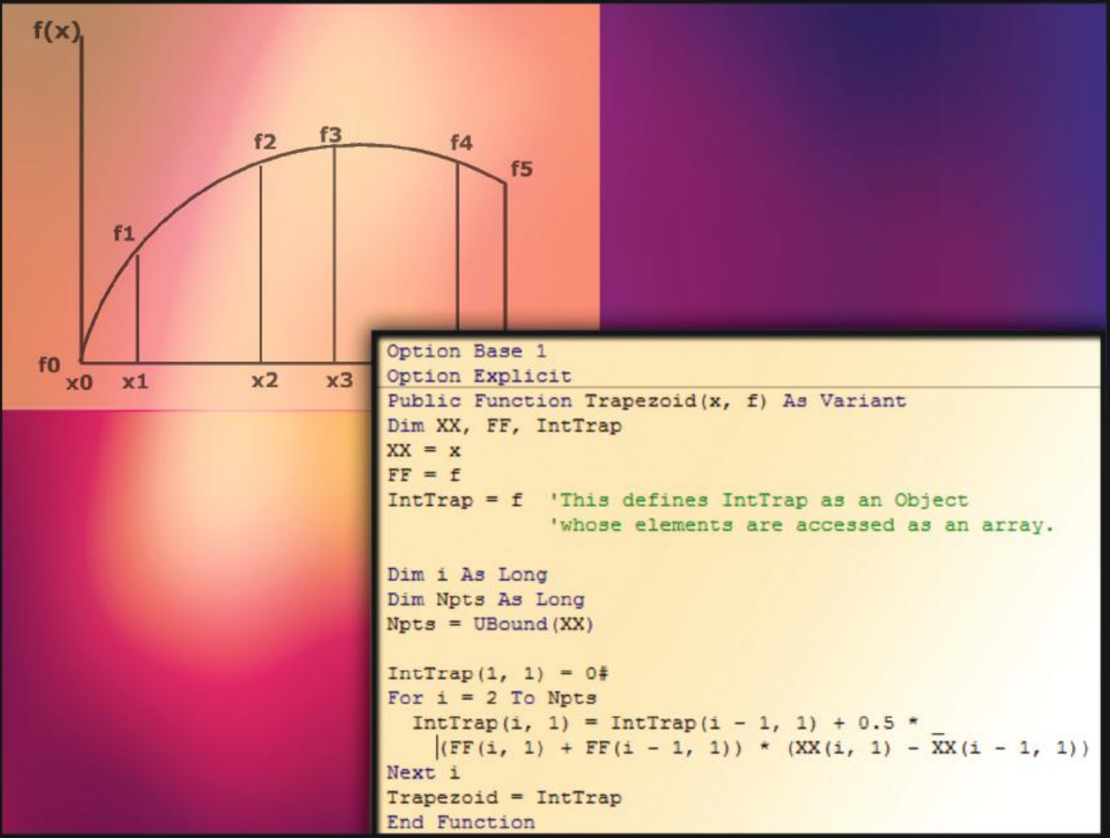
Numerical Methods for Chemical Engineers

Using Excel®, VBA, and MATLAB®



```
Option Base 1
Option Explicit
Public Function Trapezoid(x, f) As Variant
Dim XX, FF, IntTrap
XX = x
FF = f
IntTrap = f  'This defines IntTrap as an Object
             'whose elements are accessed as an array.

Dim i As Long
Dim Npts As Long
Npts = UBound(XX)

IntTrap(1, 1) = 0#
For i = 2 To Npts
  IntTrap(i, 1) = IntTrap(i - 1, 1) + 0.5 * _
    (FF(i, 1) + FF(i - 1, 1)) * (XX(i, 1) - XX(i - 1, 1))
Next i
Trapezoid = IntTrap
End Function
```

Victor J. Law

CRC Press
Taylor & Francis Group

NUMERICAL METHODS for CHEMICAL ENGINEERS

Using Excel®, VBA, and MATLAB®

NUMERICAL METHODS for CHEMICAL ENGINEERS

Using Excel®, VBA, and MATLAB®

VICTOR J. LAW

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

MATLAB® is a trademark of The MathWorks, Inc. and is used with permission. The MathWorks does not warrant the accuracy of the text or exercises in this book. This book's use or discussion of MATLAB® software or related products does not constitute endorsement or sponsorship by The MathWorks of a particular pedagogical approach or particular use of the MATLAB® software.

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

No claim to original U.S. Government works
Version Date: 20130313

International Standard Book Number-13: 978-1-4665-7535-6 (eBook - PDF)

This book contains information obtained from authentic and highly regarded sources. Reasonable efforts have been made to publish reliable data and information, but the author and publisher cannot assume responsibility for the validity of all materials or the consequences of their use. The authors and publishers have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission to publish in this form has not been obtained. If any copyright material has not been acknowledged please write and let us know so we may rectify in any future reprint.

Trademark Notice: Product or corporate names may be trademarks or registered trademarks, and are used only for identification and explanation without intent to infringe.

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

Dedication

To my wonderful family:

Penny (wife)

Preston (elder son)

Sandy (younger son)

Vicki (daughter)

Jo (daughter-in-law)

CeCe (granddaughter)

Blake (grandson)

They make even writing a book worthwhile.

Contents

Preface

This book has been written using notes developed for the course Numerical Methods for Chemical Engineers at Tulane University. The author has written two previous textbooks: one on FORTRAN® programming and one using the language Pascal. On a personal note, when I completed the Pascal book, I asked my wife to *break my fingers* if I ever decided to write another book! Well, that was a long time ago, and having been granted a sabbatical leave to write this book, my wife decided that she would *look the other way*.

While there are many textbooks whose title would indicate that they are suitable for the course Numerical Methods for Chemical Engineers, every one that has been tried has been a failure in one way or another. Either they were too elementary and the applications and problems were not ideal or they did not offer instruction in Excel® and Visual Basic® for Applications (VBA). This led to the development, over a 6-year period, of detailed notes to be used in place of a textbook. These notes have been enhanced and put into textbook form to produce the present book.

The primary reason for using Excel is that it is generally available software, and it comes with every computer system (both PC and Mac) with Microsoft Office® installed. VBA is a programming environment that comes with Excel and greatly enhances the capabilities of basic Excel spreadsheets. It is available on systems running Microsoft operating systems and Mac OS. Beware, however, that VBA is available only on the latest (2011) version of Microsoft Office for the Mac.

Other programming software systems that are often used in chemical and biomolecular engineering numerical methods courses are the following:

- MATLAB®
- Mathematica®
- MathCad®
- C/C++
- FORTRAN
- PolyMath

C/C++ and FORTRAN are compiler-based programming languages. Courses that deal with them must devote large amounts of time to learning the language itself rather than emphasizing problem solving.

The first three examples are programming environments with relatively easy-to-use interfaces. Mathematica and MathCad offer powerful built-in methods for solving many common problem types, and both are particularly suited to symbolic problem solving (such as performing analytical differentiation or integration and solving differential equations). MATLAB is by far the most popular of the "proprietary" packages, and at least two textbooks have been written that combine chemical engineering problem solving with the MATLAB system. A significant difficulty with using MATLAB is that it requires rather expensive licenses. In all likelihood, MATLAB will not be available to practicing engineers in industry. This adds to the attractiveness of using Excel with VBA.

PolyMath is a specialized software package and has its roots in academia. It is especially suitable for a number of chemical engineering applications, and at least one textbook has been written using PolyMath as the base programming tool.

Obviously, there is no panacea when choosing which programming system to use, and any choice will have both backers and detractors. As a compromise, MATLAB is *introduced* in the last chapter of this text. This introduction is sufficient for students to grasp the basics of MATLAB and how it differs from using Excel and VBA. Also, MATLAB programming is easily mastered by those who know VBA.

The vast majority of problems presented in this text, including in-class examples, homework problems, and exam problems, are related to chemical and biomolecular engineering. Application areas include (but are not limited to)

- Material and energy balances
- Thermodynamics
- Fluid flow
- Heat transfer
- Mass transfer
- Reaction kinetics (including biokinetics)
- Reactor design and reaction engineering
- Process design
- Process control

In the course taught by the author, exams (including most of the final exam) are of the "take-home" variety. It is not practical to give a timed, in-class exam when numerical methods and using a computer are involved. In order to encourage individual work, each student is given a unique set of input data so that no two students are expected to get the same "answers."

In the text, when mentioning a topic for which there is neither time nor space to elaborate, the statement "Google it" appears. This is not a plug for any specific search engine but an easy way for the author to suggest getting more information if the reader's interest is sparked. Another feature is the use of "Did You Know" boxes. These are used to remind about features of Excel that are assumed known.

MATLAB® is a registered trademark of The MathWorks, Inc. For product information, please contact:

The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098 USA
Tel: 508-647-7000
Fax: 508-647-7001
E-mail: info@mathworks.com
Web: www.mathworks.com

Author

In the Fall of 2012, Victor J. Law, PhD, FAIC hE, FICH EME, CE, started his 50th year on the faculty of Tulane University. Dr. Law graduated with a BS (ChE) degree from Tulane in 1960, an MS (ChE) degree in 1962, and a PhD degree in 1963. He joined the faculty of the Tulane School of Engineering (Department of Chemical Engineering) on July 1, 1963. Dr. Law's PhD thesis was in the area of automatic process control. Early on, he taught graduate and upper-level undergraduate courses in process control, transport phenomena, applied mathematics, and applied statistics. He began using computers while still an undergraduate, and his research centered on the use of computers for process control and process simulation. Dr. Law worked for several summers at the Monsanto research campus in St. Louis, Missouri, and was peripherally involved in the development of one of the first "process simulators" called FLOWTRAN. While working at Monsanto, Dr. Law became interested in numerical optimization methods. He worked with Monsanto colleague Dr. Robert H. Fariss on general-purpose software for nonlinear equations, nonlinear regression, nonlinear programming, and constrained nonlinear regression. In 1967, Dr. Law was promoted to associate professor (with tenure) and in 1970 to professor.

In 1973, Dr. Law initiated a program that was to become the Department of Computer Science at Tulane. He was the head of that department from 1979 to 1982. During his tenure in computer science, Dr. Law wrote two textbooks on introductory computer programming (one on FORTRAN77 and another on Pascal). In 1988, Dr. Law returned to the chemical engineering department in order to resume his research career. Since returning to chemical engineering, he has taught classes in process control, transport phenomena, process design, engineering statistics, and numerical methods for chemical engineers. His research has included projects in coastal erosion, methane emissions from rice paddies, thermochemical processes for hydrogen production from water, and butanol production from biomass.

Dr. Law is a fellow of the American Institute of Chemical Engineers; a fellow of the Institution of Chemical Engineers; a chartered engineer in the United Kingdom and Europe, No. 20514794; and a registered professional engineer in the State of Louisiana, No. 10961. He is a member of Tau Beta Pi, Sigma Xi, and Omega Chi Epsilon.

Notes to the Instructor

HISTORY OF AND THE REASON FOR EXCEL®/VBA

The material in this book has been developed over a 6-year period while teaching a class entitled Numerical Methods for Chemical Engineers. The author has taught this class (not continually) for the past 15 years. Early on, the computing platform was FORTRAN® running on a mainframe. At that time, students (as freshmen) took a required course in FORTRAN programming. By the time they took this class (as juniors), they needed considerable refreshing in FORTRAN. The course concentrated on linear algebra and the solution of ordinary differential equations. Many of the problems were generic rather than chemical engineering oriented.

When PCs became prevalent, a switch was made to the MATLAB® platform. Considerable time was spent getting students familiar with MATLAB, but the range of problems was greater because of MATLAB's function availability. However, students returning from summer internships complained that MATLAB was not available at their employer's sites. They wanted a tool that they could use in any setting. The result was to settle on Excel and VBA. There is no panacea; other students who went to graduate school came back for reunions and complained that MATLAB, MathCad, and Mathematica were the popular computing tools at the institutions they attended. It was then decided to add some MATLAB training to the Numerical Methods for Chemical Engineers class. In recent years, almost all example and homework problems have been related to chemical and biomolecular engineering.

Mac Users Beware: The most recent version (2011) of MS Office for the Mac *does* include VBA. Students have reported that the Mac version presents no significant differences from the PC version. In Chapter 3, a *free* software package called Matrix.xla is introduced. It offers a host of matrix-based functions not available directly in Excel. While this package can be downloaded to a Mac, the Matrix.xla functions are not available at the Excel level. They can, however, be utilized in VBA mode. If nothing else, these functions allow students to view very well written code in VBA. At some point, it is hoped that the publishers of Matrix.xla will support its features on the Mac.

MATERIAL AVAILABLE FOR INSTRUCTORS

Files with Excel/VBA or MATLAB programs for all of the examples in the book, any concluding comments for those programs, and solutions to all of the end-of-chapter exercises are available on a DVD from the publisher with a qualifying course adoption. Additionally, solutions to all of the end-of-chapter exercises are provided. When possible, concluding comments are included along with the programs.

