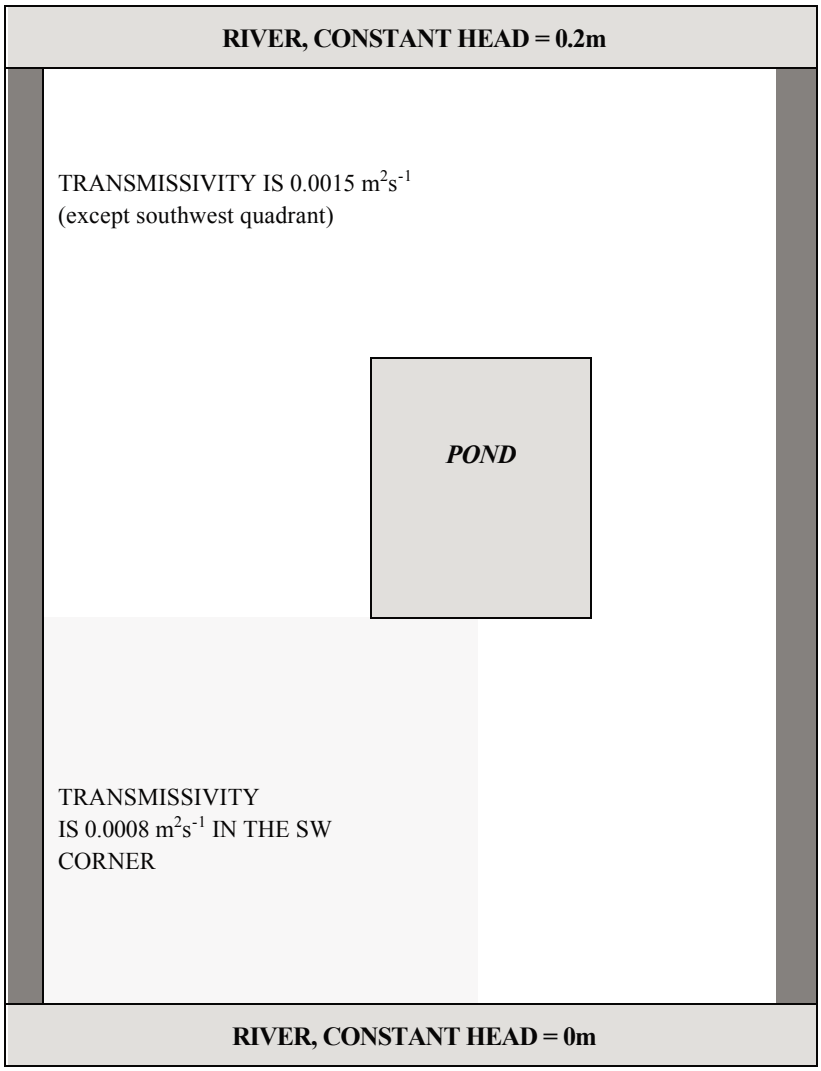


Figure 6.4. Schematic diagram of the example problem.

For the example, we use a very simple (coarse) grid (Figure 6.5). The values in the schematic below give the values for the blocks in the grid. For the blocks representing the area of the pond, the size is 25 m x 25 m, the transmissivity is 0.0015 m^2 s^{-1}, and the recharge is 1.25×10^{-7} m s^{-1}. There are 25 cells (unknowns), numbered from "1" in the upper left, "2" in the next cell to the right, ..., "6" in the cell in the second row leftmost position, ..., and "25" in the lower right cell.

As mentioned above, the difficult part of finite-difference solutions is in setting up the matrices. Review the m-file listed below to see how one might proceed to write a code for this problem.

Δx 40 Δy 75 T1 1.5e-3	40 75 1.5e-3	25 75 T1	25 75 T1	50 75 1.5e-3
40, 25, 1.5e-3	40, 25, 1.5e-3			50, 25, 1.5e-3
40, 25, 1.5e-3	40, 25, 1.5e-3			50, 25, 1.5e-3
40, 50, T2 (8e-4)	40, 50, 8e-4	25 50 T2	25, 50, T1	50, 50, 1.5e-3
40, 50, 8e-4	40, 50, 8e-4	25 50 T2	25, 50 T1	50, 50, 1.5e-3

Figure 6.5. Schematic of the finite-difference blocks for the example problem. The dark shaded squares indicate the overlying pond and the lighter shaded cells are where the transmissivity is relatively lower.

```
% example code for poisson equation
% set the grid spacing and transmissivity matrices
[X,Y]=meshgrid(cumsum([0 40 40 25 25 50]),cumsum([0 75 25 25 50 50]));
dx=diff(X');dy=diff(Y);
dx=dx(:,1:5);dy=dy(:,1:5);dx=dx';
T=0.0015*ones(5,5);
% account for heterogeneity in the southwest corner
```

```
T(4,1:3)=0.0008;T(5,1:3)=0.0008;
% form "minus half" and "plus half" arrays for bredehoeft equations
Tjm1=[zeros(5,1),T(:,1:4)];Tim1=[zeros(1,5);T(1:4,:)];
Tjp1=[T(:,2:5), zeros(5,1)];Tip1=[T(2:5,:);zeros(1,5)];
dxim1=[zeros(1,5);dx(1:4,:)];dyjm1=[zeros(5,1),dy(:,1:4)];
dxip1=[dx(2:5,:); zeros(1,5)];dyjp1=[dy(:,1:4),zeros(5,1)];
% preallocate zero matrices for the coefficients
a=zeros(1,25);b=a;c=a;d=a;e=a;rhs=a;
% set up the coefficients
Toverdxminus=(2*T.*Tim1)./(dxim1.*T+dx.*Tim1);
Toverdxplus=(2*T.*Tip1)./(dxip1.*T+dx.*Tip1);
Toverdyminus=(2*T.*Tjm1)./(dyjm1.*T+dy.*Tjm1);
Toverdyplus=(2*T.*Tjp1)./(dyjp1.*T+dy.*Tjp1);
cx=Toverdxminus./dx;cy=Toverdyminus./dy;
cxp1=Toverdxplus./dx;cyp1=Toverdyplus./dy;
[ii,jj]=find(isnan(cx)==1);cx(ii,jj)=0;
[ii,jj]=find(isnan(cy)==1);cy(ii,jj)=0;
[ii,jj]=find(isnan(cxp1)==1);cxp1(ii,jj)=0;
[ii,jj]=find(isnan(cyp1)==1);cyp1(ii,jj)=0;
c=reshape(cx',1,25);b=reshape(cxp1',1,25);
e=reshape(cy',1,25);d=reshape(cyp1',1,25);
a=-(c+b+e+d);
a(1:5)=a(1:5)-b(1:5);
a(21:25)=a(21:25)-c(21:25);
a(1:5:21)=a(1:5:21)-d(1:5:21);
a(5:5:25)=a(5:5:25)-e(5:5:25);
% set the top boundary -- head of 0.2m
for j=1:5
      rhs(j)=rhs(j)-b(j)*0.2;
end
% set the no-flow conditions at the sides --note that the following loop
% is ok for this example, but does not work for general heterogeneity!!
for i=1:5
      k=(i-1)*5+1;
      d(k)=2*d(k);
      k=k+4;
      e(k)=2*e(k);
end
% recharge at the pond grid cells
w=-1.25e-7;
rhs(8)=w;rhs(9)=w;rhs(13)=w;rhs(14)=w;
% use the sparse matrix commands to set the full matrix and then solve the
equations
U1_diag=sparse(6:25,1:20,b(1:20),25,25);
L1_diag=sparse(1:20,6:25,c(6:25),25,25);
U2_diag=sparse(2:25,1:24,d(1:24),25,25);
L2_diag=sparse(1:24,2:25,e(2:25),25,25);
M_diag=sparse(1:25,1:25,a,25,25);
A=M_diag+U1_diag+U2_diag+L1_diag+L2_diag;
h=A'\rhs';
aa=reshape(h,5,5);
x=X(1,2:6);
y=Y(2:6,1);
mesh(x,y,aa);xlabel('distance, x, in m')
```

```
ylabel('distance, y, in m');zlabel('ground-water head, m')
pause
[xx,yy]=meshgrid(35:5:185,70:5:230);
zz=interp2(x,y,aa,xx,yy);surfc(zz);
zlabel('head, m');xlabel('regular grid point # (x)')
ylabel('regular grid point # (y)')
```

The matrix "aa" from the code above gives the heads arranged in an order to ease interpretation. You should keep in mind that the heads in matrix "aa" are for nodal points and, for this example problem, are ***not*** equally spaced. The *MATLAB* "contour" command and the related others that we used previously assume that the matrix elements are representative of equally spaced points. Thus, the output above should not be contoured directly, at least not with the *MATLAB* "contour" command. One thing we can do with unequally spaced data is use the "mesh" command (Figure 6.6) once the x and y values are specified.

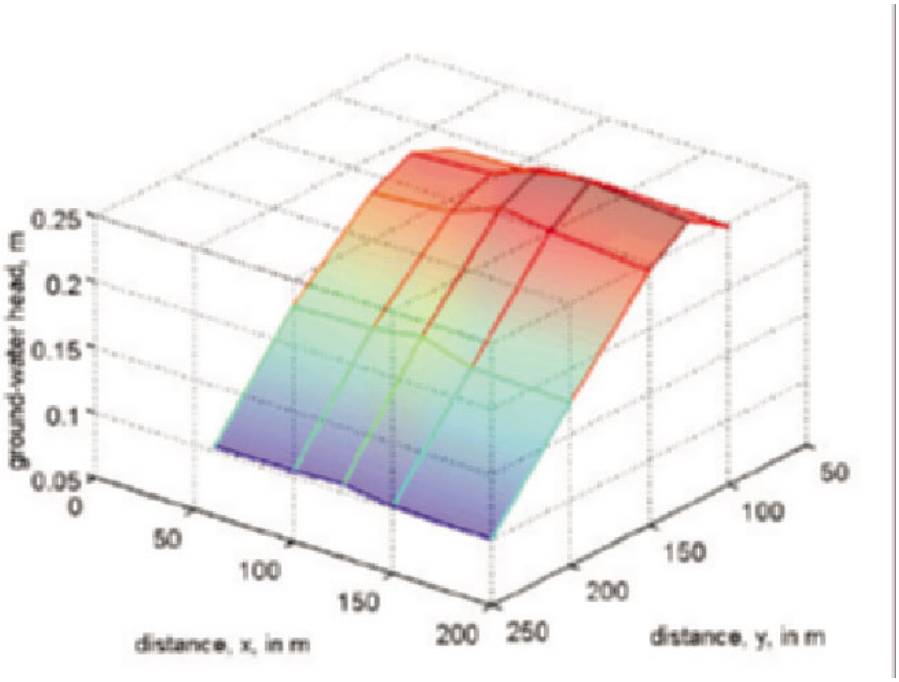


Figure 6.6. Meshplot of solution to example problem.

If we want to contour the heads, we can use the *MATLAB* `interp2` command to interpolate the unequally spaced data onto a regular grid.

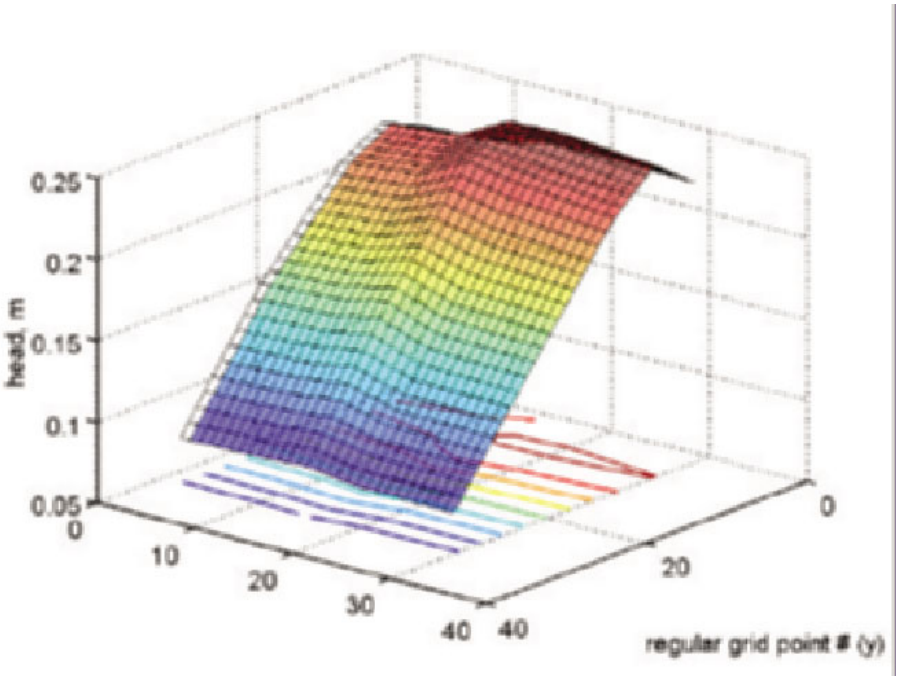


Figure 6.7. Data interpolated to a regular grid can be contoured.

6.7. Problems

The Brookhaven landfill was constructed on (Wisconsin) glacial outwash deposits. The landfill overlies the upper glacial aquifer, the lower boundary of which is the Gardiners Clay. The aquifer is composed primarily of sand and gravel with small amounts of feldspar, mica, and other minerals. See Wexler (1988) for details.

Precipitation in the area averages 47.4 inches annually. Under natural conditions, about half this amount recharges the aquifer, with the remainder going to evapotranspiration. The bulk of the recharge water flows toward the south in the upper glacial aquifer, discharging to streams and bays.

Depth to the water table ranges from 0 to 55 feet, depending on surface elevation. Saturated thickness of the upper aquifer ranges from 100 to 130 feet. The sands and gravels are quite permeable. Estimates of hydraulic conductivity range from 187 to 267 feet per day. Beaverdam Creek, which is fed almost totally by groundwater from the upper glacial aquifer, drains the area (Figure 6.8).

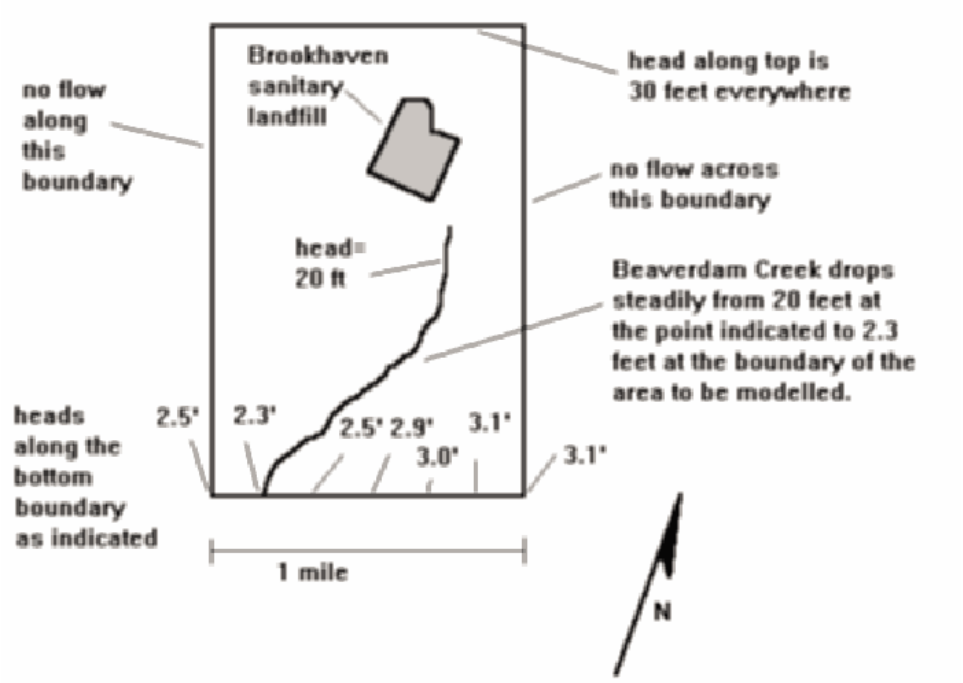


Figure 6.8. Schematic layout for the Brookhaven Landfill problem.

The landfill was constructed with a PVC liner, but despite this precaution, leachate has entered the aquifer. Landfill operations began in 1973 and by 1982 a plume extended several thousand feet to the southeast of the landfill.

As a consultant to the local government, your assignment is to develop a ground-water flow model for the site using the information available. The flow model is needed to interpret the directions of leachate migration. This interpretation is direct, in part, in that a flow net is generally a good guide to picturing contaminant migration. The interpretation is indirect, in part, in that the results from a flow model are required to implement a transport model. (Note: such assignments are

"imprecise". You have great latitude in designing the model, but keep in mind that approximations must be made -- no model can capture even a fraction of the complexity of the "real world". The trick in developing a useful model is choosing useful approximations.) [Solve the problem first using a transmissivity two orders of magnitude smaller than suggested by the data. This will allow you to visualize what is going on more easily.]

One hint that you may find useful is how to assign a fixed head to an interior node. Suppose the general finite-difference equation for the interior node is:

$$e_i h_{i-J} + c_i h_{i-1} + b_i h_{i+1} + d_i h_{i+J} - a_i h_i = 0$$

where J is the number of columns in the finite-difference grid. Rather than change the value of h_i in all of the equations in which it appears, we can adjust just the particular equation above and carry on with the solution in an "as usual" fashion. To do this we change a_i to 1, b_i, c_i, d_i, and e_i to zero and change the 0 on the right-hand side to h_*, where h_* is the value at which we want to fix h_i. Note what this does. The values of e, c, b, and d are zero. Thus, the equation is

$$h_i = h_*$$

fixing h at this node at the desired constant value.

6.8. References

Bredehoeft, J.D., Microcomputer codes for simulating transient ground-water flow in two and three space dimensions. U.S. Geological Survey Open-file Report 90-559, 1990.

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

McDonald, M.C., and Harbaugh, A.W., A modular three-dimensional finite-difference ground-water flow model: U.S. Geological Survey Techniques of Water-Resources Investigations, Book 6, Chap. A1, 586 pp., 1988.

Harbaugh, A.W., Banta, E.R., Hill, M.C., and McDonald, M.G., MODFLOW-2000, the U.S. Geological Survey modular ground-water model -- User guide to modularization concepts and the Ground-Water Flow Process: U.S. Geological Survey Open-File Report 00-92, 121 pp., 2000.

Tóth, J.A., A theoretical analysis of ground-water flow in small drainage basins, *J. Geophys. Res., 68*: 4795-4811, 1963.

Wexler, E.J., Ground-water flow and solute transport at a municipal landfill site on Long Island, New York. Part 1. Hydrogeology and water quality. U.S.Geological Survey Water Resources Investigations Report 86-4070, 1988.

Box 6.1. Groundwater flow equations

The basis of the equations used to describe the flow of groundwater is the conservation of mass equation, which, when applied to a fixed control volume, basically says that the rate of mass inflow minus rate of mass outflow equals rate of change of mass storage – what goes in minus what goes out equals the change in what's inside.

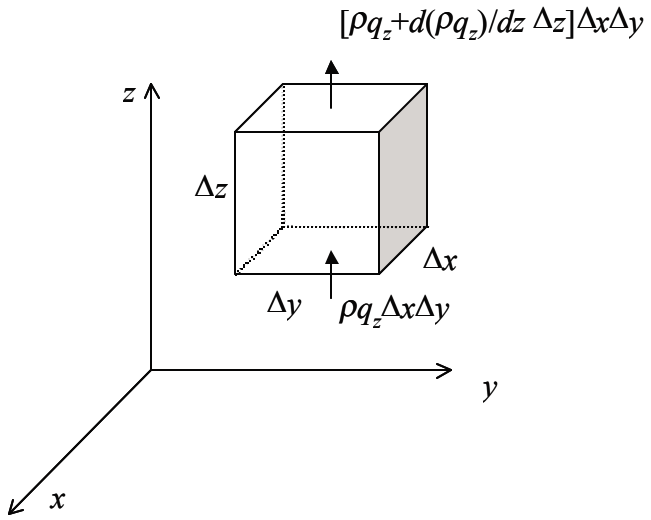


Figure B6.1.1. Control volume for deriving the conservation equation.

One way to derive the general equation of continuity for pore-fluid flow is to specify an arbitrary control volume to be a small rectangular parallelepiped in a fixed, Cartesian coordinate frame with sides of length Δx, Δy, and Δz (Fig. B6.1.1). Without any loss of generality, we take the directions designated by the arrows on the axes as positive (Fig. B6.1.1) and consider the case of positive flows. Consider first the inflow of mass into the control volume. The inflow into the parallelepiped in the z-direction is $\rho q_z \Delta x \Delta y$, where ρ is the density of water and q_z is the specific discharge (volumetric discharge per unit area) in the z direction. Density times the specific discharge gives the mass flux (mass per area per time) so multiplication by the area, $\Delta x \Delta y$, yields the mass inflow in the z direction. The mass flows in the x- and y-directions can be similarly calculated. Because specific discharge can change with distance, the value of q_z at the top face need not be the same as that at the bottom face. We can estimate the specific discharge at the top face using the Taylor series (Chapter 3.2). Because the distance separating the two faces (Δz) is as small as we wish to make it, we can get an acceptable approximation of the flux at the top face by just retaining the first two terms of the series (i.e., a linear extrapolation):

$$q_z(z + \Delta z) = q_z(z) + \frac{\partial q_z}{\partial z}\Delta z \qquad \text{(B6.1.1)}$$

The expression for the mass outflow in the z direction is then

$$\left[\rho\, q_z + \frac{\partial\, \rho q_z}{\partial z}\Delta z\right]\Delta x\Delta y \qquad \text{(B6.1.2)}$$

Now the expression that we need for the continuity equation is the **net** inflow of mass – the difference between the inflow and the outflow. For the z direction,

$$\text{net mass flow}_Z = (\rho q_z(z) - \rho q_z(z+\Delta z))\Delta x\Delta y = -\frac{\partial \rho q_z}{\partial z}\Delta x\Delta y\Delta z$$

Using similar expressions for the x- and y-directions, the total net mass inflow can be obtained:

$$\text{total net mass inflow} = -\left[\frac{\partial\, \rho q_x}{\partial x} + \frac{\partial\, \rho q_y}{\partial y} + \frac{\partial\, \rho q_z}{\partial z}\right]\Delta x\Delta y\Delta z \qquad \text{(B6.1.3)}$$

For *steady* flow there can be no change of mass in the control volume so the net inflow must be zero. If we further assume that the density is constant, equation (B6.1.3) implies that

$$\frac{\partial q_x}{\partial x} + \frac{\partial q_y}{\partial y} + \frac{\partial q_z}{\partial z} = 0 \qquad \text{(B6.1.4)}$$

The forces driving flow through an aquifer are due to gravity and pressure gradients. These forces, expressed on a per unit weight basis, are represented in groundwater flow equations in terms of a *head gradient*. Groundwater head is defined as pressure per unit weight plus elevation, which is the head due to gravity.

Darcy's law relates the specific discharge to the head gradient. Darcy's law states that this relationship is linear, with the constant of proportionality between specific discharge and head gradient being the *hydraulic conductivity*, K. If we make the assumption that the aquifer is *isotropic*, i.e., that K is independent of direction, Darcy's law can be written as follows.

$$\begin{aligned} q_x &= -K\frac{\partial h}{\partial x} \\ q_y &= -K\frac{\partial h}{\partial y} \\ q_z &= -K\frac{\partial h}{\partial z} \end{aligned} \qquad \text{(B6.1.5)}$$

Combining equations (B6.1.4) and (B6.1.5), we obtain an equation for steady groundwater flow.

$$\frac{\partial}{\partial x}\left(K\frac{\partial h}{\partial x}\right) + \frac{\partial}{\partial y}\left(K\frac{\partial h}{\partial y}\right) + \frac{\partial}{\partial z}\left(K\frac{\partial h}{\partial z}\right) = 0 \qquad \text{(B6.1.6)}$$

For the case of a *homogeneous* aquifer, one for which K is constant, equation (B6.1.6) reduces to the Laplace equation.

$$\frac{\partial^2 h}{\partial x^2}+\frac{\partial^2 h}{\partial y^2}+\frac{\partial^2 h}{\partial z^2}=0 \qquad \text{(B6.1.7)}$$

For the case of a horizontal aquifer of constant thickness, b, we assume that there is no vertical flow so equation B6.1.4 takes the form

$$b\frac{\partial q_x}{\partial x}+b\frac{\partial q_x}{\partial x}=0 \qquad \text{(B6.1.8)}$$

When (B6.1.8) is combined with Darcy's law, we obtain

$$\begin{aligned}&\frac{\partial}{\partial x}\left(Kb\frac{\partial h}{\partial x}\right)+\frac{\partial}{\partial x}\left(Kb\frac{\partial h}{\partial x}\right)=0\\&\frac{\partial}{\partial x}\left(T\frac{\partial h}{\partial x}\right)+\frac{\partial}{\partial x}\left(T\frac{\partial h}{\partial x}\right)=0\end{aligned} \qquad \text{(B6.1.9)}$$

where T is the *transmissivity* of the aquifer. If there is recharge to the aquifer, e.g., by slow flow through an overlying confining layer, the equation is modified accordingly:

$$\frac{\partial}{\partial x}\left(T\frac{\partial h}{\partial x}\right)+\frac{\partial}{\partial x}\left(T\frac{\partial h}{\partial x}\right)=-w \qquad \text{(B6.1.10)}$$

where w is the recharge rate.

If the aquifer is homogeneous, T is constant and can be brought outside the derivative in (B6.1.9). The result is

$$\frac{\partial^2 h}{\partial x^2}+\frac{\partial^2 h}{\partial y^2}=0 \qquad \text{(B6.1.11)}$$

so again we find that the Laplace equation describes the steady flow of groundwater through a horizontal aquifer.

Finally, note that Darcy's law indicates that flow is down the *gradient* in head. This implies that flow lines for groundwater in an isotropic aquifer are perpendicular to lines of constant head. In *MATLAB* this means that if groundwater heads are computed and contour lines drawn, the `gradient` and `quiver` commands can be used to depict the flow.

Box 6.2. MODFLOW

The most widely used computer program in the world for simulating groundwater flow MODFLOW, a modular code developed by the U.S. Geological Survey in the early 1980's and continually improved since then (McDonald and Harbaugh, 1988; Harbaugh et al., 2000). The code along with documentation can be downloaded from the USGS Web site.

MODFLOW uses the finite difference method to solve the groundwater equations using the block centered node approach. An aquifer system is divided into rectangular blocks by a grid organized by rows, columns, and layers. Each block is called a "cell." Hydraulic properties are assigned to the cells, and boundary conditions are set by specifying cells to be constant head, no-flow, and so forth (see Figure B6.2.1).

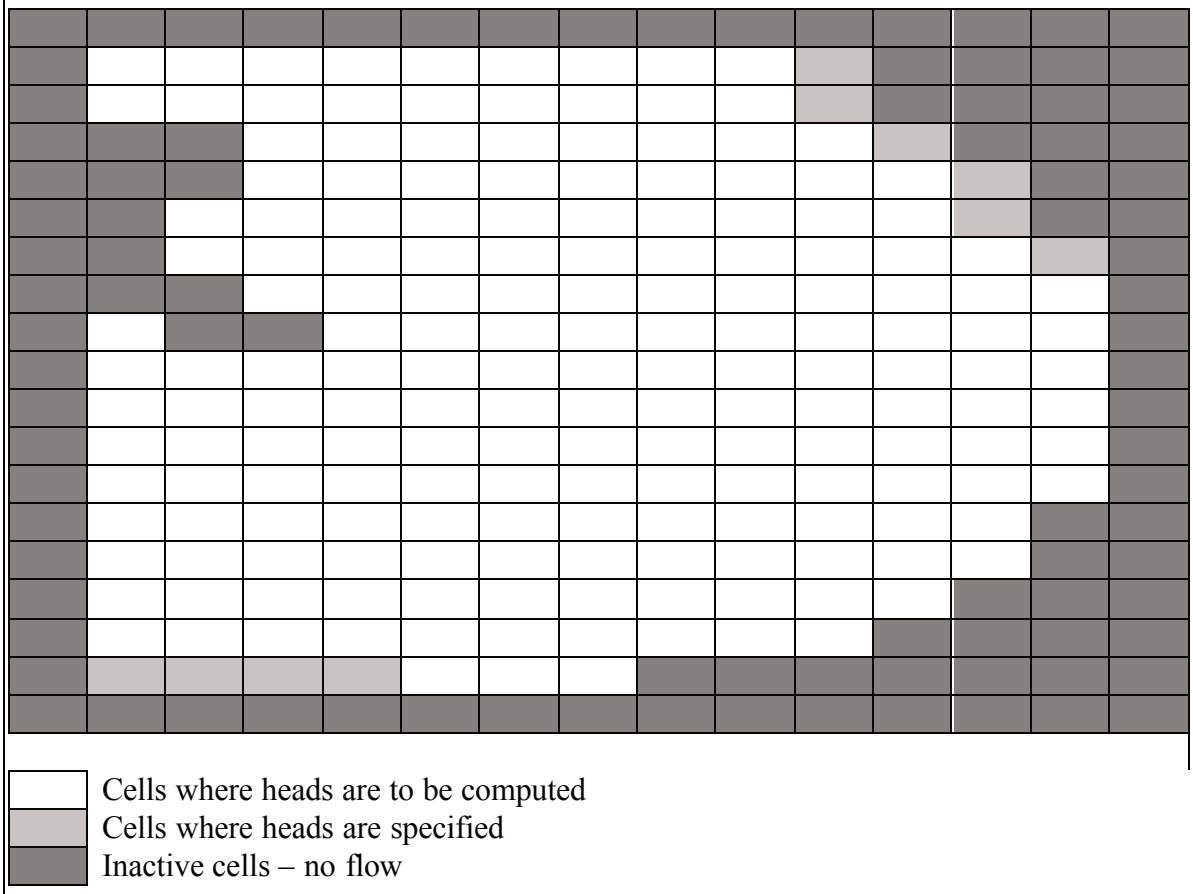


Figure B6.2.1. Example of a MODFLOW grid. Finite difference equations are written for cell-centered nodes.

Box 6.3. A block-centered finite difference approximation

For the block-centered approach, the transmissivity (and other properties) are assigned to blocks. The nodes, at which the heads are calculated, are at the center of the blocks.

Block i-1	Block i	Block i+1
T_{i-1}	T_i	T_{i+1}
h_{i-1}	h_i	h_{i+1}
$\bullet \leftarrow \Delta x_{i-1}/2 \rightarrow$	$\leftarrow \Delta x_i/2 \rightarrow \bullet \leftarrow \Delta x_i/2 \rightarrow$	$\leftarrow \Delta x_{i+1}/2 \rightarrow \bullet$

The partial derivative in the x direction, $\frac{\partial}{\partial x}\left(T\frac{\partial h}{\partial x}\right)$, is approximated as:

$$\frac{\left(T\frac{\partial h}{\partial x}\right)_{i+1/2} - \left(T\frac{\partial h}{\partial x}\right)_{i-1/2}}{\Delta x_i}$$

The derivatives at the halfway points are approximated in a straightforward manner. For example,

$$\left(T\frac{\partial h}{\partial x}\right)_{i+1/2} \approx \frac{2T_iT_{i+1}}{T_i + T_{i+1}}\frac{(h_{i+1} - h_i)}{\Delta x_i}$$

in which the transmissivity is approximated using the *harmonic mean*, $\frac{2T_iT_{i+1}}{T_i + T_{i+1}}$.

If the block sizes are unequal, the equations are modified slightly; see Bredehoeft (1990) for further details.

CHAPTER 7

Iterative Solution of Systems of Equations

Numerical Methods in the Hydrological Sciences
Special Publication Series 57

doi:10.1029/057SP12

7. Iterative Solution of Systems of Equations

7.1. Introduction

The methods for solving partial differential equations presented in Chapter 6 relied on Gaussian elimination (conveniently implemented with *MATLAB*'s "\" command) to solve the system of equations obtained when finite differences are used to approximate the derivatives. Gaussian elimination is not the only way to solve a system of equations. Iterative methods provide an alternative that can be advantageous. For example, when programming in one of the popular languages (FORTRAN, PASCAL, C), it is much easier to code the "loop" calculations needed for typical iterative solutions than it is to code Gaussian elimination. The matrix approach used in Chapter 6 is convienent for *MATLAB* but is not the way the calculations generally are done when working with a "non-matrix" language. [Consult the text by Wang and Anderson (1982) for more detail.] Also, some of the very large matrices that arise in complex hydrological problems are not solved directly very easily – iterative methods are actually preferred to Gaussian elimination. The basic idea of iterative solutions to equations was already presented in Chapter 2. In this chapter, we explore how iterative methods can be used to solve systems of equations.

7.2. Fixed-point iteration revisited

Recall that "fixed-point iteration" is one option for solving nonlinear equations [Chapter 2.6]. The equation is written in the form

$$x = g(x)$$

and the iteration formula is simply

$$x_{k+1} = g(x_k).$$

Convergence of the iteration is not guaranteed in general and the convergence criterion typically used is a check on whether x_{k+1} is "sufficiently close" to x_k.

Of course a linear equation is just a special case of a nonlinear equation. Although iteration doesn't make good sense computationally for a single linear equation, we can look at this case to motivate the actual use that interests us – solution of systems of linear equations. So, let's take the simple example

$$3x = 6.$$

To develop an iteration formula, we must "split" the 3. For example, we can say that $3x=2x+x$, subtract x from both sides of the equation, divide by 2, and have the iteration formula

$$x_{k+1} = \frac{-x_k}{2} + 3.$$

The first several steps in the iteration are

$$x_1 = \frac{-x_0}{2} + 3$$

$$x_2 = \frac{-x_1}{2} + 3 = \frac{-1}{2}\left(\frac{-x_0}{2} + 3\right) + 3 = \left(\frac{-1}{2}\right)^2 x_0 + 3\left(1 - \frac{1}{2}\right)$$

$$x_3 = \frac{-x_2}{2} + 3 = \left(\frac{-1}{2}\right)^3 x_0 + 3\left(1 - \frac{1}{2} + \left(\frac{1}{2}\right)^2\right)$$

$$\vdots$$

$$\vdots$$

$$x_{k+1} = \left(\frac{-1}{2}\right)^{k+1} x_0 + 3\left(1 - \frac{1}{2} + \left(\frac{1}{2}\right)^2 - \left(\frac{1}{2}\right)^3 + \ldots + \left(\frac{1}{2}\right)^k\right).$$

We can see from inspection of the general equation above that the iteration converges to the right answer. The first term goes to zero as $k \rightarrow \infty$ (i.e., 1/2 to the k^{th} power approaches zero). The term in brackets is the geometric series. The limit of this series is [1/(1+1/2)]=2/3, so in the limit, x_k approaches 3*(2/3)=2.

There are other ways – in fact, an infinite number of ways – we can "split" 3 in the original equation. The general case can be represented as

$$x_{k+1} = \frac{-(3-\alpha)}{\alpha} x_k + \frac{6}{\alpha} = \left[\frac{-(3-\alpha)}{\alpha}\right]^{k+1} x_0 + \left(\frac{6}{\alpha}\right)\left(1 - \frac{3-\alpha}{\alpha} + \left(\frac{3-\alpha}{\alpha}\right)^2 - \ldots + \left(\frac{3-\alpha}{\alpha}\right)^k\right).$$

We can see the requirement for convergence of the method. The term $-(3-\alpha)/\alpha$ must be less than one in magnitude. (Otherwise, as the term is raised to higher and higher powers, the iteration diverges.) This condition requires that $\alpha > 3/2$. Is there a "best" value for α? It should be clear from an examination of the steps above that the smaller our "iteration number", the faster the effect of the initial guess goes to zero and the faster the terms in the series goes to zero and hence the faster this series converges. Therefore, we want to have the absolute value of the "iteration number" $(3-\alpha)/\alpha$ be as small as possible. We can look at values for the iteration number as α varies from 3/2 to 10.

```
alpha=3/2:1/6:10;
iter=abs((3-alpha)./alpha);
plot(alpha,iter);
xlabel('value of the splitting parameter alpha')
ylabel('magnitude of the iteration number')
```

The graph that results from the *MATLAB* operations above (Fig. 7.1) shows that there is a "best" value. As we would have expected for this simple (simple-minded?) problem, 3 is the best splitting parameter because then no iteration at all is required! The example is nonetheless instructive because one can see that the progress of the iteration – how fast the process converges – depends on the choice for splitting. To make good use of information on whether a split is "good", we need to know something about the magnitude of the iteration number. We will return to this notion directly, but first, let's look at solutions for matrix-vector equations.

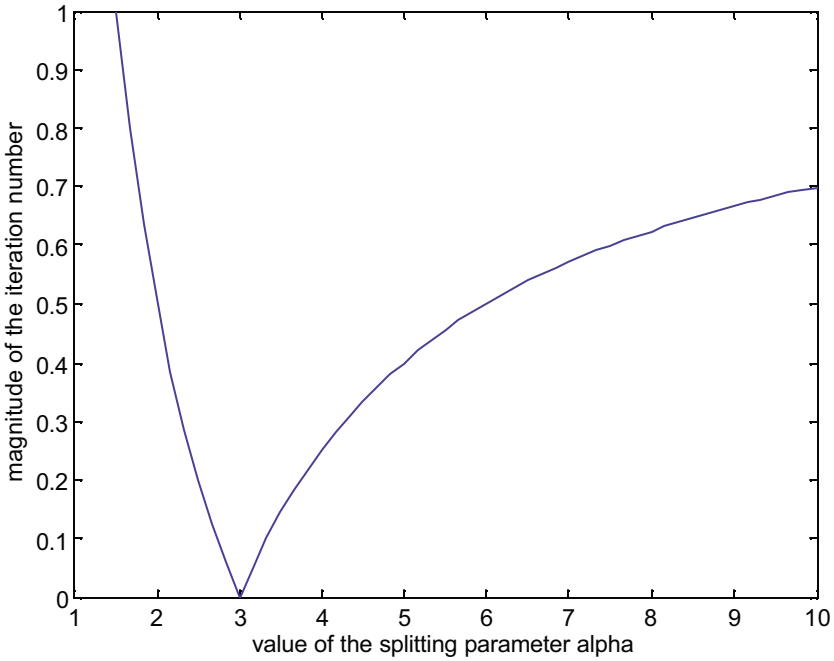


Figure 7.1. Splitting parameter for the example $3x=6$.

7.3. Iterative solution of a system of equations

The notions above generalize to matrix-vector equations pretty well. Take a system of equations given by

$$Ax = b\,.$$

Split the matrix A into $D+B$. Rewrite the equation as follows,

$$Dx = -Bx + b$$

$$x = -D^{-1}Bx + D^{-1}b$$

which leads to the iteration formula

$$x_{k+1} = -D^{-1}Bx_k + D^{-1}b\,.$$

General iteration formulas have this form. The specifics differ because the choice of how to split the matrix differs. The simplest iteration (or at least one very simple one) uses the diagonal elements of matix A to form D. This method uses the sum of the part of A that is below the diagonal (L) and the part of A that is above the diagonal (U) as the B matrix. The iteration formula is therefore

$$x_{k+1} = -D^{-1}\left(L+U\right)x_k + D^{-1}b = \left[Q\right]^{k+1}x_0 + \left(I + Q + Q^2 + \ldots + Q^k\right)D^{-1}b,$$

where $Q=D^{-1}(L+U)$. Note that because D is a diagonal matrix, D^{-1} is trivial to compute – it is just the inverse of the individual diagonal elements of D. This is a feature sought in iteration methods because computational ease is one of the reasons for using iterative methods. This simple iteration

formula is known as *Jacobi* iteration. For finite-difference approximations to the Laplace equation [Chapter 6.3], this method "works", that is, it converges. Note that this means that $Q^k \rightarrow 0$ as $k \rightarrow \infty$. But how fast will it converge? And are there better methods? How can we judge these methods? For the single linear equation that we used as an example in the first section of these notes, it turned out that the magnitude of the "iteration number" was important. But now we have an iteration matrix, Q, rather than an iteration number. We need to examine the notion of the "magnitude" of a matrix before we proceed.

7.4. Vector and matrix norms

You are already familiar with the idea of magnitude applied to a vector. If the x component of velocity is 10 m s^{-1} and the y component is 10 m s^{-1}, you know that the magnitude of the velocity is about 14.14 m s^{-1}. You arrive at this answer by squaring the lengths of the x and y components, adding, and taking the square root. This measure of the magnitude of a vector is known as the L_2 norm. A "norm" is just the generalization of a measure of size.

The norm of a vector x is an associated, non-negative number, $\|x\|$, that satisfies the following requirements:

1) $\|x\| > 0$ for $x \neq 0$; $\|0\| = 0$

2) $\|cx\| = |c|\|x\|$

3) $\|x + y\| \leq \|x\| + \|y\|$

Here are the three most popular ways of assigning a vector norm.

i) $\|x\|_\infty = \max_i |x_i|$ (the maximum component of the vector)

ii) $\|x\|_1 = |x_1| + |x_2| + \ldots + |x_n|$ (the sum of the lengths of all of the componenents)

iii) $\|x\|_2 = \sqrt{x_1^2 + x_2^2 + \ldots + x_n^2}$ (the "standard" norm with which you are familiar).

These are often referred to as the L_∞, L_1, and L_2 norms, respectively. These are the most commonly used forms of the more general L_p norm, $\|x\|_p = [x_1^p + x_2^p + x_3^p + \circ\circ\circ + x_n^p]^{1/p}$ [Box 7.1].