HOW THE AUTHOR TEACHES THE CLASS

All classes are held in a departmental computer lab. Usual class size is about 20, and each student has his or her own computer. At the beginning of coverage of a new chapter, a short lecture (at the chalk board—no PowerPoint) presents the highlights of the material. Chapter examples are then shown on a projector connected to a PC. Sometimes the lecture/example sequence is repeated when appropriate. Finally, students are given an in-class exercise to perform—usually one of the end-of-chapter exercises. They are given time to attempt the solution on their own; after a while, they are given "helpful hints" as to how to proceed. If they do not complete the exercise within the allotted class time, they are encouraged to finish on their own. A homework assignment is given that is again one of the end-of-chapter exercises. Students are usually given about 1 week to complete a homework assignment. They are expected to turn in Excel/VBA or MATLAB files along with a Word file if needed. The homework assignments are sent via email to a teaching assistant (TA), who grades the work. The instructor usually spends some time with the TA regarding the assignment and how to grade it.

Exams are all of the "take-home" variety. It is not reasonable to give a programming assignment within the time limits of a typical class. In order to discourage *collaboration*, each student is given his or her own set of data for the exam. In addition, an honor code statement is made at the beginning of the exam document. Students are warned that plagiarism on programs is usually very easy to detect and that the spirit of the honor code will be upheld. The instructor grades all exams. Typically, two exams are given during the semester and often consist of about six problems.

The final exam is usually handed out during the second to last week of the class. The due date of the exam is the day that an in-class final exam would have taken place (this is pre-scheduled by the university). So, students usually have about 10–14 days to work on the final exam. Enterprising students can complete the exam well before other final exams begin (students are encouraged to do this). The final exam usually consists of about 10–12 problems. Again, each student is given his or her own set of data required for the exam problems. Since MATLAB is the last subject covered, there are several MATLAB problems on the final exam.

1 Roots of a Single Nonlinear Equation

1.1 INTRODUCTION

Many engineering problems require the solution of a single nonlinear equation. Such an equation can always be cast into the form

$$f(x) = 0 \tag{1.1}$$

The objective of this chapter is to study methods and learn of Excel® tools for finding the root(s) of a nonlinear equation, that is, for finding x such that $f(x) = 0$.

Simple algebra provides the root for a *linear* equation. However, for more complex (nonlinear or transcendental) equations, it is often the case that no analytical solution is available, or is difficult to obtain, so that numerical methods must be used.

An example of a nonlinear equation is the van der Waals equation of state, which is given by

$$\left(P + \frac{a}{V^2}\right)(V - b) = RT \tag{1.2}$$

where

$$a = \frac{27}{64}\left(\frac{R^2 T_c^2}{P_c}\right)$$

$$b = \frac{RT_c}{8P_c}$$

with subscript c referring to the "critical" values of temperature and pressure for the gas. Note that, if a and b are zero, this reduces to the ideal gas equation of state.

A typical problem is to find the molar volume, V, given the temperature and pressure (and the type of gas). While it is possible to find analytic solutions to the van der Waals equation of state for V (this is simply a cubic equation), a numerical solution is often preferred. The equation of state in the form $f(x) = 0$ is obtained by simple rearrangement:

$$f(V) = \left(P + \frac{a}{V^2}\right)(V - b) - RT = 0 \tag{1.3}$$

Note that this particular form $f(x) = 0$ is not unique, and other algebraic rearrangements are possible.

1.2 ALGORITHMS FOR SOLVING $f(x) = 0$

Clearly there is a need to find good methods for determining roots. Four such methods are now presented (though *many* more have been developed): fixed-point iteration (direct substitution), bisection, Newton's method, and the secant method. These methods are *iterative*. That is, given a *guess* of a root, or the interval in which a root lies, the algorithm refines that guess repeatedly, obtaining (hopefully) better and better guesses, until a value "close enough" to the true root is found. Also, graphing the equation can add insight into the roots of interest and can provide good initial estimates of the roots for the iterative algorithms. It is recommended to always prepare a graph of the function to give insight into possible solutions.

1.2.1 Plotting the Equation

Excel has very good plotting capabilities. Unfortunately, it is not possible in Excel to simply give a command such as plot($f(x)$). It is necessary to produce a list or table of x and $f(x)$ values and to graph the resulting data. This is best illustrated by an example.

Example 1.1: Plotting the Equation

A table of data and an Excel graph of the van der Waals equation for ammonia at 250°C and 10 atm are shown in Figure 1.1.

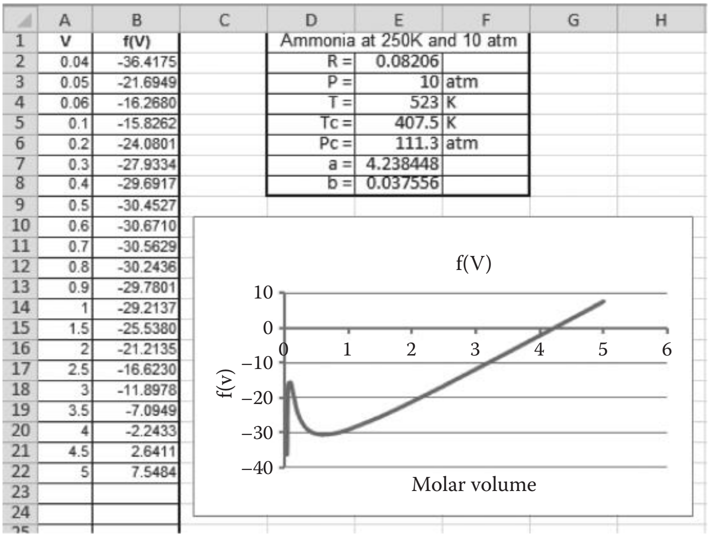
From the graph, it is easy to see that there is one real root between 0 and 5 (the other roots are complex conjugates). Remember that the van der Walls equation of state is an attempt to model the *nonideality* of the gas. Since the given temperature and pressure are not severe, it is expected that the calculated molar volume from the equation of state should not be greatly different from that predicted by the ideal gas law. From the ideal gas law, the molar volume is 4.29 L/gmol, which is very close to the root shown in the graph.

1.2.2 Fixed-Point Iteration (Direct Substitution)

To apply this method, the equation must be cast into the form

$$x = g(x)$$

or, more generally,



	A	B	C	D	E	F	G	H
1	V	f(V)		Ammonia at 250K and 10 atm				
2	0.04	-36.4175		R =	0.08206			
3	0.05	-21.6949		P =	10	atm		
4	0.06	-16.2680		T =	523	K		
5	0.1	-15.8262		Tc =	407.5	K		
6	0.2	-24.0801		Pc =	111.3	atm		
7	0.3	-27.9334		a =	4.238448			
8	0.4	-29.6917		b =	0.037556			
9	0.5	-30.4527						
10	0.6	-30.6710						
11	0.7	-30.5629						
12	0.8	-30.2436						
13	0.9	-29.7801						
14	1	-29.2137						
15	1.5	-25.5380						
16	2	-21.2135						
17	2.5	-16.6230						
18	3	-11.8978						
19	3.5	-7.0949						
20	4	-2.2433						
21	4.5	2.6411						
22	5	7.5484						
23								
24								

FIGURE 1.1 Roots of the van der Walls equation for ammonia at 250°C and 10 atm.

$$x^{k+1} = g(x^k) \tag{1.4}$$

where k is an iteration counter.

Example 1.2: Direct Substitution

Note that this can always be accomplished by adding x to each side of $f(x) = 0$, if necessary. The van der Walls equation can be cast into the following form:

$$V = b + \frac{RT}{\left(P + \dfrac{a}{V^2}\right)} \tag{1.5}$$

The pertinent data for ammonia are shown in Figure 1.1. If a value for $V = 4$ is guessed (based on the graph of Figure 1.1) and is used on the right-hand side, a new (and hopefully better) value is calculated from Equation 1.5. The iterations produce the following sequence:

```
4.00000 4.21854 4.22946 4.22996 4.22998.
```

The solution is `4.22998` L/gmol. If more significant digits are required, then more iterations can be carried out.

It should be noted that direct substitution can be a *divergent* process. That is, the successively calculated values actually get worse rather than closer to the correct value. Without proof, the following statements apply:

Let g' be the first derivative of the function g in Equation 1.4. Then,

- If $|g'| < 1$, the error will decrease with each iteration.
- If $|g'| > 1$, the error grows at each iteration.
- If $g' > 0$, the error will have the same sign at each iteration.
- If $g' < 0$, the error will alternate signs at each iteration.

Clearly, the equation should be arranged so that the magnitude of g' is less than 1. This might take some experimentation. Often, a form of g is tried, and if the process does not converge, then other forms are attempted.

1.2.3 Bisection

If it is (somehow) known that a root lies in the interval $[a, b]$, then by simply halving the interval in which the root lies, the interval can be reduced to an acceptable level. This idea is at the heart of the bisection method as shown in Figure 1.2.

The restriction is that $f(a)$ and $f(b)$ *must* have opposite signs—one of them must be positive, the other negative (it does not matter which). Then, because f is assumed to be continuous, it must be a zero somewhere in $[a, b]$. Let c be the midpoint of $[a, b]$. Either c is the root, or the root lies in $[a, c]$ or in $[a, b]$. If $f(c)$ is close enough to zero (see below regarding tolerance), then the root has been found. Otherwise, one pair of $[f(a),f(c)]$or $[f(c),f(b)]$ has opposite signs. Keep the half-interval with opposite signs and discard the other. Repeat the process until either (1) f, evaluated at the midpoint of the interval, is sufficiently small or (2) the interval has been shrunk to a suitably small value.

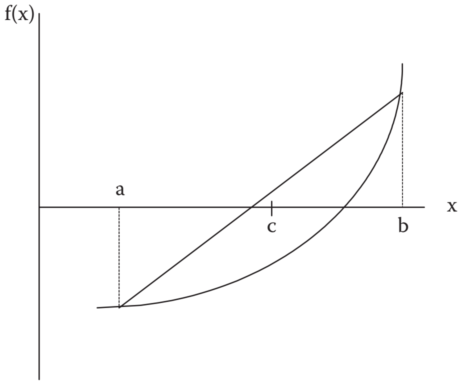


FIGURE 1.2 Bisection: $c = (a + b)/2$.

The term “sufficiently small” is usually tested using a “tolerance,” which represents a number small enough to be considered zero based on the application. This can be stated more formally as follows:

If |f(x)|<tolerance then (x) is sufficiently small.

Example 1.3: Bisection Applied to the van der Waals EOS

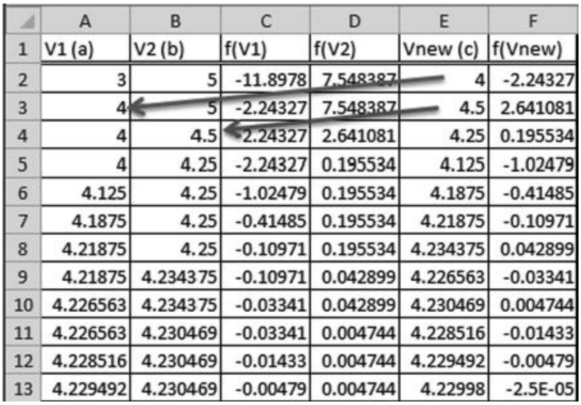
Recall the van der Waals EOS in the form

$$f(V)=\left(P+\frac{a}{V^2}\right)(V-b)-RT=0 \tag{1.6}$$

The table below shows the progression of applying the bisection method. It uses the Excel IF function to choose whether f(a) or f(b) is replaced by f(c) (and correspondingly whether a or b is to be replaced by c). The syntax for the IF function is as follows (see the explanation of the Excel spreadsheet below for a full explanation of how the IF function is used):

```
IF(test, True Value, False Value)
```

In this example (again for ammonia at 10 atm and 250°C), the function in the cell below the label V1 (corresponding to the point a in the nomenclature of the figure) is = IF(F2<0, E2, A2), where cell F3 holds f(Vnew) [corresponding to f(c)], E2 contains Vnew (corresponding to c), and A2 holds V1. So, if f(Vnew) is negative (test is TRUE), V1 (corresponding to a) gets the Vnew value; otherwise, it retains the old V1. Likewise, the formula in the cell below the V2 label is = IF(F2<0,B2,E2), which replaces V2 (corresponding to b) with the value Vnew if the test (F2 < 0) is FALSE. After the first row of formulas has been entered, these cells are copied down to repeat the iterations. Iterations should be repeated until f(Vnew) is sufficiently small (in absolute value). The initial guesses for V are 3 and 5, respectively, which give opposite signs for the function.



	A	B	C	D	E	F
1	V1 (a)	V2 (b)	f(V1)	f(V2)	Vnew (c)	f(Vnew)
2	3	5	-11.8978	7.548387	4	-2.24327
3	4	5	-2.24327	7.548387	4.5	2.641081
4	4	4.5	-2.24327	2.641081	4.25	0.195534
5	4	4.25	-2.24327	0.195534	4.125	-1.02479
6	4.125	4.25	-1.02479	0.195534	4.1875	-0.41485
7	4.1875	4.25	-0.41485	0.195534	4.21875	-0.10971
8	4.21875	4.25	-0.10971	0.195534	4.234375	0.042899
9	4.21875	4.234375	-0.10971	0.042899	4.226563	-0.03341
10	4.226563	4.234375	-0.03341	0.042899	4.230469	0.004744
11	4.226563	4.230469	-0.03341	0.004744	4.228516	-0.01433
12	4.228516	4.230469	-0.01433	0.004744	4.229492	-0.00479
13	4.229492	4.230469	-0.00479	0.004744	4.22998	-2.5E-05

Did You Know?: That there are two ways to copy cell contents down. One way is to select the cells to be copied and then grab the small box in the lower right corner of the selection. Pulling down will copy the cells. The second method is to select the cells to be copied and while holding down the mouse button pull down as far as desired and finally hitting CTRL/d (hold down the CTRL key and hit the d key).

1.2.4 Newton's Method

The next algorithm to be considered is the Newton's method for finding roots. Newton's method does not always converge. But, when it does converge, it usually does so very rapidly (at least once it is "close enough" to the root). Newton's method also has the advantage of not requiring a bracketing interval with positive and negative values. So Newton's method allows the solution of equations such as

$$x^2 - 2x + 1 = 0 \tag{1.7}$$

whereas bisection does not. This parabola just *touches* the zero axis and has no negative values.

The basic idea of the Newton algorithm is this: given an initial guess, call it x^1 to a root of $f(x) = 0$, a refined guess, x^2, is computed based on the x-intercept of the line tangent to $f(x)$ at x^1. That is, consider the equation of the line tangent to $f(x)$ at x^1 (this is just the Taylor series expansion of the function ignoring all but linear terms):

$$f(x) = f(x^1) + f'(x^1)(x - x^1) \tag{1.8}$$

This is the point-slope equation of a line, where x^1 is the base point and $f'(x^1)$ is the slope [derivative of $f(x)$ evaluated at x^1]. Solving Equation 1.8 for x at which $f(x) = 0$ gives

$$0 = f(x^1) + f'(x^1)(x^2 - x^1)$$

$$x^2 = x^1 - \frac{f(x^1)}{f'(x^1)}$$

or more generally,

$$x^{k+1} = x^k - \frac{f(x^k)}{f'(x^k)} \tag{1.9}$$

The value of x^{k+1} is the new guess at the root. The process is repeated, computing successively x^2, x^3, x^4,… until an x^K is found at which

$$|f(x^K)| < tol \tag{1.10}$$

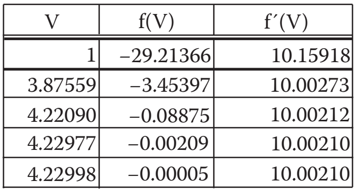
where *tol* is a prescribed tolerance.

Example 1.4: Newton's Method Applied to the van der Waals Equation

Newton's method is now applied for ammonia at 10 atm and 250°C with an initial guess for $V = 1$ L/gmol. To begin using Newton's method, the derivative of Equation 1.2 is required and is as follows:

$$f'(V) = \left(P + \frac{a}{V^2}\right) + (V - b)\left(\frac{-a}{V^3}\right) \tag{1.11}$$

Here is an Excel spreadsheet that solves this problem:



V	f(V)	f′(V)
1	−29.21366	10.15918
3.87559	−3.45397	10.00273
4.22090	−0.08875	10.00212
4.22977	−0.00209	10.00210
4.22998	−0.00005	10.00210

The root is found in about 3 iterations. Note the rapid rate of convergence.

An examination of Equation 1.9 for the new iterate x^k reveals a potential for failure of Newton's method, namely,

$$f'(x^k) \approx 0 \tag{1.12}$$

This may lead to a wildly divergent iterative process. There are other possible reasons why this method might not converge. In general, Newton's method is prone to failure, but when it does work, it converges rapidly. Good initial guesses are the key to success.

1.2.5 Secant Method

In Chapter 4, methods for approximating the derivative of a function using finite differences are presented. The secant method uses the idea of finite differences to approximate the derivative in the Newton method formula. Starting with *two* initial guesses x^0 and x^1, *which **need not** bracket the root of interest*, the approximation to $f'(x)$ can be written as follows:

$$f'(x^1) \cong \frac{f(x^1) - f(x^0)}{x^1 - x^0} \tag{1.13}$$

Or, in general, after k steps or iterations,

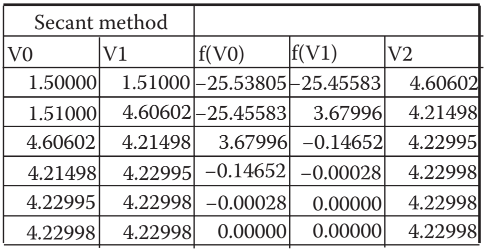
$$f'(x^k) \cong \frac{f(x^k) - f(x^{k-1})}{x^k - x^{k-1}} \tag{1.14}$$

Substituting this approximation into the Newton formula (Equation 1.9), the following iteration formula results for the secant method:

$$x^{k+1} = x^k - \frac{f(x^k)}{f(x^k) - f(x^{k-1})}(x^k - x^{k-1}) \tag{1.15}$$

Example 1.5: The Secant Method for van der Waals Equation

The following shows the van der Waals example solved using the secant method. The two initial guesses for *V* are 1.50 and 1.51—the second was chosen arbitrarily close to the first one. Note that the rate of convergence is about the same as that of Newton's method. Beware that the secant method is subject to the same potential shortcomings of Newton's method.



Secant method				
V0	V1	f(V0)	f(V1)	V2
1.50000	1.51000	−25.53805	−25.45583	4.60602
1.51000	4.60602	−25.45583	3.67996	4.21498
4.60602	4.21498	3.67996	−0.14652	4.22995
4.21498	4.22995	−0.14652	−0.00028	4.22998
4.22995	4.22998	−0.00028	0.00000	4.22998
4.22998	4.22998	0.00000	0.00000	4.22998

1.3 USING EXCEL® TO SOLVE NONLINEAR EQUATIONS (GOAL SEEK)

Any of the methods discussed previously can be implemented quite easily using Excel. However, Excel has *built into it* two tools for solving nonlinear equations, Goal Seek and Solver. Solver is discussed at length in Chapter 9. The *Goal Seeking* feature can solve many single nonlinear equations (good initial guesses are important). Goal Seek uses whatever value is placed in the "By Changing Cell" location as an initial guess.

According to Microsoft® documentation on Goal Seek,

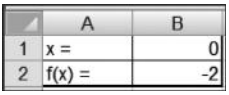
> The Goal Seek command uses a simple linear search beginning with guesses on the positive or negative side of the value in the source cell (**By Changing Cell**). Excel uses the initial guesses and recalculates the formula. Whichever guess brings the formula result closer to the targeted result (**To Value**) is the direction (positive or negative) in which Goal Seek heads. If neither direction appears to approach the target value, Goal Seek makes additional guesses that are further away from the source cell. After the direction is determined, Goal Seek uses an iterative process in which the source cell is incremented or decremented at varying rates until the target value is reached. (http://esupport.lenovo.com/mss/mss.pl?doctype=kb&docid=MTAwNzgy)

Therefore, the algorithm used by Goal Seek is somewhat similar to the secant method with some enhancements. If Goal Seek fails, it is usually due to a poor initial guess, so changing the guess might lead to success. To see how Goal Seek works, consider the following simple example:

$$x - x^{1/3} - 2 = 0$$

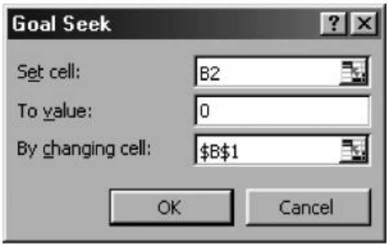
In the following spreadsheet, cell `B1` contains the *initial guess* for the independent variable, x. When a correct value is found for x, then the function $f(x) = 0$ [the formula for $f(x)$ is in cell `B2` and is `= B1-B1^(1/3) - 2`].

Initial spreadsheet:



	A	B
1	x =	0
2	f(x) =	-2

The following is the "Goal Seek" window obtained from the `Data/What If Analysis/Goal Seek` Menu.



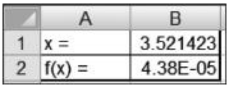
Goal Seek

Set cell: B2

To value: 0

By changing cell: $B$1

After hitting the OK button on the Goal Seek window, the spreadsheet changes to the following:



	A	B
1	x =	3.521423
2	f(x) =	4.38E-05

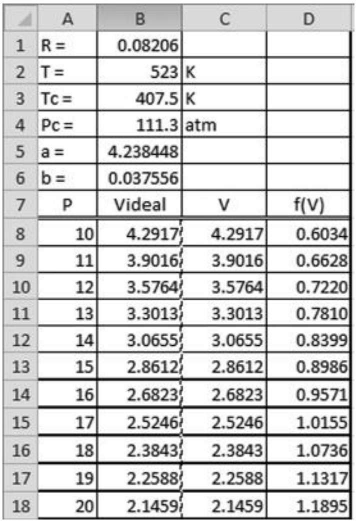
The function $f(x)$ is "close" to zero and the solution is shown for x.

Did You Know?: That the F4 key can be used to toggle between the four cell referencing methods. When a $ sign appears before a row or column indicator, the reference is absolute (if the cell contents are copied, the reference with the $ sign is unchanged). If there is no $ sign and a cell's contents are copied to another cell, the reference is relative and changes accordingly. The four cell references are (for cell B3, for example) B3, $B3, B$3, and $B$3.

Example 1.6: Automating Goal Seek

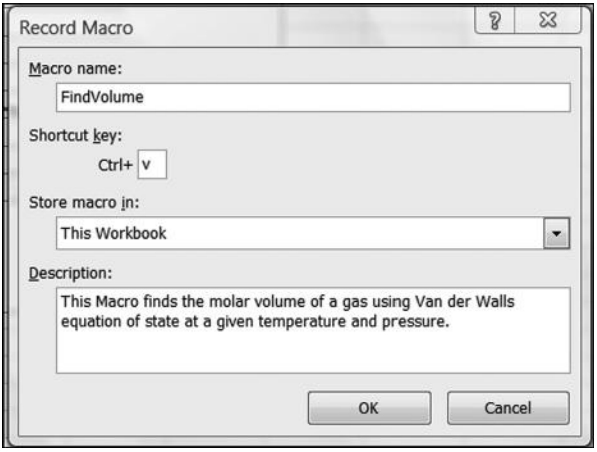
This example gives a very first look at Visual Basic® for Applications (VBA), which is the programming language associated with Excel (as well as all other Microsoft Office applications). The example also introduces the use of Keystroke Macros, which allow the recording of a sequence of keystroke and mouse commands that are recorded for repeated use.

The following spreadsheet applies to the same problem as in previous examples of this chapter. It is desired to compute the molar volume of ammonia at 250°C, but now for a *range* of pressure from 10 to 20 atm (in increments of 1 atm). To do this, Goal Seek can be manually applied for each pressure, but a much more efficient way is shown that involves first recording a Keystroke Macro and then *Editing* the resulting VBA program that was automatically produced by the Macro recorder. When invoking the edited VBA program, the molar volume is found at each pressure automatically. In the spreadsheet, the molar volume from the ideal gas law appears in the second column and is copied to the third column to give a good initial guess of the nonideal volume.



	A	B	C	D
1	R =	0.08206		
2	T =	523	K	
3	Tc =	407.5	K	
4	Pc =	111.3	atm	
5	a =	4.238448		
6	b =	0.037556		
7	P	Videal	V	f(V)
8	10	4.2917	4.2917	0.6034
9	11	3.9016	3.9016	0.6628
10	12	3.5764	3.5764	0.7220
11	13	3.3013	3.3013	0.7810
12	14	3.0655	3.0655	0.8399
13	15	2.8612	2.8612	0.8986
14	16	2.6823	2.6823	0.9571
15	17	2.5246	2.5246	1.0155
16	18	2.3843	2.3843	1.0736
17	19	2.2588	2.2588	1.1317
18	20	2.1459	2.1459	1.1895

The first step is to record a *Keystroke Macro* to find the molar volume (`V`) at 5 atm. Begin by going to the `Developer` tab and choosing (in the upper left set of menus) `Use Relative Reference`. This is necessary so that the recorded Macro works from any proper initial cell. Place the cursor on the `f(V)`value when `P = 10` (the current value is `0.6034`). Next choose `Record Macro` and the following window appears. The name of the Macro (`FindVolume`) and the Shortcut Key (`v`) were chosen to be appropriate for the problem.



Record Macro

Macro name: FindVolume

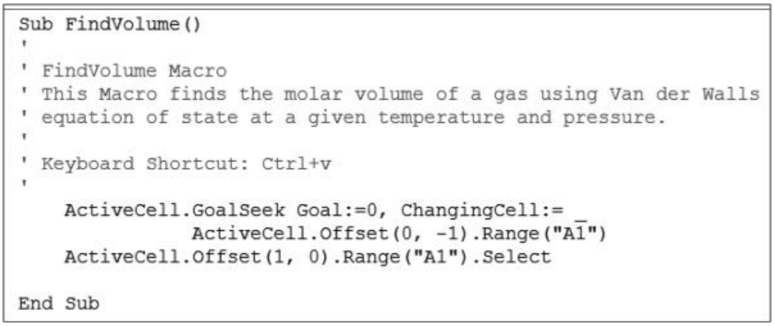
Shortcut key: Ctrl+ v

Store macro in: This Workbook

Description: This Macro finds the molar volume of a gas using Van der Walls equation of state at a given temperature and pressure.