The norm of a square matrix A is a non-negative number, $\|A\|$, satisfying

1) $\|A\| > 0$ *for* $A \neq 0$; $\|Z\| = 0$, iff $Z = 0$

2) $\|cA\| = |c|\|A\|$

3) $\|A + B\| \leq \|A\| + \|B\|$

4) $\|AB\| \leq \|A\|\|B\|$

Because in many problems we have both matrices and vectors, it is convienent to introduce the norm of a matrix in such a way that it will be consistent with a vector norm. A matrix norm is said to be compatible with a given vector norm if

$\|Ax\| \le \|A\|\|x\|$.

A matrix norm constructed to be compatible is said to be subordinate to the given vector norm.

For the three "popular" vector norms given above, the subordinate matrix norms are:

i) $\|A\|_\infty = \max_i \sum_{k=1}^{n} |a_{ik}|$, (the maximum sum of row elements)

ii) $\|A\|_1 = \max_k \sum_{i=1}^{n} |a_{ik}|$, (the maximum sum of column elements)

iii) $\|A\|_2 = \sqrt{\lambda_1}$, where λ_1 is the largest eigenvalue of the matrix $A'A$.

Next consider how matrix-vector norms are useful in examining iteration. Here are three useful theorems about matrix norms (given without proof).

Theorem 1. In order that $A^m \to 0$ as $m \to \infty$, a necessary and sufficient condition is that the eigenvalues of A be less than unity in magnitude.

Theorem 2. No eigenvalue of a matrix exceeds any of its norms in modulus. (This means that, from Theorem 1, sequential powers of a matrix will converge to zero if any matrix norm is <1.)

Theorem 3. In order that the series $I + A + A^2 + ... + A^m + ...$ converge, it is necessary and sufficient that $A^m \to 0$. In this case, the sum of the series converges to $(I-A)^{-1}$.

Armed with this knowledge, we go back to Jacobi iteration,

$$x_{k+1} = -D^{-1}(L+U)x_k + D^{-1}b = [Q]^{k+1} x_0 + (I + Q + Q^2 + ... + Q^k) D^{-1}b,$$

from which we see that the method converges if $\|Q\| < 1$. Note that, in direct analogy with the trivial example that we used for motivation – iterative solution of a single linear equation – the rate of convergence of the matrix iteration problem is determined by the "size" of the iteration matrix. The smaller the norm of Q, the faster will the method converge. The aim of improved methods for solving linear equations iteratively is to reduce the norm of the iteration matrix. In particular, because it is the maximum eigenvalue of the iteration matrix that controls convergence, "good" iterative methods will have iteration matrices with small eigenvalues.

7.5. Iterative methods for finite difference equations

Recall the example problem from Chapter 6.3 (for the Laplace equation).

```
M_diag=sparse(1:21,1:21,-4,21,21);
L_diag1=sparse(2:21,1:20,1,21,21);
L_diag2=sparse(8:21,1:14,1,21,21);
L_diag1(8,7)=0; L_diag1(15,14)=0;
A=M_diag+L_diag1+L_diag2+L_diag1'+L_diag2';
```

Remember that these steps give the blocked matrix with -4's on the main diagonal and 1's on a few off-diagonals.

```
full(A(1:9,1:9))

ans =
    -4     1     0     0     0     0     0     1     0
     1    -4     1     0     0     0     0     0     1
     0     1    -4     1     0     0     0     0     0
     0     0     1    -4     1     0     0     0     0
     0     0     0     1    -4     1     0     0     0
     0     0     0     0     1    -4     1     0     0
     0     0     0     0     0     1    -4     0     0
     1     0     0     0     0     0     0    -4     1
     0     1     0     0     0     0     0     1    -4
```

The right-hand vector for this example problem has three values of -100.

```
b=zeros(21,1); b(7)=-100; b(14)=-100; b(21)=-100;
```

Now rather than use the *MATLAB* backslash command to get the solution, suppose we use the Jacobi iteration approach. The matrix D is just what we defined as `M_diag` in the *MATLAB* statements above. The $L+U$ portion is everything else; we can define them in *MATLAB* as

```
L=L_diag1+L_diag2; U=L'; LnU=L+U;
```

To get D^{-1}, just write `Dinv=inv(M_diag)`. The iteration matrix is `Q=-Dinv*LnU`. We solve the equations using a loop that ends when the fractional change in h is less than 10^{-3}.

```
convcrit=1e9;
h_old=ones(21,1);
kount=0;
while convcrit>1e-3
   kount=kount+1;
   h=Q*h_old+Dinv*b;
   convcrit=max(abs(h-h_old)./h);
   h_old=h;
end
```

It takes 40 iterations to reduce the relative change in the iterative values to $< 10^{-3}$. Compare the results from this iteration (below) with the results in Chapter 6.3 obtained with the "\" command.

```
output=[h(1:7) h(8:14) h(15:21)]

output =
    0.3519    0.4973    0.3519
    0.9112    1.2865    0.9112
    2.0076    2.8287    2.0076
    4.2929    6.0153    4.2929
    9.1505   12.6501    9.1505
   19.6612   26.2865   19.6612
   43.2090   53.1759   43.2090
```

What about the rate of convergence? A good iteration method will have a norm that is "small". The closer to unity the norm, the slower will be the rate of convergence. For the Jacobi iteration matrix for the example problem, `abs(max(eig(Q)))` gives 0.815. For the small example problem this isn't horrendous, but as the size of the problem gets bigger (the mesh spacing for a fixed area gets finer), the norm of the Jacobi iteration matrix gets close to one very quickly. The method always converges, but progress can be agonizingly slow.

The method known as "successive over-relaxation" (SOR) is the simplest of the useful iterative methods for solving systems of linear equations. The iteration formula is

$$x_{k+1} = S(\omega)\, x_k + \left(\omega^{-1} D + L\right)^{-1} b,$$

where the iteration matrix, S, is

$$S(\omega) = -(\omega^{-1} D + L)^{-1} \left(U + (1 - \omega^{-1}) D\right).$$

The Greek letter ω in the iteration formula is an iteration parameter, chosen to accelerate convergence. The identification of an optimal value for ω is not easy for complex problems. Theoretically, the optimal value of ω for the Laplace equation is

$$\omega_{opt} = \frac{2}{1 + \sqrt{1 - \rho^2(Q)}},$$

where $\rho(Q)$ is the magnitude of the largest eigenvalue of the Jacobi iteration matrix.

For the example problem, we can calculate ω_{opt} using 0.815 for $\rho(Q)$, a value that can be found using the *MATLAB* `normest` command, `wopt=2/(1+sqrt(1-(normest(Q))^2))`, which gives a value of about 1.27. [Note that the approximate norm will fail to give a usable estimate when the maximum eigenvalue of the Jacobi matrix is near unity. In this case, the command in the example code, `max(eig(full(S))`, should be used.] We can apply the SOR method to the example problem:

```
y=inv(D*(1/wopt)+L);
S=-y*(U+(1-(1/wopt))*D);
convcrit=1e9;
h_old=ones(21,1);
kount=0;
while convcrit>1e-3
   kount=kount+1;
   h=S*h_old+y*b;
   convcrit=max(abs(h-h_old)./h);
   h_old=h;
end
```

For the SOR method with ω_{opt} =0.815, the convergence criterion is satisfied after 14 iterations. The results are,

```
output=[h(1:7) h(8:14) h(15:21)]

output =
    0.3529    0.4988    0.3530
    0.9131    1.2893    0.9132
    2.0103    2.8323    2.0103
    4.2957    6.0194    4.2957
    9.1532   12.6538    9.1532
   19.6632   26.2894   19.6632
   43.2101   53.1774   43.2101
```

How "big" is the iteration matrix for the SOR method? `abs(max(eig(full(s))))`

```
for i=1:max(size(w))
    y=inv(D*(1/w(i))+L);
    S=-y*(U+(1-(1/w(i)))*D);
    rho(i)=abs(max(eig(full(S))));
end
plot(w,rho)
xlabel('SOR iteration parameter')
ylabel('Magnitude of maximum eigenvalue of the iteration matrix')
```

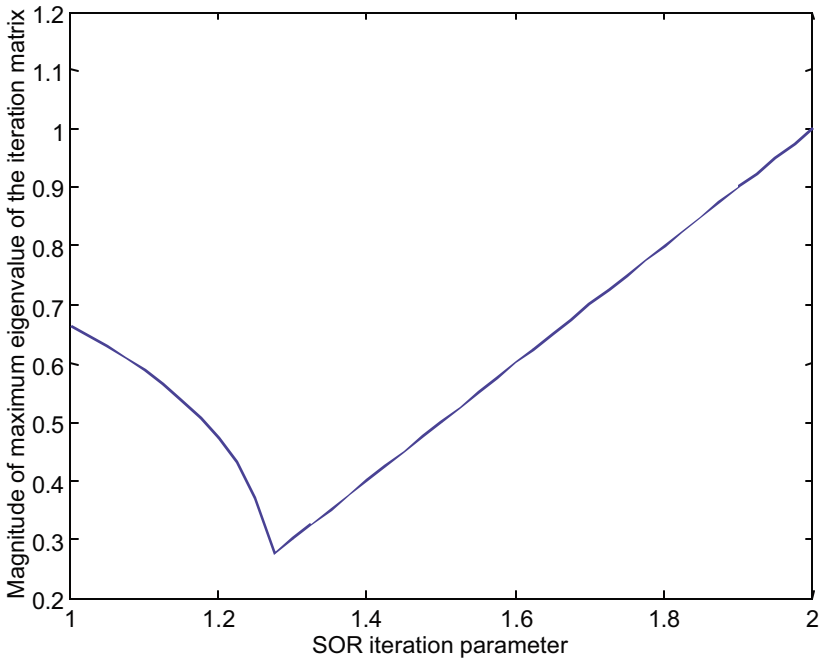


Figure 7.2. Splitting parameter for the SOR example.

Compare this figure with the one we drew for our simple-minded, one-linear-equation example. The analogy isn't perfect by any means, but the outcome is the same – there is an optimum value of an iteration parameter that speeds convergence of the iterative procedure.

The big trick is, of course, to choose a good iteration parameter. Sometimes it is sensible to calculate the maximum eigenvalue of the Jacobi matrix, but that procedure is in no case the "universal" solution. If you need to pursue this topic further, you will have to consult a text on numerical solution of systems of linear equations. Although the text is a bit "long in the tooth" now, you might try Remson et al. (1971) for a start.

The iterative methods that are used most widely today are more efficient (smaller matrix norms) than the very simple methods presented above (including SOR). One of these methods is called the Strongly Implicit Procedure (SIP). MODFLOW [Box 6.2] uses SIP to solve the groundwater flow equations (McDonald and Harbaugh, 1988). The SIP method uses a handful of iteration parameters; choosing them is an art, although the performance of the method does not depend critically on having exact values.

Another relatively new method is the conjugate-gradient method (e.g., Hill, 1990). The conjugate-gradient technique is not only powerful, but the acceleration parameter is calculated as part of the solution so no special tricks are needed to make it work efficiently. The conjugate-gradient method is really an iterative method for finding the minimum of a nonlinear function. Its use in solving sparse linear systems stems from recognizing that the minimum of

$$f(x) = \frac{1}{2}\left|Ax - b\right|^2$$

is unique and occurs for a value of x that satisfies the equation $Ax=b$. *MATLAB* has several built-in conjugate-gradient solvers, including `cgs`, which uses the "conjugate gradients squared" method. The syntax is `x=cgs(A,b)` where $Ax=b$ is the system to be solved (A is an $n \times n$ square matrix and b has length n). Error tolerance and maximum number of iterations can be set as described in "`help cgs`". Another *MATLAB* conjugate-gradient solver is `pcg`, which uses a preconditioned conjugate gradient method. However, for `pcg` the $n \times n$ coefficient matrix A must be positive definite and symmetric, which is often not true of the coefficient matrices in finite difference solutions to partial differential equations.

7.6. Problem

For the problem on the Brookhaven landfill from Chapter 6.7, use the Jacobi, SOR, and conjugate-gradient methods to solve the finite-difference equations. Write your own code for the Jacobi and SOR methods; use *MATLAB*'s `cgs` function for the conjugate-gradient method. Compare the number of iterations needed to achieve a solution using the different methods. (The number of iterations can be set in `cgs`.) Examine the effect of changing the value of the convergence criterion. Examine the effect of changing grid spacing. (Note that *MATLAB's* preconditioned conjugate gradients solver, `pcg`, will not work for this case because the coefficient matrix is not symmetric.)

7.7. References

Hill, M.C., Preconditioned conjugate-gradient 2(PCG2), a computer program for solving ground-water flow equations. U.S. Geological Survey Water-Resources Investigations Report 90-4048, 1990.

McDonald, M.G. and A.W. Harbaugh, A modular three-dimensional ground-water flow model. U.S. Geological Survey Techniques of Water-Resources Investigations, Book 6, Chap. A1, 586 pp., 1988.

Remson, I., Hornberger, G.M., and F.J. Molz, *Numerical Methods in Subsurface Hydrology*, 389 pp., Wiley-Interscience, New York, 1971.

Wang, H.F. and M.P. Anderson, *Introduction to Groundwater Modeling: Finite Difference and Finite Element Methods*, 237 pp., W.H. Freeman, San Francisco, 1982.

Box 7.1. L_p norms and circles

The general equation for the L_p norm is

$$\|x\|_p = [x_1^p + x_2^p + x_3^p + \circ\circ\circ + x_n^p]^{1/p}$$

As noted in the text, p=2 gives the "standard" distance; p=1 gives the sum of absolute values; and p=∞ gives the maximum component. Other values of p don't have as intuitive a meaning. One interesting way to think of these norms is in terms of the associated circles. In two dimensions, the line of constant value of a norm is a circle. The L_2 (Euclidean) norm produces the circle we are used to. But other norms also correspond to "circles." We can view these "circles" using the following m-file.

```
function y=isoLp(p)
x=[];y=[];
x=0:0.01:1;
y=(1-x.^p).^(1/p);
plot(x,y,-x,y,'b',x,-y,'b',-x,-y,'b')
axis('square')
```

The figure below shows the resulting "circles" for four values of p.

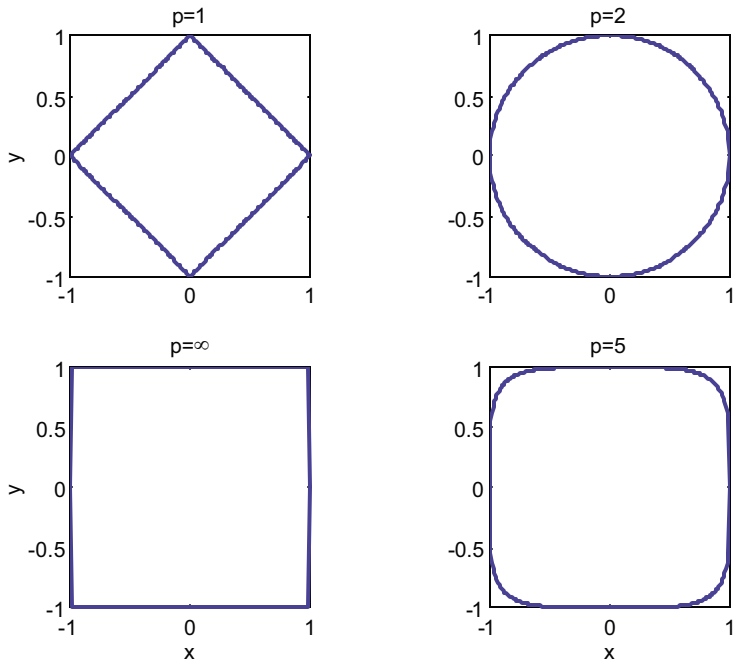


Figure B7.1. "Circles" associated with the L_p norm for several values of p.

Box 6.2. MODFLOW

The most widely used computer program in the world for simulating groundwater flow MODFLOW, a modular code developed by the U.S. Geological Survey in the early 1980's and continually improved since then (McDonald and Harbaugh, 1988; Harbaugh et al., 2000). The code along with documentation can be downloaded from the USGS Web site.

MODFLOW uses the finite difference method to solve the groundwater equations using the block centered node approach. An aquifer system is divided into rectangular blocks by a grid organized by rows, columns, and layers. Each block is called a "cell." Hydraulic properties are assigned to the cells, and boundary conditions are set by specifying cells to be constant head, no-flow, and so forth (see Figure B6.2.1).

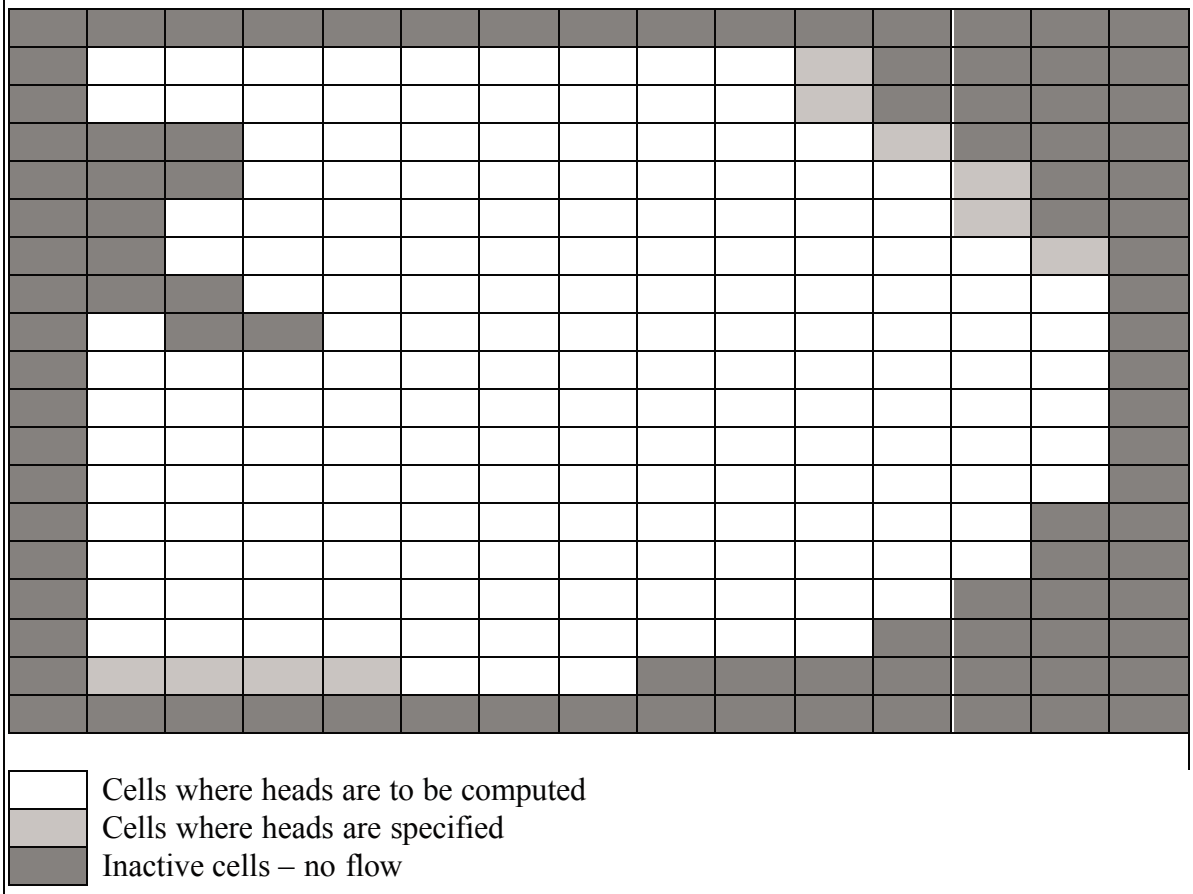


Figure B6.2.1. Example of a MODFLOW grid. Finite difference equations are written for cell-centered nodes.

CHAPTER 8

Finite Difference Solutions for Transient Problems

Numerical Methods in the Hydrological Sciences
Special Publication Series 57

doi:10.1029/057SP13

8. Finite Difference Solutions for Transient Problems

8.1. Background

Hydrological problems often involve time as a variable. The propagation of flood waves through river systems, the decline in water levels in aquifers as water is pumped from wells, the speading of a contaminant through a waterway or aquifer, and the drying of surface layers of soil by evapotranspiration are examples of problems where the variablesof interest – river stage, ground-water head, or soil moisture – vary temporally.

A particularly important type of equation used to describe *transient* (i.e., time-varying) problems in hydrology is the *heat equation* (Chapter 6.1). This equation describes not only heat flow in the natural environment (e.g., see Domenico and Schwartz 1998), but such things as diffusion of chemicals in sediments (e.g., Boudreau 1997), the movement of flood waves in channels using the diffusion analogy (e.g., Bras 1990), and the change in groundwater head in an aquifer (e.g., Fetter 2001).

8.2. Unidirectional flow in an aquifer

To illustrate the numerical methods used to solve the heat equation, consider a horizontal aquifer with constant formation properties, T (transmissivity) and S (storativity)[1]. The head in the aquifer is taken to vary with only one spatial coordinate, x. The equation that describes the variation of head in time and space is then

$$\frac{\partial h}{\partial t} = \frac{T}{S}\frac{\partial^2 h}{\partial x^2}. \tag{8.1}$$

Further suppose that the aquifer is bounded by constant-head streams at x=0 and 2 km and that at some initial time the head in the aquifer is elevated at the center of the aquifer relative to the streams because of a recharge event. The head distribution is given by

$$h = 0.1x, \qquad \text{for } 0 \le x \le 1000 \text{ m}$$
$$h = 0.1(2000 - x), \quad \text{for } 1000 \text{ m} \le x \le 2000 \text{ m}.$$

The parameter T/S for the aquifer is 151.5 $m^2 s^{-1}$. The problem is to determine heads in the aquifer as a function of time assuming that recharge has ceased.

8.3. A forward difference (or *explicit*) approximation

Finite difference approximations for transient problems involve differencing both time and space derivatives. We will adopt the convention of using a "j" superscript to denote grid points in time and an "i" subscript to denote space grid points. Thus h_i^j denotes head at spatial grid point "i" and time grid point "j".

[1] For transient conditions, the derivation of the conservation equation shown in Box 6.1 must be adjusted to accommodate changes in time within the control volume. This leads to inclusion of the time derivative of head multiplied by a storage coefficient. See a text on hydrogeology, e.g., Fetter (2001), for more details.

One finite-difference approximation to equation (8.1) is formed by using the heads at time j in the approximation of the right-hand side:

$$\frac{h_i^{j+1} - h_i^j}{\Delta t} = \frac{T}{S}\left(\frac{h_{i+1}^j - 2h_i^j + h_{i-1}^j}{(\Delta x)^2} \right). \tag{8.2}$$

The way that we solve time-varying problems is to start at an initial time when conditions are known and then calculate conditions one time step into the future. Conditions at this step in the future are then "known" and one can proceed to calculate conditions at $2\Delta t$. And so forth. Thus, in general terms, conditions at time "j" are known and conditions at time "j+1" are to be calculated. Because the heads on the right side of equation (8.2) are at the "j" time, the approximation to the time derivative is a forward-difference approximation.

The unknowns in equation (8.2) form a vector $\left[h_1^{j+1} \; h_2^{j+1} \; h_3^{j+1} \ldots \; h_n^{j+1} \right]'$. We can write equation (8.2) in matrix-vector form as

$$h^{j+1} = W h^j \tag{8.3}$$

where W has diagonal entries equal to $\left(1 - 2\left(\frac{T\Delta t}{S(\Delta x)^2} \right)\right)$ and sub- and super-diagonal entires equal to $\frac{T\Delta t}{S(\Delta x)^2}$. The method is *explicit* because each component equation of the set of equations (8.3) is independent of the others. The equations are not linked directly, so no solution of a system of equations is required. Equation (8.3) requires only matrix multiplication.

Let's look at how the solution works. We choose a spacing in the x direction of 250 m and a time step of 206 s $\left[= (\Delta x)^2 \, S / 2T \right]$.

```
% Explicit solution to example one-D groundwater problem
% Set parameter values and grid spacing
dx=250; x=250:dx:1750; ToverS=151.5; dt=dx^2/(2*ToverS);
h_old=[0.1*x(1:4) 0.1*(2000-x(5:7))]';  %Initial head values
alpha=ToverS*dt/dx^2;   %Transmissivity divided by storage coefficient
M_diag=sparse(1:7,1:7,1-2*alpha,7,7);  %Diagonal of W matrix
L_diag=sparse(2:7,1:6,alpha,7,7);  %Off-diagonal of W matrix
W=M_diag+L_diag+L_diag';   %W matrix
hh=zeros(7,10);  %Pre-allocate matrix for solution
for j=1:10   %Do ten time steps
      h_new=W*h_old;          % The explicit solution
      hh(:,j)=h_new;
      h_old=h_new;
end
hh'
ans =
   25.0000   50.0000   75.0000   75.0000   75.0000   50.0000   25.0000
   25.0000   50.0000   62.5000   75.0000   62.5000   50.0000   25.0000
   25.0000   43.7500   62.5000   62.5000   62.5000   43.7500   25.0000
```

```
   21.8750   43.7500   53.1250   62.5000   53.1250   43.7500   21.8750
   21.8750   37.5000   53.1250   53.1250   53.1250   37.5000   21.8750
   18.7500   37.5000   45.3125   53.1250   45.3125   37.5000   18.7500
   18.7500   32.0312   45.3125   45.3125   45.3125   32.0312   18.7500
   16.0156   32.0312   38.6719   45.3125   38.6719   32.0312   16.0156
   16.0156   27.3438   38.6719   38.6719   38.6719   27.3438   16.0156
   13.6719   27.3438   33.0078   38.6719   33.0078   27.3438   13.6719
```

We can do the comparison with the analytical solution as well (Figure 8.1) and show that the approximate nature of the numerical solution is evident with the relatively coarse spatial grid spacing used in this example.[2]

```
% Analytical solution is an infinite Fourier series; see, e.g.,
% Carslaw and Jaeger 1959
time=0:dt:10*dt; fdtime=time(2:11);  % set time vector for solution
x1=250; x2=750;  % choose values of x for solution
n=0:25;   % use 25 terms in Fourier series solution
h250=zeros(1,11); h750=zeros(1,11);  % preallocate solution vectors
% compute the exact solution for 11 times
for j=1:11
     h250(j)=800*sum((ones(size(n))./(pi^2*(2*n+1).^2)).*cos((2*n+1).* ...
        pi*(x1/1000-1)/2).*exp(0.3738*(2*n+1).^2*time(j)/1000));
     h750(j)=800*sum((ones(size(n))./(pi^2*(2*n+1).^2)).*cos((2*n+1).*...
        pi*(x2/1000-1)/2).*exp(-0.3738*(2*n+1).^2*time(j)/1000));
end
plot(fdtime,hh(1,:),'or',fdtime,hh(3,:),'or',time,h250,'b',time,h750,'b')
xlabel('Time, seconds')
ylabel('Heads at 250m (lower curve) and 750m (upper curve)')
```

[2] The analytical solution is in the form of an infinite Fourier series. See Carslaw and Jaeger (1959).

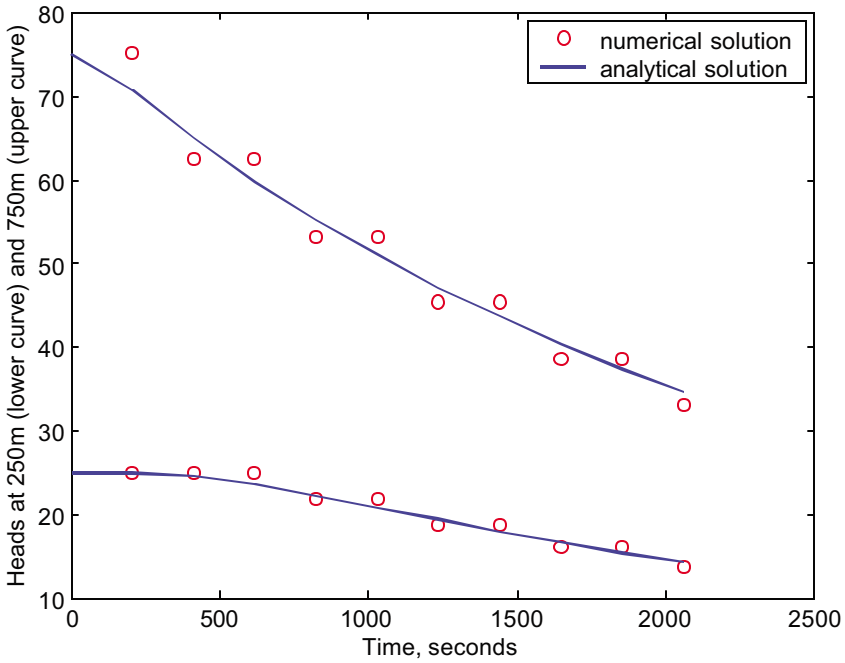


Figure 8.1. Heads calculated using explicit method with the analytical solution.

8.4. Stability problems with the explicit method

The explicit method is attractive because of its simplicity. To be useful, however, the time steps in the explicit method must be kept small. To illustrate the problem, let's look at the solution above for a Δt double that of our original choice, i.e. $\Delta t = (\Delta x)^2 S / T$.

```
% set time and space steps and parameters
dx=250; x=250:dx:1750; ToverS=151.5; dt=dx^2/(ToverS);
alpha=ToverS*dt/dx^2;
% place initial heads in vector h_old
h_old=[0.1*x(1:4) 0.1*(2000-x(5:7))]';
% construct the finite-difference matrix
M_diag=sparse(1:7,1:7,1-2*alpha,7,7);
L_diag=sparse(2:7,1:6,alpha,7,7);
W=M_diag+L_diag+L_diag';
% preallocate solution matrix
hh=zeros(7,10);
for j=1:10              % solve the equations for 10 time steps
      h_new=W*h_old;
      hh(:,j)=h_new;
      h_old=h_new;
end
hh(1:4,:)'

ans =
          25          50          75          50
```

```
    25          50          25         100
    25           0         125         -50
   -25         150        -175         300
   175        -350         625        -650
  -525        1150       -1625        1900
  1675       -3300        4675       -5150
 -4975        9650      -13125       14500
 14625      -27750       37275      -40750
-42375       79650     -105775      115300
```

Obviously, something is drastically wrong! The calculated heads are oscillating with absolute values getting ever larger with simulation time. This is a classic example of instability.

The way to study stability problems is to look at how errors (in computation there will *always* be small round-off errors) propagate through the solution. If the true solution to the finite-difference equations is given by H, the best that we can calculate is $H+\varepsilon$, where ε is the error. For one of the individual equations of (8.3), the true computation should be

$$H_i^{j+1} = rH_{i-1}^j + (1-2r)H_i^j + rH_{i-1}^j \tag{8.4}$$

where $r = \dfrac{T\Delta t}{S(\Delta x)^2}$. Our computation is imperfect, however, and is actually given by the equation

$$H_i^{j+1} + \varepsilon_i^{j+1} = r\left(H_{i-1}^j + \varepsilon_{i-1}^j\right) + (1-2\,r)\left(H_i^j + \varepsilon_i^j\right) + r\left(H_{i+1}^j + \varepsilon_{i+1}^j\right). \tag{8.5}$$

Subtracting (8.4) from (8.5), we find that the errors are governed by exactly the same equation as are the true heads:

$$\varepsilon_i^{j+1} = r\left(\varepsilon_{i-1}^j\right) + (1-2r)\left(\varepsilon_i^j\right) + r\left(\varepsilon_{i+1}^j\right). \tag{8.6}$$

A rigorous analysis of errors can be made in a number of ways. It turns out that we can get the right answer using a very simple analysis. We want a way to calculate a bound on the error at time j+1. Looking at equation (8.6), we ask, "What is the *worst* thing that could happen?" (It turns out that in computation, the worst thing always does happen.) Clearly, the error at time j+1 will be largest if ε at the i-1 and i+1 spatial nodes have the opposite sign to the error at node i. Without loss of generality, we can assume that the errors are also equal in magnitude. In that case (8.6) becomes:

$$\varepsilon_i^{j+1} = (1-4r)\varepsilon_i^j. \tag{8.7}$$

Noting that this equation would serve as a bound for all values of j, we find that, after m time steps, the error is related to any initial error by

$$\varepsilon_i^{j+1} = (1-4r)^m\,\varepsilon_i^0. \tag{8.8}$$

We see that, if $(1\text{-}4r) > 1$ in magnitude, the error will grow as a power of the number of time steps. Such unbounded growth in error is what we mean by instability. The stability criterion for the explicit method is read directly from (8.8).

$$|1-4r| \le 1, \quad \text{or}$$

$$r \le \frac{1}{2}.$$

The stability restriction renders the explicit method cumbersome for many applications. For any approximation to be reasonably good, Δx must be kept small. Because Δt must be scaled by $(\Delta x)^2$, each time step is very small and a large number of computational steps may be required. If the stability restriction on the time step could be removed, Δt could be selected independently of Δx and the computation could be much more efficient. A modification to the finite-difference approximation accomplishes just that aim.

8.5. A backward difference (or *implicit*) approximation

To show how to avoid the rather strict stability criterion for explicit methods, let us revisit the finite difference equation (8.2) and see what happens if we use the unknowns at the j+1 time step to approximate the spatial derivative.

$$\frac{h_i^{j+1} - h_i^j}{\Delta t} = \frac{T}{S}\left(\frac{h_{i+1}^{j+1} - 2h_i^{j+1} + h_{i-1}^{j+1}}{(\Delta x)^2} \right). \tag{8.9}$$

With this approximation, we are "looking backward" in time from the time step at which we take the spatial derivative, so the finite difference approximation in time is a backward difference. With this equation, if we place the "unknowns" (all heads at the j+1 time step) on the left-hand side of the equation and all the "knowns" (the heads at time j) on the right-hand side, we get

$$Ah^{j+1} = h^j, \tag{8.10}$$

where the diagonal elements of A are (1+2r) and the sub- and super-diagonal elements are -r. The solution is implicit because a system of matrix-vector equations must be solved.

How about the stability? The same type of analysis that we used for the explicit equation gives the equation for the error as

$$-r\varepsilon_{i-1}^{j+1} + (1+2r)\varepsilon_i^{j+1} - r\varepsilon_{i+1}^{j+1} = \varepsilon_i^j.$$

The absolute worst that can happen in this case is for the errors at time j+1 for the i-1 and i+1 nodes to be equal to the error for node i. Then the error at the j+1 time step would be equal to the error at the j time step. No explosive growth. No instability. Even under the worst conditions. Thus, the implicit method is unconditionally stable for the ground-water equation. We can use any time steps that we want. Of course, large time steps will not result in a very accurate solution to the equation, but the solution will be stable.

How does the implicit method work in practice? The modification to the *MATLAB* code for the explicit method isn't very difficult.

```
% set time and space steps and parameters
dx=250; x=250:dx:1750; ToverS=151.5; dt=dx^2/(2*ToverS);
alpha=ToverS*dt/dx^2;
```

```
% place initial heads in vector h_old
h_old=[0.1*x(1:4) 0.1*(2000-x(5:7))]';
% construct the finite-difference matrix
M_diag=sparse(1:7,1:7,1+2*alpha,7,7);
L_diag=sparse(2:7,1:6,-alpha,7,7);
A=M_diag+L_diag+L_diag';
% preallocate solution matrix
hh=zeros(7,10);
for j=1:10                    % solve the equatios for 10 time steps
      h_new=A\h_old;          % solution uses the backslash command
      hh(:,j)=h_new;
      h_old=h_new;
end
hh'

ans =
   24.7423   48.9691   71.1340   85.5670   71.1340   48.9691   24.7423
   24.1471   47.1038   66.3301   75.9486   66.3301   47.1038   24.1471
   23.2616   44.7521   61.5390   68.7438   61.5390   44.7521   23.2616
   22.1756   42.1791   57.0369   62.8903   57.0369   42.1791   22.1756
   20.9760   39.5530   52.8777   57.8840   52.8777   39.5530   20.9760
   19.7308   36.9713   49.0482   53.4661   49.0482   36.9713   19.7308
   18.4872   34.4872   45.5190   49.4926   45.5190   34.4872   18.4872
   17.2755   32.1276   42.2606   45.8766   42.2606   32.1276   17.2755
   16.1137   29.9039   39.2467   42.5616   39.2467   29.9039   16.1137
   15.0115   27.8186   36.4551   39.5083   36.4551   27.8186   15.0115
```

The numerical solution is "smoother" for the implicit method than for the explict method (compare Figure 8.2 with Figure 8.1).

To illustrate that the solution is stable for larger values of r, rerun the example with r=1, the value that led to explosive instability with the explicit method. As the results on the next page show, the calculated heads "behave" even when r exceeds 0.5. It is the ability to take relatively large time steps that leads to the preference for the use of implicit methods in finite-difference solutions of ground-water-flow problems.

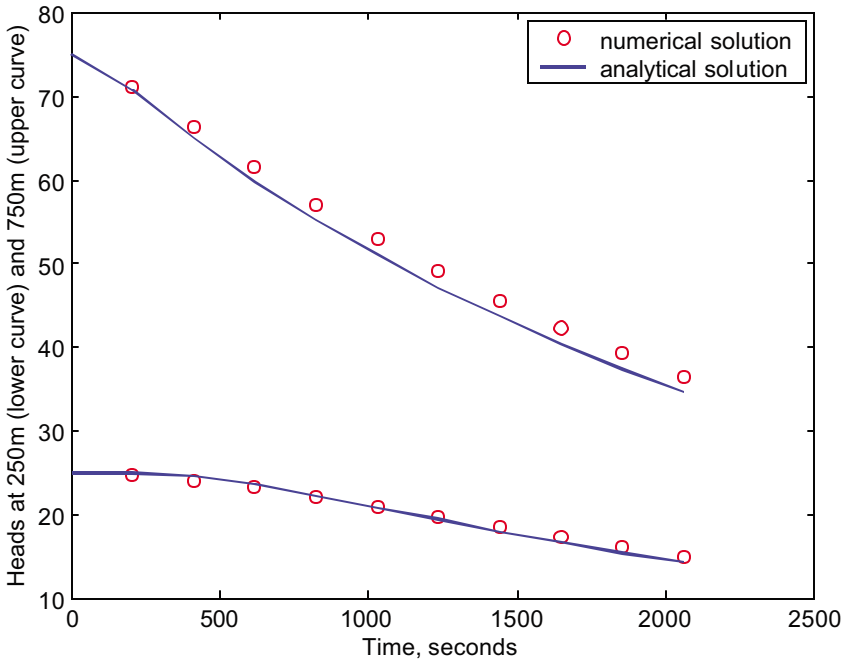


Figure 8.2. Heads calculated using implicit method with the analytical solution.

```
dx=250; x=250:dx:1750; ToverS=151.5; dt=dx^2/(ToverS);
h_old=[0.1*x(1:4) 0.1*(2000-x(5:7))]';
alpha=ToverS*dt/dx^2;
M_diag=sparse(1:7,1:7,1+2*alpha,7,7);
L_diag=sparse(2:7,1:6,-alpha,7,7);
A=M_diag+L_diag+L_diag';
hh=zeros(7,10);
for j=1:10
      h_new=A\h_old;
      hh(:,j)=h_new;
      h_old=h_new;
end
hh'

ans =
   23.9362   46.8085   66.4894   77.6596   66.4894   46.8085   23.9362
   22.0349   42.1684   57.6618   64.3278   57.6618   42.1684   22.0349
   19.7685   37.2706   49.8750   54.6926   49.8750   37.2706   19.7685
   17.4690   32.6386   43.1762   47.0150   43.1762   32.6386   17.4690
   15.3076   28.4537   37.4150   40.6150   37.4150   28.4537   15.3076
   13.3526   24.7502   32.4444   35.1679   32.4444   24.7502   13.3526
   11.6191   21.5048   28.1451   30.4860   28.1451   21.5048   11.6191
   10.0979   18.6745   24.4207   26.4425   24.4207   18.6745   10.0979
    8.7699   16.2120   21.1915   22.9418   21.1915   16.2120    8.7699
    7.6140   14.0721   18.3904   19.9075   18.3904   14.0721    7.6140
```

8.6. The γ method

We can "mix and match" the explicit and implicit methods. That is, we can approximate the spatial derivative as γ times the finite-difference approximation for the spatial derivative at the j+1 time step plus (1-γ) times the approximation at time j. This results in a set of equations

$$Gh^{j+1} = Hh^{j},$$

where the diagonal elements of G are (1+2$r\gamma$) and the sub- and super-diagonal elements are $-r\gamma$, and the diagonal elements of H are [1-2r(1-γ)] and the sub- and super-diagonal elements are r(1-γ). When γ=0, we recover the explicit method; if γ=1, we get the implicit method.

For γ=1/2, this method is known as the Crank-Nicolson method. Recall that the forward- and backward-difference approximations to a first derivative in time have truncation errors O(Δt) whereas the central-difference approximation has truncation error O((Δt)^2). The Crank-Nicolson approximation is, in a sense, a central-difference approximation in time. In fact, one can show that the truncation error for this case is O((Δt)^2).

8.7. Flow in an unsaturated soil

Problems involving flow in an unsaturated soil generally are attacked by solving the Richards equation [Box 8.1]. Such problems are quite difficult because the relationships between matric potential, ψ, and moisture content, θ, and between hydraulic conductivity, K, and moisture content are highly nonlinear. The material in the following section is meant to serve as an introduction to how such problems can be approached.

Consider the problem of vertical flow of soil moisture. The Richards equation for this problem can be written (with "z" measured vertically downward from the surface)

$$\frac{\partial \theta}{\partial t} = \frac{\partial}{\partial z}\left[K(\theta)\frac{\partial \psi}{\partial z}\right] - \frac{\partial K(\theta)}{\partial z}. \qquad (8.11)$$

There are a number of ways for expressing the dependence of ψ and K on θ. One of these is as follows.

$$s = \frac{\theta}{n}, \quad \text{where } n \text{ is the porosity;}$$

$$\psi\ (s) = \psi_{sat} s^{-1/m}, \quad \text{where } m \text{ is a parameter;}$$

$$K(s) = K_{sat} s^{c}, \quad \text{where } c \text{ is a parameter.}$$

where K_{sat} and ψ_{sat} are the values of these parameters for s=1. Given these forms for the nonlinear relationships, equation (8.11) can be written

$$\frac{\partial \theta}{\partial t} = \frac{\partial}{\partial z}\left[K_{sat}\left(\frac{\theta}{n}\right)^{c}\left(\frac{-\psi_{sat}}{m}\right)\left(\frac{1}{n}\right)\left(\frac{\theta}{n}\right)^{-1-1/m}\frac{\partial \theta}{\partial z}\right] - \left(K_{sat} c\left(\frac{1}{n}\right)\left(\frac{\theta}{n}\right)^{c-1}\right)\frac{\partial \theta}{\partial z}.$$

The equation is obviously nonlinear. There are no "standard" ways to solve such equations. Suffice it to say that solution is often achieved with considerable difficulty.

Solutions to nonlinear equations can be obtained by "linearizing" the nonlinear equations and accepting the solution to the linear equations as an approximation to the solution to the nonlinear equations. Often, iterative methods are used. One non-iterative approach for solving the nonlinear finite-difference soil moisture equations is a "predictor-corrector" method. In predictor-corrector methods, the nonlinear terms are taken out of the equations by substituting known values in these terms. The basic idea behind predictor-corrector methods is to eliminate the nonlinear terms by approximating them with values at the j^{th} time level, solve these equations to get a "predicted" value of the dependent variable at, say, time level $j+1/2$, use these "predicted" values to linearize the nonlinear terms in the original equation, and obtain "corrected" values by solving these equations.

To implement a predictor-corrector method for the soil-moisture equation, it is rewritten as:

$$\frac{\partial^2\theta}{\partial z^2} = \frac{1}{y}\frac{\partial\theta}{\partial t} - \left[\frac{u}{y}\frac{\partial\theta}{\partial z} - \frac{w}{y}\right]\frac{\partial\theta}{\partial z}$$

where y, u, and w are functions of moisture content:

$$y = K\frac{d\psi}{d\theta}, \quad w = \frac{dK}{d\theta}, \text{ and } u = \frac{dK}{d\theta}\frac{d\psi}{d\theta} + K\frac{d^2\psi}{d\theta^2}.$$

Note that the y, w, and u terms are the nonlinear terms in the equation; if these were *known* and constant (i.e., not functions of θ), the soil-moisture equation would be linear. The equation is solved from t^j to t^{j+1} in two steps. First a step (the "predictor") is made to an intermediate point half way between t^j and t^{j+1}. Then, using the results from this half step, the solution is advanced to time t^{j+1} (the "corrector"). The predictor equation is:

$$\frac{\theta_{i-1}^{j+1/2} - 2\theta_i^{j+1/2} + \theta_{i+1}^{j+1/2}}{\Delta z^2} = \frac{1}{y_i^j}\frac{\theta_i^{j+1/2} - \theta_i^j}{\Delta t/2} - \left[\frac{u_i^j}{y_i^j}\left(\frac{\theta_{i+1}^j - \theta_{i-1}^j}{2\Delta z}\right) - \frac{w_i^j}{y_i^j}\right]\left(\frac{\theta_{i+1}^j - \theta_{i-1}^j}{2\Delta z}\right)$$

and the corrector equation is:

$$\frac{1}{2}\left[\frac{\theta_{i-1}^{j+1} - 2\theta_i^{j+1} + \theta_{i+1}^{j+1}}{\Delta z^2} + \frac{\theta_{i-1}^{j} - 2\theta_i^{j} + \theta_{i+1}^{j}}{\Delta z^2}\right] =$$

$$\frac{1}{y_i^j}\frac{\theta_i^{j+1} - \theta_i^j}{\Delta t} - \left[\frac{u_i^{j+1/2}}{y_i^{j+1/2}}\left(\frac{\theta_{i+1}^{j+1/2} - \theta_{i-1}^{j+1/2}}{2\Delta z}\right) - \frac{w_i^{j+1/2}}{y_i^{j+1/2}}\right]\left(\frac{1}{2}\right)\left(\frac{\theta_{i+1}^{j+1} - \theta_{i-1}^{j+1}}{2\Delta z} + \frac{\theta_{i+1}^{j} - \theta_{i-1}^{j}}{2\Delta z}\right)$$

The predictor uses an implicit method covering a time step $\Delta t/2$, linearized by approximating y,w, and u at the j^{th} time level. The corrector is a Crank-Nicholson method covering the full time step – level j to level $j+1$ – and linearized by approximating y, w and u using the results from the predictor, i.e., the $j+1/2$ values. (See Remson et al., 1971 for details.)

An implementation of the predictor-corrector in *MATLAB* may be helpful. Consider the problem of infiltration into an initially dry sandy loam soil (K_s=3.4 x 10^{-3} cm s^{-1}, ψ_s=-25 cm, n=0.25, m=5.4, c=3.4). Moisture content at the start of the infiltration event is equal to 0.10

everywhere. The moisture content at the soil surface is instantaneously raised to 0.25 and held there (e.g., by applying a pond of water in an infiltrometer). The time evolution of the moisture profile can be studied by integrating the Richards equation numerically.

```
% 3 function files precede the main code
function y=Kdpsi(param,theta)     % function "y"
Ks=param(1); c=param(2); n=param(3); psis=param(4); m=param(5);
y=(Ks*(-psis/m)*(theta/n).^(c-1-1/m))/n;
function w=dk(param,theta)        % function "w"
Ks=param(1); c=param(2); n=param(3); psis=param(4); m=param(5);
w=(Ks*c*(theta/n).^(c-1))/n;
function u=dKdpsi(param,theta)      % function "u"
Ks=param(1); c=param(2); n=param(3); psis=param(4); m=param(5);
u1c=Ks*c*(-psis/m);
u2c=Ks*psis*(1/m)*(1+1/m);
u=((u1c+u2c)*(theta/n).^(c-2-1/m))/n^2;
% main predictor-corrector loop (using the y, w, and u functions above)
Ks=3.4e-3; psis=-25; n=0.25; m=3.3; c=3.6;   %Values of parameters
param=[Ks c n psis m]; nz=30;
dz=100/nz; dt=60;                       %set time and space steps
ysat=-Ks*psis/m; wsat=Ks*c;
theta_old=ones(nz+1,1)*0.1;
theta_old(1)=0.25;
output=zeros(15,nz+1)';
time=zeros(19,1);
outkount=1;
for j=1:75                            % time loop
% predictor to advance one-half of the time step
   y=Kdpsi(param,theta_old(2:nz));
   w=dK(param,theta_old(2:nz));
   u=dKdpsi(param,theta_old(2:nz));
   bracketterm=((u./y).*(theta_old(3:nz+1)-theta_old(1:nz-1))/(2*dz)-w./y);
   mterm=-2*ones(nz-1,1)/dz^2-2*ones(nz-1,1)./(y*dt);
   M_diag=sparse(1:nz-1,1:nz-1,mterm,nz-1,nz-1);
   Lterm=ones(nz-1,1)./dz^2;
   Uterm=ones(nz-1,1)./dz^2;
   L_diag=sparse(2:nz-1,1:nz-2,Lterm(2:nz-1),nz-1,nz-1);
   U_diag=sparse(1:nz-2,2:nz-1,Uterm(1:nz-2),nz-1,nz-1);
   A=M_diag+L_diag+U_diag;
   rhs=-2*theta_old(2:nz)./(y*dt)-(bracketterm/(2*dz)).*...
      (theta_old(3:nz+1)-theta_old(1:nz-1));
   rhs(1)=rhs(1)-low(1)*theta_old(1);
   rhs(nz-1)=rhs(nz-1)-up(nz-1)*theta_old(nz+1);
   soln=A\rhs;
   thetahalf=[theta_old(1);soln;theta_old(nz+1)];
% corrector to advance to the next time
   y=Kdpsi(param,thetahalf(2:nz));
   w=dK(param,thetahalf(2:nz));
   u=dKdpsi(param,thetahalf(2:nz));
   bracketterm=((u./y).*(thetahalf(3:nz+1)-thetahalf(1:nz-1))/(2*dz)-w./y);
   Mterm=-ones(nz-1,1)/dz^2-ones(nz-1,1)./(y*dt);
   M_diag=sparse(1:nz-1,1:nz-1,d,nz-1,nz-1);
   Lterm=ones(nz-1,1)./(2*dz^2)-bracketterm./(4*dz);
```

```
    Uterm=ones(nz-1,1)./(2*dz^2)+bracketterm./(4*dz);
    L_diag=sparse(2:nz-1,1:nz-2,Lterm(2:nz-1),nz-1,nz-1);
    U_diag=sparse(1:nz-2,2:nz-1,Uterm(1:nz-2),nz-1,nz-1);
    A=M_diag+L_diag+U_diag;
    rhs=-theta_old(2:nz)./(y*dt)-(theta_old(1:nz-1)-2*theta_old(2:nz)...
       +theta_old(3:nz+1))/(2*dz^2)-bracketterm.*(theta_old(3:nz+1)...
       -theta_old(1:nz-1))/(4*dz);
    rhs(1)=rhs(1)-low(1)*theta_old(1);
    rhs(nz-1)=rhs(nz-1)-up(nz-1)*theta_old(nz+1);
    soln=A\rhs;
    theta_new=[theta_old(1);soln;theta_old(nz+1)];theta_old=theta_new;
    if rem(j,15)==0
       output(:,outkount)=theta_new;
       time(outkount)=dt*15*outkount; outkount=outkount+1;
    end
end
zz=0:-1/nz:-1; axis([0 0.26 -1 0]);
plot(output(:,1),zz,output(:,2),zz,output(:,4),zz)
xlabel('Volumetric moisture content');ylabel('Depth, meters')
time=time/60;
text(0.11,-0.6,['profiles for times ' num2str(time(1)) ' ' num2str(time(2))
' ' num2str(time(4)) ' minutes'])
```

The results of the computation (Figure 8.3) show that the moisture propagates into the soil with a fairly sharp front. In cases with a sharp front, time and space increments need to be kept small to avoid "glitches" in the solution. (Try the code above with nz=20, instead of 30, for example.) You should keep your skepticism intact when you are generating solutions to complex equations and not believe everything immediately as it comes out of the computer.

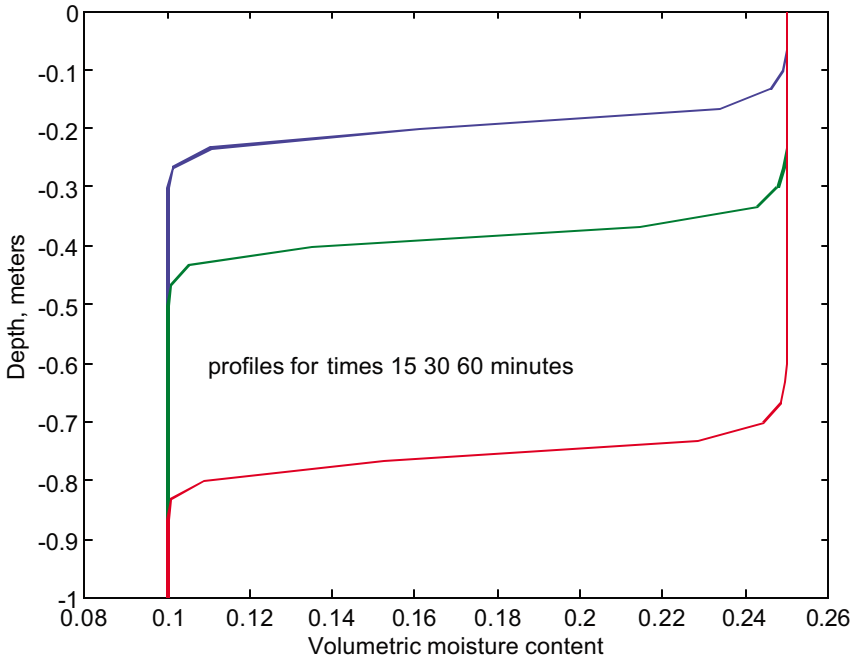


Figure 8.3 Solution to the Richards equation.

8.8. Problems

1. Heat flow in the earth is governed by the processes of conduction and convection. In regions where water is free to move, heat flow in the near surface (the top several hundred meters of the earth's surface) is strongly affected by convection and the analysis of temperature changes is quite complicated. In the arctic, however, permafrost essentially renders water motion meaningless as a heat-flow mechanism. In these areas, conduction is the primary mechanism by which heat is transported in the crust and a relatively simple analysis may be appropriate. The equation for such a problem is:

$$\frac{\partial T}{\partial t} = \frac{K}{\rho\, c}\frac{\partial^2 T}{\partial z^2}$$

where T is temperature, z is depth below the surface, t is time, K is thermal conductivity, c is heat capacity, and ρ is density.

Consider heat flow in the top 1km of the crust. The surface boundary condition is a specified temperature. The bottom boundary condition (at 1km) is that the upward heat flux, q, be equal to $K\Gamma_0$, where Γ_0 is the geothermal gradient, about 3°C per 100m. The thermal conductivity of rock and of permafrost is about 0.5 cal m^{-1} s^{-1} °C^{-1} and ρc is about 0.5 cal cm^{-3} °C^{-1}.

a) Write a code, using the γ method, to solve the temperature problem for permafrost regions. Explore the effect of changing grid spacing and γ.

b) Use your code to calculate the steady-state temperature profile for a surface temperature of -15°C. Start the computation with T=0°C everywhere.

c) Starting with the steady-state temperature profile as the initial condition, calculate the temperature profile every decade under conditions of a steadily increasing surface temperature at a rate of 3.5°C per century.

d) Mann et al. (1998) suggest that global surface temperature remained relatively steady for several centuries prior to the 20th century and then the temperature rose at a rate of about 0.5°C per century. Of course surface temperature trends at any locale can depart from the global mean trend. The table below gives measured temperatures from a borehole on the north slope of Alaska in 1984. Using your code, offer an interpretation of these data. By asking you to use your code, the intention is that you should be quantitative in your answer. You will want to run your code emphasizing the top tens of meters of the temperature profile – otherwise the fine detail of the changing temperature profile may be obscured by the coarse spatial discretization. [After you have finished this problem, you may want to look at the article by Lachenbruch and Marshall (1986) which is the source for these data. For a more up-to-date discussion of the analysis of borehole temperature profiles relative to climate, see Pollack and Huang (1998). For data from boreholes around the world, see http://www.ngdc.noaa.gov/paleo/borehole/borehole.html. Finally, in case you are prone to accept the results of the analysis of borehole temperatures uncritically, see Mann and Schmitt (2003) for a discussion of difficulties associated with inferring climate change from borehole temperature profiles.]

Depth below surface, meters	Temperature, degrees C
700	11.99
600	9.00
500	5.99
400	3.04
300	-0.01
200	-3.00
150	-4.49
125	-5.24
100	-5.92
90	-6.11
80	-6.37
75	-6.46
70	-6.50
65	-6.60
60	-6.69
55	-6.71
50	-6.75
45	-6.76
40	-6.78
35	-6.73

2. Explore the solution of the Richards equation for unsaturated flow. How are results sensitive to the time and space steps? How does the infiltration front progress for various soil types? (The values below are from Bras, 1990.)

SOIL TYPE	K_{sat} (cm s^{-1})	ψ_{sat} (cm)	n	m	c
Clay	3.4 x 10-5	-90	0.45	0.44	7.5
Silty loam	3.4 x 10-4	-45	0.35	1.2	4.7
Sand	8.6 x 10-3	-15	0.2	5.4	3.4

8.9. References

Boudreau, BP., *Diagenetic Models and their Implementation*, 414 pp., Springer-Verlag, Berlin, 1997.

Bras, R.L., *Hydrology,* 643 pp., Addison-Wesley, Reading, MA, 1990.

Carslaw, H.S. and J.C. Jaeger, *Conduction of Heat in Solids*, 2nd ed., 510 pp, Clarendon Press, Oxford, 1959.

Domenico, P.A. and F.W. Schwartz, *Physical and Chemical Hydrogeology*, 2nd ed., 506 pp., Wiley, New York, 1998.

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

Lachenbruch, A.H. and B.V. Marshall, Changing climate: geothermal evidence from permafrost in the Alaskan Arctic, *Science, 234*: 689-696, 1986.

Mann, ME, Bradley, RS, and MK Huges, Global-scale temperature patterns and climate over the past six centuries. *Nature, 392*: 779-787. 1998.

Mann ME and GA Schmidt, Ground vs. surface air temperature trends: Implications for borehole surface temperature reconstructions, *Geophys. Res. Let. 30*: art. no. 1607, 2003.

Pollack, HN and S. Huang, Underground temperatures reveal changing climate, *Geotimes, 43*:16-19, 1998.

Remson, I., Hornberger, G.M., and F.J. Molz, *Numerical Methods in Subsurface Hydrology*. 389pp., Wiley-Interscience, New York, 1971.

Box 8.1. Equation for flow in an unsaturated soil

Water in the subsurface can be conveniently divided into *groundwater*, or water under pressure > 0, and *soil moisture*, or water under pressure < 0. That is, generally speaking, soil moisture is water above the water table (the surface defined by $p = 0$) that is under "tension" (under pressure less than atmospheric).

Water in the unsaturated zone, like water in the saturated zone, moves down a gradient in *head*. For soil moisture, the total head, h, is taken as the sum of the head due to gravity, $-z$ (for the z axis directed downward), and capillary pressure head or matric head, ψ.

$$h = \psi - z.$$

Matric head is a function of the moisture content. That is, in a very dry soil, capillary (and other) forces hold the water very strongly (high ψ) while in a moist soil the water is held less strongly (lower ψ). Matric head also depends on pore size, which generally scales with grain size, so that finer-grained sediment or soil has a larger matric head at a given level of saturation than coarse-grained soils . The relationship between matric head, ψ, and moisture content, θ, for a given soil is known as the *matric characteristic*.

Darcy's law is used to describe the flow of water in the unsaturated zone. The law has the same form as for flow in saturated soils – specific discharge is proportional to the gradient in total head – but in the unsaturated zone, the hydraulic conductivity is a function of moisture content: $K=K(\theta)$. Darcy's law is then written

$$q = -K(\theta)\frac{dh}{dl}$$

where "l" represents a distance variable in the direction of flow.

For many applications we are concerned with the flow of soil moisture in the vertical, or "z" direction. In this case Darcy's law is (for z measured downward)

$$q_z = -K(\theta)\frac{dh}{dz} = -K(\theta)\frac{d}{dz}(\psi - z)$$

$$q_z = -K(\theta)\left[\frac{d\psi}{dz} - 1\right]$$

For the case of vertical flow of water, the appropriate continuity equation [Box 6.1] is:

$$\frac{\partial\theta}{\partial t} = -\frac{\partial q_z}{\partial z}$$

The left side of this equation represents the rate of change of mass in a small control volume, and the right side is the difference between the inflow rate and the outflow rate, each expressed on a per unit volume basis. Combining the continuity equation with Darcy's law results in the *Richards equation*.

$$\frac{\partial\theta}{\partial t} = \frac{\partial}{\partial z}\left[K(\theta)\frac{\partial\psi}{\partial z}\right] - \frac{\partial K(\theta)}{\partial z}.$$

Box 6.1. Groundwater flow equations

The basis of the equations used to describe the flow of groundwater is the conservation of mass equation, which, when applied to a fixed control volume, basically says that the rate of mass inflow minus rate of mass outflow equals rate of change of mass storage – what goes in minus what goes out equals the change in what's inside.

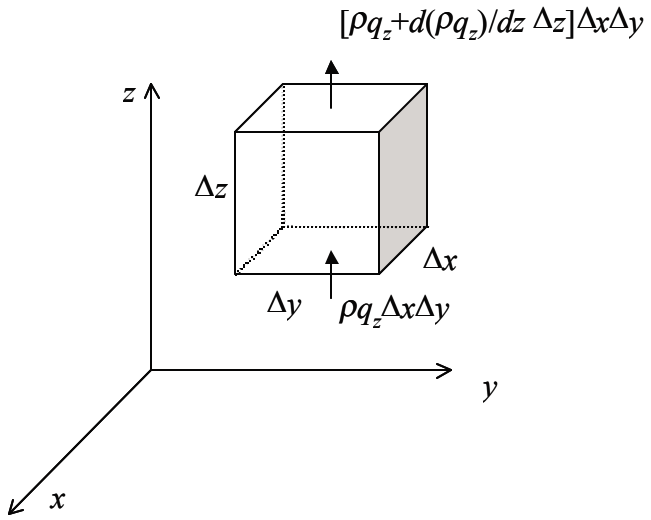


Figure B6.1.1. Control volume for deriving the conservation equation.

One way to derive the general equation of continuity for pore-fluid flow is to specify an arbitrary control volume to be a small rectangular parallelepiped in a fixed, Cartesian coordinate frame with sides of length Δx, Δy, and Δz (Fig. B6.1.1). Without any loss of generality, we take the directions designated by the arrows on the axes as positive (Fig. B6.1.1) and consider the case of positive flows. Consider first the inflow of mass into the control volume. The inflow into the parallelepiped in the z-direction is $\rho q_z \Delta x \Delta y$, where ρ is the density of water and q_z is the specific discharge (volumetric discharge per unit area) in the z direction. Density times the specific discharge gives the mass flux (mass per area per time) so multiplication by the area, $\Delta x \Delta y$, yields the mass inflow in the z direction. The mass flows in the x- and y-directions can be similarly calculated. Because specific discharge can change with distance, the value of q_z at the top face need not be the same as that at the bottom face. We can estimate the specific discharge at the top face using the Taylor series (Chapter 3.2). Because the distance separating the two faces (Δz) is as small as we wish to make it, we can get an acceptable approximation of the flux at the top face by just retaining the first two terms of the series (i.e., a linear extrapolation):

$$q_z(z + \Delta z) = q_z(z) + \frac{\partial q_z}{\partial z}\Delta z \qquad \text{(B6.1.1)}$$

The expression for the mass outflow in the z direction is then

$$\left[\rho\, q_z + \frac{\partial\, \rho q_z}{\partial z}\Delta z\right]\Delta x\Delta y \qquad \text{(B6.1.2)}$$

Now the expression that we need for the continuity equation is the **net** inflow of mass – the difference between the inflow and the outflow. For the z direction,

$$\text{net mass flow}_Z = (\rho q_z(z) - \rho q_z(z+\Delta z))\Delta x\Delta y = -\frac{\partial \rho q_z}{\partial z}\Delta x\Delta y\Delta z$$

Using similar expressions for the x- and y-directions, the total net mass inflow can be obtained:

$$\text{total net mass inflow} = -\left[\frac{\partial\, \rho q_x}{\partial x} + \frac{\partial\, \rho q_y}{\partial y} + \frac{\partial\, \rho q_z}{\partial z}\right]\Delta x\Delta y\Delta z \qquad \text{(B6.1.3)}$$

For *steady* flow there can be no change of mass in the control volume so the net inflow must be zero. If we further assume that the density is constant, equation (B6.1.3) implies that

$$\frac{\partial q_x}{\partial x} + \frac{\partial q_y}{\partial y} + \frac{\partial q_z}{\partial z} = 0 \qquad \text{(B6.1.4)}$$

The forces driving flow through an aquifer are due to gravity and pressure gradients. These forces, expressed on a per unit weight basis, are represented in groundwater flow equations in terms of a *head gradient*. Groundwater head is defined as pressure per unit weight plus elevation, which is the head due to gravity.

Darcy's law relates the specific discharge to the head gradient. Darcy's law states that this relationship is linear, with the constant of proportionality between specific discharge and head gradient being the *hydraulic conductivity*, K. If we make the assumption that the aquifer is *isotropic*, i.e., that K is independent of direction, Darcy's law can be written as follows.

$$\begin{aligned} q_x &= -K\frac{\partial h}{\partial x} \\ q_y &= -K\frac{\partial h}{\partial y} \\ q_z &= -K\frac{\partial h}{\partial z} \end{aligned} \qquad \text{(B6.1.5)}$$

Combining equations (B6.1.4) and (B6.1.5), we obtain an equation for steady groundwater flow.

$$\frac{\partial}{\partial x}\left(K\frac{\partial h}{\partial x}\right) + \frac{\partial}{\partial y}\left(K\frac{\partial h}{\partial y}\right) + \frac{\partial}{\partial z}\left(K\frac{\partial h}{\partial z}\right) = 0 \qquad \text{(B6.1.6)}$$

For the case of a *homogeneous* aquifer, one for which K is constant, equation (B6.1.6) reduces to the Laplace equation.

$$\frac{\partial^2 h}{\partial x^2}+\frac{\partial^2 h}{\partial y^2}+\frac{\partial^2 h}{\partial z^2}=0 \qquad \text{(B6.1.7)}$$

For the case of a horizontal aquifer of constant thickness, b, we assume that there is no vertical flow so equation B6.1.4 takes the form

$$b\frac{\partial q_x}{\partial x}+b\frac{\partial q_x}{\partial x}=0 \qquad \text{(B6.1.8)}$$

When (B6.1.8) is combined with Darcy's law, we obtain

$$\begin{aligned}&\frac{\partial}{\partial x}\left(Kb\frac{\partial h}{\partial x}\right)+\frac{\partial}{\partial x}\left(Kb\frac{\partial h}{\partial x}\right)=0\\&\frac{\partial}{\partial x}\left(T\frac{\partial h}{\partial x}\right)+\frac{\partial}{\partial x}\left(T\frac{\partial h}{\partial x}\right)=0\end{aligned} \qquad \text{(B6.1.9)}$$

where T is the *transmissivity* of the aquifer. If there is recharge to the aquifer, e.g., by slow flow through an overlying confining layer, the equation is modified accordingly:

$$\frac{\partial}{\partial x}\left(T\frac{\partial h}{\partial x}\right)+\frac{\partial}{\partial x}\left(T\frac{\partial h}{\partial x}\right)=-w \qquad \text{(B6.1.10)}$$

where w is the recharge rate.

If the aquifer is homogeneous, T is constant and can be brought outside the derivative in (B6.1.9). The result is

$$\frac{\partial^2 h}{\partial x^2}+\frac{\partial^2 h}{\partial y^2}=0 \qquad \text{(B6.1.11)}$$

so again we find that the Laplace equation describes the steady flow of groundwater through a horizontal aquifer.

Finally, note that Darcy's law indicates that flow is down the *gradient* in head. This implies that flow lines for groundwater in an isotropic aquifer are perpendicular to lines of constant head. In *MATLAB* this means that if groundwater heads are computed and contour lines drawn, the `gradient` and `quiver` commands can be used to depict the flow.

CHAPTER 9

Finite Difference Methods for Transport Equations

Numerical Methods in the Hydrological Sciences
Special Publication Series 57

doi:10.1029/057SP14

9. Finite Difference Methods for Transport Equations

9.1. Background

In Chapter 8, we considered problems related to time-dependent diffusion of quantities such as heat, solutes, and head. In those problems, diffusion was the only process responsible for the flux or transport of the quantity of interest. However, a solute in a moving fluid not only will be transported by diffusion, but will also be advected with the fluid. In this case both transport processes – advection and diffusion – must be included in the equation for the time-dependent change in concentration,

$$\frac{\partial c}{\partial t} + u\frac{\partial c}{\partial x} = D\frac{\partial^2 c}{\partial x^2} \tag{9.1}$$

where u is the advection velocity and D is the diffusion or dispersion coefficient. (Vertical and transverse variations in flow velocity increase rates of mixing compared to molecular or turbulent diffusion. The increased rate of mixing is commonly parameterized by a dispersion coefficient. See, e.g, Fischer et al., 1979 or Hemond and Fechner-Levy, 2000.) Equation (9.1) is commonly referred to as the advection-diffusion or advection-dispersion equation (or simply the transport equation). Here we'll refer to (9.1) as the advection-dispersion equation.

9.2. Numerical dispersion

When considering numerical solutions to the advection- dispersion equation, it is helpful to non-dimensionalize equation (9.1) using the dimensionless parameters $\hat{t} = tu/L$, and $\hat{x} = x/L$,

$$\frac{\partial c}{\partial \hat{t}} + \frac{\partial c}{\partial \hat{x}} = \frac{1}{\mathbf{Pe}}\frac{\partial^2 c}{\partial \hat{x}^2} \tag{9.2}$$

where L is a length scale. The dimensionless coefficient on the right-hand-side of the equation is the Peclet number, $\mathbf{Pe} = uL/D$. The Peclet number represents the ratio of the time scale for advection to the time scale for diffusion or dispersion. When the Peclet number is large, the term on the right can be neglected, i.e., dispersion is relatively unimportant. When the Peclet number is small, transport is dominantly by dispersion.

To illustrate one of the difficulties associated with solving equations having the form of the advection-dispersion equation numerically, we first consider the equation for large Peclet numbers. In this case the dispersion term is negligible compared to the advection term and the equation we want to solve is:

$$\frac{\partial c}{\partial \hat{t}} + \frac{\partial c}{\partial \hat{x}} = \frac{\partial c}{\partial t} + u\frac{\partial c}{\partial x} = 0.$$

Let c be the concentration of some tracer in a stream flowing at velocity u. The problem is to describe the downstream transport of the tracer when it is injected at location $x = 0$ such that the concentration in the stream at the injection point is held at 1 unit. That is, initially c is equal to zero everywhere, then $c = 1$ at $x = 0$ for all subsequent time. We can see

intuitively what the solution to the problem is: the concentration "front" ($c = 1$ on the upstream side and $c = 0$ on the downstream side) propagates downstream at speed u.

The advection equation provides an easy demonstration that apparently sensible finite-difference schemes do not always work. Take the straightforward explicit representation of the (dimensional version of the) advection equation:

$$c_i^{j+1} = c_i^j - \left(\frac{u\Delta t}{2\Delta x}\right)\left(c_{i+1}^j - c_{i-1}^j\right) \tag{9.3}$$

This benign-looking method is actually *unconditionally* unstable! That is, there are no values of the time and space steps, no matter how small, that will make this method stable. You can do the simple "worst-case" stability analysis (Chapter 8.3) to show this.

The Lax scheme often is used to solve equations in which the advective term is dominant. To use this scheme, we replace the c_i^j on the right-hand side of the finite-difference equation above with $\left(c_{i+1}^j + c_{i-1}^j\right)/2$.

$$c_i^{j+1} = \left(\frac{c_{i+1}^j + c_{i-1}^j}{2}\right) - \left(\frac{u\Delta t}{2\Delta x}\right)\left(c_{i+1}^j - c_{i-1}^j\right), \text{ or,}$$

$$c_i^{j+1} = \left(\frac{1}{2} + \frac{u\Delta t}{2\Delta x}\right)c_{i-1}^j + \left(\frac{1}{2} - \frac{u\Delta t}{2\Delta x}\right)c_{i+1}^j$$

Now if you use our simple stability analysis procedure, you will find that the Lax scheme is stable if the following stability criterion (referred to as the Courant condition) is met:

$$\frac{|u|\Delta t}{\Delta x} \le 1.$$

Let's look at the solution to our problem using the Lax scheme.

```
% Lax solution for the advection problem
c=zeros(1,21);       % initial conditions
c(1)=1;           % boundary condition
u=1; dx=2; dt=2;        % set velocity and grid steps
% note: solution for dt=dx/u=2 should preserve sharp front
for i=1:20
   c(2:20)=(0.5+u*dt/(2*dx))*c(1:19)+(0.5-u*dt/(2*dx))*c(3:21);
   cc(i,1:21)=c;
end
% repeat for dt=1.5
c=zeros(1,21);
c(1)=1;
dt=1.5;
for i=1:20
      c(2:20)=(0.5+u*dt/(2*dx))*c(1:19)+(0.5-u*dt/(2*dx))*c(3:21);
   cc2(i,1:21)=c;
end
% plot the profile at t=24 (12 steps for dt=2; 16 for dt=1.5)
z=0:dx:40;
```

```
plot(z,cc(12,:),'b',z,cc2(16,:),'r')
xlabel('Distance from injection point')
ylabel('Reduced concentration')
legend('dt=2','dt=1.5')
```

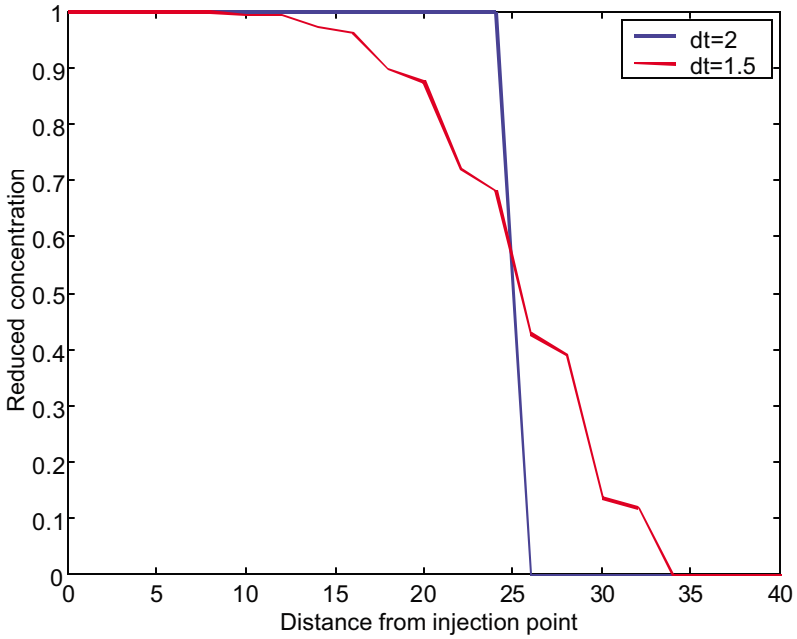


Figure 9.1. Lax solution for a simple advection problem for two time steps

Curiously, the solution for $\Delta t = 2$ is better than the solution for $\Delta t = 1.5$. What is going on? It turns out that the change we made to ensure stability was not totally benign. Let's consider our modification of equation (9.3) more carefully. The forward-in-time, central-in-space approximation that is unconditionally unstable is:

$$\frac{c_i^{j+1}}{\Delta t} = \frac{c_i^j}{\Delta t} - \left(\frac{u}{2\Delta x}\right)\left(c_{i+1}^j - c_{i-1}^j\right)$$

To achieve stability, we replaced the first term on the right-hand side with an average of the concentrations at the surrounding nodes. It turns out that this is equivalent to adding the term $\frac{1}{2}\frac{(\Delta x)^2}{\Delta t}\left(\frac{c_{i+1}^j - 2c_i^j + c_{i-1}^j}{(\Delta x)^2}\right)$ to the right-hand side of the equation. But note that the term that we had to add is a numerical approximation to a dispersion term equal to $\frac{1}{2}\frac{(\Delta x)^2}{\Delta t}\frac{\partial^2 c}{\partial x^2}$. That is, we introduced a *numerical* dispersion term.

Of course, the dispersion term in a differential equation is not exactly represented by the numerical approximation – recall that we truncated a Taylor series to get the approximation to the second derivative [Chapter 3.2]. The actual differential equation that we are approximating in using the Lax scheme can be written

$$\frac{\partial c}{\partial t} + u\frac{\partial c}{\partial x} = \frac{1}{2}u\Delta x\left(\frac{1}{\mathbf{Cr}} - \mathbf{Cr}\right)\frac{\partial^2 c}{\partial x^2} + \cdots$$

where **Cr** is the Courant number ($\mathbf{Cr} = u\Delta t / \Delta x$) and the ellipses (…) represent higher order terms in the equation. Now it is clear why the solution for $\Delta t = 2$ is good – the Courant number is exactly unity so the dispersion term is knocked out of the equation above. The solution for $\Delta t = 2$ "moves" the front exactly one grid space in a time interval and all of the truncation errors magically cancel. In general, however, the only way we have to control numerical dispersion errors in solving transport problems is to use small time and space grid sizes. (There are some special ways to use finite-differences to control numerical dispersion – for example, the "method of characteristics" – but we will not consider these here.)

9.3. The advection-dispersion equation

With this bit of background on numerical dispersion, we return to the full advection-disperion equation.

$$\frac{\partial c}{\partial \hat{t}} + \frac{\partial c}{\partial \hat{x}} = \frac{1}{\mathbf{Pe}}\frac{\partial^2 c}{\partial \hat{x}^2}$$

A Crank-Nicolson approximation for this equation is:

$$\frac{c_i^{j+1} - c_i^j}{\Delta t} + \frac{1}{2}\left[\frac{c_{i+1}^{j+1} - c_{i-1}^{j+1}}{2\Delta x} + \frac{c_{i+1}^{j} - c_{i-1}^{j}}{2\Delta x}\right]$$
$$= \left(\frac{1}{\mathbf{Pe}}\right)\left(\frac{1}{2}\right)\left(\frac{c_{i+1}^{j+1} - 2c_i^{j+1} + c_{i-1}^{j+1}}{\Delta x^2} + \frac{c_{i+1}^{j} - 2c_i^{j} + c_{i-1}^{j}}{\Delta x^2}\right).$$

(Note that the Lax method is unstable when the dispersion term is included in the equation.) Let's look at a *MATLAB* implementation of this solution for a couple of grid spacings and compare the solution to an analytical solution for a semi-infinite domain. (The comparison will be valid as long as the concentration at the downstream end of the domain that we are simulating remains very close to the zero background concentration.)

```
% adv_disp.m
% Crank-Nicolson solution to advection-dispersion equation
% Solution for a Peclet number of 30, a distance of 4 dimensionless
% units and a total time of 1.5.
Pe=30; length=4; endtime=1.5;
%
% First do the solution for dx=0.2 (21 nodes) and dt=0.25.
nnodes=21; nsoln=nnodes-2; dx=length/(nnodes-1); dt=0.25;
ntimes=endtime/dt;
x1=0:dx:length;
cold=zeros(nnodes,1); cold(1)=1;  %set initial and boundary conditions
Mterm=1/dt+1/(Pe*dx^2);
Uterm=1/(4*dx)-1/(2*Pe*dx^2);
Lterm=-1/(4*dx)-1/(2*Pe*dx^2);
M_diag=sparse(1:nsoln,1:nsoln,Mterm,nsoln,nsoln);
U_diag=sparse(1:nsoln-1,2:nsoln,Uterm,nsoln,nsoln);
L_diag=sparse(2:nsoln,1:nsoln-1,Lterm,nsoln,nsoln);
```

```
A=M_diag+U_diag+L_diag;   %construct the left-hand-side matrix
Mterm=1/dt-1/(Pe*dx^2);
Uterm=-1/(4*dx)+1/(2*Pe*dx^2);
Lterm=1/(4*dx)+1/(2*Pe*dx^2);
M_diag=sparse(1:nsoln,1:nsoln,Mterm,nsoln,nsoln);
U_diag=sparse(1:nsoln-1,2:nsoln,Uterm,nsoln,nsoln);
L_diag=sparse(2:nsoln,1:nsoln-1,Lterm,nsoln,nsoln);
rhsmatrix=M_diag+U_diag+L_diag;   %construct the right-hand-side matrix
for j=1:ntimes
   rhs=rhsmatrix*cold(2:nnodes-1);   %calculate rhs vector
      %adjust rhs vector for boundary condition at x=0
   rhs(1)=rhs(1)+2*(1/(4*dx)+1/(2*Pe*dx^2))*cold(1);
   cnew=A\rhs;   %solve the equation
   cnew=[cold(1);cnew;cold(nnodes)];
   cold=cnew;
end
c1=cnew;
%
% now do the solution for 81 nodes and for dt=0.1
%
nnodes=81; nsoln=nnodes-2; dx=length/(nnodes-1); dt=0.1;
ntimes=endtime/dt;
x2=0:dx:length;
cold=zeros(nnodes,1); cold(1)=1;
Mterm=1/dt+1/(Pe*dx^2);
Uterm=1/(4*dx)-1/(2*Pe*dx^2);
Lterm=-1/(4*dx)-1/(2*Pe*dx^2);
M_diag=sparse(1:nsoln,1:nsoln,Mterm,nsoln,nsoln);
U_diag=sparse(1:nsoln-1,2:nsoln,Uterm,nsoln,nsoln);
L_diag=sparse(2:nsoln,1:nsoln-1,Lterm,nsoln,nsoln);
A=M_diag+U_diag+L_diag;
Mterm=1/dt-1/(Pe*dx^2);
Uterm=-1/(4*dx)+1/(2*Pe*dx^2);
Lterm=1/(4*dx)+1/(2*Pe*dx^2);
M_diag=sparse(1:nsoln,1:nsoln,Mterm,nsoln,nsoln);
U_diag=sparse(1:nsoln-1,2:nsoln,Uterm,nsoln,nsoln);
L_diag=sparse(2:nsoln,1:nsoln-1,Lterm,nsoln,nsoln);
rhsmatrix=M_diag+U_diag+L_diag;
for j=1:ntimes
   rhs=rhsmatrix*cold(2:nnodes-1);
   rhs(1)=rhs(1)+2*(1/(4*dx)+1/(2*Pe*dx^2))*cold(1);
   cnew=A\rhs;
   cnew=[cold(1);cnew;cold(nnodes)];
   cold=cnew;
end
c2=cnew;
%
% The analytical solution.
xanalyt=0:0.02:4;
arg1=(xanalyt-endtime)./(2*sqrt(endtime/Pe));
arg2=(xanalyt+endtime)./(2*sqrt(endtime/Pe));
o=ones(size(xanalyt));
canalyt=0.5*(o-erf(arg1)+exp(1/Pe)*(o-erf(arg2)));
plot(x1,c1,'bo-',x2,c2,'r+-',xanalyt,canalyt,'g')
xlabel('Dimensionless distance')
ylabel('Reduced concentration')
legend('coarse spacing','finer spacing','analytical solution')
```

The results of the computation (Figure 9.2) indicate that the effects of numerical dispersion are easily noticeable for the coarse spacing but appear to be greatly diminished for the finer spacing. One way to check for numerical dispersion is to halve the grid spacing and see if the answer remains the same. Note also the "overshoot" near the front for the coarse spacing. This, like the more subtle effects of numerical dispersion, is a feature that also is seen quite often in numerical solutions for problems in which advection is important.