After clicking the OK button, invoke the `Data/What-If-Analysis/Goal Seek` menu and select the appropriate cells so that the volume for the first pressure is calculated. Go back to the `Developer` menu and select `Stop Recording`. The recorded Macro could now be used manually at each pressure by placing the cursor over the next f(V) and hitting Ctrl/v (hold down Ctrl and hit v). However, the Macro can be Edited to *make it work repeatedly* until the data are exhausted.

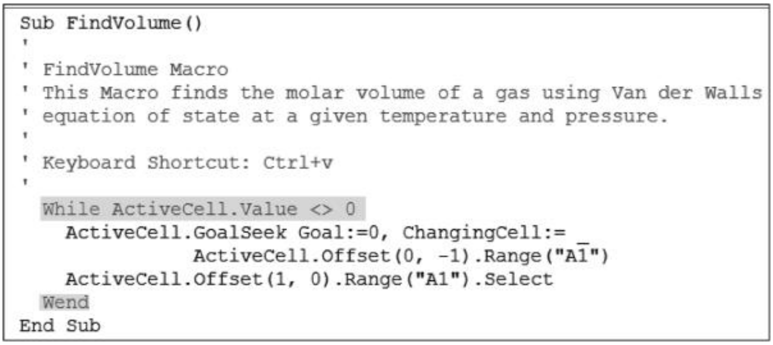
By choosing `Developer/Macros/FindVolume/Edit`, the following VBA code appears:



```
Sub FindVolume()
'
' FindVolume Macro
' This Macro finds the molar volume of a gas using Van der Walls
' equation of state at a given temperature and pressure.
'
' Keyboard Shortcut: Ctrl+v
'
    ActiveCell.GoalSeek Goal:=0, ChangingCell:= _
               ActiveCell.Offset(0, -1).Range("A1")
    ActiveCell.Offset(1, 0).Range("A1").Select

End Sub
```

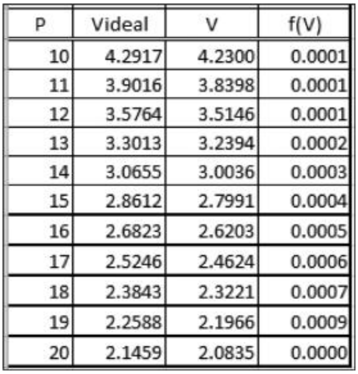
Note that this code was generated *automatically* during the recording of the Keystroke Macro. The code has been edited somewhat to fit properly on the printed page without altering its accuracy. The lines beginning with apostrophe (`'`) are comments and can be ignored. The first executable line begins with `ActiveCell.GoalSeek`, which invokes the Goal Seek algorithm. The remainder of the line specifies a goal value of `0` and that the "changing cell" is on the same row, one column to the left `(0,-1)`. The underscore at the very end of the first line is a "continuation" marker, which states that information on the next line is part of the current line (but there was no room for it). The second line that involves the word `Offset` is a command to move the cursor down one row but in the same column `(1,0)`.

This code can be altered so that, when invoked, it repeats over and over until the next item in the *f*(*x*) column is blank (which also represents a value of 0). The edited code appears below, where the additional code is shaded for emphasis:



```
Sub FindVolume()
'
' FindVolume Macro
' This Macro finds the molar volume of a gas using Van der Walls
' equation of state at a given temperature and pressure.
'
' Keyboard Shortcut: Ctrl+v
'
  While ActiveCell.Value <> 0
    ActiveCell.GoalSeek Goal:=0, ChangingCell:= _
              ActiveCell.Offset(0, -1).Range("A1")
    ActiveCell.Offset(1, 0).Range("A1").Select
  Wend
End Sub
```

Details concerning the VBA language are covered in Chapter 2. The statement `While ActiveCell.Value <> 0` is a looping command that says, in effect, perform all statements below this until the `Wend` is encountered as long as the value in the `ActiveCell` (the one where the cursor is) is not zero (or not blank). When this revised Macro is invoked by typing Ctrl/v, the spreadsheet changes to that shown below. As expected, the difference between ideal and non-ideal volume becomes larger as pressure increases.



P	Videal	V	f(V)
10	4.2917	4.2300	0.0001
11	3.9016	3.8398	0.0001
12	3.5764	3.5146	0.0001
13	3.3013	3.2394	0.0002
14	3.0655	3.0036	0.0003
15	2.8612	2.7991	0.0004
16	2.6823	2.6203	0.0005
17	2.5246	2.4624	0.0006
18	2.3843	2.3221	0.0007
19	2.2588	2.1966	0.0009
20	2.1459	2.0835	0.0000

The graph shown in Figure 1.3 illustrates the difference between the molar volumes for ammonia at 250°C computed by the ideal gas law and by the van der Waals equation of state.

Example 1.7: Fraction Vaporized of a Hydrocarbon Mixture

Figure 1.4 gives data for a mixture of four hydrocarbons. Included in the data are constants (*A*, *B*, and *C*) for the Antoine equation, a correlation that allows the calculation of the vapor pressure of pure components as follows:

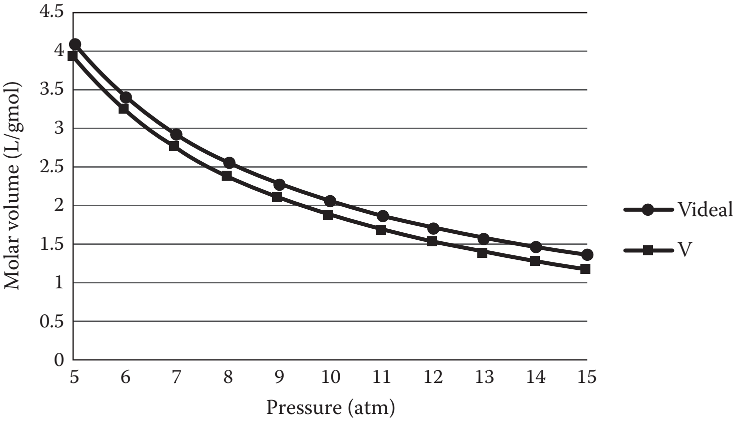


FIGURE 1.3 Comparison of ideal gas and van der Waals molar volume.

	Ethylene	Ethane	Propane	n-Butane
A	3.86690	3.93264	3.97721	3.84431
B	584.146	659.739	819.296	909.65
C	−18.307	−16.719	−24.417	−36.146
z	0.10	0.30	0.40	0.20

FIGURE 1.4 Feed mole fraction and antoine coefficients. *z* is the mole fraction of each component in the feed. Antoine constants are from the NIST Chemistry Database.

$$\log_{10} P_i^* = A_i - \frac{B_i}{C_i + T} \tag{1.16}$$

where

P_i^* = vapor pressure of component *i* (atm)
T = temperature (K)
A_i, B_i, C_i = Antoine coefficients

It is desired to find the fraction of the mixture in the vapor phase when the mixture is flashed at 60°C over a pressure range of 18, 19, …, 40 atm. A plot of the fraction vaporized versus pressure is also desired.

Consider the schematic of a flash tank as shown in Figure 1.5.

F is the molar feed rate, and *z* is the mole fraction of each component in the feed; *V* is the molar vapor rate, and *y* is the mole fraction of each component in the vapor; *L* is the molar liquid rate, and *x* is the mole fraction of each component in the liquid.

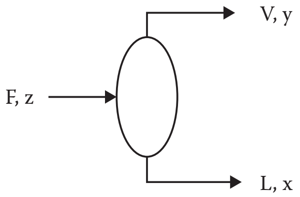


FIGURE 1.5 Schematic diagram of a flash tank.

Assuming an ideal mixture (where Raoult's law applies), the following equilibrium expression applies:

$$k_i = \frac{y_i}{x_i} = \frac{P_i^*}{P}; i = 1, 2, \ldots, n_c \tag{1.17}$$

where k_i is called the equilibrium constant for component i, P is the total pressure, and n_c is the number of components.

The material balance equations for the flash tank can then be written as follows:

$$\begin{aligned} &z_iF = x_iL + y_iV; \quad i = 1, 2, \cdots, n_c \\ &F = L + V \\ &\sum_{i=1}^{n_c} x_i = \sum_{i=1}^{n_c} y_i = 1 \end{aligned} \tag{1.18}$$

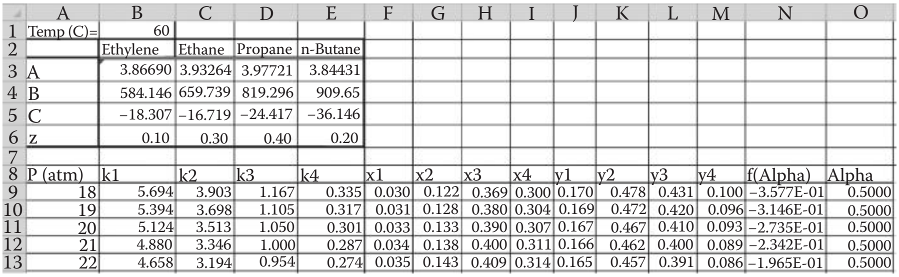
Let α = V/F, the fraction of the feed flashed to the vapor phase. These equations can be manipulated (see Henley and Rosen, p. 341) into the following single nonlinear equation in α:

$$f(\alpha) = \sum_{i=1}^{n_c} (x_i - y_i) = \sum_{i=1}^{n_c} \frac{z_i(1 - k_i)}{1 + \alpha(k_i - 1)} \tag{1.19}$$

Once α has been determined, the liquid and vapor mole fractions can be found from

$$\begin{aligned} &x_i = \frac{z_i}{1 + \alpha(k_i - 1)} \\ &y_i = k_ix_i \end{aligned} \tag{1.20}$$

The following Excel spreadsheet shows the setup for $T = 60°\text{C}$ and $P = 18, 19, \ldots, 22$ atm. Initial guesses for Alpha are entered as 0.5, which is a reasonable first estimate since the result must be between 0 and 1. The first of Equation 1.19 is used for `f(Alpha)`.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	Temp (C)=	60													
2		Ethylene	Ethane	Propane	n-Butane										
3	A	3.86690	3.93264	3.97721	3.84431										
4	B	584.146	659.739	819.296	909.65										
5	C	−18.307	−16.719	−24.417	−36.146										
6	z	0.10	0.30	0.40	0.20										
7															
8	P (atm)	k1	k2	k3	k4	x1	x2	x3	x4	y1	y2	y3	y4	f(Alpha)	Alpha
9	18	5.694	3.903	1.167	0.335	0.030	0.122	0.369	0.300	0.170	0.478	0.431	0.100	−3.577E-01	0.5000
10	19	5.394	3.698	1.105	0.317	0.031	0.128	0.380	0.304	0.169	0.472	0.420	0.096	−3.146E-01	0.5000
11	20	5.124	3.513	1.050	0.301	0.033	0.133	0.390	0.307	0.167	0.467	0.410	0.093	−2.735E-01	0.5000
12	21	4.880	3.346	1.000	0.287	0.034	0.138	0.400	0.311	0.166	0.462	0.400	0.089	−2.342E-01	0.5000
13	22	4.658	3.194	0.954	0.274	0.035	0.143	0.409	0.314	0.165	0.457	0.391	0.086	−1.965E-01	0.5000

Goal Seek can be used to find α at all of the pressures individually, or a Keystroke Macro can be recorded and edited as in Example 1.6. The result of having done this is shown in the next spreadsheet, again for pressures of 18, 19, …, 22 atm. The graph in Figure 1.6 summarizes the final result for all pressures in the range 18–40 atm.

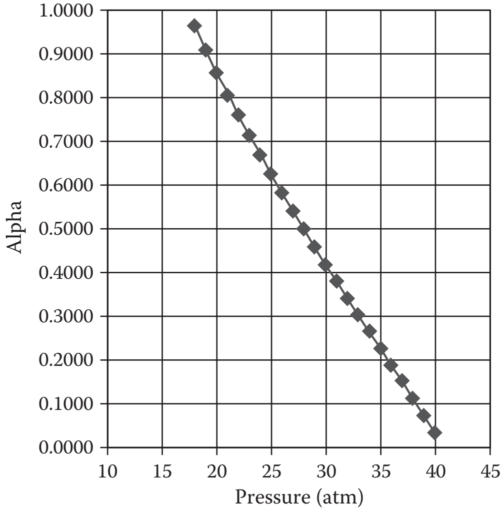
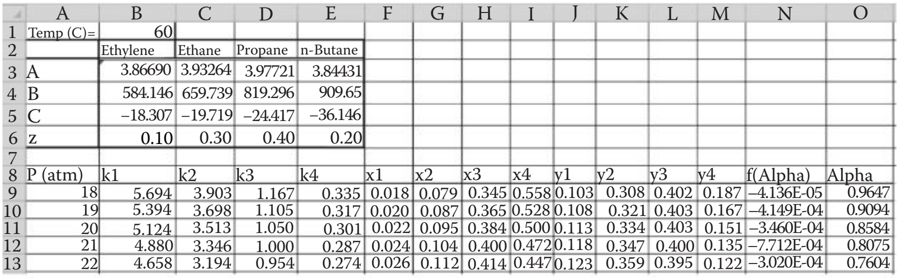


FIGURE 1.6 Results for Example 1.7. Fraction vaporized for a hydrocarbon mixture.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	Temp (C)=	60													
2		Ethylene	Ethane	Propane	n-Butane										
3	A	3.86690	3.93264	3.97721	3.84431										
4	B	584.146	659.739	819.296	909.65										
5	C	−18.307	−19.719	−24.417	−36.146										
6	z	0.10	0.30	0.40	0.20										
7															
8	P (atm)	k1	k2	k3	k4	x1	x2	x3	x4	y1	y2	y3	y4	f(Alpha)	Alpha
9	18	5.694	3.903	1.167	0.335	0.018	0.079	0.345	0.558	0.103	0.308	0.402	0.187	−4.136E-05	0.9647
10	19	5.394	3.698	1.105	0.317	0.020	0.087	0.365	0.528	0.108	0.321	0.403	0.167	−4.149E-04	0.9094
11	20	5.124	3.513	1.050	0.301	0.022	0.095	0.384	0.500	0.113	0.334	0.403	0.151	−3.460E-04	0.8584
12	21	4.880	3.346	1.000	0.287	0.024	0.104	0.400	0.472	0.118	0.347	0.400	0.135	−7.712E-04	0.8075
13	22	4.658	3.194	0.954	0.274	0.026	0.112	0.414	0.447	0.123	0.359	0.395	0.122	−3.020E-04	0.7604

1.4 A NOTE ON IN-CELL ITERATION

Suppose that in a spreadsheet the following formula is entered into cell `B1`:

```
= B1^(1/3)+2
```

Since cell reference `B1` is in the formula of the same cell, this is called a "circular reference." Normally, Excel will complain about this. However, if the *Enable Iterative Calculations* (see `File/Options/Formulas`) box is checked, Excel will immediately do the following:

1. It will automatically use an *initial guess of zero* (there is no control over this).
2. It will iterate up to 100 times or until successive values are within 0.001 (these values can be changed).
3. The final result will be the answer (hopefully).