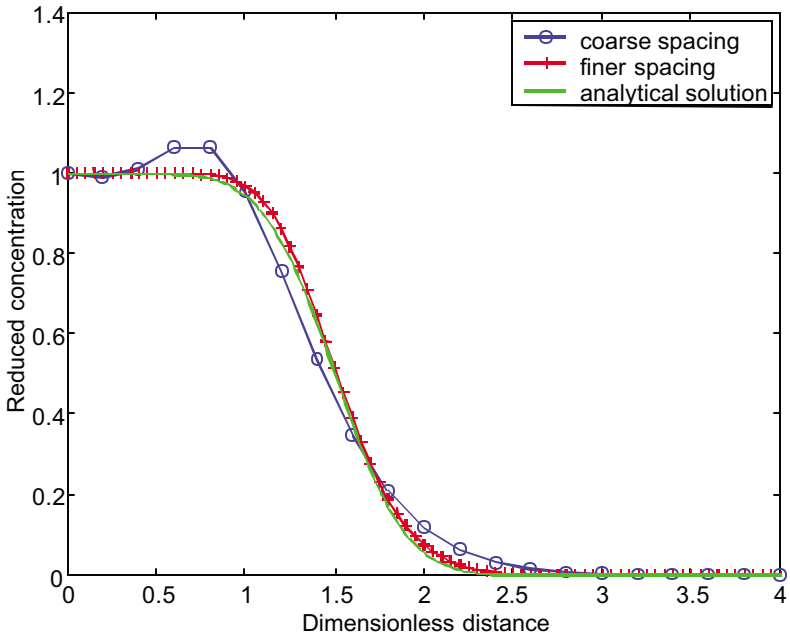


Figure 9.2. Results for numerical solution of the advection-dispersion equation.

9.4. Transport of reactive solutes

Often, we are concerned with solutes that interact with the mineral grains – of the aquifer if we are dealing with groundwater transport, of the streambed material if we are dealing with streamflow. To illustrate some of the embellishments of the methods that we have considered so far when applied to such transport problems, we will look at the transport of silica colloid through sand (Saiers et al., 1994). The interaction between the colloidal particles and the sand grains can be described using first-order kinetics. The transport equations are:

$$\frac{\partial c}{\partial t} + u\frac{\partial c}{\partial x} = D\frac{\partial^2 c}{\partial x^2} - k_f\, c + k_b\, s$$

$$\frac{\partial s}{\partial t} = k_f c - k_b\, s$$

where s is the amount of adsorbed constituent (equal to the bulk density of the aquifer material divided by the porosity times the adsorbed concentration expressed in mass of constituent per mass of mineral grains), and k_f and k_b are "forward" and "backward" rate coefficients, respectively, for the transfer of colloidal mass between the aqueous (c) and the adsorbed (s) phases. A fully implicit finite-difference approximation for these equations is:

$$\frac{c_i^{j+1} - c_i^j}{\Delta t} + u\frac{c_{i+1}^{j+1} - c_{i-1}^{j+1}}{2\Delta x} = D\frac{c_{i+1}^{j+1} - 2c_i^{j+1} + c_{i-1}^{j+1}}{\Delta x^2} - k_f c_i^{j+1} + k_b s_i^{j+1}$$

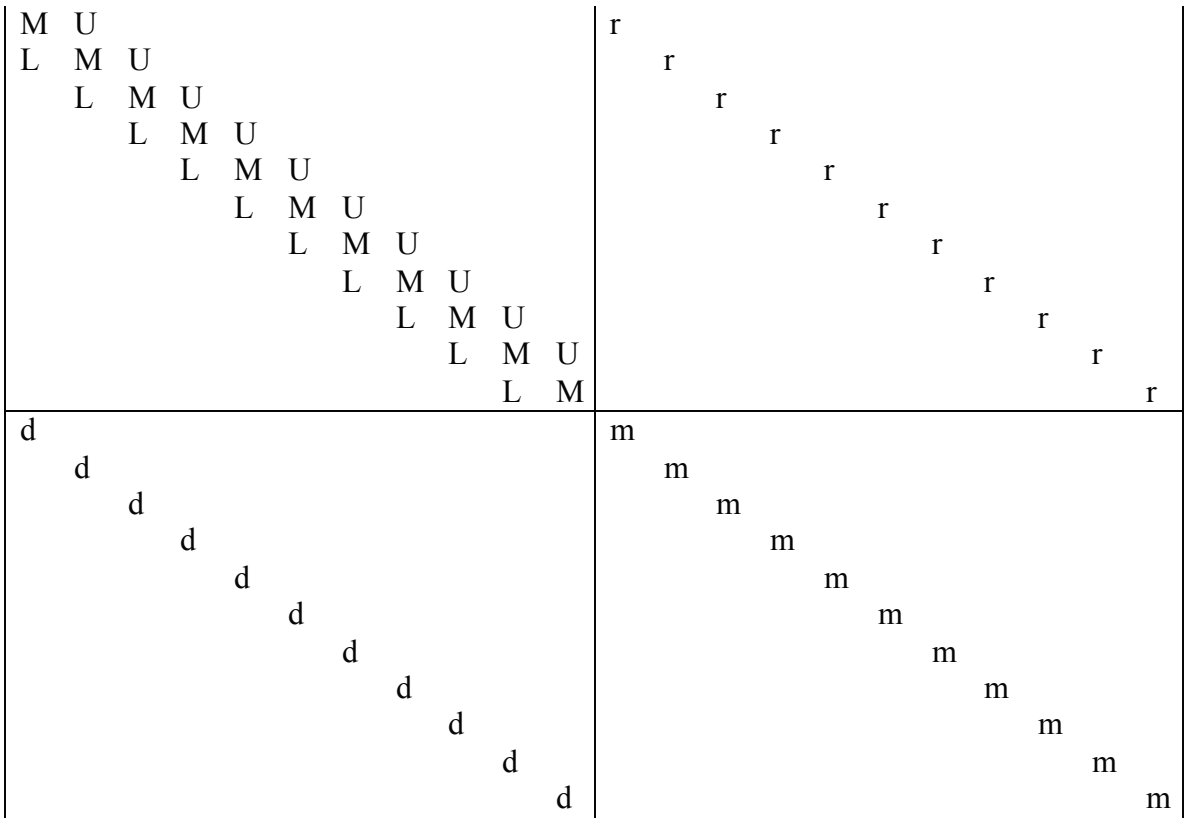
$$\frac{s_i^{j+1} - s_i^j}{\Delta t} = k_f c_i^{j+1} - k_b s_i^{j+1}$$

The way to picture the formulation of (and hence the solution to) this problem in matrix-vector notation is to note that we have twice as many unknowns now – the values of the c_i^{j+1} for the "regular" advection-dispersion problem and the s_i^{j+1} as well. To gain a solution, we again number the *c*'s sequentially, c_1 through c_n. Now, rather than doing the same for the *s*'s, we let the first *s* be c_{n+1}, the second *s* be c_{n+2}, and so on so that the n^{th} *s* is c_{2n}. Our vector of unknowns is then

$$c = [c_1, c_2, ..., c_n, c_{n+1}, c_{n+2}, ..., c_{2n}]'$$

where the second *n* elements are the unknown values of the sorbed concentration.

How does the coefficient matrix look for our approach? Consider the matrix to be blocked into $n \times n$ submatrices. The upper part of the matrix is for the equations for the aqueous concentrations. The diagonal elements in the upper left block (M in matrix below) are the coefficients on c_i^{j+1}: $1/\Delta t + 2D/\Delta x^2 + k_f$. The upper diagonal elements (U) are the coefficients on c_{i+1}^{j+1}: $u/2\Delta x - D/\Delta x^2$. Likewise the lower diagonal (L) contains the coefficients on the c_{i-1}^{j+1}: $-u/2\Delta x - D/\Delta x^2$. The matrix block in the upper right (r) contains the coefficients on the *s*'s in the equations for the aqueous concentrations. The diagonal elements (for the *block*) are the coefficients on s_i^{j+1}, $-k_b$.



The lower half of the matrix is for the equations for the adsorbed concentrations. The diagonal elements in the right, lower block (m) are $1/\Delta t+k_b$, and the coefficients in the lower left block (d) are $-k_f$. The right-hand-side vector is composed of the appropriate "known" quantities involving the j time step.

The solution in which we are interested is the breakthrough curve at the end of a column of sand. Initial conditions are zero concentration everywhere. The top boundary condition is a reduced concentration of 1 during pulse injection and a reduced concentration of zero thereafter. The bottom boundary condition is subtler. Flow occurs out of the bottom and the concentration changes with time. Thus, the bottom boundary condition is neither a fixed concentration nor a specified flux. The appropriate condition is that the flux occurs only by advection, i.e. dispersion=0. This implies that $\partial^2 c / \partial x^2 = 0$ (since $D \neq 0$) or, in terms of finite differences, $c_{n+1} - 2c_n + c_{n-1} = 0$. Solving for the concentration at the fictitious "n+1" node gives $c_{n+1} = 2c_n - c_{n-1}$.

```
% kinsorp.m  Implicit solution to advection-dispersion equation with
% kinetic sorption
% Solution for u=15.38cm/h, D=2.43cm^2/h, L=14.5cm, kf=0.13h^-1,
% kb=1.08h^-1, and a pulse input of 2.28 h (2.42 pore volumes).
v=15.38; length=14.5; D=2.43; kf=0.13; kb=1.08; pulse=2.28; endtime=5.0;
%
% do the solution for 49 nodes and for dt=0.02h.
%
nnodes=49; nsoln=nnodes; dx=length/(nnodes-1); dt=0.02;
ntimes=endtime/dt; npulse=pulse/dt;
ntotal=2*nsoln;
time=0:dt:endtime-dt;
time=time/0.94;  %correction specific to this data set
cbottom=zeros(size(time));
c=zeros(nnodes,1);c(1)=1;
s=zeros(nnodes,1);
cold=[c;s];
%
% Set the coefficient matrix
Mterm=1/dt+2*D/(dx^2)+kf;
Uterm=v/(2*dx)-D/(dx^2);
Lterm=-v/(2*dx)-D/(dx^2);
M_diagUL=sparse(1:nsoln,1:nsoln,Mterm,ntotal,ntotal);
U_diagUL=sparse(1:nsoln-1,2:nsoln,Uterm,ntotal,ntotal);
L_diagUL=sparse(2:nsoln,1:nsoln-1,Lterm,ntotal,ntotal);
M_diagUR=sparse(1:nsoln,nsoln+1:ntotal,-kb,ntotal,ntotal);
M_diagLR=sparse(nsoln+1:ntotal,nsoln+1:ntotal,1/dt+kb,ntotal,ntotal);
M_diagLL=sparse(nsoln+1:ntotal,1:nsoln,-kf,ntotal,ntotal);
A=M_diagUL+U_diagUL+L_diagUL+M_diagUR+M_diagLR+M_diagLL;
% Bottom boundary
A(nnodes,nnodes)=1/dt+v/dx+kf;
A(nnodes,nnodes-1)=-v/dx;
% Set the right hand side
rhs=cold./dt;
% Do the solution
for j=1:ntimes
   % top boundary
      if j<=npulse
        rhs(1)=rhs(1)+v/(2*dx)+D/(dx^2);
```

```
        end
    cnew=A\rhs;
    cold=cnew;
    rhs=cold./dt;
    cbottom(j)=cnew(nsoln);
end
load silica.dat;
plot(time,cbottom,'b',silica(:,1),silica(:,2),'+r')
xlabel('Dimensionless time, (pore volumes)')
ylabel('Reduced concentration')
legend('model','data')
```

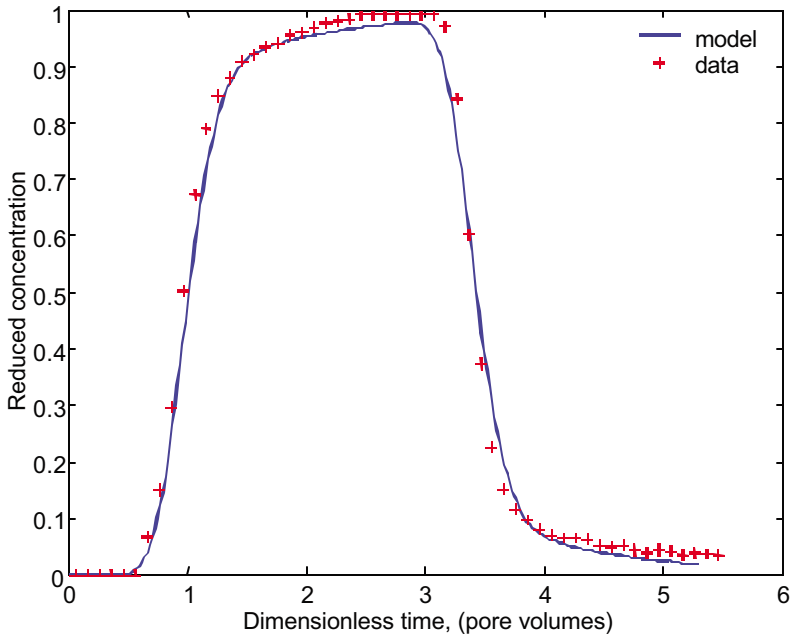


Figure 9.3. Breakthrough curve for example.

9.5. Problems

1. The transport of solute in a stream flowing over alluvium can be approached by considering the water velocity in the alluvium to be negligible with respect to that in the stream and by considering the exchange of solutes between the stream and the groundwater in the sediments to be governed by a simple first-order expression. (If you need to read up on the notion of the "transient-storage" model, you can start with the paper by Bencala and Walters, 1983.) The equations to solve (for a somewhat more simplified representation of the channel and the flow processes than that given by Bencala and Walters) are

$$\frac{\partial c}{\partial t} + u\frac{\partial c}{\partial x} = D\frac{\partial^2 c}{\partial x^2} + \alpha (s - c)$$

$$\frac{ds}{dt} = -\alpha \frac{A}{A_s}(s - c)$$

where c represents concentration in the stream, s represents concentration in the substream sediments (the "hyporheic zone"), α is a first-order exchange coefficient, A is the stream cross-sectional area , and A_s is the cross-sectional area of the storage zone.

Write a code to solve the transient-storage-zone model. Apply the code to the stream at Shaver Hollow, Virginia using the data below (from Castro and Hornberger, 1991). [To get into the ballpark with the parameter values that you will need, note that Bencala and Walters obtained values of α in the 10^{-5} per second range, D on the order of 0.1 $m^2 s^{-1}$, and A/A_s in the range of 0.3 – 1 for Uvas Creek, a small gravel-bed stream; other data indicate that A/A_s is larger for larger streams and rivers (Bencala and Walters, 1983).]

Data below are for bromide concentrations at a station 131 m downstream of the injection point. At the injection point, concentrations in the stream were maintained approximately constant at 47.8 mg L^{-1} from 1015 on 2 Aug 1988 through 0700 on 6 Aug 1988. Concentrations at the injection site were zero thereafter.

Day of August 1988	*Time*	*Bromide Concentration* (mg L^{-1})
2	1406	1.67
2	1506	4.96
2	1606	7.60
2	1704	10.38
2	1806	12.61
2	1906	14.37
2	2058	16.82
2	2214	18.92
2	2359	21.29
3	0206	23.04
3	0413	24.92
3	0613	25.92
3	0802	25.92
3	1204	28.04
3	1609	28.04
3	2016	31.55
4	0800	34.13
4	1155	31.55

4	1602	30.34
4	1955	32.81
5	0818	36.92
5	1202	34.13
5	1558	34.13
5	2004	35.50
6	0640	38.41
6	0730	36.92
6	0829	38.41
6	0956	37.31
6	1156	34.51
6	1409	27.30
6	1616	15.81
6	1815	9.89
7	0823	3.39
7	1147	2.66
7	1628	2.18
8	0825	1.64
9	0712	0.99
12	0825	0.52
20	0920	0.14

9.6. References

Bencala, K.E. and R.A. Walters, Simulation of solute transport in a mountain pool and-riffle stream: A transient storage model, *Water Resour. Res., 19*: 718-724, 1983.

Castro, N.M. and G.M. Hornberger, Surface-subsurface water interactions in an alluviated mountain stream channel, *Water Resour. Res., 27*: 1613-1621, 1991.

Fischer, H.B., E.J. List, R.C.Y. Koh, J. Imberger, and N.H. Books, *Mixing in Inland and Coastal Waters,* 483 pp., Academic Press, New York, 1979.

Hemond, H.F. and E.J. Fechner-Levy, *Chemical Fate and Transport in the Environment*, 433 pp., Academic Press, San Diego, 2000.

Hornberger, G.M., Mills, A.L., and J.S. Herman, Bacterial transport in porous media: evaluation of a model using laboratory observations, *Water Resour. Res., 28*: 915-938, 1992.

Saiers, J.E., Hornberger, G.M., and C. Harvey, Colloidal silica transport through structured, heterogeneous porous media, *J. Hydrology 163*: 271-288, 1994.

CHAPTER 10

The Finite Element Method: An Introduction

Numerical Methods in the Hydrological Sciences
Special Publication Series 57

doi:10.1029/057SP15

10. The Finite Element Method: An Introduction

10.1. Background

Finite-difference methods for solving differential equations are attractive for a number of reasons. First, they are conceptually straightforward – they "make sense". Second, the equations are relatively easy to derive, at least for a regular mesh. Third, programming to obtain a solution is not too difficult.

Finite-difference methods have some serious drawbacks, however. First, it often is desirable to have an irregular mesh (unequally spaced nodes). The finite difference equations are not easy to write in general terms under this condition. Second, it is quite cumbersome to handle irregular boundaries using finite differences. And finally, anisotropy is not easy to incorporate into finite-difference methods.

A popular numerical method that overcomes the drawbacks of finite-difference methods (admittedly at the expense of some of the attractive features of the method) is the finite-element method. In the finite-element method, the solution to the differential equation is approximated as a continuous function of the independent variables, as opposed to being approximated at only a discrete number of points, the nodal points of a mesh. The presentation below starts with the method of collocation to illustrate the idea of approximation with a continuous function and then goes on to the finite-element method itself.

10.2. Collocation

The basic idea for the method of collocation is to consider some functional approximation to the solution to a differential equation and then find coefficients in the approximate function to make it "close" to the actual solution. For example, we can consider the approximating function to be a polynomial of degree n with the (n+1) coefficients selected to match the boundary conditions and to fit the solution as well as possible. An illustrative example should help clarify the idea.

Consider the problem of groundwater flow between two drains (Figure 10.1). With the Dupuit assumptions[1], the equation describing the problem is (e.g., see Fetter 2001):

$$\begin{aligned} &\frac{d}{dx}\left[Kh\frac{dh}{dx} \right] = -w \\ &h(0) = h_1 \\ &h(L) = h_2 \end{aligned} \qquad (10.1)$$

where K is hydraulic conductivity and w is recharge rate. To make the problem even more concrete, let h_1=5 m, h_2=10 m, L=100 m, K=10 cm d^{-1}, and w=0.15 cm d^{-1}.

[1] The Dupuit assumptions are used to simplify the problem of flow in an unconfined aquifer. Basically, the assumptions are that the vertical component of flow in a horizontal aquifer is negligible – that the head gradient is the slope of the water table and that flow is horizontal.

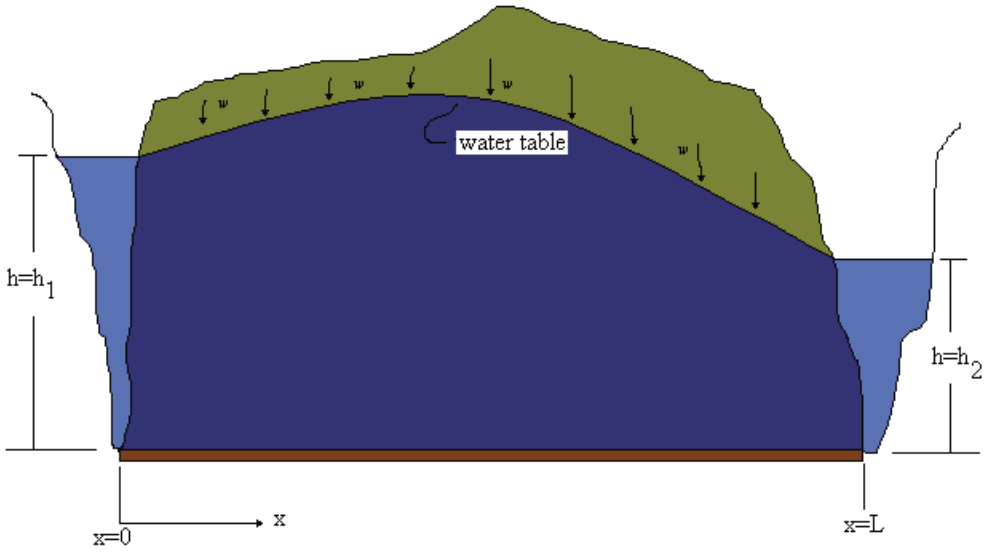


Figure 10.1. Schematic of groundwater flow between two drains.

Choose a second-degree polynomial to represent the true solution. The polynomial can be made to satisfy the boundary conditions by judicious choice of two of the coefficients.

$$\hat{h} = 5 + 0.05x + Cx(x - 100) \tag{10.2}$$

where C is a constant at our disposal to fit the approximate solution to the true solution. First, express the left side of (10.1) in terms of the trial function (10.2).

$$\hat{h} = 5 + 0.05x + Cx(x-100) = 5 + (0.05 - 100C)x + Cx^2$$

$$\frac{d\hat{h}}{dx} = (0.05 - 100C) + 2Cx$$

$$K\hat{h}\frac{d\hat{h}}{dx} = K\left[5 + (0.05 - 100C)x + Cx^2\right]\left[(0.05 - 100C) + 2Cx\right]$$

$$= K\left\{5(0.05 - 100C) + \left[(0.05 - 100C)^2 + 10C\right]x + 3C(0.05 - 100C)x^2 + 2C^2x^3\right\}$$

$$\frac{d}{dx}\left[K\hat{h}\frac{d\hat{h}}{dx}\right] = K\left\{\left[(0.05 - 100C)^2 + 10C\right] + 6C(0.05 - 100C)x + 6C^2x^2\right\}$$

For the exact solution to the differential equation, the derivative that we just calculated is equal to -w for all values of x. Because $\hat{h}$ is approximate, there will be a *residual* between the derivative and w for at least some values of x:

$$residual = r = -w - K\left\{\left[(0.05-100C)^2 + 10C\right] + 6C(0.05-100C)x + 6C^2x^2\right\}$$

The objective of the method of collocation is to make the residual as close to zero as possible. With one free coefficient at our disposal, we can select one value of x at which to make the residual equal zero exactly. A logical choice might be at x=50, halfway between the drains. This choice leads to the following solution.

$$-w - K\left\{\left[(0.05-100C)^2 + 10C\right] + 6C(0.05-100C)50 + 6C^2(50)^2\right\} = 0$$
$$-w/K - \left\{\left[(0.05-100C)^2 + 10C\right] + 6C(0.05-100C)50 + 6C^2(50)^2\right\} = 0$$
$$-0.0175 - 15C + 5000C^2 = 0$$
$$C = \text{-}8.9792\text{x}10^{-4}.$$

[Note that there are two solutions for C (the equation is quadratic) but only the one that makes sense for the problem at hand is presented. Also note that reducing the residual at x=50 to zero is *not* the same as reducing the difference between approximate and actual values of h at x=50 to zero.]

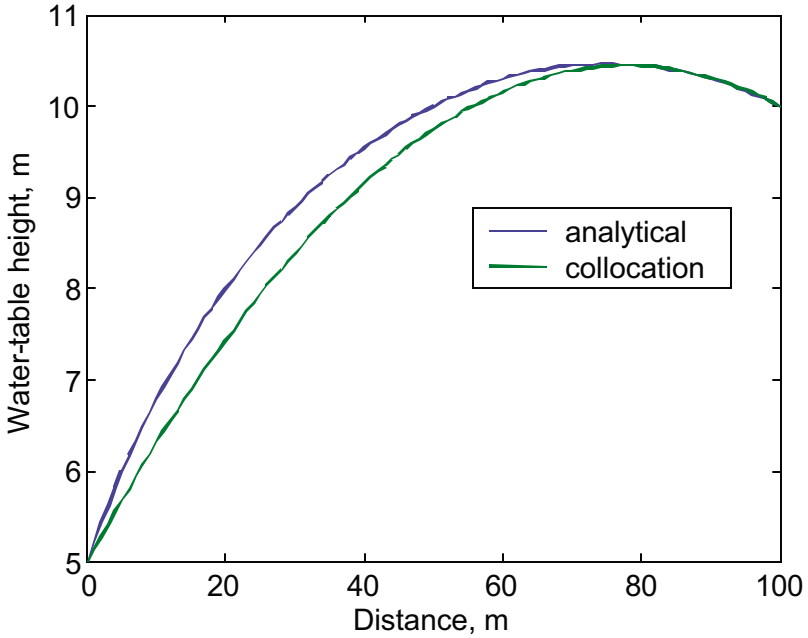


Figure 10.2. Approximate solution to the groundwater-drain problem determined using collocation compared with the analytical solution.

The approximate solution is "similar" to the analytical solution[2] (Figure 10.2), but quite possibly not as good as we might want. In the method of collocation, a better fit can be obtained by using a higher order polynomial as the trial function. For example, if we used a cubic equation, we would

[2] The *MATLAB* symbolic toolbox can be used to derive the analytical solution. Letting u be dh/dx, the problem can be represented as two differential equations, h*du/dx+u^2=-w/K and dh/dx=u. The solution is obtained by the statement: `[h,u]=dsolve('h*Du+u^2=-.015','Dh=u','h(0)=5','h(100)=10','x').`

have had two free coefficients to fit and could have chosen two points in the domain to match the differential equation exactly.

The approximate solution with the second-degree polynomial would have been much better for the example problem had the drains been only 10 m apart and the head difference between them only 0.5 m. Given the stated problem, however, increased accuracy requires a more complex approximating polynomial and, consequently, a solution to more complicated equations. The finite-element method overcomes the problems of collocation by breaking the domain into a number of elements and using simple polynomials to approximate the solution over each "small" element.

10.3. Weighted residual method

In the method of collocation applied to the example problem, the residual between the approximate solution and the right-hand side of the differential equation was minimized (set to zero) at one point in the domain. A general method for determining a good approximation is to reduce the integral of the residuals over the domain to zero. In fact, in the general method of *weighted residuals*, the residuals are multiplied by some function $W(x)$ (the weighting function) and the integral of the product is set to zero. For the example problem, we would require:

$$\int_0^{100} W(x) r(x) dx = 0 \tag{10.3}$$

The method of collocation as we applied it is a special case of the weighted residual method with the weighting function set to the Dirac delta function, a function that "selects" the collocation points (x=50 in the example) from the integral and sets the residual at these points to zero.

10.4. The finite element approach: Galerkin weighted residual method

Consider again the example problem of groundwater flow between drains. This time, we will simplify the problem by considering the linearized version – i.e., by replacing the term Kh with a constant transmissivity, T, which we will take to be 80 m^2d^{-1}.

$$\frac{d}{dx}\left[T\frac{dh}{dx}\right] = T\frac{d^2h}{dx^2} = -w$$

or,

$$\begin{aligned} &T\frac{d^2h}{dx^2} = -w = -0.15 cm/day \\ &h(0) = 5m \\ &h(100) = 10m \end{aligned} \tag{10.4}$$

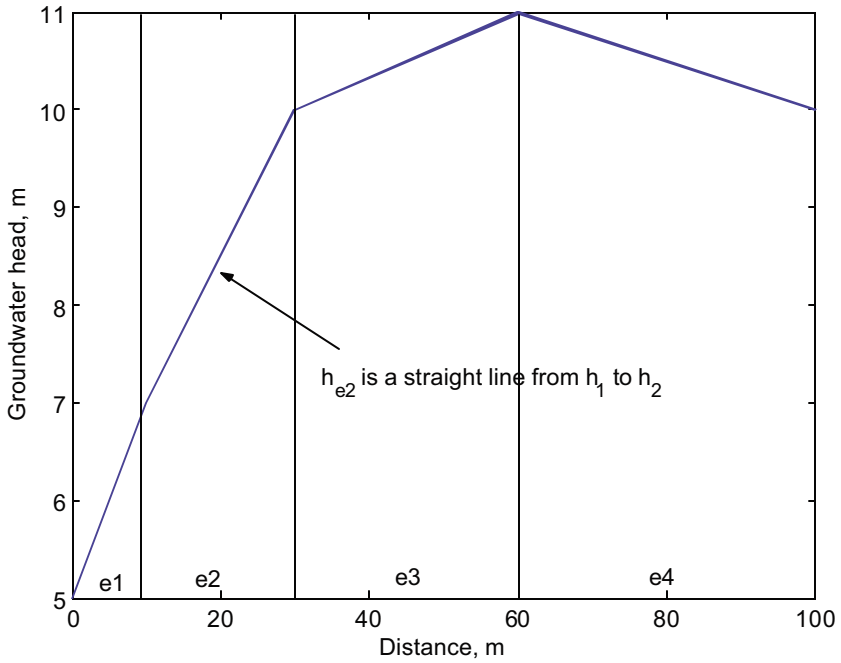


Figure 10.3. The domain is broken into four elements, e1 through e4. The groundwater head is approximated by a straight line over each element, with the endpoints denoted by the heads at the nodes, h_i.

The first step in the finite-element method is to divide the domain into *elements*, in this case four line segments (Figure 10.3). The elements are bounded by *nodes*, the endpoints of the line segments. The next step is to approximate the solution over each element. The simplest approximating polynomials are straight lines, which can be expressed as:

$$h_{e1} = h_1\left(\frac{x_2 - x}{x_2 - x_1}\right) + h_2\left(\frac{x - x_1}{x_2 - x_1}\right)$$

$$h_{e2} = h_2\left(\frac{x_3 - x}{x_3 - x_2}\right) + h_4\left(\frac{x - x_2}{x_3 - x_2}\right) \qquad (10.5)$$

$$h_{e3} = h_3\left(\frac{x_4 - x}{x_4 - x_3}\right) + h_4\left(\frac{x - x_3}{x_4 - x_3}\right)$$

$$h_{e4} = h_4\left(\frac{x_5 - x}{x_5 - x_4}\right) + h_5\left(\frac{x - x_4}{x_5 - x_4}\right)$$

It is easy to check that equations (10.5) are straight lines connecting the heads at the two endpoints of an element.

The straight-line approximation for an element can be written in terms of general "basis functions", $\xi(x)$, defined for each element. The i^{th} basis function has a value of one at node i and drops linearly to zero at the adjacent nodes (Figure 10.4). The basis functions are written explicitly as functions of x as follows.

$$\xi_i = \begin{cases} \dfrac{x_{i+1} - x}{x_{i+1} - x_i}, & \text{for } x_i \le x \le x_{i+1} \\ \dfrac{x - x_{i-1}}{x_i - x_{i-1}}, & \text{for } x_{i-1} \le x \le x_i \end{cases} \tag{10.6}$$

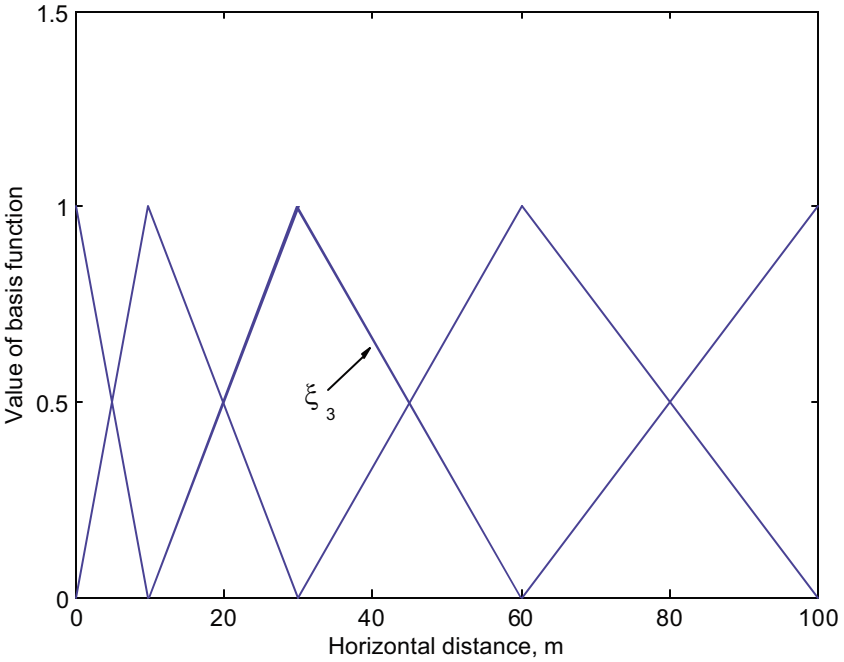


Figure 10.4. The basis functions ("chapeau functions") for the example problem. There is a basis function for each node. The function for node three is shown by the bold line.

The approximations for the groundwater heads in terms of the basis functions are:

$$\hat{h}_{ei} = h_i \xi_i + h_{i+1} \xi_{i+1} \tag{10.7}$$

where i represents the ith element. (Compare equation 10.7 with equation 10.5 given the definition of the ξ_i in equation 10.6.) The expression for h over the entire domain is then:

$$\hat{h} = \sum_k \xi_k h_k \tag{10.8}$$

The next step is to substitute the approximate function for h into the governing equation and form the residuals as we did in section 10.2.

$$residual = r = T\frac{d^2\hat{h}_e}{dx^2} + w$$

This residual is weighted and integrated over the domain of h as suggested in equation (10.3). The Galerkin method uses the basis functions themselves as the weighting functions. Although this choice may seem arbitrary, it turns out that it can be shown that use of the basis functions as weights is "best" in a sense. The specific form for equation 10.3 for the example problem is then:

$$\int_R \xi\left(T\frac{d^2\hat{h}_e}{dx^2} + w\right)dx = 0 \qquad (10.9)$$

where R is the domain over which the integration is carried out. The second derivative is eliminated from equation (10.9) by integrating by parts[3]. Let u be ξ and dv be $T\frac{d^2\hat{h}_e}{dx^2}dx$ (giving $v = T\frac{dh_e}{dx}$). Equation (10.9) can be written:

$$\xi T\frac{d\hat{h}_e}{dx}\bigg|_0^{100} - \int_0^{100}\frac{d\xi}{dx}T\frac{d\hat{h}_e}{dx}dx + \int_0^{100}\xi w dx = 0 \qquad (10.10)$$

We started our solution to the problem by dividing the domain into a number of elements. Instead of integrating across the entire domain R – from $x=0$ to $x=100$ – it makes more sense to integrate over the elements and then sum them to get the total integral. That is, we can rewrite equation (10.10) as:

$$\sum_e\left\{\xi_e T_e\frac{d\hat{h}_e}{dx}\bigg|_{x_i}^{x_{i+1}} - T_e\int_{x_i}^{x_{i+1}}\frac{d\xi_e}{dx}\frac{d\hat{h}_e}{dx}dx\right\} + \int_{x_i}^{x_{i+1}}\xi w dx = 0 \qquad (10.11)$$

where the e subscripts refer to elements. Note that in this particular example T_e is constant, but even if it were variable, it could be approximated as constant over an element and still be brought outside the integral for the element. In such a case, the coefficients derived would vary from element to element as T_e varied.

Next we evaluate equation (10.11) for a single element. First, note that the first term on the left-hand side of (10.11) is simply the total water flow into the element at x_i ($Q_i = T\frac{dh}{dx}\Big|_{x=x_i}$) minus the flow out of the element at x_{i+1} (Q_{i+1}). Now consider the second integral in equation (10.11). For the element, there are two portions of the basis functions to consider (Figure 10.5). For ξ_i, the second integral on the left-hand side of (10.11) reads:

$$T_e\int_{x_i}^{x_{i+1}}\frac{d\xi_i}{dx}\frac{d}{dx}\left(h_i\xi_i + h_{i+1}\xi_{i+1}\right)dx = T_e\int_{x_i}^{x_{i+1}}\left(\frac{d\xi_i}{dx}\frac{d\xi_i}{dx}h_i + \frac{d\xi_i}{dx}\frac{d\xi_{i+1}}{dx}h_{i+1}\right)dx \qquad (10.12)$$

and for ξ_{i+1} it is:

[3] $\int udv = uv - \int vdu.$

$$T_e \int_{x_i}^{x_{i+1}} \frac{d\xi_{i+1}}{dx}\frac{d}{dx}\left(h_i\xi_i + h_{i+1}\xi_{i+1}\right)dx = T_e \int_{x_i}^{x_{i+1}} \left(\frac{d\xi_{i+1}}{dx}\frac{d\xi_i}{dx}h_i + \frac{d\xi_{i+1}}{dx}\frac{d\xi_{i+1}}{dx}h_{i+1}\right)dx \quad (10.13)$$

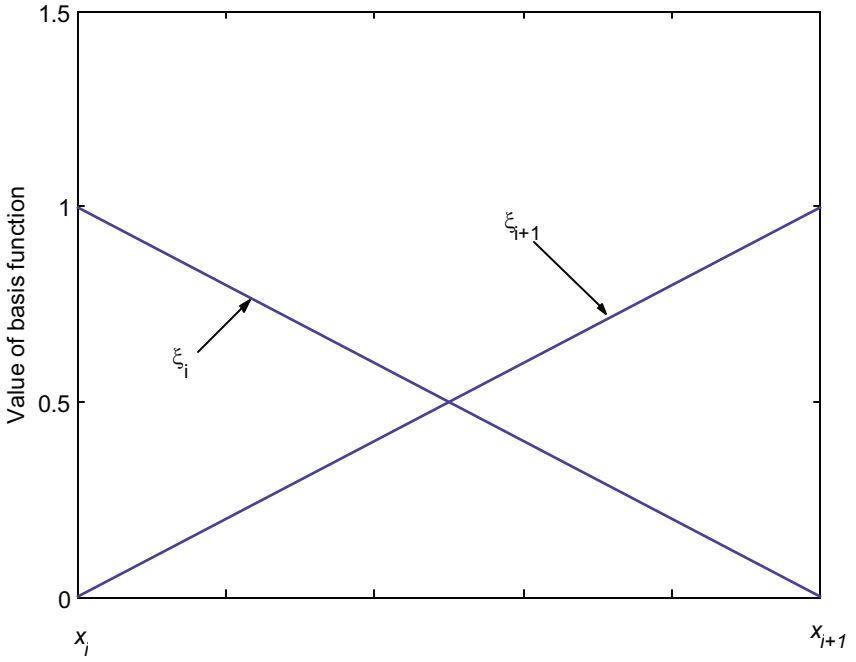


Figure 10.5. The two segments of the basis function for a general element (cf. Figure 10.4).

The integral can be evaluated easily once we find the derivatives of the basis functions with respect to x. From equation (10.6) it is clear that:

$$\frac{d\xi_i}{dx} = \frac{1}{(x_{i+1} - x_i)}, \quad \frac{d\xi_{i+1}}{dx} = \frac{-1}{(x_{i+1} - x_i)}$$

Thus, the integrands in equations (10.12) and (10.13) are not functions of x and the necessary evaluation is simply the integral of dx.

The last integral on the left-hand side of equation (10.11) is easily evaluated. Consider the integral for ξ_i and note that we can use equation (10.6) to make a substitution to convert the integral with respect to x to an integral with respect to ξ_i. In making the substitution, the limits of integration change to 0 to 1 (when $x=x_i$, $\xi_i = 0$) and dx is replaced by $\left(x_{i+1} - x_i\right)d\xi_i$.

$$w\int_{x_i}^{x_{i+1}} \frac{x - x_i}{x_{i+1} - x_i}\,dx = w\int_0^1 \xi_i \left(x_{i+1} - x_i\right) d\xi_i$$

$$= w\left(x_{i+1} - x_i\right)\frac{\xi_i^2}{2}\Bigg|_0^1$$

$$= \frac{w\left(x_{i+1} - x_i\right)}{2}$$

The integration for ξ_{i+1} is similar.

Carrying out all of the integrations, the equations corresponding to (10.11) [treating (10.12) and (10.13) separately] become:

$$\begin{aligned} Q_i - T_e\left\{\frac{h_i}{\left(x_{i+1}-x_i\right)^2} - \frac{h_{i+1}}{\left(x_{i+1}-x_i\right)^2}\right\}\left(x_{i+1}-x_i\right) + w\left(x_{i+1}-x_i\right)/2 = 0 \\ -Q_{i+1} - T_e\left\{\frac{-h_i}{\left(x_{i+1}-x_i\right)^2} + \frac{h_{i+1}}{\left(x_{i+1}-x_i\right)^2}\right\}\left(x_{i+1}-x_i\right) + w\left(x_{i+1}-x_i\right)/2 = 0 \end{aligned} \tag{10.14}$$

Now note that when the sum is done for all elements, the outflow from one element is equal to the inflow to the next. Thus, the "Q's" for all internal nodes cancel and these terms can be dropped. For the example problem, we do not have flow boundary conditions, so the Q's can be dropped there as well. The equations for a single element can be written in matrix-vector form.

$$\begin{pmatrix} 1/\Delta_e & -1/\Delta_e \\ -1/\Delta_e & 1/\Delta_e \end{pmatrix}\begin{pmatrix} h_i \\ h_{i+1} \end{pmatrix} = \begin{pmatrix} w\Delta_e/2T_e \\ w\Delta_e/2T_e \end{pmatrix} \tag{10.15}$$

where Δ_e is the element size, in this case $\left(x_{i+1}-x_i\right)$.

The finite-element procedure is completed by assembling all of the element matrices – the leftmost matrix in equation (10.15) – into a "global" matrix and solving the resulting equations for the nodal values of head. The global matrix for our 4-element example will be 3 x 3 because the heads at each end of the domain are fixed. (For the first element, the first row and first column of the element matrix disappear because the head is known. The known value of head at node 1 is transposed to the right-hand side of the equation for the first element. Likewise for the last element, the second row and column disappear.) The contribution from the first element to the global matrix is:

$$\begin{pmatrix} 1/\left(x_2-x_1\right) & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

The contribution from the second element is:

$$\begin{pmatrix} 1/\left(x_3-x_2\right) & -1/\left(x_3-x_2\right) & 0 \\ -1/\left(x_3-x_2\right) & 1/\left(x_3-x_2\right) & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 1/(x_4 - x_3) & -1/(x_4 - x_3) \\ 0 & -1/(x_4 - x_3) & 1/(x_4 - x_3) \end{pmatrix}$$

and

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1/(x_5 - x_4) \end{pmatrix}$$

The global matrix is simply the sum of the element matrices. In the same fashion, the right-hand vector is assembled by adding contributions from each element. Also, the known values of head at the endpoints of the domain are transfered to the right-hand vector. The resulting equations follow.

$$\begin{pmatrix} 1/(x_2 - x_1) + 1/(x_3 - x_2) & -1/(x_3 - x_2) & .0 \\ -1/(x_3 - x_2) & 1/(x_3 - x_2) + 1/(x_4 - x_3) & -1/(x_4 - x_3) \\ 0. & -1/(x_4 - x_3) & 1/(x_4 - x_3) + 1/(x_5 - x_4) \end{pmatrix} \begin{pmatrix} h_2 \\ h_3 \\ h_4 \end{pmatrix} =$$

$$\begin{pmatrix} w(x_2 - x_1)/2T_1 + w(x_3 - x_2)/2T_2 + 5/(x_2 - x_1) \\ w(x_3 - x_2)/2T_2 + w(x_4 - x_3)/2T_3 \\ w(x_4 - x_3)/2T_3 + w(x_5 - x_4)/2T_4 + 10/(x_5 - x_4) \end{pmatrix}$$

(The boundary conditions have been placed in the first and last equations as appropriate. Alternatively, we could have kept 5 equations and simply made the first and fifth simple statements of the boundary conditions – e.g., h_1=5.)

To obtain the numerical solution, assume that transmissivity is constant at 0.8 $m^2 day^{-1}$. The finite-element equations are then:

$$\begin{pmatrix} 1/10 + 1/20 & -1/20 & 0 \\ -1/20 & 1/20 + 1/30 & -1/30 \\ 0 & -1/30 & 1/30 + 1/40 \end{pmatrix} \begin{pmatrix} h_2 \\ h_3 \\ h_4 \end{pmatrix} = \begin{pmatrix} 0.0281 + 0.5 \\ .0469 \\ .0656 + 0.25 \end{pmatrix}$$

The numerical solution is an excellent approximation to the exact solution[4] (Figure 10.6), even with the fairly coarse mesh.

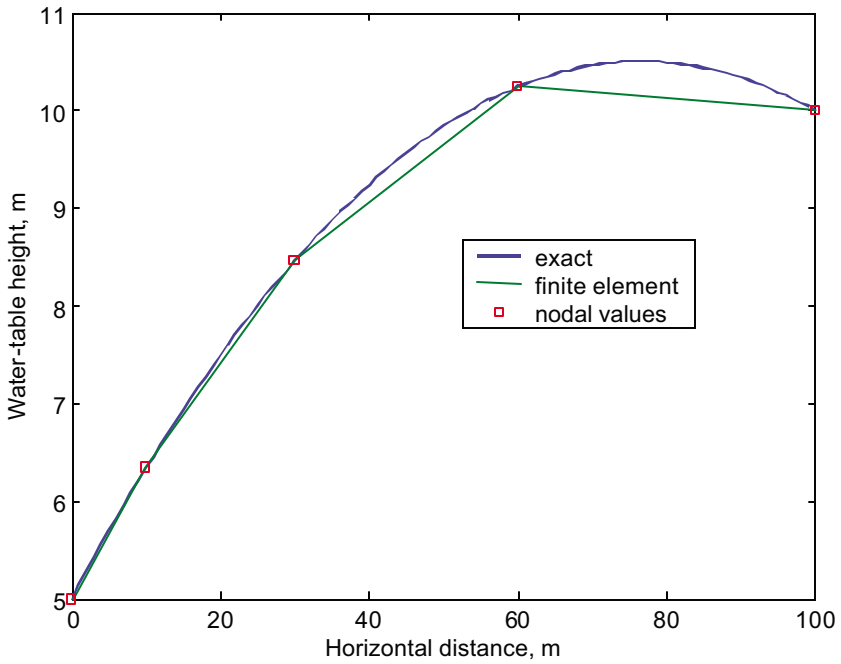


Figure 10.6. Finite element solution to the example problem.

10.5. Steady diffusion into sediment

As an example that demonstrates how variable properties, as well as unequal element size, can be accommodated in the finite-element method, consider the diffusion of oxygen into the sediment at the bottom of a lake or estuary. The equation used is a combination of Fick's law of diffusion and conservation of mass – much the same as the groundwater equation is a combination of Darcy's law and conservation of mass (see Boudreau, 1997). For steady-state conditions, the equation to solve is:

$$\frac{d}{dz}\left(\phi D \frac{dc}{dz}\right) + R = 0. \qquad (10.16)$$

[4] The *MATLAB* statement to obtain the exact solution is:

```
h=dsolve('D2h=-0.0015/0.8','h(0)=5','h(100)=10','x').
```

where ϕ is the sediment porosity, D is the diffusion coefficient of oxygen in the pore water (corrected for tortuosity), and R is the rate of oxygen consumption by reactions in the sediment. Extension of the method to conditions other than those assumed here is straightforward [Box 10.1].

Consider a 0.005-m thick layer of sediment with z=0 at depth and z=0.005 m at the sediment surface. The flux of oxygen into the sediment from the overlying water is 0.0125 n mol cm^{-2} s^{-1}. The oxygen concentration at the 0.005m (z=0) depth is zero. The conditions are given mathematically by:

$$c(0) = 0$$

$$J = \phi D \left.\frac{dc}{dz}\right|_{z=0.005} = 0.0125 \text{ n mol cm}^{-2}\text{s}^{-1}$$

The diffusion coefficient, D is the product of the free-water diffusion coefficient, D_{fw} (=11.6 $\times 10^{-6}$ cm^2 s^{-1} for this problem) and the square of the sediment porosity. That is, $D = \phi^2 D_{fw}$. The porosity, ϕ, varies with depth:

Depth interval (cm) below surface	z at bottom of interval (cm)	Porosity	Rate of oxygen consumption n mol cm^{-3} s^{-1}
0-0.05	0.45	0.9	0.02
0.05-0.1	0.4	0.85	0.07
0.1-0.2	0.3	0.8	0.04
0.2-0.3	0.2	0.75	0.02
0.3-0.5	0	0.7	0.01

We proceed as above to define the basis functions, do the integrations, and so forth, assuming that R and ϕ vary from element to element but are constant within any given element. The finite-element equations are very similar to those for the groundwater problem (the equations are nearly identical.). There are some differences. First, the top boundary (the n^{th} nodal value) condition is a *flux* condition. Look at equation (10.14). The "natural" boundary condition for the finite-element method *is* a flux condition. Thus, all that needs to be done is to place the flux on the right-hand side of the equation. Also, the porosity and diffusion coefficient vary and so are kept as part of the coefficient matrix. To simplify the notation somewhat, define λ as $\phi D = \phi\left(\phi^2 D_{fw}\right) = \phi^3 D_{fw}$. The finite-element equations (i.e., the global matrix-vector equation) are:

$$\begin{bmatrix} 1 & & & & & & \\ -\lambda_1/\Delta_1 & \lambda_1/\Delta_1+\lambda_2/\Delta_2 & -\lambda_2/\Delta_2 & & & & \\ & \bullet & & & & & \\ & & \bullet & & & & \\ & & & \bullet & & & \\ & & & & -\lambda_{n-2}/\Delta_{n-2} & \lambda_{n-2}/\Delta_{n-2}+\lambda_{n-1}/\Delta_{n-1} & -\lambda_{n-1}/\Delta_{n-1} \\ & & & & & -\lambda_{n-1}/\Delta_{n-1} & \lambda_{n-1}/\Delta_{n-1} \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \bullet \\ \bullet \\ \bullet \\ c_{n-1} \\ c_n \end{bmatrix} = \begin{bmatrix} cbot \\ R_1\Delta_1/2+R_2\Delta_2/2 \\ \\ \\ \\ R_{n-2}\Delta_{n-2}/2+R_{n-1}\Delta_{n-1}/2 \\ R_{n-1}\Delta_{n-1}/2-0.0125 \end{bmatrix}$$

where the subscripted Δ's indicate the element lengths.

The solution in *MATLAB* is now easy to construct. The m-file below does the calculation and plots the results.

```
% diagen.m
% This is a finite element solution to calculate the steady state
% concentration distribution in a heterogeneous (layered) sediment.
% Set free-water diffusion coefficient
D_fw=11.6e-6;      %cm^2 s^-1
% set nodal values of z
z=[0 0.05 0.1 0.15 0.2 0.25 0.3 0.325 0.35 0.375 0.4 0.4125 0.425 0.4375
0.45 0.4625 0.475 0.4875 0.5];
dz=diff(z);
% n=number of nodes, ne=number of elements
n=length(z);
ne=length(dz);
% phi=array of element porosities; R is array of consumption rates.
i1=find(z<0.2);i2=find(z>=0.2&z<0.3);i3=find(z>=0.3&z<0.4);
i4=find(z>=0.4&z<0.45);i5=find(z>=0.45&z<0.5);
phi=zeros(size(z));
phi(i1)=0.7;phi(i2)=0.75;phi(i3)=0.8;phi(i4)=0.85;phi(i5)=0.9;
lam_e=phi.^3*D_fw;
R=zeros(size(z));
R(i1)=-0.01;R(i2)=-0.02;R(i3)=-0.04;R(i4)=-0.07;R(i5)=-0.02;
% flux=flux at top in n mol cm^-2 s^-1, cbot=constant bottom boundary
% concentration in n mol cm^-3.
flux=0.0125;
cbot=0;
% Set the coefficient matrix and the right-hand vector.
% preallocate matrix
G=zeros(n,n);
rhs=zeros(n,1);
for e=1:ne
    node1=e;node2=e+1;          %nodes for the element
    delta=z(node2)-z(node1);    %size of element
    elmat=zeros(2,2);           %initialize element matrix
    elmat(1,1)=lam_e(e)/delta;
    elmat(1,2)=-lam_e(e)/delta;
```

```
    elmat(2,1)=-lam_e(e)/delta;
    elmat(2,2)=lam_e(e)/delta;
    % assemble the global matrix by adding the element matrix
    G(node1,node1)=G(node1,node1)+elmat(1,1);
    G(node2,node2)=G(node2,node2)+elmat(2,2);
    G(node1,node2)=G(node1,node2)+elmat(1,2);
    G(node2,node1)=G(node2,node1)+elmat(2,1);
    % assemble rhs
    rhs(node1)=rhs(node1)+R(node1)*delta/2;
    rhs(node2)=rhs(node2)+R(node1)*delta/2;
end
% Change the first equation for constant concentration
% and the last to include the flux
G(1,1:n)=0; G(1,1)=1;
rhs(1,1)=cbot;
rhs(n)=rhs(n)+flux;
% The finite element equations are in the form G*c=rhs, where "G" is
% the global coefficient matrix, "c" is the vector of unknowns,
% and "rhs" is the vector of known quantities. The MATLAB "\" function
% solves the system of equations.
c=G\rhs;
% Plot the concentrations.
plot(c,z,'r+',c,z,'-')
xlabel('Oxygen concentration (n mol cm^-^3)');
ylabel('z (cm)');
```

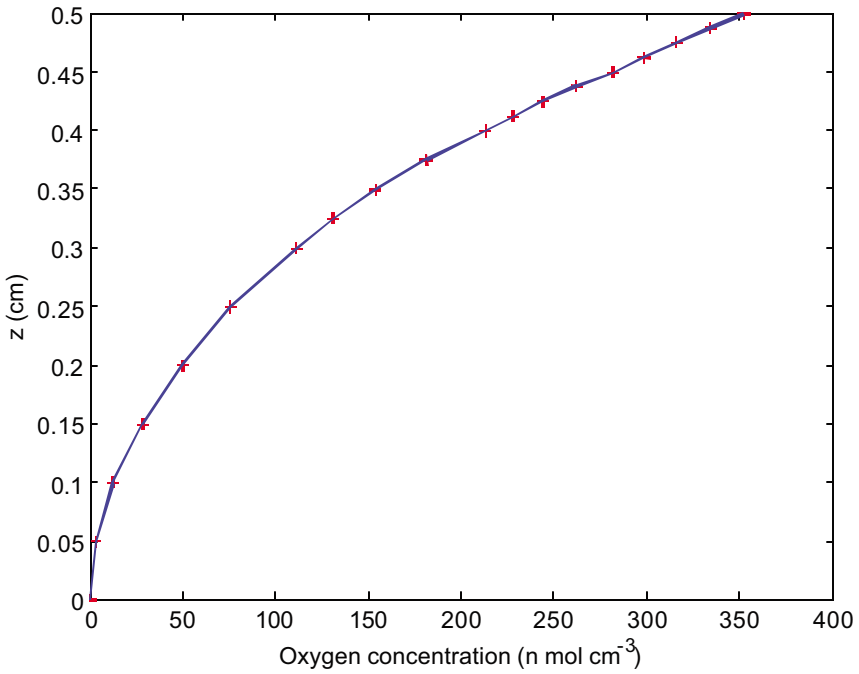


Figure 10.7. Oxygen profile calculated using the finite-element code. Note that the steady-state profile is curved because porosity (and thus diffusion coefficient) and rate of consumption vary with depth.

10.6. Problems

1. Write the finite difference equations for equation (10.1) and compare with the matrix derived using finite elements. What are the similarities and differences?

2. Berg et al. (1998) report measurements of "microprofiles" of NO_3 in freshwater sediment. In the case of nitrate, there can be "consumption" (reducing processes) or "production" (oxidizing processes). The porosity of the sediment is 0.93 and the free-water diffusion coefficient for NO_3 is 1.58×10^{-5} cm^2 s^{-1}. Given estimates of NO_3 consumption in layers of the sediment (Table 1), use a finite-element code to compute and plot the nitrate profile in the sediment. Plot your results against the measurements (Table 2) of the concentration profile. Use the surface flux boundary condition to "calibrate" the model, i.e., adjust the flux to get a good match between the computed and measured values. You might try a flux value of 0.0005 n mol cm^{-2} s^{-1} as a starting point. (Note that **negative** consumption rates represent production.)

Depth interval (cm)	NO_3 consumption rate (n mol cm^{-3} s^{-1})
0-0.16	-0.01
0.16-0.23	-0.08
0.23-0.28	0.11
0.28-0.32	0.01
0.32-0.40	0.00

Measured concentrations.

Depth	NO_3 (n mol cm^{-3})
0	69
0.05	63
0.1	57
0.15	47
0.2	30
0.25	13
0.3	2
0.35	0
0.4	0

3. The equation for early diagenesis in sediments can be modified to take account of *irrigation*, the pumping activity of tube-dwelling animals. In this case equation (10.16) is expanded to:

$$\frac{d}{dz}\left(\phi D\frac{dc}{dz}\right)+\varphi\alpha\left(c_0-c\right)+R=0,$$

where α is an irrigation coefficient and c_0 is the concentration at the sediment-water interface. Write a finite-element code to solve this problem and apply it to the conditions of problem 1. Examine results for $\alpha = 1\times10^{-5}$ s^{-1} and 5×10^{-5} s^{-1}.

4. Write a finite-element code to solve the problem of heat flow in the earth, including advection of thermal energy:

$$\frac{d}{dz}\left(\lambda \frac{dT}{dz}\right) - \rho_f c_f q \frac{dT}{dz} = 0.$$

where q is the specific discharge (units of, e.g., m s^{-1}), and ρ_f, and c_f, are the fluid density and heat capacity, respectively. [A code for the case of heat flow without advection, `geotherm.m`, is available as a template (see the m-file index).] Use a constant λ= 2.5 W m^{-1} °C and a constant ρ_f = 1000 kg m^{-3} and c_f = 4200 J kg^{-1} °C. Solve the problem subject to the boundary conditions: T_{top}=20 °C and T_{bottom}=145 °C. Plot results for the following values of q:

q = 0.0 cm yr^{-1} (static case)
q = +0.1 cm yr^{-1} (up)
q=+*0.5* cm yr^{-1} (up)
q=+*l.0* cm yr^{-1} (up)
q = –*0.*1 cm yr^{-1} (down)
q = –0.5 cm yr^{-1} (down)
q=–*l.0* cm yr^{-1} (down)

Comment on the implications of the results for interpretations made in Problem 1 in Chapter 8.

10.7. References

Berg, P., Risgaard-Petersen, N., and S. Rysgaard, Interpretation of measured concentration profiles in sediment pore water. *Limnol. Oceanogr. 43*: 1500-1510, 1998.

Boudreau, BP. *Diagenetic Models and their Implementation*, 414 pp., Springer-Verlag, Berlin. 1997.

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

Domenico, P.A. and F.W. Schwartz, *Physical and Chemical Hydrogeology*, 2nd edition, 506 pp., Wiley, New York, 1998.

Box 10.1. The finite element approach for transient conditions with advection

As an example of how the finite element method is extended to problems beyond the solutions for the steady state heat equation, consider the transport of a solute through a porous medium under transient conditions including advection (movement of the solute with the average velocity of the water) and dispersion (the "mixing" of solute due to differing velocities along different flow paths). Recall that we solved a form of the advection-dispersion equation using finite differences in Chapter 9. Here, we also let the solute decay according to a first-order rate law. More details about processes and the equation representing them can be found in hydrogeology texts (e.g., Fetter, 2001; Domenico and Schwartz, 1998). The equation is

$$\frac{\partial c}{\partial t} - D\frac{\partial^2 c}{\partial x^2} + u\frac{\partial c}{\partial x} + \lambda c = 0$$

where c is solute concentration, t is time, x is distance, D is the dispersion coefficient, u is average velocity, and λ is the decay coefficient.

The weighted residual integral for a typical element is

$$\int_{x_i}^{x_{i+1}} \xi \left(\frac{\partial c}{\partial t} - D\frac{\partial^2 c}{\partial x^2} + u\frac{\partial c}{\partial x} + \lambda c \right) dx$$

The dispersion term is similar to the one for the steady state heat equation discussed in Chapter 10. The element matrix for the dispersion term is (cf. equation (10.15)

$$\begin{pmatrix} D/\Delta_e & -D/\Delta_e \\ -D/\Delta_e & D/\Delta_e \end{pmatrix}$$

The integrations for the other terms are straightforward. First take the advection term for ξ_i. Note that the element concentration can be written

$$c = c_i\xi_i + c_{i+1}\xi_{i+1} = c_i\xi_i + c_{i+1}(1-\xi_i)$$

We can then evaluate the integral (for ξ_i) as follows, using once again a substitution of ξ_i for x in the integral and noting that $\frac{\partial c}{\partial x}dx = \frac{\partial c}{\partial \xi_i}d\xi_i$.

$$\begin{aligned}
\int_{x_i}^{x_{i+1}} \xi_i u \frac{\partial c}{\partial x} dx &= u\int_0^1 \xi_i \frac{\partial c}{\partial \xi_i} d\xi_i \\
&= u\int_0^1 \xi_i \left(c_i - c_{i+1} \right) d\xi_i \\
&= u(c_i - c_{i+1}) \frac{\xi_i^2}{2}\Bigg|_0^1 \\
&= \frac{u}{2}c_i - \frac{u}{2}c_{i+1}
\end{aligned}$$

The integral for ξ_{i+1} is evaluated in a similar way. The element matrix for the advection term is thus

$$\begin{pmatrix} \frac{u}{2} & -\frac{u}{2} \\ \frac{u}{2} & -\frac{u}{2} \end{pmatrix}$$

The integral for the term that accounts for the decay of the solute (for ξ_i) is evaluated as

$$\int_{x_i}^{x_{i+1}} \xi_i \lambda c_e dx = \int_o^1 \lambda \xi_i \left[c_i \xi_i + c_{i+1}(1-\xi_i) \right] \Delta_e d\xi_i$$

$$= \lambda \Delta_e \left(\frac{\xi_i^3}{3} c_i + \frac{\xi_i^2}{2} c_{i+1} - \frac{\xi_i^3}{3} c_{i+1} \right) \Bigg|_0^1$$

$$= \lambda \Delta_e \left(\frac{c_i}{3} + \frac{c_{i+1}}{6} \right)$$

The evaluation for ξ_{i+1} is similar. Thus the element matrix for the decay term is:

$$\begin{pmatrix} \frac{\lambda \Delta_e}{3} & \frac{\lambda \Delta_e}{6} \\ \frac{\lambda \Delta_e}{6} & \frac{\lambda \Delta_e}{3} \end{pmatrix}$$

The time derivative also must be integrated. The order of differentiation and integration can be interchanged, so the weighted residual to be evaluated is:

$$\frac{\partial}{\partial t} \int_{x_i}^{x_{i+1}} \xi c_e dx$$

Note that the integral is essentially the same as the term evaluated for the decay term. Thus the element matrix equation for the time derivative term is

$$\begin{pmatrix} \frac{\Delta_e}{3} & \frac{\Delta_e}{6} \\ \frac{\Delta_e}{6} & \frac{\Delta_e}{3} \end{pmatrix} \begin{pmatrix} \frac{\partial c_i}{\partial t} \\ \frac{\partial c_{i+1}}{\partial t} \end{pmatrix}$$

The time derivative term is replaced with an implicit finite difference approximation. The part of the approximation containing the known concentration at time t^j is placed on the right-hand side of the equation. The part of the approximation containing the unknown concentrations at time t^{j+1} is added to the left-hand side of the equation. The element matrices for the left and right sides are the same because of the sign change when the knowns are moved to the right-hand side:

$$\frac{\Delta_e}{3\Delta t}$$

The global matrix is assembled and the equations are solved one time step at a time as for the finite difference method. The code below shows how the solution can be implemented in *MATLAB*.

```
% ad_w_decay.m
% This is a finite element solution to calculate transport of
% a dissolved substance undergoing decay.
% dc/dt+u(dc/dx)=D(d2c/dx2)-(lambda)c

% Assume a 100-meter reach with constant concentration boundary
% conditions
% set nodal values of z and delta_t
z=0:1:100;
dz=diff(z);
delta_t=0.2;       % second
% n=number of nodes, e=number of elements
n_nodes=length(z);
n_elements=length(dz);
% set parameter values
u=0.2;       %m/s
D=0.1;       %m^2/s
lambda=0.01;  % per second
% conc at x=0 is one and at x=100 is zero
c0=1;c100=0;
% Set the coefficient matrix.
% preallocate matrix for left and rh sides
% The time derivative is done using a finite difference with
% an implicit approximation.
G=zeros(n_nodes,n_nodes);
R=zeros(n_nodes,n_nodes);
for e=1:n_elements
    node1=e;node2=e+1;          %nodes for the element
    delta=z(node2)-z(node1);    %size of element
    elmat=zeros(2,2);           %initialize element matrix
    rmat=zeros(2,2);            %initialize rhs matrix
    elmat(1,1)=D/delta-u/2+lambda*delta/3+delta/(3*delta_t);
    elmat(1,2)=-D/delta+u/2+lambda*delta/6+delta/(6*delta_t);
    elmat(2,1)=-D/delta-u/2+lambda*delta/6+delta/(6*delta_t);
    elmat(2,2)=D/delta+u/2+lambda*delta/3+delta/(3*delta_t);
    rmat(1,1)=delta/(3*delta_t);
    rmat(1,2)=delta/(6*delta_t);
    rmat(2,1)=delta/(6*delta_t);
    rmat(2,2)=delta/(3*delta_t);
    % assemble the global matrix by adding the element matrix
         G(node1,node1)=G(node1,node1)+elmat(1,1);
         G(node2,node2)=G(node2,node2)+elmat(2,2);
         G(node1,node2)=G(node1,node2)+elmat(1,2);
         G(node2,node1)=G(node2,node1)+elmat(2,1);
    % assemble the matrix for the rhs
         R(node1,node1)=R(node1,node1)+rmat(1,1);
         R(node2,node2)=R(node2,node2)+rmat(2,2);
         R(node1,node2)=R(node1,node2)+rmat(1,2);
         R(node2,node1)=R(node2,node1)+rmat(2,1);
end
```

```
% set the first and last equation for a fixed concentration boundary
% conditions
G(1,:)=0;G(1,1)=1; G(n_nodes,:)=0; G(n_nodes,n_nodes)=1;
R(1,:)=0;R(1,1)=1; R(n_nodes,:)=0; R(n_nodes,n_nodes)=1;
%
% Set the vector of "knowns" using specified concentration values
rhs=zeros(n_nodes,1);
rhs(1)=c0;
rhs(n_nodes)=c100;
endtime=300;       % seconds
% time loop
dz=dz';
time=0;j=1;k=1;
c_old=zeros(n_nodes,1);c_old(1)=rhs(1);c_old(n_nodes)=rhs(n_nodes);
while time<endtime
    rhs=R*c_old;
    c_new=G\rhs;
    if mod(j,300)==0
        conc(:,k)=c_new;
        k=k+1;
    end
    c_old=c_new;
    j=j+1;
    time=time+delta_t;
end
% Plot the profiles.
plot(z,conc)
ylabel('concentration'); xlabel('z (m)');
```

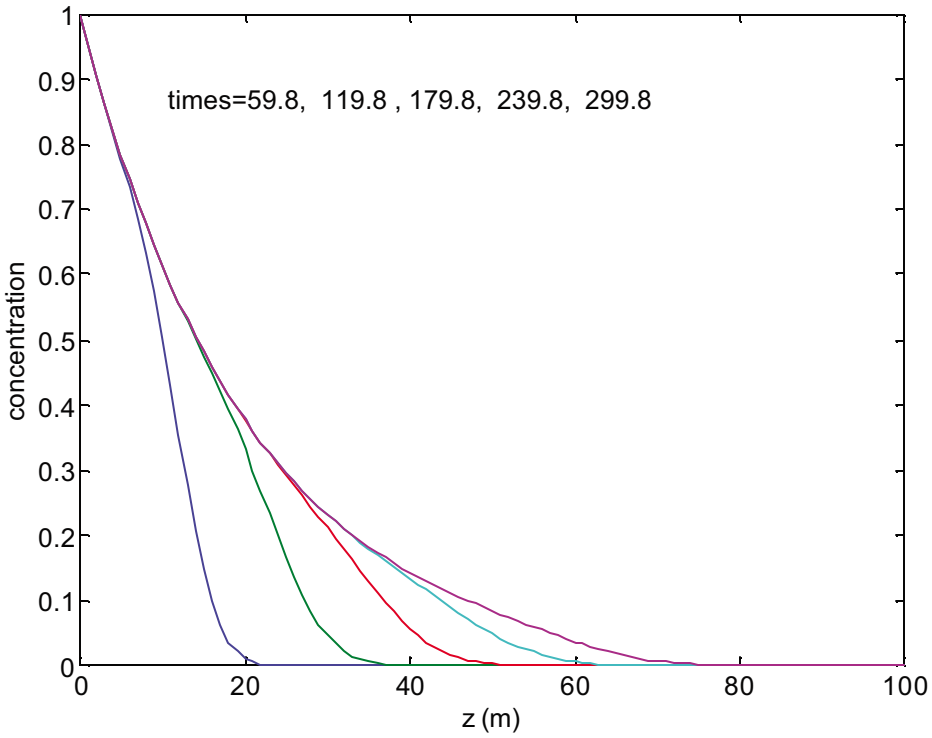


Figure B10.1.1 Finite element solution to advection-dispersion equation.

CHAPTER 11

The Finite Element Method: Steady Flow of Groundwater in Two Dimensions

Numerical Methods in the Hydrological Sciences
Special Publication Series 57

doi:10.1029/057SP16

11. The Finite Element Method: Steady Flow of Groundwater in Two Dimentions

11.1 Background

The real power of the finite-element approach to solving hydrological problems comes when more than one spatial dimension is involved. In such cases, the ability to approximate flow in regions with boundaries of various shapes and to use variable-sizes for elements makes the complexity of integrating the Galerkin residual equations worth the effort. The discussion of the application of the finite-element method to problems with one space dimension in Chapter 10 is nevertheless important because it essentially introduces all of the concepts needed to extend the method to problems with higher spatial dimension. In this Chapter we will show how to use the finite-element method to solve two-dimensional problems.

11.2 An example problem

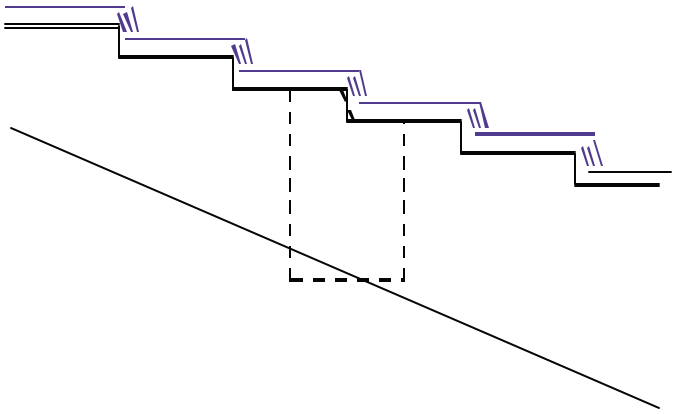


Figure 11.1. Schematic of a longitudinal profile of bottom topography in a stream. The dashed box shows a section of the stream subsurface where the approximation of no flow can be made for the bottom and sides. The bottom of the box is considered to be impervious rock and the sides have no flow by reason of symmetry.

A problem of current interest is how streams interact with the *hyporheic zone*, a region of porous material surrounding the stream channel (e.g., see Woessner, 2000) Consider the problem of flow induced in the bed of an alluvial channel by a series of drops in the bed (Figure 11.1), representing schematically a step-pool channel topography. (A discussion of the problem of flow into and from channel bedforms can be found in a number of papers, for example in Elliot and Brooks, 1997 and in Kasahara and Wondzell, 2003.) We assume that Darcy's law holds for the flow and approximate the boundary conditions as illustrated in Figure 11.1. The hydraulic conductivity is K_x in the horizontal and K_z in the vertical directions. The governing equation is:

$$\frac{\partial}{\partial x}\left(K_x \frac{\partial h}{\partial x}\right) + \frac{\partial}{\partial z}\left(K_z \frac{\partial h}{\partial z}\right) = 0 \,. \qquad (11.1)$$

We will use this problem to illustrate the application of the finite-element method in two spatial dimensions. (Extension of the method to problems with different characteristics, e.g., transient conditions is straightforward. See Box 11.1 for an example.)

11.3 Triangular elements

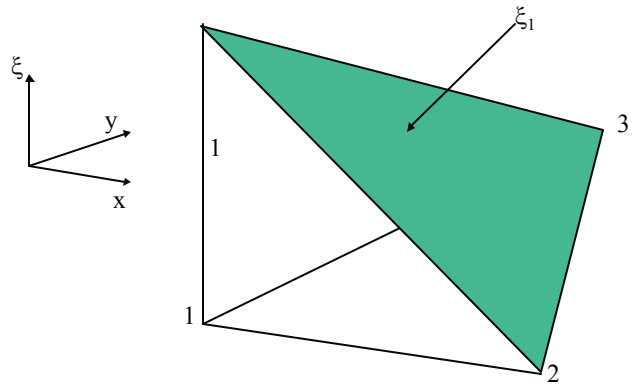


Figure 11.2. A basis function (ξ_1 is shown) for a triangular finite element defined by nodes 1, 2, and 3

A useful choice for element shape is a triangle. To make the coming integrations easier, we again use linear basis functions to rewrite the approximation above. The basis function ξ_i is defined to be equal to one at node *i* and zero at the surrounding nodes (Figure 11.2). This choice of basis functions is directly analogous to the linear segments used for the one-dimensional problems in Chapter 10 (compare Figure 11.2 with Figures 10.4 and 10.5). We divide the region up conveniently (Figure 11.3) into 320 triangles (elements). The triangles are identified by three nodes, *i, j,* and *k,* always taken in counter-clockwise rotation (more about this later). The trial solution is approximated as a linear function – a plane – over the element (Figure 11.2).

$$\hat{h}_e = C_1 + C_2 x + C_3 z$$

The basis functions can be expressed mathematically as:

$$\begin{aligned}
\xi_i &= \left[\left(x_j z_k - x_k z_j\right) + \left(z_j - z_k\right)x + \left(x_k - x_j\right)z\right] \Big/ 2\Delta_e \\
\xi_j &= \left[\left(x_k z_i - x_i z_k\right) + \left(z_k - z_i\right)x + \left(x_i - x_k\right)z\right] \Big/ 2\Delta_e \\
\xi_k &= \left[\left(x_i z_j - x_j z_i\right) + \left(z_i - z_j\right)x + \left(x_j - x_i\right)z\right] \Big/ 2\Delta_e
\end{aligned} \qquad (11.2)$$

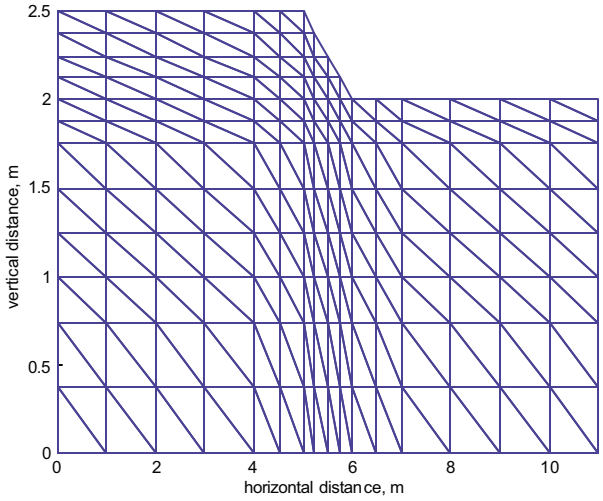


Figure 11.3. Triangular mesh for the example problem.

where the subscripted x's and y's refer to nodal coordinate values and Δ_e is the area of the element, which can be calculated as:

$$2\Delta_e = \left(x_i z_j - x_j z_i\right) + \left(x_k z_i - x_i z_k\right) + \left(x_j z_k - x_k z_j\right) \qquad (11.3)$$

The piecewise-linear approximation to head in each element is then just the sum of the three basis functions multiplied by the corresponding value of head at the node:

$$\hat{h}_e = \xi_i h_i + \xi_j h_j + \xi_k h_k \qquad (11.4)$$

11.4 The weighted residual equation and its integral over an element

The Galerkin method follows exactly the same course as for the one-dimensional example of the previous Chapter. The only difference is that the integration must be carried out over an area rather than over a line segment. The weighted residual equation for an element is:

$$\iint_{\Delta} \xi \left(\frac{\partial}{\partial x}\left(K_x \frac{\partial h}{\partial x} \right) + \frac{\partial}{\partial z}\left(K_z \frac{\partial h}{\partial z} \right) \right) dA = 0 \qquad (11.5)$$

We eliminate the second derivatives by using Green's theorem (the 2-D analog to integration by parts). Equation 11.5 can then be written:

$$-\iint_{\Delta_e} \left(K_x \frac{\partial \hat{h}}{\partial x}\frac{\partial \xi}{\partial x} + K_z \frac{\partial \hat{h}}{\partial z}\frac{\partial \xi}{\partial z} \right) dx dz + \int_{\Gamma} \left(K_x \frac{\partial \hat{h}}{\partial x} n_x + K_z \frac{\partial \hat{h}}{\partial z} n_z \right) \xi \, ds = 0 \qquad (11.6)$$

where Γ is the boundary of the element and n_x and n_z are components of an outwardly directed unit vector. (See a text on multivariable calculus for a discussion of Green's theorem, e.g., Stewart,

where Γ is the boundary of the element and n_x and n_z are components of an outwardly directed unit vector. (See a text on multivariable calculus for a discussion of Green's theorem, e.g., Stewart, 2002.) The surface integral again represents the net flux through the perimeter of the element and needs to be retained only for boundaries with a specified flux. The element conductance matrix is obtained from the integral over the area in the equation above.

To make the notation less cumbersome for the integration, some definitions help. For general subscript, n,

$$\xi_n(x,z) = \frac{1}{2\Delta}(\alpha_n + \beta_n x + \gamma_n z)$$

where the coefficients for node i are:

$$\alpha_i = x_j z_k - x_k z_j$$
$$\beta_i = z_j - z_k$$
$$\gamma_i = x_k - x_j$$

and are similar for node j and node k (obvious from equation 11.2). From (11.4), we see that the derivatives of the approximating function, $\hat{h}$, involve derivatives of the basis functions as well. Again, the analogy with the one-dimensional case is fairly transparent; compare equation (11.7) below with equations (10.11) to (10.13). The integral in (11.6) can be written:

$$\begin{aligned}\iint_{\Delta}\left(K_x\frac{\partial\hat{h}}{\partial x}\frac{\partial\xi}{\partial x}+K_z\frac{\partial\hat{h}}{\partial z}\frac{\partial\xi}{\partial z}\right)dxdz &= \Delta_e\left(K_x\frac{\partial\xi_i}{\partial x}\frac{\partial\xi_L}{\partial x}+K_z\frac{\partial\xi_i}{\partial z}\frac{\partial\xi_L}{\partial z}\right)h_i \\ &+\Delta_e\left(K_x\frac{\partial\xi_j}{\partial x}\frac{\partial\xi_L}{\partial x}+K_z\frac{\partial\xi_j}{\partial z}\frac{\partial\xi_L}{\partial z}\right)h_j+\Delta_e\left(K_x\frac{\partial?_k}{\partial x}\frac{\partial?_L}{\partial x}+K_z\frac{\partial?_k}{\partial z}\frac{\partial?_L}{\partial z}\right)h_k\end{aligned} \tag{11.7}$$

where L is i, j, or k. When L is taken as i, for example, the terms in (11.7) are the (i,i), (i,j), and (i,k) entries in the element conductance matrix. As L goes through the three indices, all entries of the 3x3 element matrix are defined. (Compare with equation 10.15, where the element matrix was 2x2 because two nodes defined the element. For a triangular element in the two-dimensional case, there are three nodes per element, thus a 3x3 element matrix obtains.)

In terms of the notation introduced following equation (11.6) above, the element conductance matrix is:

$$K_x\begin{bmatrix}\frac{\beta_i\beta_i}{4\Delta} & \frac{\beta_i\beta_j}{4\Delta} & \frac{\beta_i\beta_k}{4\Delta}\\ \frac{\beta_j\beta_i}{4\Delta} & \frac{\beta_j\beta_j}{4\Delta} & \frac{\beta_j\beta_k}{4\Delta}\\ \frac{\beta_k\beta_i}{4\Delta} & \frac{\beta_k\beta_j}{4\Delta} & \frac{\beta_k\beta_k}{4\Delta}\end{bmatrix}+K_z\begin{bmatrix}\frac{\gamma_i\gamma_i}{4\Delta} & \frac{\gamma_i\gamma_j}{4\Delta} & \frac{\gamma_i\gamma_k}{4\Delta}\\ \frac{\gamma_j\gamma_i}{4\Delta} & \frac{\gamma_j\gamma_j}{4\Delta} & \frac{\gamma_j\gamma_k}{4\Delta}\\ \frac{\gamma_k\gamma_i}{4\Delta} & \frac{\gamma_k\gamma_j}{4\Delta} & \frac{\gamma_k\gamma_k}{4\Delta}\end{bmatrix} \tag{11.8}$$

11.5 Defining node connectivities

The assembly of the global matrix is facilitated by choice of a systematic node numbering scheme. Consider the simple mesh depicted in Figure 11.4.

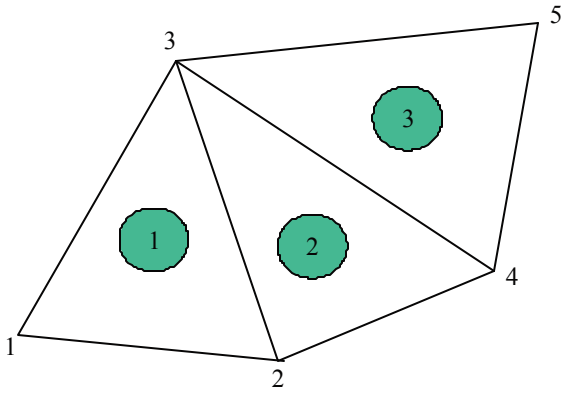


Figure 11.4. A simple three-triangle mesh with nodes numbered 1 through 5 and elements numbered within circles in the triangles.

The assembly of the element matrices for this (or any) mesh is done by mapping the specific node numbers, proceeding in the counterclockwise direction, into the "general" nodes of the element conductance matrix, *i, j,* and *k*. A way to organize the assembly is to construct a table that indicates the mapping.

Element number	Nodes		
	i	*j*	*k*
1	1	2	3
2	2	4	3
3	3	4	5

Given this map, the assembly is simply done by addition. For example, the connectivity table above shows that all three elements contribute a component to node 3. These components would be added together to get the entry in the global matrix.

For a large mesh, writing the connectivity table by hand would be quite tedious. It often is useful to create the necessary linkages automatically and, when this is the goal, it is sensible to use a few simple rules for the grid. We will use some constraints that are not necessary for the method, but make handling the mesh relatively easy. First, let each element have one leg horizontal and one leg vertical (i.e., have an underlying *x*-*z* grid as in the mesh for the example problem shown in Figure 11.3). Next, make the triangular elements by dividing the rectangles defined by the *x*-*z* grid with a

diagonal from the upper left to the lower right – again, see Figure 11.2. Finally, we number nodes beginning at the lower left of each column proceeding upward and number the elements the same way (Figure 11.5); e.g., element 1 has nodes 1,4, 2 and element 2 has nodes 2,4,5. The regularity of such a mesh can be exploited to derive formulas for the connectivity table entries.