In-cell iteration is not to be encouraged when robustness is a goal. It often fails to find a solution.

EXERCISES

Exercise 1.1: For the following functions, graph the function and then use Goal Seek to find the root(s).

a. $f(x) = x - x^{1/3} - 2$
b. $f(x) = x\tan x - 1$
c. $f(x) = x^4 - e^x + 1$
d. $f(x) = x^2 e^x - 1$

Exercise 1.2: Find the roots of the functions given in Exercise 1.1 using the bisection method. Use the graph of each function to choose points that bracket the root of interest.

Exercise 1.3: Set up a spreadsheet that implements the secant method and then solve each of the problems from Exercise 1.1. Use the graph of each function to select an initial guess. Recall the iteration formula for the secant method:

$$x^{k+1} = x^k - \frac{f(x^k)}{f(x^k) - f(x^{k-1})}(x^k - x^{k-1})$$

Hint: Set up the first row of the spreadsheet for your problem with headings such as the following:

xk−1	xk	f (xk−1)	f (xk)	xk+1

Put the formula for the function under the headings `f(xk-1)` and `f(xk)`. In the cell under `xk+1`, put the secant method iteration formula. In the second row, replace the previous `xk-1` with `xk` and then `xk` with `xk+1`. Now copy the two formulas down one row. At this point, one iteration of the secant method is displayed. To see more iterations, just copy the second row down for as many iterations as desired. If too many iterations are copied and the function difference (the denominator of the iteration formula) becomes exactly zero, a "divide by zero" error will appear.

Exercise 1.4: Use Goal Seek to find root(s) of the following functions. Plot the functions first to obtain an approximation of a desired root.

a. $f(x) = x^3 - 17x + 12 = 0$

b. $J_1(x) = 0$ J_1 is the Bessel function of the first kind of order 1. It can be computed in Excel using the `BESSELJ()`function. This function has an infinite number of roots; find the root between 2 and 5.

c. Solve for the molar volume of a gas at 400 K and 1200 kPa using the van der Waals equation of state. The critical temperature and pressure are 500 K and 80 atm, respectively. Use the ideal gas solution for your initial guess.

d. Solve the Colebrook equation for the Darcy friction factor, f, for a Reynolds number (N_{Re}) of 10^5 and a roughness factor, ε/D, of 10^{-4} (this equation holds for Reynolds numbers > 4000):

$$\sqrt{\frac{1}{f}} + 0.86\ln\left(\frac{\varepsilon/D}{3.7} + \frac{2.51}{N_{\text{Re}}\sqrt{f}}\right) = 0$$

e. Repeat part d using the same roughness factor, but for a range of Reynolds numbers from 5000 to 30,000 (pick a reasonable increment). Plot the results (friction factor versus Reynolds number). Automate Goal Seek for this.

Exercise 1.5: Use the secant method as described in Exercise 1.3 to find the root(s) of the functions given in Exercise 1.4. Carefully choose the two initial guesses so that the function values have opposite signs. The roots found may or may not correspond to those found using Goal Seek in Exercise 1.3—it depends on the initial guesses.

Exercise 1.6: Solve the problems of Exercise 1.1 using the Newton method.

Exercise 1.7: Repeat Exercise 1.4 parts a and b using the Newton method. The derivative of $J_1(x)$ is given by

$$J_1'(x) = J_0(x) - \frac{1}{x} J_1(x)$$

Exercise 1.8: Repeat Exercise 1.4 using the bisection method.

Exercise 1.9: An additional method for solving single nonlinear equation is the "Regula–Falsi" or method of "false position." Like the bisection method, the false position method starts with two points a_0 and b_0 such that $f(a_0)$ and $f(b_0)$ are of opposite signs, which implies that the function f has a root in the interval $[a_0, b_0]$. The Regula–Falsi method proceeds by producing a sequence of shrinking intervals $[a_k, b_k]$ that always contain a root of f.

At iteration number k, the value

$$c_k = \frac{f(b_k)a_k - f(a_k)b_k}{f(b_k) - f(a_k)}$$

is computed. As explained below, c_k is the root of the secant line through $(a_k, f(a_k))$ and $(b_k, f(b_k))$. If $f(a_k)$ and $f(c_k)$ have the same sign, then set $a_{k+1} = c_k$ and $b_{k+1} = b_k$; otherwise set $a_{k+1} = a_k$ and $b_{k+1} = c_k$. This process is repeated until the root is approximated sufficiently well.

The above formula is also used in the secant method, but the secant method always retains the last two computed points, while the false position method retains two points that bracket a root. On the other hand, the only difference between the false position method and the bisection method is that the latter uses $c_k = (a_k + b_k)/2$.

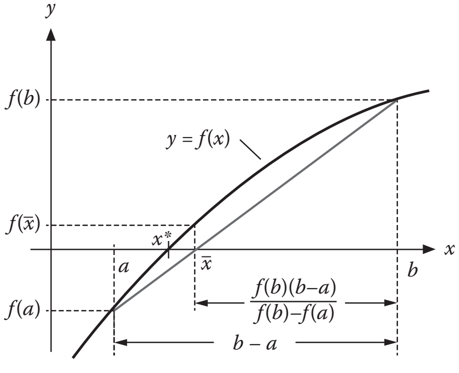


FIGURE 1.7 Regula–Falsi method.

A schematic description of the Regula–Falsi method is shown in Figure 1.7. Repeat either Exercise 1.1 or 1.4 using the method of false position.

Exercise 1.10: Refer to the diagram and data of Example 1.7. The dew point of a vapor is the temperature at a given pressure at which the first drop of liquid is formed. Thus, at the dew point, the ratio of vapor to feed (*V/F*) is essentially one. Conversely, the bubble point of a liquid is the temperature at a given pressure at which the first bubble of vapor is formed and at this condition $V/F = 0$.

Assume, as in Example 1.7, an ideal mixture (where Raoult's law applies). Also, assume the same mixture and data as shown in Figure 1.4.

a. At the dew point, $V/F = 1$ [there is an infinitesimal amount of liquid, but it *does* exist and has a mole fraction for each component (x_i); however, since there is only an infinitesimal amount of liquid, $V = F$ and $y_i = z_i$]. Applying the fact that the sum of $x_i = 1$, the following equation results:

$$1-\sum_{j=1}^{n_c}\left(\frac{z_j}{k_j}\right)=0$$

Since k_j are functions of temperature, this nonlinear equation can be solved for the temperature (dew point).

b. At the bubble point, $V/F = 0$ ($L/F = 1$) and $x_i = z_i$, and if that fact that the sum of $y_i = 1$ is applied, the following equation results:

$$1-\sum_{j=1}^{n_c} z_j k_j = 0$$

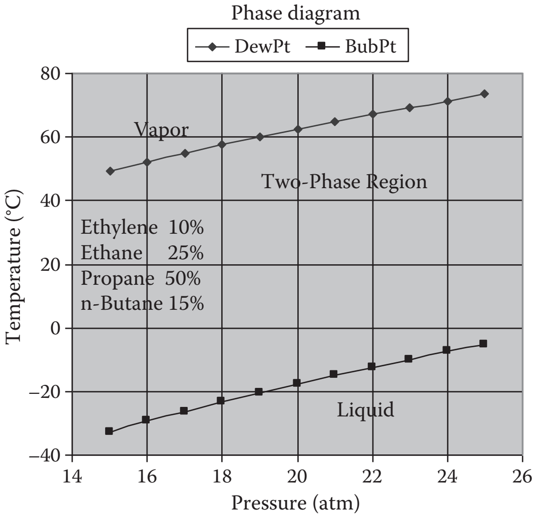
This nonlinear equation can be solved for the temperature (bubble point).

The specific assignment is as follows.

1. Prepare an Excel spreadsheet that calculates for a mixture the bubble point at pressures of 15, 16, …, 25 atm and produces a plot of the bubble point versus pressure with suitable annotations and title. Use Goal Seek to solve the single nonlinear equation at each pressure. Data for the system are given in Figure 1.4.

   Prepare a *Keystroke Macro* to find the bubble point at each pressure. That is, record the macro and edit it using a While statement so that one keystroke finds all bubble points.
2. Prepare an Excel spreadsheet that is similar to the one for the bubble point, but is used to compute the dew point of the same mixture at the same pressures. Put these calculations on the same Worksheet as those for the bubble point.

3. Transfer the pressure, bubble point, and dew point data to a separate portion of the Worksheet and graph both the bubble and dew points versus pressure. This produces a *phase diagram* for the hydrocarbon mixture. It should look *similar* to the following:



Note that all of the text to fully describe the graph (as in the example plot) must be added manually.

REFERENCES

Henley, E.J., and E.M. Rosen, *Material and Energy Balance Computation*, Wiley, New York (1969).

NIST Data Gateway, Chemistry WebBook. http://webbook.nist.gov/chemistry/.

2 Visual Basic® for Applications Programming

2.1 INTRODUCTION

Visual Basic® for Applications (VBA) is a computer programming language. This short introduction is not intended as a complete course on computer programming. However, programming involves things that engineers are good at: calculating the results of formulas and equations, making logical decisions, and designing algorithms (steps to be used to solve a particular problem). VBA is only one of a host of programming languages, some of which are FORTRAN, C, or C++. The reason for using VBA instead of any other language is that it allows the use of Excel® worksheets for both input and output of data. Excel also makes it easy to put the output data into graphical form. Therefore, VBA provides the Excel system with highly flexible programming capabilities. Furthermore, VBA is *part* of Excel—no licensing is required.

A computer program consists of *code* in the syntax of a specific programming language that implements an *algorithm*. An algorithm can be thought of as a recipe, much like the instruction one sees in a cookbook. If the algorithm can be expressed (e.g., through words or graphically or by any other means), then the algorithm can be *translated* into any programming language with relative ease. For simple algorithms, one can simply begin by writing code in the target language such as VBA; but for complex algorithms, it can be foolhardy and frustrating to try to jump directly to code writing without some means that encourages *algorithm design*. The earliest method for algorithm design was a graphical tool known as a *flowchart*. The method shown here is related to a flowchart but is known as a *structure chart* (Bowles 1979; Law 1983, 1985) since it enforces a high level of structure to the algorithm.

2.2 ALGORITHM DESIGN

Consider, for example, the task of changing a flat tire. The steps involved can be expressed in English as follows (these steps are not unique):

1. Remove the spare and be sure it is not itself flat.
2. If the spare is not flat then proceed; otherwise call AAA!
3. Set the emergency brake and chock the wheels.
4. Remove the jack and set it up in the appropriate place.
5. Loosen the lugs slightly.
6. Jack up the vehicle until the spare no longer touches the ground.

7. Remove the lugs.
8. Remove the flat tire and put on the spare.
9. Put the lugs back on and tighten slightly.
10. Lower the jack all the way.
11. Tighten the lugs snugly.
12. Done!

Another good idea when several sequential steps are involved is to split them into categories (as when preparing an outline for a paper or report). Here is a typical way in which this might be done for the flat tire problem:

1. Get ready.
   a. Remove the spare and be sure it is not flat.
   b. Set the brake and chock the wheels.
   c. Remove the jack and set it up.
2. Remove the flat tire.
   a. Loosen the lugs.
   b. Jack up the car.
   c. Remove the lugs.
   d. Remove the tire.
3. Put on the spare.
   a. Place spare on wheel.
   b. Slightly tighten the lugs.
   c. Lower the jack.
   d. Tighten the lugs.
4. Finish up.
   a. Place the flat tire in the trunk.
   b. Take the spare to be repaired.