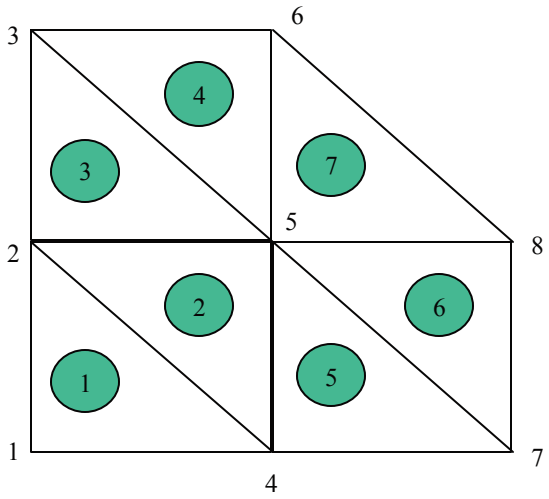


Figure 11.5. A mesh showing the numbering scheme for an eight-node, 7-element mesh with the rules as described in the text. Node numbers are at the nodes and element numbers are in circles within the element.

Here is a *MATLAB* code for assigning the nodes for each element in the example problem.

```
% program to calculate node connectivities for the example problem
%
% there are 7 vertical lines with 13 nodes, one with 12, one with 11,
% one with 10 and then seven with 9
nz=[13 13 13 13 13 13 13 12 11 10 9 9 9 9 9 9 9];
nx=length(nz);
% the number of elements per column is the sum of the number of nodes
% minus one for the bounding vertical lines
nelpercol=(nz(1:nx-1)-1)+(nz(2:nx)-1);
nelem=sum(nelpercol);
% define a vector with the number of nodes at the top of the previous
% column
nnodeprev=[0 cumsum(nz)];
% define a vector with the number of elements at the top of the
% previous column
nelemprev=[0 cumsum(nelpercol)];
% the algorithm for assigning nodes in the "i-j-k" counterclockwise
% order is set each for element depending on whether it is the lower half
% of the rectangle (j odd) or the top half of the rectangle (j even)
for i=1:nx-1
   for j=1:nelpercol(i)
      elem=nelemprev(i)+j;
```

```
      if mod(j,2)==0
         n1=nnodeprev(i)+j-floor(j/2)+1;
         n2=nz(i)-1+n1;
         n3=n2+1;
      else
         n1=nnodeprev(i)+j-floor(j/2);
         n2=nz(i)+n1;
         n3=n1+1;
      end
      nodes(elem,1:3)=[n1 n2 n3];
   end
end
```

Look at the first portion of the output from the code (below) and verify that the computation does indeed assign the correct values of "*i-j-k*" nodes to each of the elements.

```
nodes =
     1    14     2
     2    14    15
     2    15     3
     3    15    16
     3    16     4
     4    16    17
     4    17     5
     5    17    18
     5    18     6
```

11.6. Assigning *x* and *z* coordinates to the mesh

Calculation of the entries in the element conductance matrix (equation 11.8) requires specification of nodal values of *x* and *z*. These can be typed into a file and read in to the code. Alternatively, we can use the regular nature of the mesh to construct the values within a code. The *MATLAB* code below defines the node coordinates and plots the grid – useful for checking that the node connectivity table is correct.

```
% given the output "nodes" for the mesh connectivity, (and the other
% variables defined in the code above for generating the node
% connectivities) calculate the x and z values of the nodes and plot
% the values for the mesh coordinates
x=[0 2 4 5 5.5 6 7 9 11];
z=[0 0.75 1.25 1.75 2 2.25 2.5];
% assign x and z values to the nodes
for i=1:nx
   for j=1:nz(i)
      node=j+nnodeprev(i);
      nodex(node)=x(i);
      nodez(node)=z(j);
   end
end
% plot the mesh
for i=1:nelem
   line([nodex(nodes(i,1)) nodex(nodes(i,2))],[nodez(nodes(i,1))
nodez(nodes(i,2))])
```

```
   line([nodex(nodes(i,2)) nodex(nodes(i,3))],[nodez(nodes(i,2))
nodez(nodes(i,3))])
   line([nodex(nodes(i,3)) nodex(nodes(i,1))],[nodez(nodes(i,3))
nodez(nodes(i,1))]); end
```

11.7 Assembling the global conductance matrix

The pieces are now in hand to assemble the global conductance matrix. Equation (11.8) is used to compute the element conductance matrix (calculating the values of the β's, γ's, and Δ from the definitions provided) and the connectivity table is used to map the general *i, j,* and *k* into the correct position in the global matrix. The *MATLAB* code below assembles the global matrix, *G*.

```
% assemble the global conductance matrix
% set Kx and Kz
Kx=1;  %m/day
Kz=0.3;
% preassign G as a zero matrix
nnodes=sum(nz);
G=zeros(nnodes,nnodes);
% main loop for the matrix
for elem=1:nelem
   % retrieve values for i,j, and k from the connectivity table
   i=nodes(elem,1);j=nodes(elem,2);k=nodes(elem,3);
   % calculate the element area
   delta=(nodex(i)*nodez(j)-nodex(j)*nodez(i))+...
         (nodex(k)*nodez(i)-nodex(i)*nodez(k))+...
         (nodex(j)*nodez(k)-nodex(k)*nodez(j));
   % calculate the betas (b here) and gammas (gam here)
   bi=nodez(j)-nodez(k);
   bj=nodez(k)-nodez(i);
   bk=nodez(i)-nodez(j);
   gami=nodex(k)-nodex(j);
   gamj=nodex(i)-nodex(k);
   gamk=nodex(j)-nodex(i);
   % add the element values to the global matrix
   G(i,i)=G(i,i)+Kx*bi*bi/(4*delta)+Kz*gami*gami/(4*delta);
   G(i,j)=G(i,j)+Kx*bi*bj/(4*delta)+Kz*gami*gamj/(4*delta);
   G(i,k)=G(i,k)+Kx*bi*bk/(4*delta)+Kz*gami*gamk/(4*delta);
   G(j,i)=G(j,i)+Kx*bj*bi/(4*delta)+Kz*gamj*gami/(4*delta);
   G(j,j)=G(j,j)+Kx*bj*bj/(4*delta)+Kz*gamj*gamj/(4*delta);
   G(j,k)=G(j,k)+Kx*bj*bk/(4*delta)+Kz*gamj*gamk/(4*delta);
   G(k,i)=G(k,i)+Kx*bk*bi/(4*delta)+Kz*gamk*gami/(4*delta);
   G(k,j)=G(k,j)+Kx*bk*bj/(4*delta)+Kz*gamk*gamj/(4*delta);
   G(k,k)=G(k,k)+Kx*bk*bk/(4*delta)+Kz*gamk*gamk/(4*delta);
end
```

11.8 The boundary conditions – setting the right-hand side of the equations

The "natural" boundary condition for the finite-element method is zero flux. This is evident from equation (11.6), where the second integral – the integral over a boundary line of the element – gives the flux through the boundary of the element. If that flux is zero, nothing needs to be done to the equation on the boundary; the flux is automatically zero.

If a non-zero flux is specified, the line integral in equation (11.6) is not zero and must be shifted to the right-hand side of the equation. The integration assigns a flux value to the two nodes at the ends of a boundary segment of an element, say i and j. The integration along the boundary segment uses the basis-function weights, which are 1 for node i and 0 for node j for the appropriate basis function and vice-versa for the other. That is, the fluxes are multiplied by the area of a triangle with base equal to the distance between nodes and height one. The specified flux, q, is multiplied by one-half times the boundary length. This is intuitively appealing, indicating that half of the total mass flow through the element boundary is assigned to node i and the other half to node j. In some problems, we need to treat additional fluxes at internal elements. For example, in some groundwater problems, either recharge to or discharge from internal elements might be specified. In such cases integration over triangular elements indicates that one-third of the recharge to or discharge from an element is assigned to each node of the element; see Box 11.1 for an application of this method to simulate a pumping well. (For a lucid discussion of the treatment of flux boundary conditions, as well as other aspects of the finite-element method, see Wang and Anderson (1982).)

For fixed-head boundary conditions, we can simply replace the equation for the node with the equation stating that $h=h_{boundary}$. Because the example problem has only zero-flux and fixed-head boundaries, it is only the latter that must be set. The boundary condition at the top of the box shown in Figure 11.1 is a constant total head, the sum of pressure head, taken as the depth of water in the stream, and elevation head, the elevation of the streambed. (See a text on hydrology, e.g. Hornberger et al., 1998, for a discussion of hydraulic head.) For this example, we take the stream depth, i.e., the pressure head, to be constant at 0.3 m. Once the right-hand vector has been set, the solution is obtained by the standard "backslash" operation. The *MATLAB* code below sets the boundary conditions, solves the equations, and plots the results.

```
% set rhs and solve
% note that all zero-flow boundaries are "natural" -- no work necessary
%
% preassign rhs
rhs=zeros(nnodes,1);
%
% put the node numbers for the top boundary in a vector
topnodes=cumsum(nz);
%
% for the fixed-head boundary conditions, replace the equation for that
% node with h(topnode)=0.3+nodez(topnode)
for i=1:nx
   rhs(topnodes(i))=0.3+nodez(topnodes(i));
   G(topnodes(i),:)=zeros(1,nnodes);
   G(topnodes(i),topnodes(i))=1;
end
%
% solve the finite-element equations using the backslash operator
```

```
h=G\rhs;
%put the solution vector, h, back onto the two-D mesh in matrix hh
for i=1:nx
   for j=1:nz(i)
      hh(i,j)=h(nnodeprev(i)+j);
   end
   if nz(i)<nz(1)
      for j=nz(i)+1:nz(1)
         hh(i,j)=NaN;
      end
   end
end
% orient the matrix
hh=rot90(hh);
% set x and y values for the mesh and orient them as for hh
for i=1:nx
   for j=1:nz(i)
      xx(i,j)=x(i);
      zz(i,j)=z(j);
   end
   if nz(i)<nz(1)
      for j=nz(i)+1:nz(1)
         xx(i,j)=NaN;
         zz(i,j)=NaN;
      end
   end
end
xx=rot90(xx);zz=rot90(zz);
%
% plot the results
[c,handle]=contour(xx,zz,hh);
clabel(c,handle)
[px,pz]=gradient(hh,x,z);
hold on;
quiver(xx,zz,-Kx*px,Kz*pz,2);
line([x(1) x(1)],[z(1) z(nz(1))]);
line([x(nx) x(nx)],[z(1) z(nz(nx))]);
line([x(1) x(nx)],[z(1) z(1)]);
for i=1:nx-1
   line([x(i) x(i+1)],[z(nz(i)) z(nz(i+1))]);
end
axis equal; hold off
```

The solution to the example problem shows the expected flow pattern with flow into the bed on the upstream boundary and back out of the bed in the downstream pool (Figure 11.6). Note that the anisotropy makes the flow have larger horizontal components than vertical.

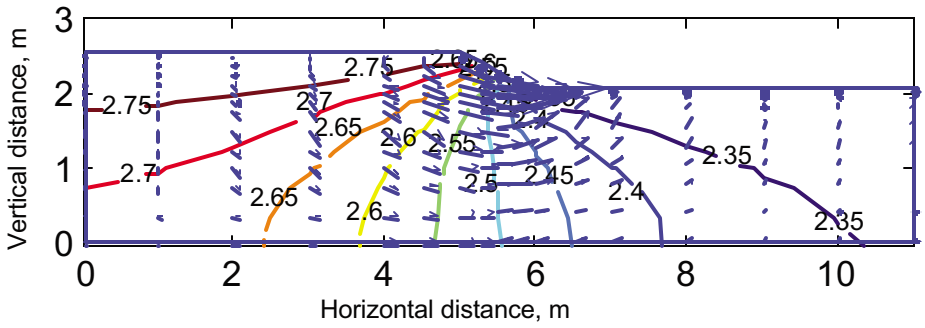


Figure 11.6. Flow through the bed of an idealized step-pool stream as calculated by the finite-element code.

11.9 Problems

A waste-disposal pond is designed to lose water to evaporation and to "trap" the contaminants in the sludge at the bottom of the pond (See Figure 6.5). Unfortunately, the pond, which is about 50m on a side, leaks (as do all such ponds!) and contaminated water is recharged to the underlying aquifer at a rate of 6 m/y. Recharge over the rest of the area in this semi-arid environment is negligible. Streams that dissect the area bound the aquifer to the north and south. The stream to the north, 75 m from the north end of the pond, is topographically higher and, on average, is a "losing stream", with water infiltrating into the aquifer. The boundary can be approximated as having a constant head of 0.2 m above datum. The boundary to the south, some 100 m from the south end of the pond, also can be approximated by a constant-head boundary with a head of 0m. Because the undisturbed flow in this stream-aquifer system tends to be directly from north to south, we can choose east and west boundaries (away from the pond) to be "no-flow" boundaries. The west no-flow boundary is 80 m from the edge of the pond and the east no-flow boundary is 50 m from the pond. The transmissivity of the aquifer is estimated to be 0.0015 $m^2 s^{-1}$, except in the southwestern part of the aquifer where the transmissivity is thought to be half of that value.

1. Construct a finite-element solution for this problem. Comment on the complexity of coding in comparison with the finite-difference solution presented in Chapter 6.
2. After your study had been undertaken, it was discovered that on the southern half of the eastern boundary there is a flux of water into the area. This flux is estimated to be 10 m^3/y per meter of length. Repeat the solution with this segment of the boundary identified as a specified flux boundary.
3. Examine the effect of placing a pumping well in the northwest quadrant. Consider the constant pumping rate to be 25 m^3/day, but also consider the effect of pumping at 125 m^3/day.
4. The codes presented in the chapters on finite elements use full matrices, mainly to make the presentation clear. For large meshes this is a badly inefficient procedure. Write your code using sparse matrices.

11.10 References

Bear, J. ***Modeling Groundwater Flow and Pollution***. Dordrecht, Boston, 414p., 1987.

Elliott, A. H., and N. H. Brooks, Transfer of non-sorbing solutes to a streambed with bed forms: Laboratory experiments, ***Water Resour. Res.***, ***33***: 137–151, 1997.

Fetter, C.W. Jr., ***Applied Hydrogeology***, 598pp., Prentice-Hall, Inc. Inc., 2001.

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

Kasahara, T., and S. M. Wondzell, Geomorphic controls on hyporheic exchange flow in mountain streams , ***Water Resour. Res.***, ***39***: 1005, doi:10.1029/2002WR001386, 2003.

Kinzelbach, W. ***Groundwater Modelling***. Elsevier, Amsterdam, 333p., 1986.

Stewart, J. ***Multivariable Calculus*** , Fifth edition. Brooks/Cole, Pacific Grove CA, 630p., 2002.

Wang, H.F. and M.P. Anderson. ***Introduction to Groundwater Modeling: Finite Difference and Finite Element Methods***. W.H. Freeman and Co., San Francisco, 237p., 1982

Woessner, W.W. Stream and fluvial plain groundwater interactions: Rescaling hydrogeologic thought. ***Groundwater 38***: 423-429. 2000.

Box 11.1. The finite element approach to a two-dimensional transient problem

The extension of the finite element method in two dimensions to problems with terms over and above the second spatial derivatives proceeds in a manner analogous to that for the one-dimensional problems discussed in Chapter 10. As an example, consider the addition of a time derivative; in particular, consider the problem of drawdown in the piezometric surface caused by a fully penetrating pumping well. A discussion of the mathematical problem for describing this situation can be found in hydrogeology text; for example, see Fetter (2001). The equation to solve is

$$\frac{\partial h}{\partial t}+\frac{T}{S}\frac{\partial^2 h}{\partial x^2}+\frac{T}{S}\frac{\partial^2 h}{\partial y^2}=0$$

where h is hydraulic head;

t is time;

x is distance;

y is distance;

T is transmissivity;

S is storativity.

The weighted residual integral for a typical element is

$$\iint_{\Delta}\xi\left(\frac{\partial h}{\partial t}\right)dA+\iint_{\Delta}\xi\left(\frac{\partial}{\partial x}\left(\frac{T}{S}\frac{\partial h}{\partial x}\right)+\frac{\partial}{\partial z}\left(\frac{T}{S}\frac{\partial h}{\partial z}\right)\right)dA=0$$

The second integral is similar to the one for steady state hyporheic zone flow done in Chapter 11, and thus the coefficient matrix for this term has already been presented. The integration for the term containing the time derivative is easily done by transforming to "local" coordinates, i.e., to coordinates relative to the element. Details of the integration for this (and other terms in the groundwater equation) can be found in texts on the use of the finite element method in hydrology (e.g., Bear, 1987; Kinzelbach, 1986;, Wang and Anderson, 1982). For the problem of transient drawdown caused by a pumping well, the finite element equations can be expressed as:

$$P\left\{\frac{dh_e}{dt}\right\}+G\left\{h_e\right\}=0$$

where G is the coefficient matrix associated with the spatial derivative terms and P is the matrix that derives from integrating the weighted residual integral for the time derivative term. For any element, the local P matrix is given as (e.g., see Kinzelbach, 1986)

$$P_e=\frac{\Delta_e}{12}\begin{pmatrix}2&1&1\\1&2&1\\1&1&2\end{pmatrix}$$

The global *P* matrix is assembled the same way as the *G* matrix. The *MATLAB* code below demonstrates how the solution can be coded.

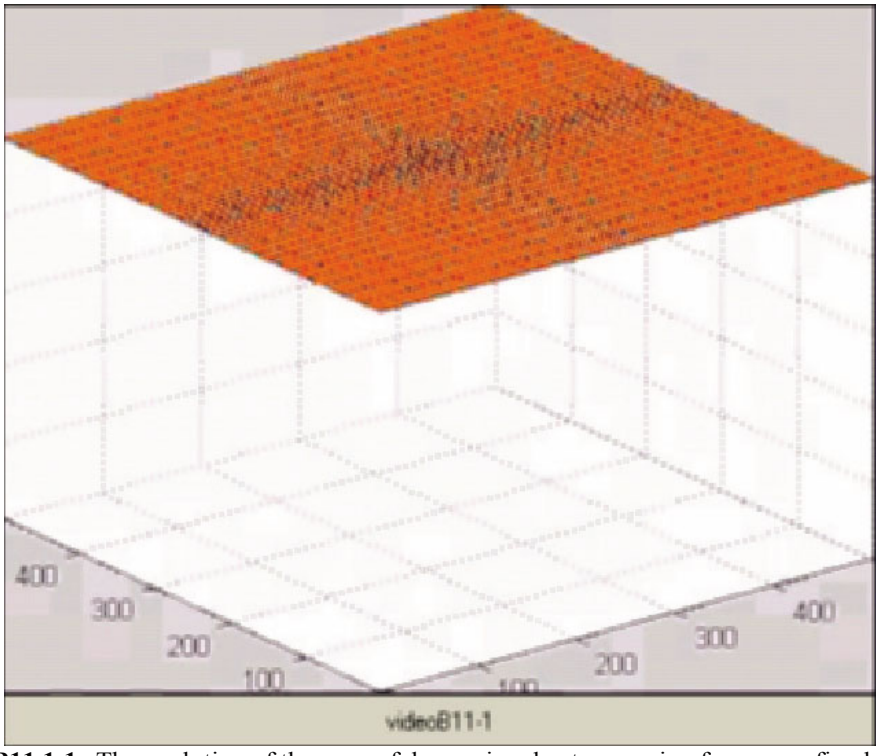
```
% pumping well in a rectangular region
% calculate node connectivities
ny=38*ones(1,38);
nx=length(ny);
% the number of elements per column is the sum of the number of nodes
% minus one for the bounding vertical lines
nelpercol=(ny(1:nx-1)-1)+(ny(2:nx)-1);
nelem=sum(nelpercol);
% define a vector with the number of nodes at the top of the previous
% column
nnodeprev=[0 cumsum(ny)];
% define a vector with the number of elements at the top of the previous
% column
nelemprev=[0 cumsum(nelpercol)];
% the algorithm for assigning nodes in the "i-j-k" counterclockwise
% order is set for element depending on whether it is the lower half
% of the rectangle (j odd) or the top half of the rectangle (j even)
for i=1:nx-1
   for j=1:nelpercol(i)
      elem=nelemprev(i)+j;
      if mod(j,2)==0
         n1=nnodeprev(i)+j-floor(j/2)+1;
         n2=ny(i)-1+n1;
         n3=n2+1;
      else
         n1=nnodeprev(i)+j-floor(j/2);
         n2=ny(i)+n1;
         n3=n1+1;
      end
      nodes(elem,1:3)=[n1 n2 n3];
   end
end
% given the output "nodes" for the mesh connectivity, calculate
% the x and y values of the nodes and plot the mesh
% the values for the mesh coordinates are:
unequal_spacing=cumsum([0 5 5 5 4 4 4 3 3 3 2 2 2 1.5 1.5 1.5 1 1 1 1 1
1 1 1.5 1.5 1.5 2 2 2 3 3 3 4  4 4 5 5 5]);
x=5*unequal_spacing;
y=5*unequal_spacing;
% assign x and y values to the nodes
for i=1:nx
   for j=1:ny(i)
      node=j+nnodeprev(i);
      nodex(node)=x(i);
      nodey(node)=y(j);
   end
end
%set T and S
nn=1:nelem;
T(nn)=299.5;    %m2/d
S(nn)=0.0051;

% preassign G and P as zero matrices
nnodes=sum(ny);
```

```
G=zeros(nnodes,nnodes);
P=zeros(nnodes,nnodes);
% assemble global conductance matrix
% main loop for the matrix
for elem=1:nelem
   % retrieve values for i,j, and k from the connectivity table
   i=nodes(elem,1);
   j=nodes(elem,2);
   k=nodes(elem,3);
   % calculate the element area
   delta=((nodex(i)*nodey(j)-nodex(j)*nodey(i))+...
      (nodex(k)*nodey(i)-nodex(i)*nodey(k))+...
      (nodex(j)*nodey(k)-nodex(k)*nodey(j)))/2;
  if delta<=0
      pause
  end
   % calculate the betas, gammas
   ai=nodex(j)*nodey(k)-nodex(k)*nodey(j);
   aj=nodex(k)*nodey(i)-nodex(i)*nodey(k);
   ak=nodex(i)*nodey(j)-nodex(j)*nodey(i);
   bi=nodey(j)-nodey(k);
   bj=nodey(k)-nodey(i);
   bk=nodey(i)-nodey(j);
   gami=-nodex(k)+nodex(j);
   gamj=-nodex(i)+nodex(k);
   gamk=-nodex(j)+nodex(i);
 % add the element values to the global matrices
G(i,i)=G(i,i)+(T(elem)/S(elem))*(bi*bi/(4*delta)+gami*gami)/(4*delta);
G(i,j)=G(i,j)+(T(elem)/S(elem))*(bi*bj/(4*delta)+gami*gamj)/(4*delta);
G(i,k)=G(i,k)+(T(elem)/S(elem))*(bi*bk/(4*delta)+gami*gamk)/(4*delta);
G(j,i)=G(j,i)+(T(elem)/S(elem))*(bj*bi/(4*delta)+gamj*gami)/(4*delta);
G(j,j)=G(j,j)+(T(elem)/S(elem))*(bj*bj/(4*delta)+gamj*gamj)/(4*delta);
G(j,k)=G(j,k)+(T(elem)/S(elem))*(bj*bk/(4*delta)+gamj*gamk)/(4*delta);
G(k,i)=G(k,i)+(T(elem)/S(elem))*(bk*bi/(4*delta)+gamk*gami)/(4*delta);
G(k,j)=G(k,j)+(T(elem)/S(elem))*(bk*bj/(4*delta)+gamk*gamj)/(4*delta);
G(k,k)=G(k,k)+(T(elem)/S(elem))*(bk*bk/(4*delta)+gamk*gamk)/(4*delta);
P(i,i)=2;P(j,j)=2*delta/12;P(k,k)=2*delta/12;
P(i,j)=1*delta/12;P(i,k)=1*delta/12;
P(j,i)=1*delta/12;P(j,k)=1*delta/12;
P(k,i)=1*delta/12;P(k,j)=1*delta/12;
end
% preassign rhs
rhs=zeros(nnodes,1);
%add pumping well
pumpingrate=2725/2;%m3/d
pumpelem1=nelem/2;
pumpelem2=nelem/2+1;
pumpelemnodes1=nodes(pumpelem1,:);
pumpelemnodes2=nodes(pumpelem2,:);
% plot the mesh
for i=1:nelem
    line([nodex(nodes(i,1)) nodex(nodes(i,2))],[nodey(nodes(i,1))
nodey(nodes(i,2))])
    line([nodex(nodes(i,2)) nodex(nodes(i,3))],[nodey(nodes(i,2))
nodey(nodes(i,3))])
    line([nodex(nodes(i,3)) nodex(nodes(i,1))],[nodey(nodes(i,3))
nodey(nodes(i,1))])
end
```

```
hold on
fill(nodex(pumpelemnodes1),nodey(pumpelemnodes1),'r');
fill(nodex(pumpelemnodes2),nodey(pumpelemnodes2),'r');
hold off; pause
% set x and y values for the mesh and orient them as for hh
    for i=1:nx
        for j=1:ny(i)
            xx(i,j)=x(i);
            yy(i,j)=y(j);
        end
        if ny(i)<ny(1)
            for j=ny(i)+1:ny(1)
                xx(i,j)=NaN;
                yy(i,j)=NaN;
            end
        end
    end
xx=xx';yy=yy';
% time loop
alltime=[0 logspace(-3,0,15)];
h_old=ones(nnodes,1)*100;
hh=reshape(h_old,max(ny),nx);
aviobj = avifile('wellmovie.avi','fps',2);
Frame=1;
figure(2)
surfc(xx,yy,hh);
axis([min(x) max(x) min(y) max(y) 90 100]);
colormap('hot');
Frame=getframe;
aviobj = addframe(aviobj,Frame);
K=0;
for ii=2:length(alltime)
    time=alltime(ii);
    delta_t=alltime(ii)-alltime(ii-1);
    K=K+1;
    F=G+2*P/delta_t;
    rhs=2*P*h_old/delta_t;
    rhs(pumpelemnodes1)=rhs(pumpelemnodes1)-…
      pumpingrate./(3*S(pumpelemnodes1))';
    rhs(pumpelemnodes2)=rhs(pumpelemnodes2)-…
      pumpingrate./(3*S(pumpelemnodes2))';
    % put the node numbers for the top and bottom boundaries in a vector
    topnodes=[cumsum(ny)];
    bottomnodes=[(cumsum(ny)+1)-(ny)];
    % fixed head boundary conditions
    for i=1:nx
        rhs(topnodes(i))=100;%m
        F(topnodes(i),:)=zeros(1,nnodes);
        F(topnodes(i),topnodes(i))=1;
    end
    for i=1:nx
        rhs(bottomnodes(i))=100;%m
        F(bottomnodes(i),:)=zeros(1,nnodes);
        F(bottomnodes(i),bottomnodes(i))=1;
    end
        F(j,:)=zeros(1,nnodes);
        F(j,j)=1;
    end
```

```
    for j=2:ny(1)-1
        rhs(j)=100;%m
        F(j,:)=zeros(1,nnodes);
        F(j,j)=1;
    end
    for j=sum(ny)-ny(1)+2:sum(ny)-1
        rhs(j)=100;%m
   %solve the finite-element equations using the backslash operator
    h=F\rhs;
    %put the solution vector, h, back onto the two-D mesh in matrix hh
    hh=reshape(h,nx,max(ny));
    h_old=h;
    % plot the results
    surfc(xx,yy,hh);
    axis([min(x) max(x) min(y) max(y) 90 100]);
    drawnow
    colormap('hot');
    Frame=getframe;
    aviobj = addframe(aviobj,Frame);
end
aviobj=close(aviobj);
```

The output from this code shows how the cone of depression around the pumping well evolves through time (Video B11.1.1).



Video B11.1.1. The evolution of the cone of depression due to pumping from a confined aquifer.

CHAPTER 12

Methods of Data Analysis: Fourier Analysis of Time Series

Numerical Methods in the Hydrological Sciences
Special Publication Series 57

doi:10.1029/057SP17

12. Methods of Data Analysis: Fourier Analysis of Time Series

12.1. Periodic functions

In previous chapters we have discussed numerical methods for solving algebraic and differential equations. Another area in which numerical methods play an important role in hydrological investigations is in the analysis of data. We touched on one aspect of this in Chapter 3.3 – fitting a polynomial to a set of points for the purpose of interpolation. Another aspect is one familiar to experimental hydrologists – fitting a function (line, curve) to a set of data points in order to describe the observed relationship between dependent and independent variables in a simple mathematical form. The most common example would be using a linear regression (Chapter 1.4).

Many relationships cannot be described adequately in terms of a linear or low-order polynomial function. Among these are periodic variations typical of many hydrological phenomena, such as monthly average river discharge, hourly evapotranspiration, and tidal amplitudes. An example of a time series with obvious periodicity is the widely reported record of CO_2 at Mauna Loa (Figure 12.1). In addition, there are many other time-varying processes, such as turbulence, floods, and precipitation that, while not strictly periodic, can be analyzed using the techniques discussed in this chapter.

The most common way to analyze periodic or quasi-periodic data is through Fourier analysis. In Fourier analysis, a function is represented by the sum of an infinite series of sinusoidal curves of varying amplitude and frequency. While it seems reasonable to expect that such a series would provide a good representation of truly periodic functions, we'll see that Fourier analysis can provide important information about almost any time series.

12.2. Fourier series

The underpinning of any Fourier analysis is the Fourier series

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty}\left[a_n \cos\left(\frac{2\pi n x}{L}\right) + b_n \sin\left(\frac{2\pi n x}{L}\right)\right] \tag{12.1}$$

in which a periodic function $f(x)$, with period L, is represented as an infinite sum of sin and cos terms. The fundamental task of Fourier analysis is to determine the coefficients a_n and b_n. The sin and cos terms in the series are orthogonal, essentially representing parts of a function that are odd (antisymmetic) and even (symmetric), respectively, with respect to the origin. The orthogonality of the sin and cos terms leads to some identities that simplify the process of evaluating the coefficients a_n and b_n in (12.1).

$$\int_{-L/2}^{L/2} \sin\left(\frac{2\pi n x}{L}\right) dx = 0 \tag{12.2a}$$

$$\int_{-L/2}^{L/2} \cos\left(\frac{2\pi n x}{L}\right) dx = \begin{cases} 0, n \neq 0 \\ L, n = 0 \end{cases} \tag{12.2b}$$

$$\int_{-L/2}^{L/2} \sin\left(\frac{2\pi n x}{L}\right) \cos\left(\frac{2\pi m x}{L}\right) dx = 0 \tag{12.2c}$$

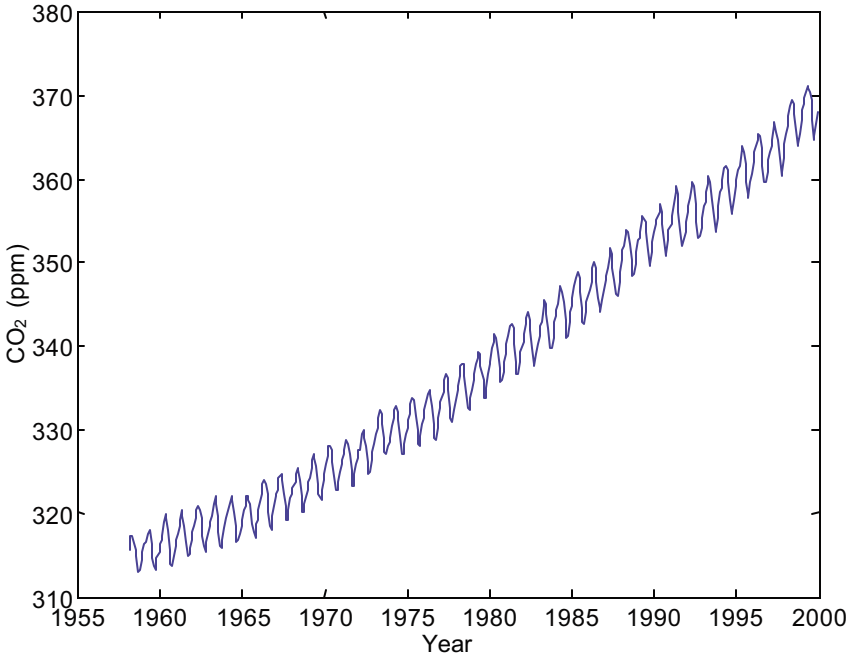


Figure 12.1. Atmospheric concentration of CO_2 measured at Mauna Loa, HI

$$\int_{-L/2}^{L/2} \sin\left(\frac{2\pi nx}{L}\right)\sin\left(\frac{2\pi mx}{L}\right)dx = \begin{cases} 0, n \neq m \\ L/2, n = m \end{cases} \tag{12.2d}$$

$$\int_{-L/2}^{L/2} \cos\left(\frac{2\pi nx}{L}\right)\cos\left(\frac{2\pi mx}{L}\right)dx = \begin{cases} 0, n \neq m \\ L/2, n = m \end{cases} \tag{12.2e}$$

To find the coefficient a_0 of the Fourier series, we integrate $f(x)$ over the interval $-L/2$ to $L/2$ and use the relationships in equations (12.2) to simplify the result.

$$\int_{-L/2}^{L/2} f(x)dx = \int_{-L/2}^{L/2} \frac{a_0}{2}dx + \sum_{n=1}^{\infty}\int_{-L/2}^{L/2} a_n \cos\left(\frac{2\pi nx}{L}\right)dx + \sum_{n=1}^{\infty}\int_{-L/2}^{L/2} b_n \sin\left(\frac{2\pi nx}{L}\right)dx$$

$$= \frac{a_0 L}{2}$$

or

$$a_0 = \frac{2}{L}\int_{-L/2}^{L/2} f(x)dx \tag{12.3}$$

Note that the 2nd and 3rd terms on the right-hand-side of the first expression disappear because of (12.2a) and (12.2b). The resulting equation for a_0 is just twice the average of $f(x)$ over the interval. Thus, the leading coefficient in the Fourier series, $a_0/2$, represents the average of the function.

To find the other a_n coefficients (for n=1,2 …), we multiply all terms in (12.1) by $\cos(2\pi mx/L)$ (m is any positive integer) and integrate

$$\int_{-L/2}^{L/2} f(x)\cos\left(\frac{2\pi mx}{L}\right)dx = \int_{-L/2}^{L/2}\frac{a_0}{2}\cos\left(\frac{2\pi mx}{L}\right)dx$$
$$+\sum_{n=1}^{\infty}\int_{-L/2}^{L/2} a_n\cos\left(\frac{2\pi nx}{L}\right)\cos\left(\frac{2\pi mx}{L}\right)dx+\sum_{n=1}^{\infty}\int_{-L/2}^{L/2} b_n\sin\left(\frac{2\pi nx}{L}\right)\cos\left(\frac{2\pi mx}{L}\right)dx$$

The 1st integral on the right-hand-side is zero (12.2b) and the 3rd integral is zero (12.2c). The 2nd integral is zero except when m=n, and then it equals $a_nL/2$ (12.2e). Therefore, all that is left is

$$a_n = \frac{2}{L}\int_{-L/2}^{L/2} f(x)\cos\left(\frac{2\pi nx}{L}\right)dx \qquad n=1,2,3,\ldots \tag{12.4}$$

We follow the same procedure to find b_n, except that now we multiply each term by $\sin(2\pi mx/L)$ before integrating

$$\int_{-L/2}^{L/2} f(x)\sin\left(\frac{2\pi mx}{L}\right)dx = \int_{-L/2}^{L/2}\frac{a_0}{2}\sin\left(\frac{2\pi mx}{L}\right)dx$$
$$+\sum_{n=1}^{\infty}\int_{-L/2}^{L/2} a_n\cos\left(\frac{2\pi nx}{L}\right)\sin\left(\frac{2\pi mx}{L}\right)dx+\sum_{n=1}^{\infty}\int_{-L/2}^{L/2} b_n\sin\left(\frac{2\pi nx}{L}\right)\sin\left(\frac{2\pi mx}{L}\right)dx$$

which, using (12.2), reduces to

$$b_n = \frac{2}{L}\int_{-L/2}^{L/2} f(x)\sin\left(\frac{2\pi nx}{L}\right)dx \qquad n=1,2,3\ldots \tag{12.5}$$

12.3. Example: Square wave

Consider a square wave (Figure 12.2),

$$f(x) = \begin{cases} -k & \text{when} \quad -\pi<x<0 \\ k & \text{when} \quad 0<x<\pi \end{cases}$$

and

$$f(x+2\pi) = f(x)$$

This is clearly periodic (period=2π), but not sinusoidal. The average of the function over the interval $-\pi$ to π is zero, so a_0=0. The function is odd (antisymmetric about the origin). As a result the Fourier coefficients a_n of the cosine terms (even terms) are also zero. To find the remaining Fourier coefficients, we use equation (12.5) to get

$$b_n = \frac{1}{\pi}\left[\int_{-\pi}^{0} -k\sin(nx)\,dx+\int_{0}^{\pi} k\sin(nx)\,dx\right] = \frac{k}{\pi}\left[\left.\frac{\cos(nx)}{n}\right|_{-\pi}^{0}-\left.\frac{\cos(nx)}{n}\right|_{0}^{\pi}\right]$$
$$=\frac{k}{\pi n}\left[\cos(0)-\cos(-n\pi)-\cos(n\pi)+\cos(0)\right]=\frac{2k}{\pi n}\left[1-\cos(n\pi)\right]$$

The resulting b_n are zero for even n and $4k/(n\pi)$ for odd n. So the Fourier series describing this square wave is

$$f(x) = \frac{4k}{\pi}\left(\sin(x) + \frac{1}{3}\sin(3x) + \frac{1}{5}\sin(5x) + \ldots\right)$$

The original square wave and the Fourier series approximation to $f(x)$ when 1, 2, or 10 terms of the infinite series are retained are shown in Figure 12.2. The shape of the square wave is relatively well captured using 10 terms of the series, but noticeable oscillations in the approximation are still present, particularly just before and after the vertical jumps in the function.

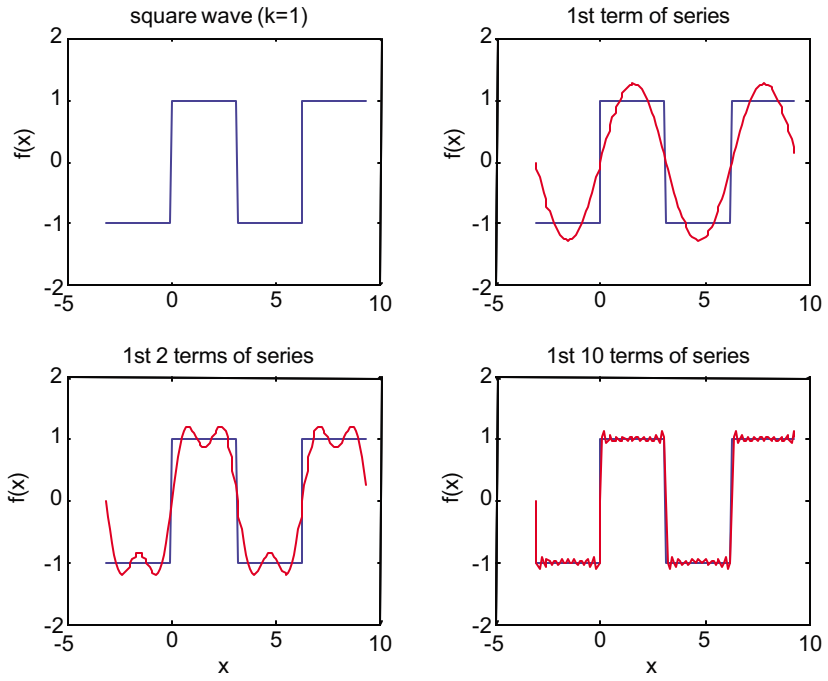


Figure 12.2. Fitting of Fourier coefficients to a square wave.

12.4. Example: CO_2 time series

In the first example, we determined the Fourier coefficients for a function that could be expressed algebraically. When analyzing measured time series using Fourier analysis, we generally are not able to represent the data with a simple algebraic expression. However, we have already investigated methods of numerical integration and their application to experimental data. In this example, we'll use one of these methods to determine the Fourier coefficients for the Mauna Loa CO_2 time series (Figure 12.1). We'll also explore *MATLAB*'s `fft` (Fast Fourier Transform) function that does most of the work for us.

In the following m-file, the *MATLAB* function `trapz` (Chapter 3.8) is used to perform the necessary integrations. The coefficients a_n and b_n are determined in a loop after detrending the data by subtracting a linear trend (top panel of Figure 12.3). It is generally advisable to

detrend a time series before performing a Fourier analysis for several reasons. From a theoretical standpoint, Fourier analysis assumes a time series is *stationary* (essentially, that is has no systematic trend in mean or variance). Detrending is also helpful from a practical point of view because retaining a trend can result in large coefficients at low frequencies (small values of n) that can dominate the results of the analysis.