Writing out instructions in this manner is fine as long as the steps are consecutive (sequential), do not require a decision (although step 2 involves a simple go/no-go decision), and do not require repetition (doing something over again several times).

Shown in Figure 2.1 is the algorithm expressed as a structure chart.

Comparing the outline form of the algorithm and the structure chart indicates that the sequence of operations is "left to right, depth first." That is, the top (or root) node simply gives a title for the algorithm. Control passes down to "Get ready" then "Remove and check spare,"…, "Set up the jack," then "Remove flat," and so forth. Studies have shown that most algorithm designers work better using a two-dimensional tree rather than a linear outline (or using code directly).

Most algorithms can be expressed using the following logical structures:

- Sequence (as in the change flat example of Figure 2.1)
- Decision making (ask a question and take different actions depending on the answer)
- Repetition (performing similar or identical tasks over and over again)

Shown in Figure 2.2 is a structure chart for the tire problem that includes all three of these types of operations.

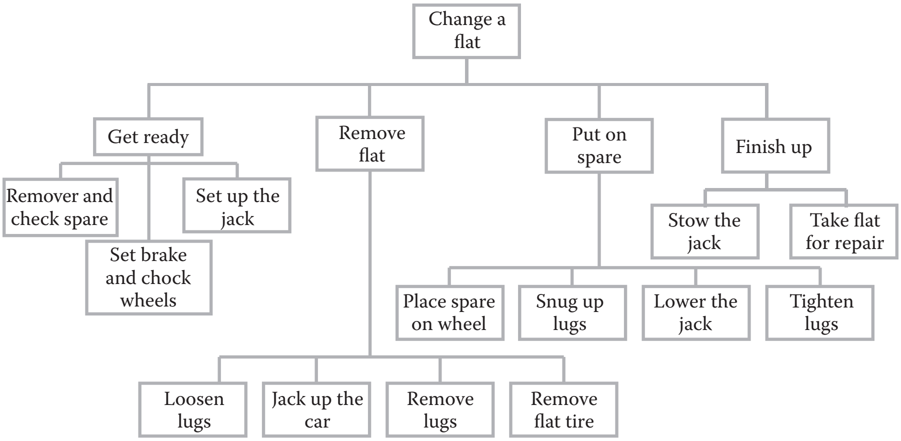


FIGURE 2.1 Structure chart algorithm design for changing a flat tire.

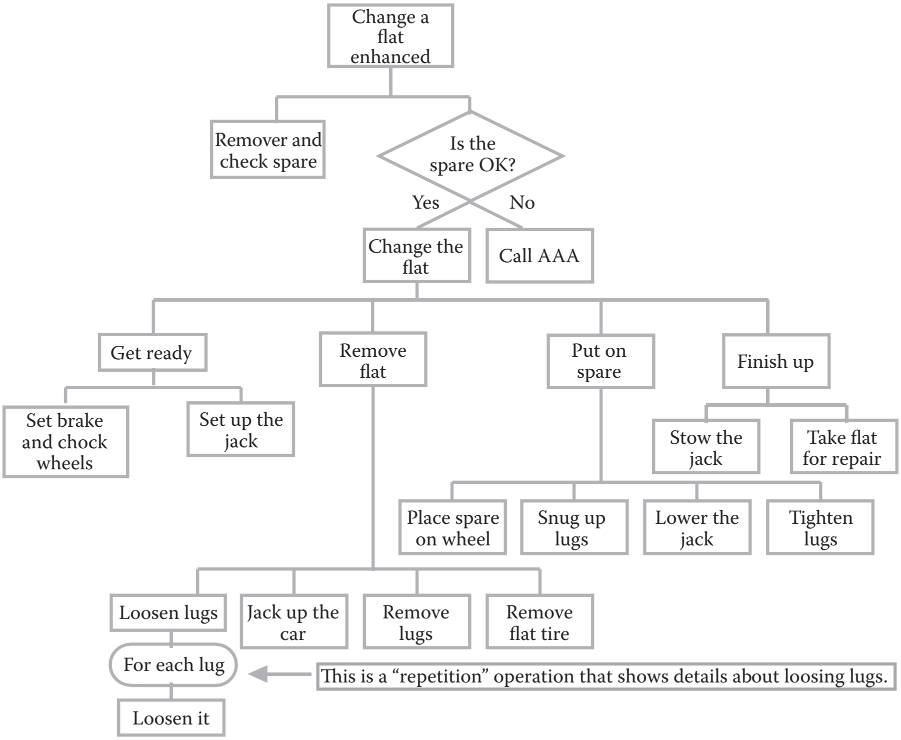


FIGURE 2.2 Enhanced structure chart algorithm design for changing a flat tire.

A “diamond” shape indicates a *decision* and the lines emanating from it are labeled Yes/No or True/False. The “oval” shape indicates *repetition*. A similar repetition operation could be added each time the lugs are manipulated, but only one repetition is shown in order to conserve space.

2.3 VBA CODING

Now that the basic idea about designing an algorithm has been introduced, it is appropriate to look at how to translate the structure chart into VBA code that can be executed from within Excel. VBA is a programming language similar to (but in some instances different from) Visual Basic. The differences in the two languages can be subtle (and frustrating). A simple way to learn VBA is to construct example programs and study them. The process of creating a simple example will be “walked through,” and it will be seen how that, at any time, *help* can be obtained from within the VBA Editor. The biggest problem when getting help is to “weed out” the myriad of topics that are likely to come up. That is, there is no lack of help, but it takes a bit of practice to know where to look for what.

Here is a list of VBA topics that are covered (at least partially) in this first example:

- Data types (especially Integer, Double)
- Declarations (especially Dim, ReDim)
- Variables and their declaration
- Strings
- Constants
- Operators
- Expressions
- Statements
- Procedures
- Control flow
  - Conditional
  - Repetition

2.4 EXAMPLE VBA PROJECT

By “project” is meant a VBA program interacting with an Excel spreadsheet. This first example is *not* one that would actually require a VBA program since it is too simple. However, it shows the basics of

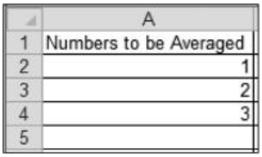
- Retrieving data from an Excel spreadsheet
- Performing operations on the input data
- Outputting data to the Excel spreadsheet

To get started, open a blank Excel spreadsheet. The first thing a programmer should do is give the “specifications” for the program. The specifications take the form of a User’s Manual. It is good programming practice to write the specifications *first* (before even thinking about the algorithm or the code).

Example Program 2.1: Averaging Numbers

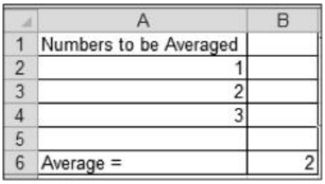
SPECIFICATION

This program takes numeric data from an Excel spreadsheet and calculates the average of the numbers. The input data look similar to the following:



	A
1	Numbers to be Averaged
2	1
3	2
4	3
5	

The user must supply a list of numbers to be averaged in a vertical column ending with a “blank” cell. It calculates the average of the numbers and outputs the average on the spreadsheet row after the blank cell. For the sample data shown, the final appearance of the spreadsheet is as follows:



	A	B
1	Numbers to be Averaged	
2	1	
3	2	
4	3	
5		
6	Average =	2

Note: Before the program is executed, the cursor (selected cell) must be the one with the text “Numbers to be Averaged” before running the program.

ALGORITHM DESIGN

The next step is to “design” the program logic (algorithm) and then (finally) to actually write the code. A structure chart for the program logic is given in Figure 2.3. There are many ways in which the desired operations can be carried out; the steps shown are merely one way in which to accomplish the desired steps.

While designing the algorithm, several variable names have been chosen (invented). These names are chosen by the programmer and should, in some way, be descriptive of the data that they represent. A description of the chosen variable names is as follows:

```
ActRow       An integer to keep tract of the row number
             where data are stored in the flowsheet
Sum          A floating point (double precision) variable
             to hold the sum of the input numbers
NumNumbers   An integer variable representing how many
             numbers
             there are
InputNumber  A floating point number holding one input data
             value
Average       A floating point value representing the average
              of the input data values
```

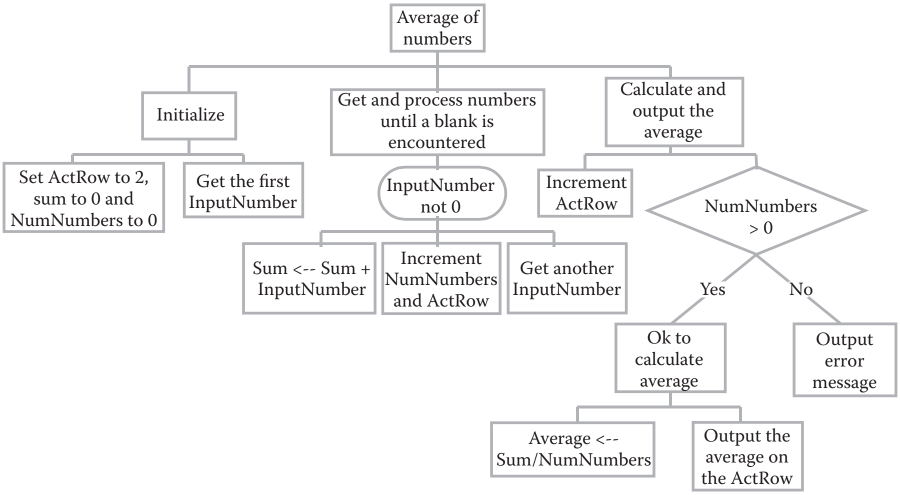
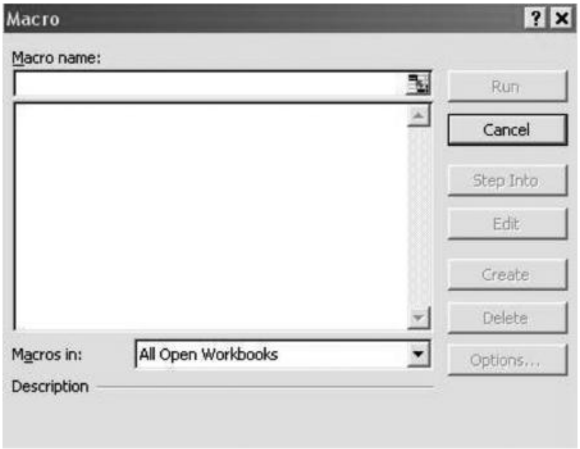

Average of numbers
Initialize
Get and process numbers until a blank is encountered
Calculate and output the average
Set ActRow to 2, sum to 0 and NumNumbers to 0
Get the first InputNumber
InputNumber not 0
Increment ActRow
NumNumbers > 0
Sum <-- Sum + InputNumber
Increment NumNumbers and ActRow
Get another InputNumber
Yes
No
Ok to calculate average
Output error message
Average <-- Sum/NumNumbers
Output the average on the ActRow

FIGURE 2.3 Algorithm for averaging numbers.

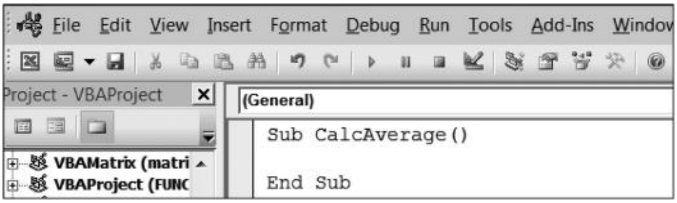
CODING IN VBA

Now that the algorithm has been sketched out in some detail, it is time to implement it in VBA code. Note that the "oval" shape indicates repetition: continue to get input numbers from the spreadsheet until a 0 or blank is encountered (note that a blank is interpreted as zero). The backward arrow (<——) indicates assignment: the entity on the right is computed and placed in the entity on the left. Again, a diamond shape represents a decision block.

To create a VBA program from within the Excel environment, go to Developer/Macros. A screen like the following appears.

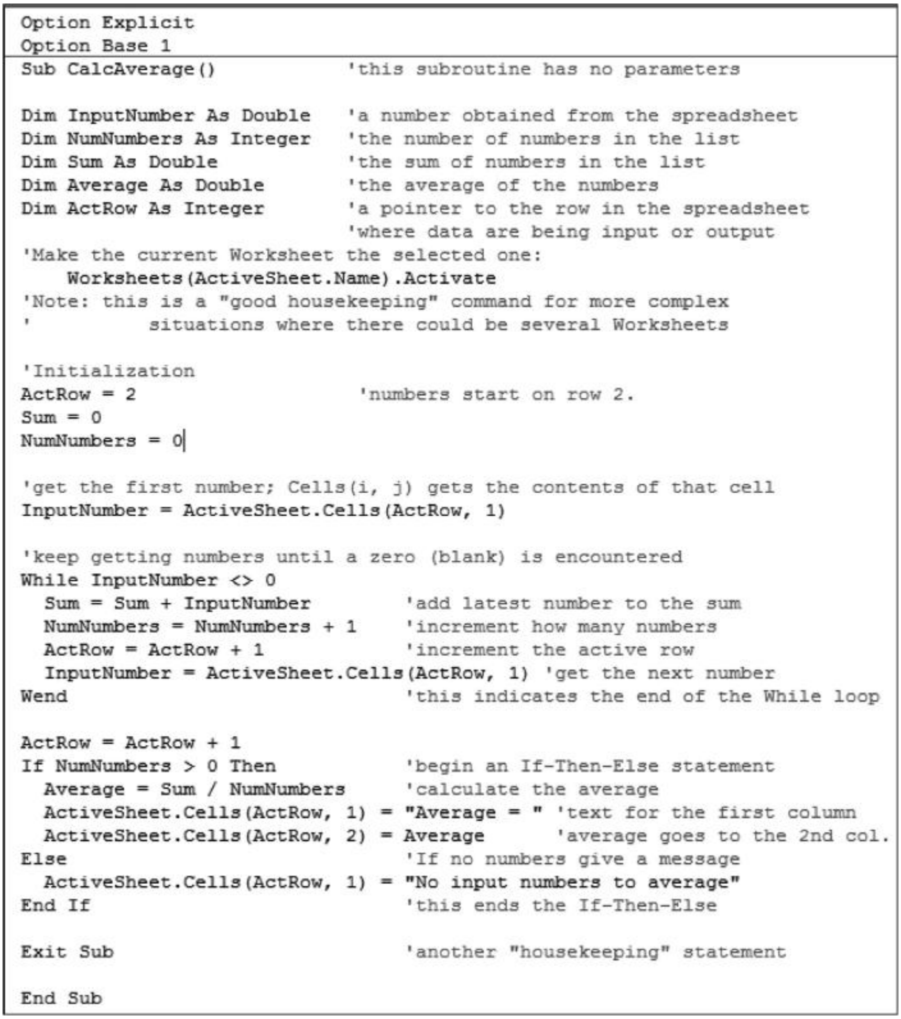


Type a descriptive name, such as CalcAverage into the Macro name field. The Create indicator will activate; hit Create. This opens the VBA Editor and that screen will look similar to the following. Note that the program or macro is automatically called a Sub (for Subroutine) (similar to a procedure or function in C). For now, all of the information to the left of the VBA Editor can be ignored.



```
Sub CalcAverage()

End Sub
```

The programmer is presented with a "template" for writing the code. The following is a listing of the code written for this problem. It follows carefully the algorithm design. Any text after an apostrophe (') is a *comment*.



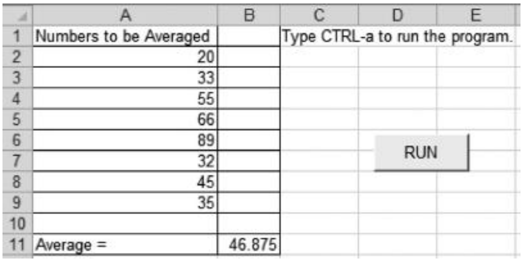
```
Option Explicit
Option Base 1
Sub CalcAverage()           'this subroutine has no parameters

Dim InputNumber As Double   'a number obtained from the spreadsheet
Dim NumNumbers As Integer   'the number of numbers in the list
Dim Sum As Double           'the sum of numbers in the list
Dim Average As Double       'the average of the numbers
Dim ActRow As Integer       'a pointer to the row in the spreadsheet
                            'where data are being input or output
'Make the current Worksheet the selected one:
    Worksheets(ActiveSheet.Name).Activate
'Note: this is a "good housekeeping" command for more complex
'          situations where there could be several Worksheets

'Initialization
ActRow = 2                   'numbers start on row 2.
Sum = 0
NumNumbers = 0

'get the first number; Cells(i, j) gets the contents of that cell
InputNumber = ActiveSheet.Cells(ActRow, 1)

'keep getting numbers until a zero (blank) is encountered
While InputNumber <> 0
  Sum = Sum + InputNumber        'add latest number to the sum
  NumNumbers = NumNumbers + 1    'increment how many numbers
  ActRow = ActRow + 1            'increment the active row
  InputNumber = ActiveSheet.Cells(ActRow, 1) 'get the next number
Wend                             'this indicates the end of the While loop

ActRow = ActRow + 1
If NumNumbers > 0 Then           'begin an If-Then-Else statement
  Average = Sum / NumNumbers     'calculate the average
  ActiveSheet.Cells(ActRow, 1) = "Average = " 'text for the first column
  ActiveSheet.Cells(ActRow, 2) = Average      'average goes to the 2nd col.
Else                             'If no numbers give a message
  ActiveSheet.Cells(ActRow, 1) = "No input numbers to average"
End If                           'this ends the If-Then-Else

Exit Sub                         'another "housekeeping" statement

End Sub
```

Here are some things to notice about this program:

- The first two items are *compiler options*. The `Option Explicit` makes it imperative that the data type of all variables be declared (using a `Dim` statement). The `Option Base 1` makes any arrays (there are none in this first program) start with a subscript of 1.
- All variables *must* be declared in a `Dim` statement.
- Variable names can be any length with a mix of upper and lower case; all names are case sensitive. The first character of a variable name must be alphabetic.
- Any text beginning with an apostrophe is a comment.
- The *built-in* function `Cells(i, j)` refers to the `i,j`th cell in the spreadsheet, where `i` is the row number and `j` is the column number. That is, the spreadsheet is treated as a matrix.
- Assignment statements use an equal sign (=).
- The `While` loop continues as long as the logical (Boolean) statement is `True`; the last statement in the loop range is the `Wend`.
- The `If-Then-Else-End If` allows one kind of action if the logical statement is `True` and another if the logical statement is `False`.
- Character strings are delimited by quote marks as in "`No input numbers to average`." Remember, anything after an apostrophe is a *comment* and is not part of the code.

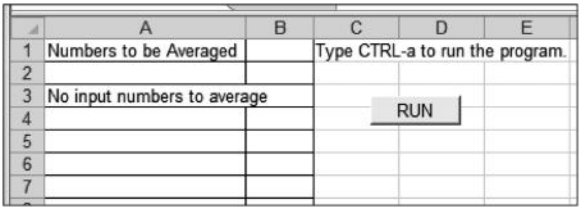
To execute the program, go to File/Close and Return to Microsoft Excel®. Enter the data to be averaged (the numbers *must* be in the first column) and then go to Developer/Macros/CalcAverage/Run. The program executes and displays the average of any numbers entered. If no numbers were entered, the message "`No input numbers to average`" is displayed. While at the Developer/Macros/CalcAverage/Run menu, Options can be chosen, which gives an opportunity to assign a Ctrl key (also called a "hot key") to the Macro (VBA Program). After this, Ctrl+(selected key) can be used to run the program. Note that the selected key is case sensitive. The lowercase letter `a` was used here.

Another way to run the program is to create a button on the spreadsheet to do so. Select Developer/Controls/Insert and a pop-up box appears. Select the "button-button" (first row, first column) and the cursor turns into a cross-hair. Use this cursor to draw a button on the spreadsheet (be sure to drag the cross-hairs cursor across an area—do not just click on a cell). A window appears and the button macro can be named (Here, RUN was used). Then hit Record (no name need be given here). Hit OK at the next window. Simply type Ctrl+a (or execute the program the "long way" via the Developer/Macros/CalcAverage/Run). When the program executes, hit the Stop Recording button. If the name on the button does not change, right click on it and select Change Text and put RUN in manually. Here is what the new spreadsheet looks like (note that different data have been entered):



	A	B	C	D	E
1	Numbers to be Averaged		Type CTRL-a to run the program.		
2	20				
3	33				
4	55				
5	66				
6	89				
7	32				
8	45				
9	35				
10					
11	Average =	46.875			

When testing a program, data should be entered that exercises all logic of the program. In this instance, data should be entered that produces the error message as in the following:



	A	B	C	D	E
1	Numbers to be Averaged		Type CTRL-a to run the program.		
2					
3	No input numbers to average				
4					
5					
6					
7					

Since no numbers were entered, the error message was output. Note that without this safeguard, a divide by zero would be attempted, which causes a system error.

2.5 GETTING HELP AND DOCUMENTATION ON VBA

While in the VBA Editor, much can be learned about the language by clicking the question mark (?). Follow the choices until getting the VBA Language Reference, and there will be a number of choices, some of which will not mean much until later. Another way to learn about VBA is by recording a Keystroke Macro and then looking at the code (recall the exercise when a While Loop was inserted into a recorded Macro involving Goal Seek).

Click on any of the items listed to get detailed information. Ones of probable particular interest are the following:

- Constants
- Data types
- Groups
- Operators
- Statements

It might be helpful to peruse the available Functions as well as other Help topics. A word of warning: there is much more information available than many people will want to investigate.

2.6 VBA STATEMENTS AND FEATURES

In Example 2.1 that calculates the average of a list of numbers, many statements and features of VBA have been introduced informally. The following VBA components are now discussed in some detail:

- Assignment statement
- Expressions
- Object-oriented programming (OOP) and the properties of objects
- Built-in functions (Abs, Exp, etc.)

- Program control
    - Branching (If-Then-Else)
    - Looping (For...Next, While-Wend)
- Data types (Integer, Long, etc.)
- Subroutines and functions
- Objects and methods
- Getting data from a worksheet
- Putting data onto a worksheet
- Alternative I/O methods
- Arrays in VBA (static and dynamic arrays)

2.6.1 Assignment Statement

The general form of the assignment statement is

```
Variable = Expression
```

Variable names must begin with an alphabetic character and can contain digits (0…9) and the underscore character (_). Variable names can be of any length, and good practice is to mix uppercase and lowercase letters to make the names self-documenting. For example, SumOfNumbers takes on obvious significance, whereas sumofnumbers is not so obvious. Variable names cannot be the same as a VBA "keyword." There are hundreds of these keywords, and this can be problematic.

2.6.2 Expressions

There are many different types of expressions in VBA. Among these are numerical, logical (Boolean), and character string expressions.

2.6.2.1 Numerical Expressions

The most common type is a numerical expression involving the algebraic operators as follows:

^	Exponentiation
*	Multiplication
/	Division
+	Addition
−	Subtraction

Parentheses can be used (and should be used liberally) to reflect the exact hierarchy of operations. The operational hierarchy is the same as for standard algebraic expressions as follows:

1. Parentheses
2. Exponentiation
3. Multiplication/division
4. Addition/subtraction

When operations are at the same hierarchical level, they are performed from left to right.

2.6.2.2 Logical (Boolean) Expressions

The result of a logical expression is either True or False (these are two VBA "keywords"). Boolean expressions are composed of comparative sub-expressions and/or logical operators. Comparative operators are <, < =, >, > =, =, < > and the logical operators are And, Or, and Not (these are VBA keywords). Here are a few examples:

`a < b And c = a` In the operator hierarchy, comparative operators come before logical operators, so this is the same as the next one. Putting parentheses around the comparatives makes the meaning clear and this practice is encouraged.

`(a < b) And (c = a)` The expression is True only if both comparisons are `True`.

`(a < b) Or (c = a)` The expression is True if either or both comparisons are `True`.

`Not (a = b)` Same as `a <> b`.

2.7 OBJECTS AND OOP

OOP may be seen as a collection of cooperating *objects*, as opposed to a traditional view in which a program may be seen as a list of instructions to the computer. In OOP, each object is capable of receiving messages, processing data, and sending messages to other objects. Each object can be viewed as an independent little machine with a distinct role or responsibility.

A thorough treatment of OOP is far beyond what can or need be covered in this text. It is only important to grasp the basic ideas and know where to get help when it is needed.

Some examples of VBA objects are the Workbook object, the Worksheet object, the Chart object, and the Range object. To get an alphabetic list of objects, when in the VBA editor, click the question mark and then select Microsoft Visual Basic Reference/Objects.

Objects have *methods*, which are invoked in the traditional syntax of OOP using a period separator. To get an alphabetic list of methods, when in the VBA editor, click the question mark and then select Microsoft Visual Basic Reference/Methods.

As an example, the following is a commonly used invocation of an `object.method` command:

```
Worksheets(ActiveSheet.Name).Activate
```

The object `Worksheets (ActiveSheet.Name)` takes on the identity of the active worksheet (the one that is currently showing when the Excel window is active).

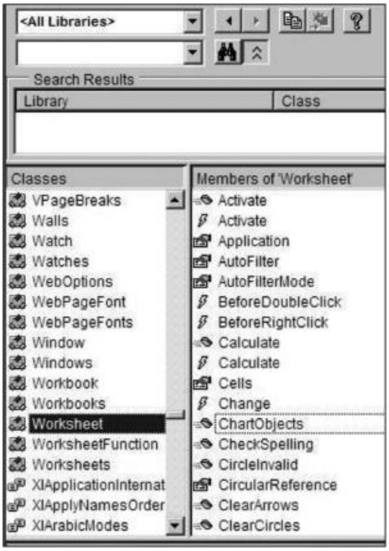
The `Activate` method tells the current VBA program that any further references to an object (`Range`, for example) refer to cells of the active worksheet. Effectively, this statement makes the current worksheet the "active" one. For Excel files with multiple worksheets, this can be an important distinction. A VBA program with this statement in it can then be invoked from any of the worksheets, and all references to, for example, `Cells(I, J)` refer to objects within the worksheet from which the VBA program is executed. In other words, if the VBA program was created while `Sheet 1` was active and later it is used when `Sheet 2` is active, then `Cells(I, J)` refers to cells on `Sheet 2`. Without the `Activate` command, the cells in Sheet 1 would be referred to by default.