When dealing with sampled data, such as the CO_2 time series, we can fit only a finite number of the terms in the infinite Fourier series. The maximum number of coefficients we can determine is constrained by the length of the time series and the sampling interval. We can never resolve frequencies higher than twice the sampling interval (monthly in this example) because it takes at least two points to define a sinusoidal function. The analysis in this example is done for a 35-year continuous record (there can't be data gaps when doing Fourier analysis). The total record length is 35×12=420, so we could determine the Fourier coefficients up to n=210. In practice, we can truncate the Fourier series well before the maximum number of coefficients that is theoretically possible. In this example, 35×4=140 coefficients are determined, enough to resolve a seasonal signal in the time series. We have taken the fundamental period T to be the length of the time series (in years), so that the coefficients for n=1 correspond to a signal with a 35-year period.

```
%co2anal.m   %finds Fourier coefficients using numerical integration
%
load maunaloa.co2   %matrix with 1 row for each year; the 1st column is
%                   the year and the next 12 columns are monthly values
y=maunaloa(:,1); [a,b]=size(maunaloa);
co2=maunaloa(:,2:b-1);
co2v=reshape(co2',(b-2)*a,1); %co2v is a row vector of all the data
m=[0:11]'./12;  M=repmat(m,length(y),1);
Y=repmat(y,1,12); Y=reshape(Y',(b-2)*a,1);
ym=Y+M;  %row vector of decimal year corresponding to each data value
i=find(co2v~=-99.99);  %-99.99 indicates missing data values

%use data section corresponding to an integer number of years without
%missing values
co2z=co2v(85:length(co2v)); ymz=ym(85:length(co2v));
%detrend data -- could also use MATLAB 'detrend' function
p=polyfit(ymz,co2z,1);
z=co2z-(p(1).*ymz+p(2));
ny=length(ymz)./12;  %length of the time series (in years)

%fit Fourier coefficients
a0=2.*mean(z);
for i=1:4*ny;
  a(i)=2./ny.*trapz(ymz,z.*cos(ymz*2*pi*i/ny));
  b(i)=2./ny.*trapz(ymz,z.*sin(ymz*2*pi*i/ny));
  s(:,i)=a(i).*cos(ymz*2*pi*i/ny)+b(i).*sin(ymz*2*pi*i/ny);
end;
fs=a0./2+sum(s,2);  %sum the terms to get full function

subplot(211), plot(ymz,z,'b',ymz,fs,'r')
xlabel('Year'); ylabel('Detrended CO_2 (ppm)');
x=1:4*ny;
subplot(425), stem(x,a,'bo')
hold on
```

```
plot(0,a0,'*',[0 2.5*ny],[0 0],'-k'); hold off
axis([-0.5 2.5*ny+0.5 -1 3])
ylabel('a_n')
subplot(427), stem(x,b,'ro')
hold on
plot([0 2.5*ny],[0 0],'-k'); hold off
axis([-0.5 2.5*ny+0.5 -1 3])
ylabel('b_n'); xlabel('n')
for i=1:4*ny,
   r(i)=sqrt(mean((z-sum(s(:,1:i),2)).^2));
end
subplot(224), plot(1:4*ny,r)
axis([-0.5 4*ny+0.5 0 2.5])
xlabel('n'); ylabel('RMS difference')
```

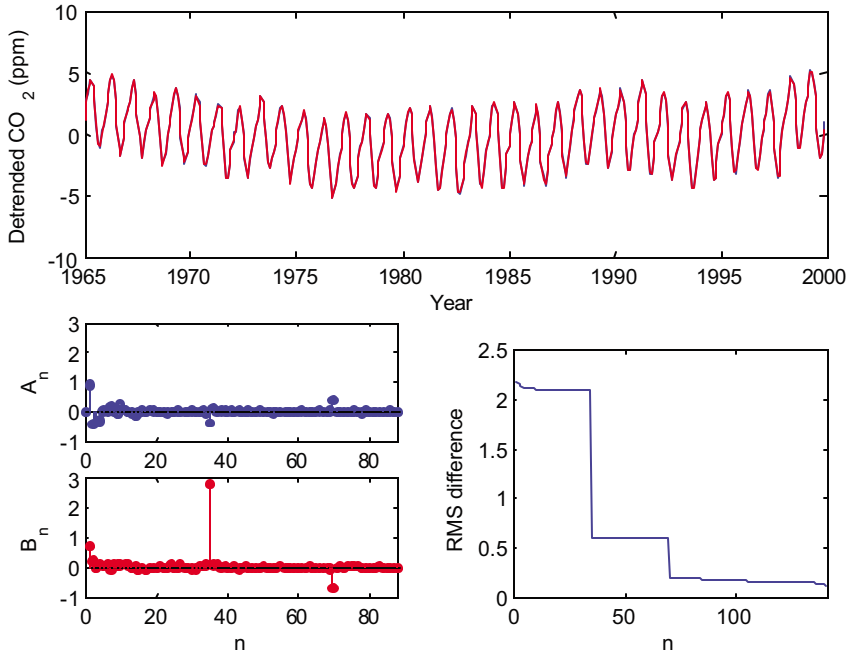


Figure 12.3. Fourier analysis of Mauna Loa CO_s time series.

The original time series and its Fourier series approximation are shown in the top panel of Figure 12.3 – only one curve is visible because they lie right on top of each other. Most of the calculated Fourier coefficients (lower left of Figure 12.3) are close to zero. Non-zero values for small values of n represent the low frequency variation apparent in the signal. The magnitude of these coefficients is relatively small, however, because the amplitude of the low-frequency variation is much smaller than the amplitude of the yearly variation. The largest coefficient is associated with n=35, the number of years in the time series, indicating a frequency of one cycle per year (period of 1 year). The b_{35} coefficient is much larger then the a_{35} coefficient, indicating that the phase of the annual signal is closer to a sine than a cosine wave. There is also a semiannual (seasonal) signal represented by the non-zero coefficients at n=70 (frequency of twice per year). The lower right panel shows the root-mean-square (rms) difference between the measured time series and the estimated time series as a function of the number of Fourier coefficients used to approximate the function. The

figure clearly shows that the annual and semiannual signals account for most of the observed variation in the time series. The top panel shows the original time series (in blue) and the Fourier series approximation (in red). They plot essentially on top of each other, indicating that the approximation provides a good representation of the original series.

12.5. *MATLAB* methods

The approach to determining the Fourier coefficients in the m-file above is straightforward, but not very efficient. Applied mathematicians have developed a more efficient method for calculating these coefficients called the "fast Fourier transform" or fft. *MATLAB* has a built-in fft function that calculates the Fourier coefficients as complex coefficients, $c_n = \sum_{k=0}^{N-1} f(k)\, e^{-i2\pi nk/L}$. c_n [the discrete Fourier transform (DFT) of $f(t)$] is related to the Fourier coefficients a_n and b_n as: $a_0 = c_0$, $a_n = c_n + c_{-n}$, $b_n = i\left(c_n - c_{-n}\right)$ [Box 12.1]. We can also write $f(k) = \sum_{n=-\infty}^{\infty} c_n\, e^{i2\pi nk/L}$ (analogous to equation 12.1), which is termed the *inverse* Fourier transform. *MATLAB* provides this information about their fft function[1]:

```
FFT Discrete Fourier transform.
    FFT(X) is the discrete Fourier transform (DFT) of vector X.  For
    matrices, the FFT operation is applied to each column. For N-D
    arrays, the FFT operation operates on the first non-singleton
    dimension.

    FFT(X,N) is the N-point FFT, padded with zeros if X has less
    than N points and truncated if it has more.

    FFT(X,[],DIM) or FFT(X,N,DIM) applies the FFT operation across the
    dimension DIM.

    For length N input vector x, the DFT is a length N vector X,
    with elements
                     N
       X(k) =       sum  x(n)*exp(-j*2*pi*(k-1)*(n-1)/N), 1 <= k <= N.
                    n=1
    The inverse DFT (computed by IFFT) is given by
                     N
       x(n) = (1/N) sum  X(k)*exp( j*2*pi*(k-1)*(n-1)/N), 1 <= n <= N.
                    k=1

    See also IFFT, FFT2, IFFT2, FFTSHIFT.
```

We can see the correspondence between the results of *MATLAB*'s `fft` function and the direct calculation of the Fourier coefficients (as in the earlier m-file) by returning to our last example. Calculating

```
X=fft(z)
```

in *MATLAB*, we obtain a complex vector of length N, the length of the time series z. (The last (N-1)/2 values of an fft are a mirror image of the (N-1)/2 values `X(2:N/2)`; the first value

[1] Reprinted with permission from MathWorks, Inc.

of X is a_0, the mean of $f(t)$. For the purposes of calculations and plotting, it is common to use twice the first half of the fft rather than the full vector.) The magnitude of X divided by N (or `2*abs(X(2:N/2))/N`) is equal to the to the magnitude of the Fourier coefficients calculated in the earlier m-file (Figure 12.4).

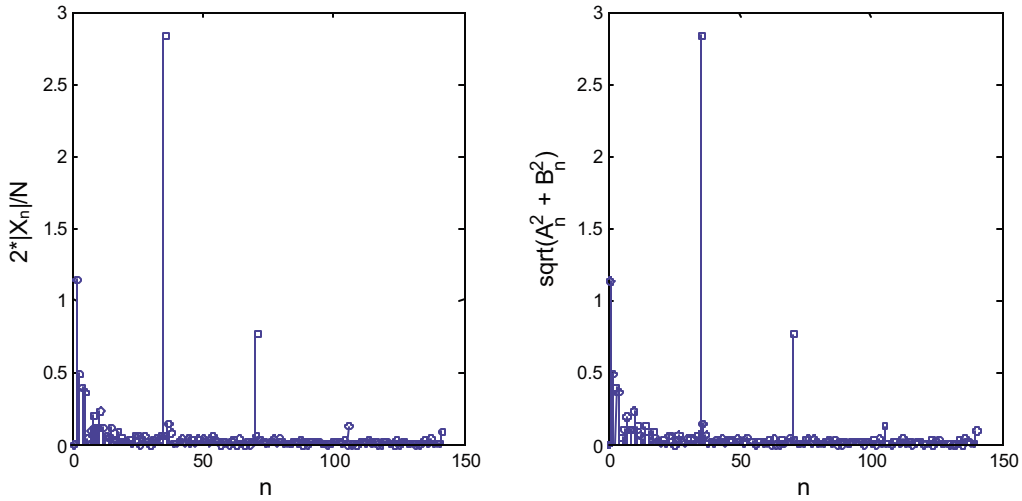


Figure 12.4. Comparison of Fourier coefficients calculated by `fft` and `co2anal.m`.

Clearly, the `fft` function provides the same information for a lot less effort than the approach used in the code presented earlier. As the *MATLAB* help file indicates, the inverse Fourier transform, `ifft(X)`, returns the original time series.

12.6. Spectral analysis and periodograms

In the figures above, the Fourier coefficients are plotted against the index n. It typically is more informative to plot the coefficients against the frequency of the corresponding sinusoidal function. The lowest frequency that can be resolved (corresponding to n=1) is dictated by the length of the time series, i.e., one cycle over the whole record. The highest frequency that can be resolved (corresponding to $n=N/2$, where N is the length of the time series) is dictated by the sampling interval dt, and is called the Nyquist frequency, 1/(2*dt).

We can calculate the frequency associated with each Fourier coefficient as

```
f=(0:N/2)./(N*dt);
```

where f=0 corresponds to a_0. So, for example, if we plot half the normalized magnitude squared of the 1st $N/2+1$ terms of the fft of the Mauna Loa CO_2 time series (`2*abs(X(2:N/2)^2)/N^2`) against frequency, we obtain the result shown in Figure 12.5a. The peaks are located at the frequencies that dominate the time series (account for most of its variance). In the example, the annual signal (frequency of 1/yr) is the largest peak; the half-yearly (2/yr) component is the second largest peak. There are two other smaller peaks at higher frequencies (3/yr and 4/yr) and some smaller peaks at lower frequencies.

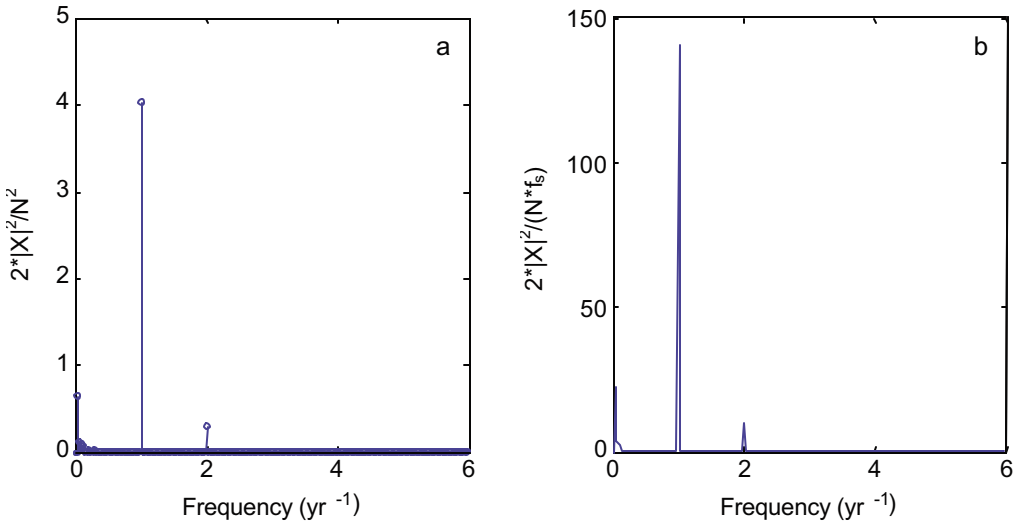


Figure 12.5. Discrete (a) and continuous (b) periodograms of Mauna Loa CO_2 data.

If we take the sum $\frac{2}{N}\sum_{n=0}^{N/2}\left|X_n^2\right|$ or $\frac{1}{N}\sum_{n=-N/2}^{N/2}\left|X_n^2\right|$, we obtain the total sum of squared deviations from the mean ($\sum_{i=1}^{N}\left(z_i-\bar{z}\right)^2$), which, when divided by N (the length of the time series), is the variance of the original time series apart from a factor of $N/(N\text{-}1)$ (Parseval's theorem; e.g., Chatfield, 2004). Therefore we can think of each value plotted in Figure 12.5a as representing the contribution of that frequency to the total variance of the time series. The sum of the values plotted in Figure 12.5a is 5.43 ppm^2; the total variance of the original time series is 5.44 ppm (N=420). We can also find the amplitude of the signal at a given frequency as (2*variance at that frequency)$^{1/2}$. So, for example, the annual signal has a variance of 4.0 ppm^2 (Figure 12.5a), giving an amplitude of 2.8 ppm (Figure 12.4) or an annual variation (high value – low value) of 5.7 ppm.

Most phenomena in the physical sciences operate over a continuous range of frequencies rather than a discrete set as plotted in Figure 12.5a. To represent the results as a continuous spectrum, we adjust the magnitudes of the coefficients so that the integral of the spectrum (total area under the curve) is equal to the variance of the original time series by multiplying the values plotted in Figure 12.5a by N/f_s, where f_s is the sampling frequency (= 12/year for our example) (Chatfield, 2004). The resulting continuous spectrum (Figure 12.5b), called a periodogram, provides an estimate of the power spectral density (psd) and shows how variance is distributed among the component frequencies in a time series; the integral over a specific range of frequencies is the variance associated with those frequencies. If we use `trapz` to calculate the integral of the full spectrum in Figure 12.5b, we again get 5.43 ppm^2.

Periodograms are informative and relatively easy to calculate, but it turns out that they have some bad statistical properties. In particular, the variance associated with a given frequency does not decrease as N increases. As a result, it is not a "consistent" estimator of the true spectral density function (e.g., Chatfield, 2004). There are two things that can be done to obtain a better estimate. One is to break the time series up into smaller segments,

calculate the spectrum for each segment, and then average the results. The segments are often overlapping. This is referred to as Welch's method. The other thing that can be done is to "window" the periodogram. We can think of dividing a time series up into segments as being equivalent to multiplying the time series by a box-car function (Figure 12.6) that has values of one over the desired segment and zeros everywhere else. For reasons that are too involved to go into here, a better spectral estimate can be obtained by replacing the simple box-car function used in the periodogram with one of a number of "windows" that are still zero outside the desired range, but have different shapes inside the range. One example is the Tukey-Hanning window (Figure 12.6), which is used in the m-file below.

MATLAB has a number of functions for spectral analysis in their signal-processing tookbox, including `periodogram`, `pwelch` and `pmtm`. `pwelch` makes use of averaging and windowing; `pmtm` is a multitaper method that employs a combination of windows. These functions calculate the power spectral density (psd) with units of power per unit frequency. (The power spectrum is given by the psd times the sampling frequency). *MATLAB* help provides more information on these and other spectral functions. The following simplified function file, `specclc.m`, calculates the psd of a time series using averaging and a Tukey-Hanning window. The function assumes a window length of 256 with a 50% overlap – a good choice for many problems – but the window length can be changed. The windows used in most windowing methods (non-rectangular windows such as the Tukey-Hanning window (Figure 12.6)) affect the power of the signal. This can be compensated for by normalizing the window so that its average power is 1. Following *MATLAB*[2], `specclc` uses a normalizing factor equal to the sum of squares of the window coefficients divided by window length – like the formulation used for the fft of the time series itself.

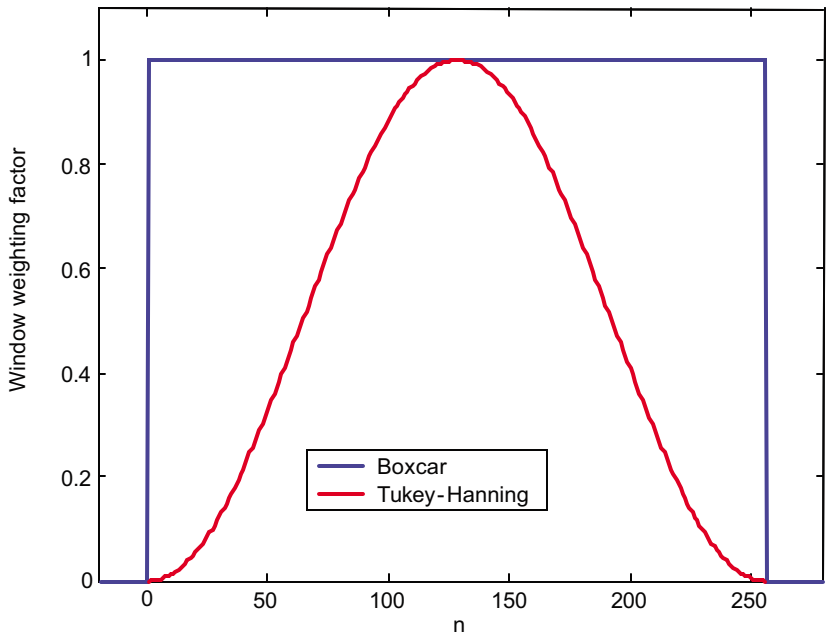


Figure 12.6. Boxcar and Tukey-Hanning windows.

```
function [cx,fx]=specclc(x,fs)
```

[2] *MATLAB* Signal Processesing Toolbox User's Guide, Version 6, MathWorks, Inc.; available through *MATLAB's* help section on Non-Parametric Spectral Estimation Methods.

```
%x = time series, fs = sampling frequency (1/dt)
%cx = vector of Fourier coefficients, fx = vector of frequencies

nw=256; %nw=window length (must be even)
no=nw/2;  %no=overlap length (50%);
nx=length(x); x=x(:);  %nx is length of record
if nx<nw, x(nx+1:nw)=0; n=nw; end; %add zeros if n<nw
nseg=floor(2*nx/nw-1);  %calculate number of segments
iw=[0:no-1]';  %window index
halfwin=0.5.*(1-cos(pi*iw/(no-1)));  %Tukey-Hanning window
win=[halfwin;flipud(halfwin)];
sswn=sum(abs(win.^2))/nw;  %sum of squares of window / nw
csum=zeros(nw,1);
for i=1:nseg,
  iseg=[1:nw]'+(i-1)*no;   %segment index
  xseg=win.*x(iseg);
  cseg=abs(fft(xseg,nw)).^2;    %fft of segment
  csum=csum+cseg;   %sum contributions from each segment
end;
cx=csum./nseg  %average summed fft
%cx is 2*normalized first half of average summed fft
cx=2*cx(1:no+1)./(nw*fs*sswn)
fx=(0:no)'*fs/nw;  %fx is frequency vector
```

We can view the psd by plotting cx vs. fx.

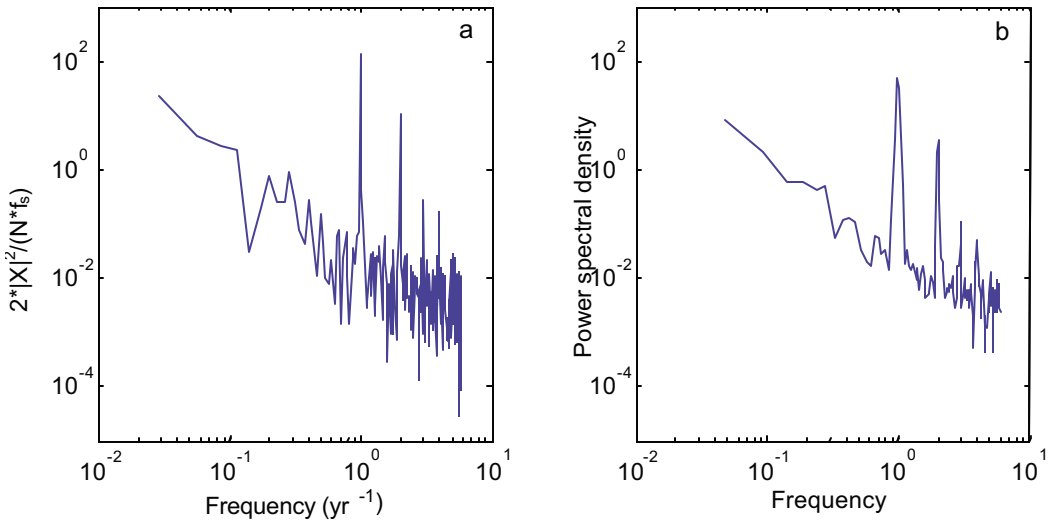


Figure 12.7. (a) Periodogram and (b) psd calculated using specclc for the Mauna Loa CO_2 data.

The application of `specclc` to the Mauna Loa time series is shown in Figure 12.7b; the spectrum shown in Figure 12.7a is the same as the one in Figure 12.5b, but plotted on log-log axes as is common for spectral data. Comparing Figure 12.7a with Figure 12.7b, we can see that the effect of averaging and windowing is to smooth the power spectral density function while preserving the main spectral peaks and their relative magnitudes. There are several large peaks in the psd as well as many smaller ones. To know which of these are significant, it is necessary to calculate confidence intervals for the psd. There is an option to calculate confidence intervals in the *MATLAB* function `pmtm`. The integral of the psd in Figure 12.7b

(e.g., `trapz(fx,cx)`), 5.11 ppm^2, is still approximately equal to the variance of the input time series, although windowing and averaging result in small differences in values. To find the contribution of a particular range of frequencies, e.g., those spanning the width of a peak in the psd, to the total variance, we can calculate the ratio of the integral over the specified range of frequencies to the integral of the entire psd.

12.7. Filtering time series

When analyzing time series, we are often more interested in some frequencies than others. For example, we might be interested in determining the flood signal in water levels in a tidally influenced channel or the multi-year climatic signal in a discharge record. In these cases, we would be concerned with frequencies lower than tidal or annual frequencies, respectively. In other cases, we might be most interested in the high frequency portion of a time series. To select or eliminate a given range of frequencies from a time series, we *filter* the data. This can be done in the *time domain* (e.g., using a running average, which is very good at smoothing data but is poor at preserving information about phase and doesn't have as sharp a frequency cutoff) or in the *frequency domain*. We will focus on the latter. If we are interested in the low frequencies, we want to *low-pass* filter the data. To retain only high frequencies, we want a *high-pass* filter. To select some intermediate range of frequencies, we must use a *band-pass* filter.

The concept behind frequency-domain filtering of a time series is straightforward once we've determined its power spectral density. Essentially, we want to apply a box-car-like function to the psd, retaining the frequencies of interest and setting the coefficients of frequencies outside the range of interest to zero. The inverse Fourier transform of the modified psd should yield a time series containing only the desired frequencies of the original time series. The biggest issue we face in doing this is the choice of the 'box-car-like' function. We want a filter that introduces the least distortion into the time series in the chosen range of frequencies. The best filters use a smoothly varying function (taper) between the 1's assigned to the frequencies to be retained and the 0's assigned to the frequencies to be removed. Optimal filter design is a problem that has received much attention. There are many approaches and many considerations. In fact, *MATLAB* has a whole toolbox devoted to filter design. Here, we consider just a simple example to illustrate the basic ideas of a frequency-domain filter.

The m-file, `lpfilt.m`, is an example of a low-pass filter for time series. The syntax is

```
xf=lpfilt(x,dt,cutoff_frequency)
```

where `x` is the original time series, `xf` is the filtered time series, `dt` is the sampling interval (used to determine the frequencies), and `cutoff_frequency` is the highest frequency to be retained. This can be modified to obtain a high-pass filter by adjusting the frequencies assigned 0's and 1's and the points assigned the taper values (0.715, 0.24, 0.024). Optimal values for the taper depend on the sampling frequency and period of the signal and the width of the frequency range to be passed in the filter (bandwidth). The taper values used in `lpfilt.m` fall near the middle of the range of values calculated by Rabiner et al. (1970; generally ±0.05 or less for 0.715 and 0.24; ±0.01 for 0.024).

```
function fdata = lpfilt(data,delta_t,cutoff_f)
```

```
%  Peforms low-pass filtering by multiplication in frequency domain using
%  a 3-point taper (in frequency space) between pass-band and stop band.
%  Coefficients suggested by D. Coats, Battelle, Ventura.
%  Written by Chris Sherwood, 1989; revised by P. Wiberg, 2003.
n=length(data);
mn=mean(data);
data=data-mn;  %linearly detrend data series
P=fft(data); N=length(P);
filt=ones(N,1);
k=floor(cutoff_f*N*delta_t);
% filt is a tapered box car, symmetric over the 1st and 2nd half of P,
% with 1's for frequencies < cutoff_f and 0's for frequencies > cutoff_f
filt(k+1:k+3)=[0.715 0.24 0.024];
filt(k+4:N-(k+4))=0*filt(k+4:N-(k+4));
filt(N-(k+1:k+3))=[0.715 0.24 0.024];
P=P.*filt;
fdata=real(ifft(P));  %inverse fft of modified data series
fdata=fdata(1:n)+mn;   %add mean back into filtered series
```

To illustrate the use of `lpfilt.m`, we can low-pass filter the Mauna Loa CO_2 record to eliminate the most prominent frequencies and focus on the variations at periods longer than a year. We use `zf=lpfilt(z,1/12,1/2)` to obtain a time series that includes only periods of two-years or longer. The result, for the detrended time series, is plotted in Figure 12.8. The low-pass filtered time series (Figure 12.8) shows that there is interannual variability in the CO_2 time series, though these variations are small compared to the annual signal (with an annual range of almost 6 ppm) and the long-term trend (about 55 ppm).

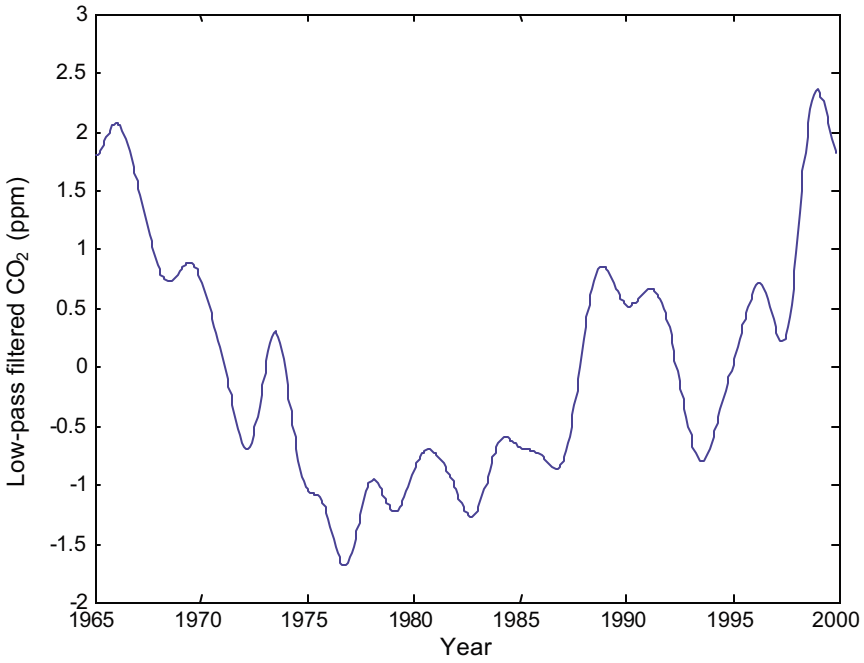
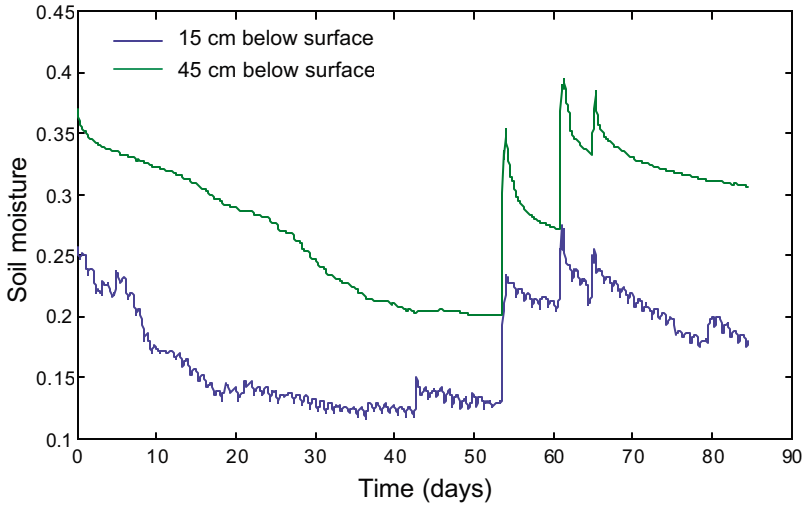


Figure 12.8. Low-pass filtered CO_2 record from Mauna Loa.

12.8. Problems

The time series of soil moisture plotted below was collected in an agricultural field at the Virginia Coast Reserve (VCR) Long-Term Ecological Research (LTER) site on the Eastern Shore of Virginia during the summer of 1998. The time series comprises hourly measurements at depths of 15 and 45 cm below the ground surface over a period of 85 days. The time series generally shows falling values of soil moisture due to evapotranspiration punctuated by sharp rises associated with precipitation. The data are in the file ltersm.dat.



1. Select a 20-day or longer segment (integer number of days) of the 15-cm record lacking any sharp rises in soil moisture. Calculate the Fourier coefficients (using `fft`) and the power spectral density (using `specclc` or one of the *MATLAB* signal processing functions such as `periodogram` or `pwelch`). What percentage of the total variance of this time series is represented by the diurnal signal? Estimate the amplitude of the diurnal variation in soil moisture.

2. a) Compare the power spectral density function for the segment used in 1) with that for the full time series. Suggest an interpretation for the difference between the two spectra.

   b) Compare the power spectral density function for the 15-cm and 45-cm records of soil moisture. Suggest an interpretation for the difference between the two spectra.

3. a) Filter the time series from the shallower depth to remove the diurnal signal. Compare the time series and spectra of the filtered and unfiltered time series to see how effective the filter was.

   b) Modify lpfilt.m to obtain a high-pass filter. Filter out everything with a lower-than-daily frequency. How does the amplitude of the signal compare with the value obtained in 1)?

12.9. References

Chatfield, C., The Analysis of Time Series: an Introductoin, 6th Ed, 333 pp, Chapman and Hall/CRC, Boca Raton, FL, 2004.

Rabiner, L.R., B. Gold, and C.A. McGonegal, An approach to the approximation problem for non-recursive digital filters. *IEEE Trans. Audio and Electroacoustics*, AU-18: 83-106, 1970.

Box 12.1. Fourier series in terms of complex numbers

Complex numbers have the form $z=a+ib$, where a and b are real numbers and $i^2=-1$. We say that a=real(z) and b=imag(z). The real numbers that we most commonly use in mathematical calculations can be considered a subset of the system of complex numbers where b=0. All standard arithmetic operations can be done with complex numbers.

The complex variable z can also be defined in terms of its magnitude $r=\sqrt{a^2+b^2}$ and direction θ as $z=r(\cos\theta+i\sin\theta)$; $a=r\cos\theta$, $b=r\sin\theta$. Defining $\cos\theta+i\sin\theta\equiv e^{i\theta}$ (Euler's identity), we can write $z=re^{i\theta}$; $e^{-i\theta}=\cos\theta-i\sin\theta$. Using Euler's identity, we can rewrite the Fourier series (equation (12.1)) as

$$f(k)=\sum_{n=0}^{\infty}\frac{a_n+ib_n}{2}(\sin\frac{2\pi nk}{L}+i\cos\frac{2\pi nk}{L})+\frac{a_n-ib_n}{2}(\sin\frac{2\pi nk}{L}-i\cos\frac{2\pi nk}{L})$$

$$=\sum_{n=0}^{\infty}c_n e^{i2\pi nk/L}+\sum_{n=0}^{\infty}c_{-n}e^{-i2\pi nk/L}=\sum_{n=-\infty}^{\infty}c_n e^{i2\pi nk/L}$$

Expressing the Fourier series in terms of complex numbers permits a range of tools of complex analysis to be used in evaluating the Fourier coefficients. In fact, the development of complex analysis is linked to the study of Fourier analysis. Methods such as the fast Fourier transform (fft) are developed using the complex form of the Fourier transform.

CHAPTER 13

Methods of Data Analysis: Spatial Data

Numerical Methods in the Hydrological Sciences
Special Publication Series 57

doi:10.1029/057SP18

13. Methods of Data Analysis: Spatial Data

13.1. Background

One-dimensional spatial data are much like time series, and many of the same techniques can be used. Often, however, the spatial data we are interested in are two or three-dimensional. In hydrology, these include topographic data, measurements of soil properties such as hydraulic conductivity and soil moisture, groundwater heads, and contaminant concentrations. A common step in analysis of spatial data is to contour the data. In some cases the data may be evenly spaced, but more often they are irregularly spaced. Contouring provides information about the distribution and gradients of a spatially distributed parameter, but typically not about the uncertainty in those distributions or gradients. A variety of geostatistical methods, such as kriging, have been developed that can provide confidence limits on spatially interpolated data. In this chapter, we introduce the basics of contouring and geostatistics. Much more sophisticated formulations of these methods are available in freely or commercially available geographic information system (GIS) software packages. Our goal is to introduce the general concepts and some of the issues that must be considered when working with spatial data.

To illustrate the methods and functions presented in this chapter, we will use measurements of land-surface and water-table elevation from Oak Ridge National Laboratory (data file ornlgw.dat). The data, which are not regularly spaced, include the "Easting" (x) and "Northing" (y) for each pair of ground (z_g) and water-table (z_w) elevations (all in feet).

13.2. Interpolating irregularly spaced data onto a regular grid

There are typically three steps involved in contouring data. First, irregularly spaced points are interpolated onto a regular grid. Second, the locations of points corresponding to the contour values is determined. Finally, a technique must be chosen for connecting the points defining each contour.

MATLAB's contouring program assumes that the variable to be contoured is specified on a regular grid. We can use the *MATLAB* function `griddata` to interpolate irregularly spaced data onto a regular grid, but first we have to specify the grid. We can use the *MATLAB* function `meshgrid` to do this[1].

```
MESHGRID   X and Y arrays for 3-D plots.
   [X,Y] = MESHGRID(x,y) transforms the domain specified by vectors
   x and y into arrays X and Y that can be used for the evaluation
   of functions of two variables and 3-D surface plots.
   The rows of the output array X are copies of the vector x and
   the columns of the output array Y are copies of the vector y.

   [X,Y] = MESHGRID(x) is an abbreviation for [X,Y] = MESHGRID(x,x).
   [X,Y,Z] = MESHGRID(x,y,z) produces 3-D arrays that can be used to
   evaluate functions of three variables and 3-D volumetric plots.

   For example, to evaluate the function  x*exp(-x^2-y^2) over the
   range  -2 < x < 2,  -2 < y < 2,
```

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

```
[X,Y] = meshgrid(-2:.2:2, -2:.2:2);
Z = X .* exp(-X.^2 - Y.^2);
mesh(Z)
```

The following m-file uses the `meshgrid` function to generate a grid for the Oak Ridge data.

```
ornlgw=tblread('ornlgw.dat');
e=ornlgw(:,1);   %Easting (ft)
n=ornlgw(:,2);   %Northing (ft)
gse=ornlgw(:,3);  %ground surface elevation (ft)
wse=ornlgw(:,4);  %water table elevation (ft)
minx=min(e); maxx=max(e);
miny=min(n); maxy=max(n);
dx=100; dy=100;   %grid spacing
[ei,ni]=meshgrid(minx-dx:dx:maxx+dx,miny-dy:dy:maxy+dy);
plot(ei,ni,'c',ei',ni','c',e,n,'ro')
axis ('equal')   %makes the x and y axis intervals the same
```

and produces this result

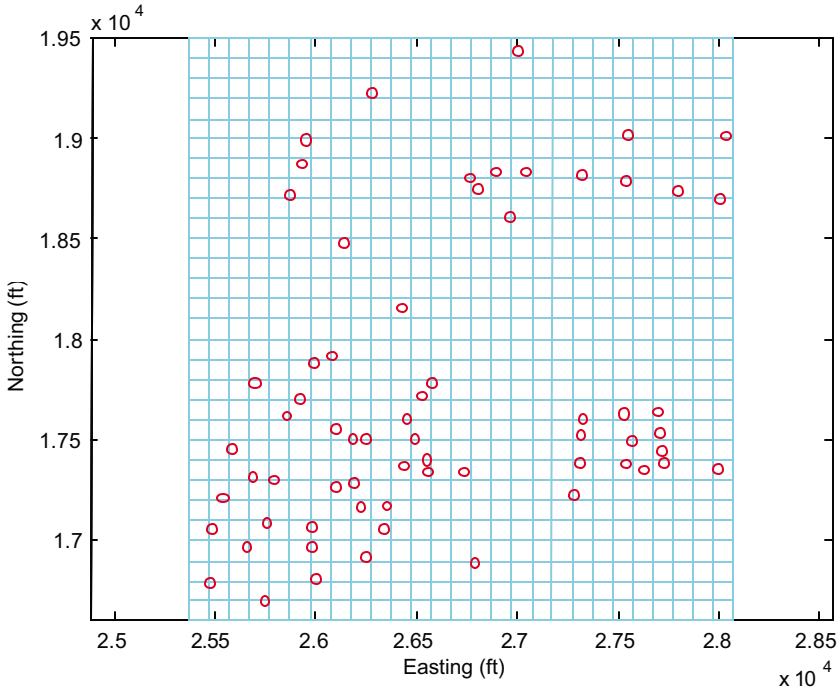


Figure 13.1. Distribution of Oak Ridge sampling points and proposed grid.

Once we have generated the grid, we can use `griddata` to interpolate the data onto the grid. The simplest gridding algorithms average the data values at the *n* control (data) points closest to a grid point. However, when the control points are unevenly distributed, this method can yield poor results. The grid interpolation methods in *MATLAB* (version 5 and later) are based on Delaunay triangulation. In this method, a triangular mesh is formed from the control points by connecting each point to its "natural neighbors". The resulting mesh of lines forms a triangular tesselation of the region with no intersecting lines except at the control points.

There are many ways to triangulate a region. Which are the "right" points to connect? Consider the subset of 9 Oak Ridge sampling locations indicated by the blue ×'s and the red ∗ in Figure 13.2. There are 28 triangles we can draw that include the location indicated by the ∗. The secret of Delaunay triangulation lies in the fact that each set of 3 points that defines a triangle also defines a circle. When all possible circles are formed, the circles with no interior data points (i.e., no data (control) points lie within the circles) identify the points that should be connected into triangles. For example, Figure 13.2 shows the 9 smallest circles that go through the red ∗ and two of the remaining 8 points (×'s). The circles shown with solid red lines are the smallest three. None of the red circles contains another point while each of the light blue circles does. The triangles (blue lines) are defined by these natural neighbors of the ∗ point. There will also be triangles that connect to the point on the far right of region shown in Figure 13.2; these will be larger than the ones on the left because of the larger distance between the points. One of the properties of a Delaunay triangulation is that the triangles are as close to equilateral as is possible given the locations of the data points. This helps to improve the accuracy of interpolations based on the triangulation. See Watson (1992) or Davis (2002) for more details of Delaunay and other triangulation schemes.

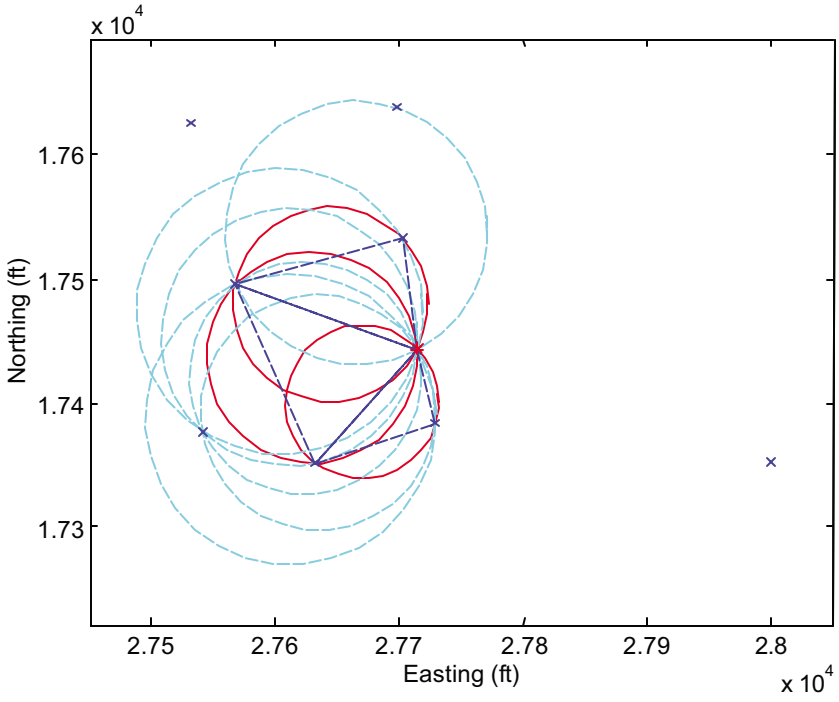


Figure 13.2. Delaunay triangulation of a subset of the Oak Ridge sampling sites.

The *MATLAB* command `delaunay` provides an implementation of this method. The following continuation of the gridding m-file uses this function to calculate the Delaunay triangulation of the Oak Ridge sampling locations.

```
plot(e,n,'o'); axis('equal'); hold on
dtm=delaunay(e,n);
trimesh (dtm,e,n,gse);
hold off; hidden off
```

The resulting triangulation is shown in Figure 13.3. Consult the *MATLAB* help file for more information about the function `delaunay`.

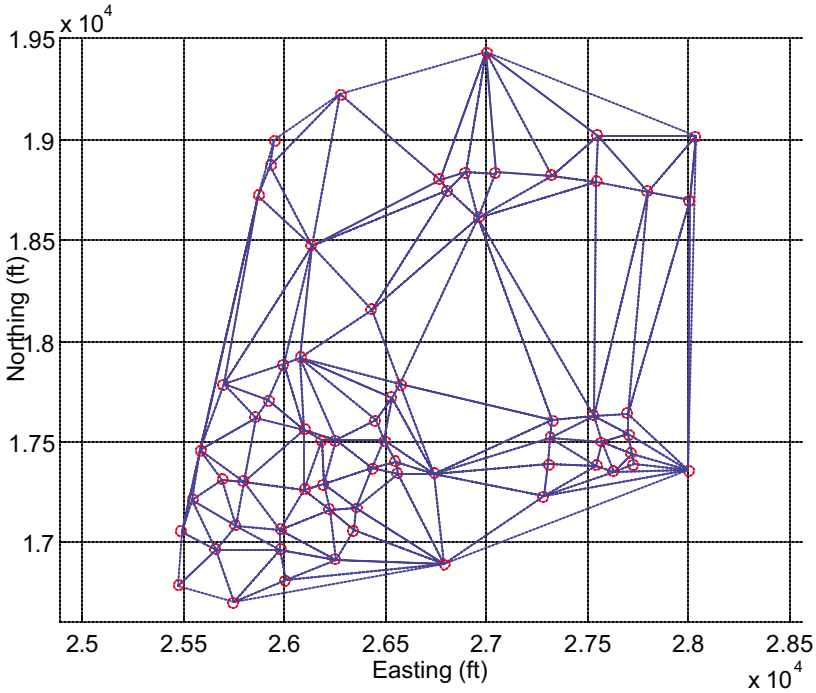


Figure 13.3. Delaunay triangulation of Oak Ridge sampling sites.

Now we have to consider how to use the known data values (control points) to interpolate the data onto our grid. One possibility is to assign to each point on the grid the value of the nearest control point. We wouldn't even need to triangulate the region for this method, but, as you might expect, the resulting surface is very blocky and not acceptable for most purposes. Another possibility is to average the three natural neighbors for a given grid location – the three control points defining the triangle in which the grid point falls. However, if a grid point is much closer to one of the control points than the other two, an equal weighting of the 3 control points might not give the best value. Intuitively, it seems that we might get the best results if we used some sort of weighted average of the natural neighbors. One way of weighting the nearest control points is by a function that depends inversely on the distance from the grid point.

The *MATLAB* `griddata` function performs the necessary interpolation. The help file for `griddata` is given below[2]. There are 3 options for how to do the interpolation: 'linear', 'cubic', and 'nearest', as well as a method carried over from an older version of *MATLAB*. 'Linear' and 'cubic' use weighted averages, while 'nearest' assigns the value of the nearest control point; 'cubic' produces a smooth surface

```
GRIDDATA Data gridding and surface fitting.
    ZI = GRIDDATA(X,Y,Z,XI,YI) fits a surface of the form Z = F(X,Y)
```

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

```
to the data in the (usually) nonuniformly-spaced vectors (X,Y,Z)
GRIDDATA interpolates this surface at the points specified by
(XI,YI) to produce ZI.  The surface always goes through the data
points.  XI and YI are usually a uniform grid (as produced by
MESHGRID) and is where GRIDDATA gets its name.

XI can be a row vector, in which case it specifies a matrix with
constant columns. Similarly, YI can be a column vector and it
specifies a matrix with constant rows.

[XI,YI,ZI] = GRIDDATA(X,Y,Z,XI,YI) also returns the XI and YI
formed this way (the results of [XI,YI] = MESHGRID(XI,YI)).

[...] = GRIDDATA(...,'method') where 'method' is one of
    'linear'    - Triangle-based linear interpolation (default).
    'cubic'     - Triangle-based cubic interpolation.
    'nearest'   - Nearest neighbor interpolation.
    'v4'        - MATLAB 4 griddata method.
defines the type of surface fit to the data. The 'cubic' and 'v4'
methods produce smooth surfaces while 'linear' and 'nearest' have
discontinuities in the first and zero-th derivative respectively.  All
the methods except 'v4' are based on a Delaunay triangulation of the
data.

See also INTERP2, DELAUNAY, MESHGRID.
```

The gridded surface calculated with `zgi=griddata(e,n,gse,ei,ni,'linear')` is plotted in Figure 13.4.

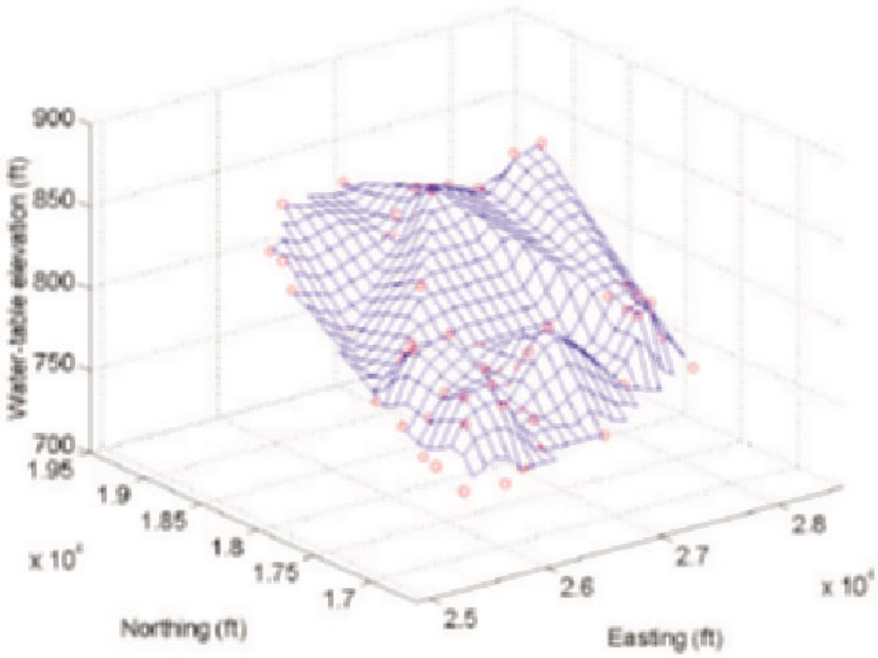


Figure 13.4. Surface elevations of the Oak Ridge sampling region calculated using linear interpolation.

13.3. Contouring

Once the data have been interpolated onto a regular grid, we can consider how to contour them. When we contour data, we are connecting points of equal value of some spatially distributed variable. The resulting curves are constrained by some contouring rules (after Middleton, 2000):

1. Contour lines don't cross or merge with other contour lines.
2. Contour lines pass between known points having higher and lower values than the contour value.
3. Contour lines are repeated to indicate a reversal of slope.
4. Contour lines must close or end at the boundaries of the region.

In addition, there are some conventions commonly adopted in contouring, such as use of a standard datum and equal contour intervals.

MATLAB has a built-in contouring routine, `contour`, that works with gridded data. The function finds the contour points and connects them with straight lines. Smoother contours require a finer grid. The command

```
[c,h]=contour(ei,ni,zi,750:10:850)
```

followed by

```
clabel (c,h,[760 780 800 820 840])
```

produces the contour plot shown in Figure 13.5a. The gridding upon which these contours were drawn is based on *MATLAB*'s default linear weighting. If instead we grid the data using the cubic weighting function, we obtain the contours plotted in Figure 13.5b. While the two contour maps are generally similar, they do differ in some details as a result of the different interpolation methods. See Davis (2002) for a more thorough discussion of interpolation and contouring. It is worth noting that manual contouring was typically done directly from a triangulation without gridding the data. The resulting contours are similar to those found from gridded data except in places where the triangles are very acute and differences in slope of the faces of the triangles is relatively large (Davis, 2002).

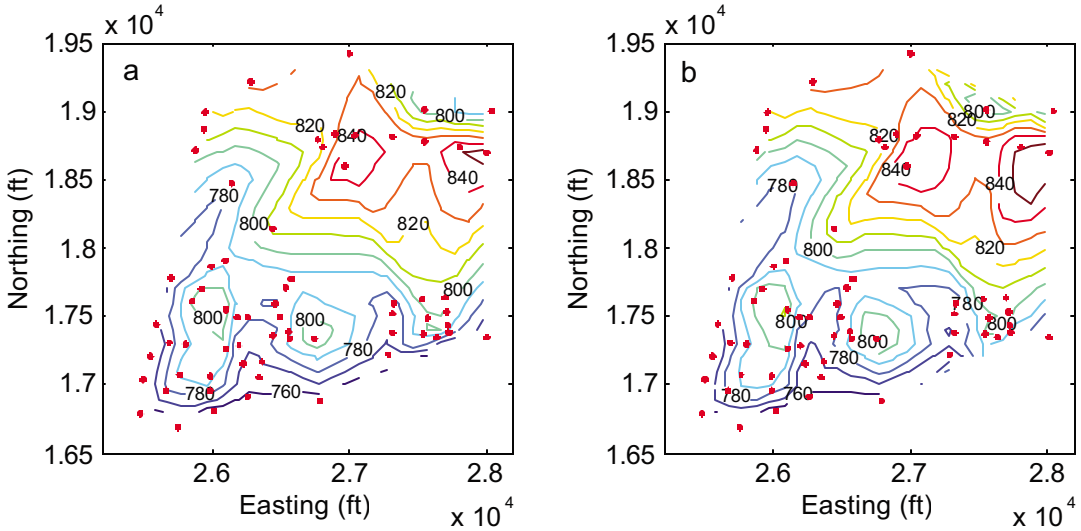


Figure 13.5. *MATLAB*-generated contours using (a) linear and (b) cubic weighting to grid the data.

13.4. Semivariance

Surfaces can be smooth, bumpy, ridged, or some combination. On a smoothly varying surface, a control point quite a distance from a point of interest might be useful in estimating the value of a variable at that point. In contrast, a nearby control point on a bumpy surface may not be useful at all. Intuitively, we might expect that if we could define the distances over which spatially distributed data were correlated, that some length scale related to the correlation distances might provide the basis for improved gridding and contouring algorithms.

We can characterize temporal correlations in a time series by calculating the autocovariance. This is done by taking two representations of a time series, sliding them past each other in time-step increments (lags), and calculating the covariance of the two time series. The autocorrelation is the autocovariance divided by the variance of the time series. The autocovariance of a time series at zero lag is equal to the variance, so the autocorrelation of a time series is 1 for lag = 0 and generally decreases with increasing lag. If the time series is periodic, then the autocorrelation will rise again at lags equal to the periodicities contained in the time series.

For spatial data, a related function called the semivariance typically is used to characterize the spatial correlation. The semivariance for a one-dimensional spatial series of equally spaced values is given by

$$\gamma_h = \frac{1}{2n}\sum_{i=1}^{n-h}(z_i - z_{i+h})^2 \tag{13.1}$$

where z represents the data values, h is the lag index and n is the length of the series (see, e.g., Middleton, 2000; Davis, 2002). In general, the semivariance is smallest when the lag is close to 0 and increases as the lag increases up to some maximum value (termed the 'sill'), after which the semivariance remains roughly constant at a value near the variance of the whole data series. The sill is reached at a lag distance $h = a$, where a is termed the 'range'. At separations greater than the range, data values are essentially uncorrelated. Several models for the general form of a semivariogram are shown in Figure 13.6. In these models, the semivariance is zero at a lag of zero and rises to a normalized semivariance of 1 (γ_h = var(z)) at a normalized lag h/a near 1. A semivariance model is typically fit to a semivariogram calculated from experimental data to provide a continuous description of the semivariance for use in interpolation methods such as those described in the next section. A linear semivariogram model has the form

$$\begin{aligned} \gamma_h &= \alpha h \quad \text{for } h < a \\ \gamma_h &= \text{var}(z) \quad \text{for } h \geq a \end{aligned} \tag{13.2}$$

while a spherical model is described by

$$\begin{aligned} \gamma_h &= \text{var}(z)\left(\frac{3h}{2a} - \frac{h^3}{2a^3}\right) \quad \text{for } h < a \\ \gamma_h &= \text{var}(z) \quad \text{for } h \geq a \end{aligned} \tag{13.3}$$

(Davis, 2002).

Equations for the exponential model and more information about semivariance models in general can be found in Davis (2002) or Middleton (2000).

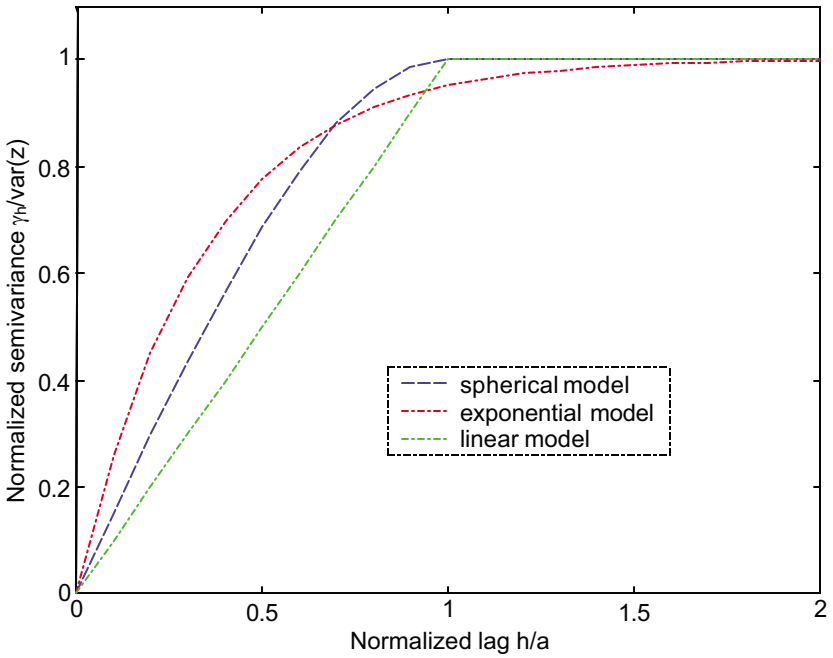


Figure 13.6. Theoretical semivariance models

To calculate the semivariogram for irregularly spaced experimental data, each point is taken pairwise with all of the other points in the region of interest, the distance between them (in the *x*-*y* plane) determined, and the squared difference in *z* values (e.g., surface elevation) calculated. Then, the results can be aggregated into distance bins (or directional bins) and averaged to obtain the semivariogram. The following m-file, `semivar.m`, does this using distance bins, but could be adapted for directional bins. By default, `semivar.m` used 20 distance bins; the code can easily be modified to change the number of bins. The surface should be detrended prior to calculating the semivariogram. An m-file (`detrendsurf.m`) to remove a 1st-order trend surface (plane) from a set of spatial data (input as an N×3 matrix of *x*, *y* (location) and *z* (data) values) is included with the m-files for this chapter.

```
%semivar.m
data=tblread('ornlgw.dat');
X=data(:,1:3)  %select x, y, and ground-surface elevation data
dtX=detrendsurf(X); %detrend surface
x=dtX(:,1); y=dtX(:,2); z=dtX(:,3); %assign x,y,z values
n=length(x); nn=n*n;
%calculate distance between each pair of points
xx=repmat(x,1,n);
yy=repmat(y,1,n);
zz=repmat(z,1,n);
dx=xx-xx'; dy=yy-yy';
h=sqrt(dx.^2+dy.^2);
%eliminate duplicate values in the lower triangular part of the matrix
% and sort values by distance
```

```
hv=reshape(triu(h),nn,1);
[nr,nc,nhv]=find(hv);  %nhv are the nonzero values of hv
[sh,ih]=sort(nhv);
%calculate the squared differences in z values for each pair of points
zvar=(zz-zz').^2; vz=reshape(zvar,nn,1); nvz=vz(nr); sv=nvz(ih);
%only compare distances that are half or less of the largest distance
hmax=max(sh)./2; nm=length(find(sh<=hmax));
nbin=20; hinc=hmax/nbin; %set the number of distance bins to 20
% adjust bin width to be round number
% The values for the magnitude of hinc (10, 100, 1000) may need to be
% adjusted depending on domain size and distance units.
if hinc<=10, hinc=floor(hinc);
   elseif hinc<=100, hinc=floor(hinc/10)*10;
   elseif hinc<=1000, hinc=floor(hinc/100)*100;
   end;
hvec=0:hinc:nbin*hinc; %vector of bin endpoints
for i=1:nbin-1;
   ib=[]; ib=find(sh>hvec(i)&sh<=hvec(i+1)); %find distances in each bin
   if ~isempty(ib),
      ni(i)=length(ib);
      gamh(i)=sum(sv(ib))./(2*ni(i)); %calculate semivariance for each bin
   else gamh(i)=0; ni(i)=0; %set semivariance to 0 if no data in a bin
   end;
end;
hvmn=(hvec(1:nbin-1)+hvec(2:nbin))./2; %find the mid-point of each bin
plot(hvmn,gamh,'bo') %plot the results
xlabel('Distance (ft)')
ylabel('Semivariance')
```

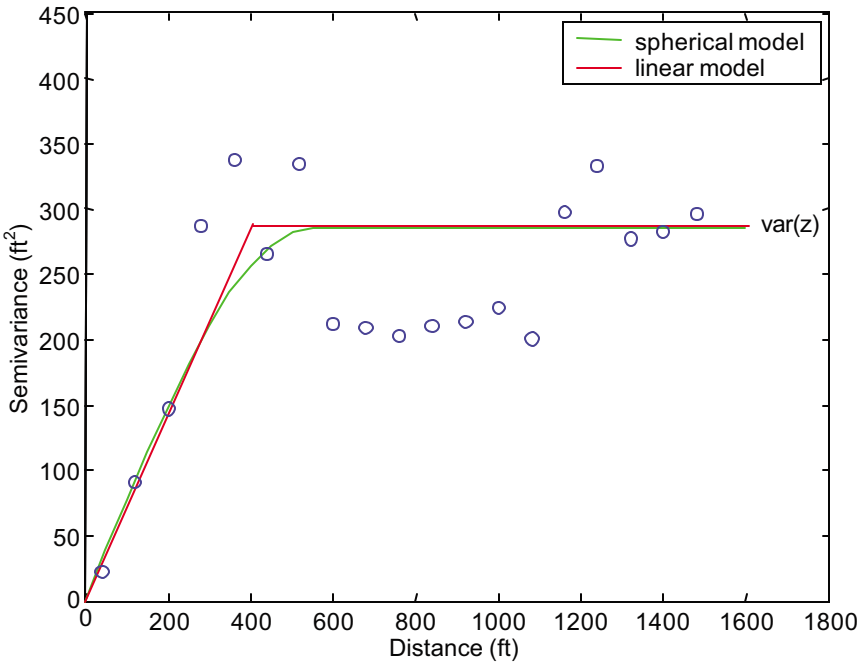


Figure 13.7. Calculated semivariance and fitted models for the detrended Oak Ridge elevation data.

The semivariogram shown in Figure 13.7 was calculated using `semivar.m` for the Oak Ridge ground surface elevation data. It doesn't look as neat as the model semivariograms shown previously, but the values appear to reach the sill ($\gamma_h = \text{var}(z)$) at a range (distance) of about 400 ft. Conceptually, the range, a, indicates the maximum distance over which the data are spatially correlated. As a result, we might expect to obtain better results when performing spatial interpolations if we weight neighboring points within the range most heavily. This is the basis for the method known as *kriging*. Fitting the surface elevation semivariogram by eye using a linear model (equation 13.2) gives $\alpha = 0.7$ and $a = 400$ (Figure 13.7). Fitting a spherical model gives $a = 550$ (Figure 13.7). Middleton (2000, p. 154) notes that "the form of the semivariogram is generally not very well known, with the data showing large deviations from the theoretical models that are used."

13.5. Kriging

Kriging, a technique developed by D.G. Krige for mining purposes, provides an alternative approach to gridding irregularly spaced data. In kriging, the semivariogram is used to determine the best weights to apply to the control points when performing spatial interpolation. It also provides an estimate of the error associated with the predicted value. There are many kriging packages available as shareware or for purchase. A simple m-file is provided below that does punctual or ordinary kriging in which the mean of the variable to be interpolated is assumed to be constant over the region containing the points used to construct the semivariogram; a more general procedure called universal kriging does not make this assumption (Davis, 2002). The purpose of this m-file is mostly to illustrate the basic ideas of kriging; more sophisticated programs are needed for more general application.

The m-file `ordkrig.m` follows the mathematical formulation provided in Davis (2002). The basic idea is that we can estimate the value of z at a specified point (x_0, y_0) as:

$$z_0 = \sum_{i=1}^{k} \lambda_i z_i \tag{13.4}$$

where λ_i are weights, z_i are the values of z at the control points and k is the number of observations used in the estimate. The problem is solved using a Lagrange multiplier with the constraints that the weights result in minimum error variance and sum to 1 (see Davis, 2002). The weights are estimated by the matrix equation

$$\mathbf{\Lambda} = \mathbf{S}^{-1}\mathbf{B} \tag{13.5}$$

where $\mathbf{\Lambda}=[\lambda_1, \lambda_2, \ldots \lambda_k, \mu]'$ (column vector), μ is the Lagrange multiplier, $\mathbf{S}$ is the matrix of semivariances for each of the pairs of control points being used, and $\mathbf{B}$ is a column vector of semivariances for each of the k control points paired with the point being interpolated, z_0. Using this, (13.4) can be rewritten as

$$z_0 = \mathbf{Z}\mathbf{\Lambda} = \mathbf{Z}\mathbf{S}^{-1}\mathbf{B} \tag{13.6}$$

where $\mathbf{Z}=[z_1, z_2, \ldots z_k, 0]$ (row vector). The variance of the estimate is given by

$$\text{var}(z_0) = \mathbf{B}'\mathbf{\Lambda} \,. \tag{13.7}$$

Calculated variances can be converted to 95% confidence intervals on the predicted value of z as $ci_{95} = \pm 1.96\sqrt{\text{var}(z_0)}$ (Davis, 2002).

```
%ordkrig.m
%uses ordinary kriging and a spherical semivariance model to predict the
%elevation z0 and its associated variance at a specified location x0, y0.

data=tblread('ornlgw.dat');
X=data(:,1:3);  %select x, y, and ground-surface elevation data
[dtX,b]=detrendsurf(X); %detrend surface
x0=input('enter the x coordinate of the point to be estimated: ');
y0=input('enter the y coordinate of the point to be estimated: ');
mz0=b(1)+b(2)*x0+b(3)*y0; %z0 value on trend surface
svrange=input('enter the range for a spherical semivariance model \n fit to
the control data: ');
x=[dtX(:,1);x0];
y=[dtX(:,2);y0];
z=dtX(:,3);
n=length(x);
xx=repmat(x,1,n);
yy=repmat(y,1,n);
dx=xx-xx'; dy=yy-yy';
h=sqrt(dx.^2+dy.^2); %calculating the distances between points
h0=h(end,:); %distances from the estimation point
dx0=dx(end,:); dy0=dy(end,:);
hn=find(h0<svrange); %select points within range to use in estimate
dxn=dx0(hn); dyn=dy0(hn);
hh=sqrt(dxn.^2+dyn.^2); %dist. of selected points from estimation point
xi=x(hn); yi=y(hn); m=length(xi);
xxi=repmat(xi,1,m);
yyi=repmat(yi,1,m);
dxxi=xxi-xxi'; dyyi=yyi-yyi';
hhi=sqrt(dxxi.^2+dyyi.^2);
gamh=var(z)*(1.5*hhi/svrange-0.5*(hhi/svrange).^3);%calculate semivariance
S_top=[gamh(1:m-1,1:m-1) ones(m-1,1)];
S_last=[ones(1,m-1) 0];
S=[S_top;S_last]; %construct semivariance matrix S
B=[gamh(1:m-1,end);1]; %semivariance vector B
lamv=S\B; %lamv are the weights
zi=lamv(1:m-1)'*z(hn(1:m-1)); %calculate estimated z
z0=zi+mz0; %add in trend surface value for z0
var0=lamv'*B; %calculate the variance in estimate of z
fprintf(1,'Estimated z0 = %5.0f at x0 = %7.2f, y0 =%7.2f \n',[z0,x0,y0])
fprintf(1,'Variance of estimate of z0 = %5.0f',var0)
```

As an example, `ordkrig.m` was used to predict the value of surface elevation at $x_0 = 2.60\times10^4$ ft, $y_0 = 1.73\times10^4$ ft, an area of topographic variation but also many control points. The range from the fit of the spherical semivariance model to the Oak Ridge data (550 ft, Figure 13.7) was used in the calculation. The predicted elevation is $z_0 = 800 \pm 19$ ft (var(z_0) = 91 ft^2). If a location in a 'smoother' region is selected, e.g., $x_0 = 2.70\times10^4$ ft, $y_0 = 1.80\times10^4$ ft (see Figures 13.4 and 13.5), `ordkrig` gives $z_0 = 789 \pm 42$ ft (var(z_0) = 461 ft^2). There is a larger error associated with this estimate because there are fewer nearby control points, suggesting we should question whether the surface really does vary smoothly in the middle of the region or whether the smoothness is an artifact of the lack of control points in that area.

13.6. Problems

One of the most common and challenging applications of the contouring and geostatistical techniques considered in this chapter is contouring topographic data. A data set from Davis (1973) has become a common test set for contouring algorithms and is provided as file davistopo.dat. Note that the x and y values are given in terms of unit distances, where one unit is 50 ft with the origin at the southwest corner of the map. The unit distances should be converted to real distance for use in `semivar.m` (or the ranges of `hinc` in `semivar.m` adjusted to allow for `hinc` < 1).

1. Develop an m-file that plots the spatial distribution of points and the Delaunay triangulation of the surface. Then calculate gridded values and contour them. Investigate the effect of grid spacing and interpolation weighting on the resulting contour plots. There is a stream network through this region. Try to identify where the stream channels are based on your contour plot. Consult Davis (2002, p. 373; also shown in earlier editions of the book) for a map showing the channel locations.

2. Use `semivar.m` to calculate the semivariogram for this data set and fit it with a linear and spherical model (equations 13.2 and 13.3). Limit the maximum semivariance to the overall variance of the data set. Then use `ordkrig.m` to grid the values. [You may want to convert `ordkrig` to a function that returns the interpolated value and variance for given x,y values.] Determine not only the gridded elevations, but also the surface defined by the error variance at each interpolation point. What does this reveal about the interpolated elevations? Contour the resulting gridded values and compare the results to those obtained in Problem 1.

13.7. References

Davis, J.C., *Statistics and Data Analysis in Geology*, 638pp, Wiley, New York, 2002.

Middleton, G.V., *Data Analysis in the Earth Sciences Using Matlab*, 260 pp., Prentice Hall, Upper Saddle River, NJ, 2000.

Watson, D.F., *Contouring*, 321pp, Pergamon Press, Oxford, 1992.

Subject Index

M-file name	*Chapter*	*Description*
ad_w_decay.m	B10.1	Finite-element solution to transient advection and dispersion of a substance undergoing decay
adv_disp.m	9.3	Crank-Nicolson solution to advection-dispersion equation
besselfd.m	5.5	Finite difference solution to the Bessel equation
bisect.m	2.5	Bisection routine for finding roots
brusselator.m	B4.1	Function file for the equations for the "brusselator"
co2anal.m	12.4	Fourier analysis of Mauna Loa CO_2 time series
deriv.m	3.4	Difference formulae for derivatives
detrendsurf.m	13.4	Removes a 1st-order trend surface (plane) from a set of 2-D spatial data
diagen.m	10.5	Finite-element solution to one-dimensional diffusion into sediment
euler_ex.m	4.2	Euler solution to first-order ordinary differential equation
explicit.m	8.3	Explicit solution to 1-D groundwater problem
G_puff.m	1.4	Gaussian puff model of contaminant dispersion
geotherm.m	10.6	Finite-element solution to one-dimensional conductive heat flow in the earth; calls the function econd.m
hamon_PET.m	1.4	Hamon calculation of PET in mm per day
hump.m	B2.3	Iterative solution showing complex behavior
implicit.m	8.5	Implicit solution to 1-D groundwater problem
kinsorp.m	9.4	Implicit solution to advection-dispersion equation with kinetic sorption
linreg.m	1.4	Simple linear regression
lpfilt.m	12.7	Simple low-pass, frequency domain filter
main_fe2d.m	11	Finite-element solution for hyporheic zone flow that calls elements.m, gridplot.m, assemble.m and setrhs_solve.m
ode_ex.m	4.6	Solution to ordinary differential equation for settling of a spherical particle in water using *MATLAB* functions ode23 and ode45
ordkrig.m	13.5	Simple ordinary kriging routine
pondex.m	6.6	Example solution of the Poisson equation for groundwater flow
rates.m	4.9	Function file for the equations describing the dehydration of talc
richards.m	8.7	Predictor-corrector solution to the one-dimensional Richards equation for flow in an unsaturated soil; calls the functions dk.m, Kdpsi.m and dKdpsi.m
semivar.m	13.4	Calculates the semivariogram for unequally

		spaced points binned by distance
specclc.m	12.6	Simple power spectral density function
taylor_ex.m	4.2	Example solution of first-order ordinary differential equation using Taylor series method
toth.m	6.4	Solution to the Tóth problem of groundwater flow
well_drawdown.m	B11.1	Finite-element solution for transient drawdown

ad_w_decay.m	Finite-element solution to transient advection and dispersion of a substance undergoing decay
Assemble.m	Assemble the global conductance matrix
brusselator.m	Function file for the equations for the "brusselator"
co2anal	Fourier analysis of Mauna Loa CO_2 time series
diagen.m	Finite-element solution to one-dimensional diffusion into sediment
Elements.m	Calculate node connectivity matrix
Euler_ex.m	Euler solution to first-order ordinary differential equation
G_puff.m	Gaussian puff model of contaminant dispersion
geotherm.m	Finite-element solution to one-dimensional conductive heat flow in the earth.
Gridplot.m	Assign coordinate values and plot mesh
hamon.m	Hamon calculation of PET in mm per day
linreg.m	Simple linear regression
Lpfilt.m	Simple low-pass, frequency domain filter
main_fe2d.m	Finite-element solution for hyporheic zone flow
ode ex.m	Solution to ordinary differential equation for settling of a spherical particle in water using *MATLAB* functions `ode23` and `ode45`
Ordkrig.m	Simple ordinary kriging routine
pond.m	Example solution of the Poisson equation for groundwater flow
rates.m	Function file for the equations describing the dehydration of talc
richards.m	Predictor-corrector solution to the one-dimensional Richards equation for flow in an unsaturated soil
Semivar.m	Calculates the semivariogram for unequally spaced points binned by distance
setrhs_solve	Solve the finite-element equations
Specclc.m	Simple power spectral density function
taylor_ex.m	Example solution of first-order ordinary differential equation using Taylor series method
toth.m	Solution to the Tóth problem of groundwater flow
well_drawdown.m	Finite-element solution for transient drawdown

AGU Lecture Notes Adobe Help

Each PDF file is an electronic book or document. The PDF file format is useful because it faithfully stores and prints-out the full content, formatting and layout of the book pages with a single data file.

System Requirements
Adobe Acrobat 6.0 or higher should provide the best experience. Acrobat 3, 4, and 5 will support some to most of the feature in these PDFs, except for the embedded files. Download the latest Acrobat version from the Adobe website at:
http://www.adobe.com/products/acrobat/readstep2.html

Features in Lecture Notes PDFs
Each chapter of these Lecture Notes is a separate PDF, with most supporting material linked or embedded. Each chapter can be used individually, or used together with the wrest of the book. There are PDFs of all book's supporting pages, such as the book table of contents and book index.

> **Bookmarks** in the PDFs show the material available. Each chapter's subsections, boxes, figures, videos and MATLAB m files are bookmarked for quick access. Each chapter's pages are (in order) chapter table of contents, chapter body, chapter box(es), and other chapter's boxes mentioned in the chapter body.
>
> **Text links** are used where ever supporting text or figures are mentioned in a text passage. These links will help you explore the supporting materials, just remember to use the back one step arrow to return from the tangent.
>
> **Embedded files**, represented by blue push pins, are files stored inside a PDF. These files, figures, videos, MATLAB m and data files, can be downloaded from the PDF and used separately. Double click a push pin to download or use the file. These files are duplicates of the files in zip or tar.gz file of data files. Most, but not all, files in the data files folders are embedded in the PDF.

Additional Lecture Note Project Features

> **Data files folders**, files distinct from the PDF are organized by chapter, and ready for use in related applications such as MATLAB. These files are figures, videos, MATLAB m and data files. These files are distributed in zip or tar.gz files by chapter.
>
> The links on the **HTML pages** of book's table of contents and index work when saved in your local lecturenotes directory.

How to Use Lecture Notes PDFs
The following pages are tips on how to use PDFs in your teaching, and how to use PDFs. If you are unfamiliar with Adobe Acrobat, this will help. For complete Adobe Acrobat Reader documentation, open Acrobat Reader, open the "Help" menu, and select "Acrobat Help."

Pages and Navigation
To flip through pages of a PDF is like using a tape cassette player.

Adobe Acrobat navigation buttons

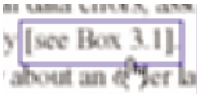
From left to right the first four buttons do the following.

|< Go to the first page of the book.
< Back one page
> Forward one page
>| Go to the last page of the book
<= Back one step (or one link)
=> Forward one step (or one link)

Links and Navigation
PDFs combine the utility of the book with the simplicity of the web link to connect ideas.

Bookmarks: stored in a panel on the left side of the screen. Clicking a bookmark text will open the connected page. We have bookmarked each chapter section, figure, chapter box, m file, and video.

Text links: indicated with a blue box in AGU PDFs. To explore the related idea, click the blue box. These links will take you to another page. When done exploring the related idea, click the back step arrow (<=) as many times as needed to return to the page where the text link was. Note: Text links on each chapter's table of contents page are invisible to ensure these page print clearly.



Example of a text link box.

Web links were not used inside the chapter PDFs to ensure reliability. No web access is required to use these PDFs. Weblinks are marked with a red link box.

PDFs in the Class Room
If you give lectures with the assistance of a computer LCD projector or big screen, Acrobat's full screen mode and magnify tool, hide clutter and help students focus on the material in the PDFs. To use less screen space on buttons and menus, it is best to use the files locally with Acrobat Reader, instead of inside a web browser.

Full Screen Mode shows only the full page of the document, and no application menus. The mode is controlled from the View menu in Acrobat or keyboard short-cuts. In Microsoft Windows, the short-cut keys are "*control L"* to activate, and either "*control L"* or *"Esc"* escape key to return to the normal view.

Full screen mode: A full page in an unclutter view

Magnifying tool can let you zoom in on a single item such as an equation. Select the tool from the application menu bar, and your mouse cursor will become a magnifying glass. Click several times on one spot to incrementally zoom in, or click-and-drag a box around what you want to show. Note the plus sign inside your mouse magnify icon.

Magnify button

Reducing magnification is done by holding the *"control"* key and mouse clicking. Note the minus sign inside you mouse magnify icon.

Returning to a standard view. There are three standard document views. **Fit to Window** and **Fit to Width** are most useful. Actual Size is another option. Fit to width will be the easiest to read from.

View buttons: Actual Size, Fit to Window and Fit to Width buttons

Embedded Files

Embedded files are useful related materials, inserted into the document. Blue push pin icons represent the embedded files. Placing your cursor over the push pin will show you the name of the file.

Right clicking will allow you to save the file locally on your hard drive. If you save the file locally, you must switch windows to your desktop or exit Acrobat and then navigate to the newly saved file to use it.

Embedded files function works in Acrobat 6, and some Acrobat 5 versions. Embedded files may not work for all users due to users computer configurations and application versions installed.

Embedded files: mouse is on the blue push pin icon, and the name is revealed.

If you double click a push pin, select “Open” in the dialog box. If the file type is supported by an installed application, it will open automatically in that application with temporary name. Otherwise you will be asked which application to use. For m and dat files, select a text editor. In Microsoft Windows, NotePad and WordPad are good choices for text editors.

###

M-file name	*Chapter*	*Description*
ad_w_decay.m	B10.1	Finite-element solution to transient advection and dispersion of a substance undergoing decay
adv_disp.m	9.3	Crank-Nicolson solution to advection-dispersion equation
besselfd.m	5.5	Finite difference solution to the Bessel equation
bisect.m	2.5	Bisection routine for finding roots
brusselator.m	B4.1	Function file for the equations for the "brusselator"
co2anal.m	12.4	Fourier analysis of Mauna Loa CO_2 time series
deriv.m	3.4	Difference formulae for derivatives
detrendsurf.m	13.4	Removes a 1st-order trend surface (plane) from a set of 2-D spatial data
diagen.m	10.5	Finite-element solution to one-dimensional diffusion into sediment
euler_ex.m	4.2	Euler solution to first-order ordinary differential equation
explicit.m	8.3	Explicit solution to 1-D groundwater problem
G_puff.m	1.4	Gaussian puff model of contaminant dispersion
geotherm.m	10.6	Finite-element solution to one-dimensional conductive heat flow in the earth; calls the function econd.m
hamon_PET.m	1.4	Hamon calculation of PET in mm per day
hump.m	B2.3	Iterative solution showing complex behavior
implicit.m	8.5	Implicit solution to 1-D groundwater problem
kinsorp.m	9.4	Implicit solution to advection-dispersion equation with kinetic sorption
linreg.m	1.4	Simple linear regression
lpfilt.m	12.7	Simple low-pass, frequency domain filter
main_fe2d.m	11	Finite-element solution for hyporheic zone flow that calls elements.m, gridplot.m, assemble.m and setrhs_solve.m
ode_ex.m	4.6	Solution to ordinary differential equation for settling of a spherical particle in water using *MATLAB* functions ode23 and ode45
ordkrig.m	13.5	Simple ordinary kriging routine
pondex.m	6.6	Example solution of the Poisson equation for groundwater flow
rates.m	4.9	Function file for the equations describing the dehydration of talc
richards.m	8.7	Predictor-corrector solution to the one-dimensional Richards equation for flow in an unsaturated soil; calls the functions dk.m, Kdpsi.m and dKdpsi.m
semivar.m	13.4	Calculates the semivariogram for unequally

		spaced points binned by distance
specclc.m	12.6	Simple power spectral density function
taylor_ex.m	4.2	Example solution of first-order ordinary differential equation using Taylor series method
toth.m	6.4	Solution to the Tóth problem of groundwater flow
well_drawdown.m	B11.1	Finite-element solution for transient drawdown

User's Guide for

Numerical Methods in the Hydrological Sciences

By George Hornberger and Patricia Wiberg

Before Using this E-Book

You may need special plugins to display the text, data, and animation files associated with this E-Text. Some .m and .dat files will not open if your computer settings do not recognize those file extensions. Visit http://www.agu.org/pubs/plugins.html for more information on plugins and viewing dynamic content.

How to Navigate this E-Book

Book Components

Table of Contents:	Detailed list of chapters and indexes
Subject Index:	Directs reader to unique terms within the text
m files Index:	Lists the file name and function of data files
Solutions:	Shows the answers to the questions posed within the text

Inside a Chapter

Within a regular chapter, the reader will find hyperlinks to sub-chapters, figures, and data and video files, as well as pertinent information in other chapters.

System Requirements

Adobe Acrobat 6.0 or higher will provide the best experience. Previous versions will not support all of the features in the PDFs. Download the latest Acrobat version from the Adobe website at: http://www.adobe.com/products/acrobat/readstep2.html

Pages and Navigation

To flip through pages of a PDF is like using a cassette player.

From left to right, the six buttons do the following.

 Go to the first page of the chapter

 Back one page

 Forward one page

 Go to the last page of the chapter

 Back one step (or one link)

 Forward one step (or one link)

Features in E-Book PDFs

Each chapter of this E-Book is a separate PDF.

PDFs combine the utility of a book with the simplicity of a web link to connect ideas.

Text links are indicated by a blue box in E-Book PDFs. To explore the related idea, click the blue box. These links will take you to another page. When done exploring the related idea, click the back step arrow as many times as needed to return to the page where the original text link was.

Figure 1.1

Bookmarks on the left side of the window in the PDFs show the available material. Each chapter's subsections, boxes, figures, videos, and MATLAB .m files are bookmarked for quick access.

Embedded files, figures, videos, MATLAB .m, and data files are represented by blue push pins and can be opened in a pop-up window to be viewed or downloaded separately. Double click a push pin to open the file in a separate pop-up window. Placing your cursor over the push pin will show you the name of the file. Right clicking will allow you to save the file locally on your hard drive.