Here is another example:

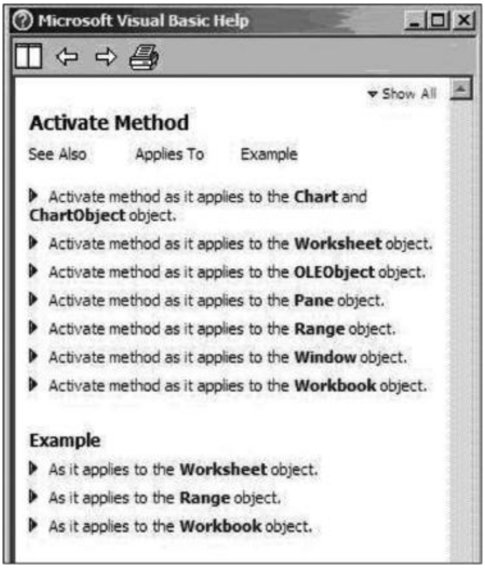
```
Range("A1:E150").Sort "Last Name", xlAscending
```

This says sort the data contained in the range `A1:E150` in ascending order using as the sort key the values in the column headed by the label `Last Name`. The label `xlAscending` is one of the many VBA built-in constants.

A list of all of the VBA objects and associated "members" can be viewed by clicking on View/Object Browser when in the VBA Editor. The Object Browser looks like this:



Notice that the `Worksheet` object is selected, so its members appear in the right-hand column (and among these members is `Activate`). There are two `Activate` entries: one is a "subroutine" and one is an "event." *Right clicking* on the one with the "lightning bolt" and selecting "Help" gives the following:



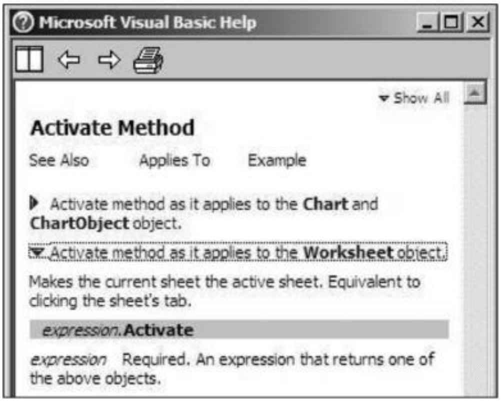
Activate Method

- Activate method as it applies to the **Chart** and **ChartObject** object.
- Activate method as it applies to the **Worksheet** object.
- Activate method as it applies to the **OLEObject** object.
- Activate method as it applies to the **Pane** object.
- Activate method as it applies to the **Range** object.
- Activate method as it applies to the **Window** object.
- Activate method as it applies to the **Workbook** object.

Example

- As it applies to the **Worksheet** object.
- As it applies to the **Range** object.
- As it applies to the **Workbook** object.

By selecting “Activate method as it applies to the **Worksheet** object,” the following window appears:



Activate Method

- Activate method as it applies to the **Chart** and **ChartObject** object.
- Activate method as it applies to the **Worksheet** object.

Makes the current sheet the active sheet. Equivalent to clicking the sheet's tab.

expression.**Activate**

expression Required. An expression that returns one of the above objects.

Thus, `Worksheet.Activate` does the same as clicking on the sheet’s tab, but it is done under program control instead of with the mouse pointer.

2.8 BUILT-IN FUNCTIONS OF VBA

There are many built-in functions in VBA, but only a small number of them apply to scientific and engineering applications. Table 2.1 shows a list of just a few of them.

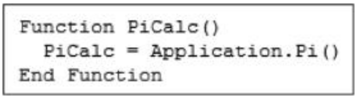
TABLE 2.1
Short List of VBA Functions and Their Excel® Counterparts

Purpose	VBA Function	Excel Function
Absolute value	`Abs(x)`	`ABS(x)`
Truncate to integer	`Int(x)`	`INT(x)`
Round x to n digits after decimal	`Round(x, n)`	`ROUND(x, n)`
Square root	`Sqr(x)`	`SQRT(x)`
Exponential	`Exp(x)`	`EXP(x)`
Natural log	`Log(x)`	`LN(x)`
Base 10 log	-	`LOG10(x)`
Base b log	-	`LOG(x, b)`
Value of π	-	`PI()`
Sine	`Sin(x)`	`SIN(x)`
Cosine	`Cos(x)`	`COS(x)`
Tangent	`Tan(x)`	`TAN(x)`
ArcSine	-	`ASIN(x)`
ArcCosine	-	`ACOS(x)`
ArcTangent	`Atn(x)`	`ATAN(x)`
ArcTangent (4 quadrant)	-	`ATAN2(x, y)`
Degrees to radians	-	`RADIANS(x)`
Radians to degrees	-	`DEGREES(x)`
Remainder (x modulo y)	`x Mod y`	`MOD(x, y)`
Random number	`Rnd()`	`RAND()`

To see the entire list, click the question mark for help. Then choose Microsoft Visual Basic Documentation/Visual Basic Language Reference/Functions. The functions are grouped alphabetically.

Also shown in Table 2.1 are the corresponding Excel functions (ones that can be accessed directly from a Worksheet). Note that the VBA functions are capitalized. Perhaps the most confusing one is **Sqr** for taking the square root. The corresponding Excel function is the more usual SQRT.

Excel functions that have no VBA counterpart can be called from VBA. For example, if it is desired to use the `Pi()` function within a VBA program, the `Application` method can be used as in the following `Function`:



```
Function PiCalc()
  PiCalc = Application.Pi()
End Function
```

This code produces the value of π when the `Function PiCalc` is called.

2.9 PROGRAM CONTROL

Program control statements include those for decision making (branching) and for looping. Many such statements available in VBA are discussed below. Ones less frequently used (or ones that are repetitious) are not covered.

2.9.1 Branching

Two popular branching or decision making statements in VBA are the If-Then-Else and the Select Case statements. Since anything that can be done with the Select Case statement can be implemented just as well using If-Then-Else, Select Case is not presented here.

2.9.1.1 If-Then-Else

The syntax of the If-Then-Else statement is

```
If Logical Expression Then
   statements 1        'this can be any number of statements on separate lines
Else
   statements 2        'this can be any number of statements on separate lines
End If
```

Simpler forms of this statement are possible, but it is easier to always use this form, which is more formally called the Block If statement. The Else clause can be omitted if there is no else “branch” in the program logic. It is good programming practice to indent the statements in each branch. Here is a simple example:

```
IF a < b Then
       c = a + b
Else
       c = a - b
End If
```

The amount of indentation to use is up to the programmer. The simplest thing is to use the Tab key to provide indentation, but at least two space indentation is recommended.

2.9.2 Looping

There are four looping statements in VBA (For...Next, Do While...Loop, Do...Loop While, For Each...Next). Only the first two of these are presented since the other two are repetitious (or can be confusing).

2.9.2.1 For…Next

This is a simple “counting” loop and it has the following syntax:

```
For Counter = Start To End [Step Increment]
       Statements
Next [Counter]
```

Counter is a variable name (usually of an integer type). On entering the loop (executing Statements the first time), Counter is initialized to the variable (or expression) Start. After the Statements are executed the first time, Counter is incremented by 1 unless the optional Step Increment is used. For example, using Step 2 means that Counter is incremented by 2 instead of 1. The Increment can be negative so that counting is backward. Looping continues until Statements are executed with Counter having the value of End. Note that if Counter is not an integer type, there can be some questions about its value the last time through the loop. Here is a very typical example:

```
For i = 1 To n
   x(i) = i
Next i
```

In the example, the Counter is the variable i, Start is 1, and End is the value of the variable n. So, i takes on the values 1, 2, …, n, after which control passes to the statement after the loop.

2.9.2.2 While...Wend Statement

This looping statement executes a series of statements as long as a given condition is **True**. The syntax of this statement is

```
While condition
   statements
Wend
```

If condition (a logical expression) is True, all statements are executed until the Wend statement is encountered. Control then returns to the While statement and condition is again checked. If condition is still True, the process is repeated. If it is not True, execution resumes with the statement following the Wend statement. While...Wend loops can be nested to any level. Each Wend matches the most recent While.

2.10 VBA DATA TYPES

All of the VBA data types are shown in Table 2.2. For floating point numbers, it is best to always use Double, which provides about 15 significant digits. Most modern computers are equipped with floating point processors, so using Double (as opposed to Single) costs essentially nothing. For integers, it is best to use Long (instead of Integer) since integers outside the range +/– 32,767 occur sometimes. Boolean data types can be useful in some applications.

The keyword DIM is used to specify the data type of a variable. When *Option Explicit* is used (as is always recommended), the type of every variable in the program *must* be specified explicitly. Later it will be seen that DIM is also used to specify an Array data structure. Table 2.2 summarizes some of the VBA built-in data types. Only the types most often used are included. A full list of data types can be viewed via the Help system.

TABLE 2.2
VBA Built-In Data Types

Data Type	Storage Required	Range of Values
`Boolean` (logical)	2 bytes	True or False
`Integer`	2 bytes	–32,767 to 32,768
`Long`	4 bytes	–2,147,283,648 to 2,147,283,647
`Real` (single precision)	4 bytes	–3.402823E+38 to –1.401298E-45 for negatives 1.401298E-45 to 3.402823E+38 for positives
`Double` (double precision)	8 bytes	–1.79769313486232E+308 to –4.94065645841247E-324 for negatives 4.94065645841247E-324 to 1.79769313486232E+308 for positives
`String`	1 byte/char.	Delimited by quote marks (").
`Variant`	16 bytes + 1/char.	Any numeric value up to the range Double or any text.

2.11 SUBS AND FUNCTIONS

A VBA program is automatically a "Sub" or Subroutine. The introductory VBA example program for getting the average of a list of numbers began with

```
Sub CalcAverage()
```

In addition to Subs, VBA also has `Function` subprograms. Here are the differences between the two:

1. A Sub can receive information (properties) and it can change or set properties.
2. A Function can only receive properties.
3. A Sub is invoked by a "Call" statement.
4. A Function is invoked simply by using its name.

A Sub has the following syntax:

```
Sub name ([arglist])
   Statements
End Sub
```

Things within square brackets are *optional*. The definition of `arglist` is as follows:

`arglist:` A list of variables representing arguments that are passed to the `Sub` procedure when it is called. Multiple arguments are separated by commas.

In its simplest form, `arglist` consists of variable names separated by commas. Consider for example

```
Call Alpha(Beta, Gamma)
```

The name of the subroutine is `Alpha`. `Beta` and `Gamma` are the arguments or parameters (often called the actual arguments) of the subroutine in the calling program. When the subroutine is coded, it might appear as follows:

```
Sub Alpha (Delta, Epsilon)
```

Here, `Delta` and `Epsilon` are what we call *dummy* parameters since they take on information provided by the actual arguments when `Alpha` is called.

The actual parameters can be "sent" to the subroutine in one of two modes:

- By *reference*: What is sent to the subroutine is the memory location of the parameter. Therefore, if the parameter is altered by the subroutine, such changes will be known to the caller when control is passed back at the end of the subroutine (often called "returning" from the subroutine). This is the *default mode* for parameter passing.
- By *value*: If an actual argument is a constant, then it is automatically passed by value (it would be chaotic if the constant 2 suddenly represented the number 3, for example). It is rare that a variable name used as an argument is to be passed by value, but if so, simply enclose it in parentheses as in

```
Call Alpha((Beta), Gamma)
```

Here, only the current value of `Beta` is sent to the subroutine. Using the same dummy variables as previously (`Delta, Epsilon`), any changes made to `Delta` by the subroutine are not transmitted back to the calling program.

2.12 INPUT AND OUTPUT

2.12.1 Getting Data from the Worksheet

In traditional programming (such as with C++), the source of input data is usually either the keyboard or (usually when there are lots of data) a file. When using VBA, which is an integral part of Excel, the natural source of input is from the Worksheet itself.

In Example Program 2.1, the following statement appeared:

```
InputNumber = ActiveSheet.Cells(ActRow, 1)
```

The `Cells` method treats the spreadsheet as a two-dimensional array representing the rows and columns. If the variable `ActRow` is 3, for example, then the variable

InputNumber is assigned whatever value is contained in the cell in the third row and first column. It is possible using the Range object to write a statement that inputs values from a collection of cells (but this tends to make things more complex than they need be).

2.12.2 Putting Data onto the Worksheet

The procedure for "writing" output data to the active worksheet simply reverses the assignment used for input. Another example taken from the sample program is as follows:

```
ActiveSheet.Cells(ActRow + 1, 2) = Average
```

The value currently stored in the variable Average is written into the cell referenced on the left of the assignment.

2.13 ARRAY DATA STRUCTURES

An Array is the equivalent of a subscripted variable. There are one-, two-, or even higher-dimensional arrays. As with scalar variables, arrays are declared in a Dim statement as in

```
Dim x(4) as Double
```

In this example, the array x consists of 5 elements: x(0), x(1), x(2), x(3), and x(4). By default, the first subscript is 0. However, starting at zero can lead to confusion, and it is recommended to always include

```
Option Base 1
```

Then subscripts begin with 1, which many find more natural. For those comfortable with subscripts starting at zero, this option need not be used. For all programs in this book use, Option Base 1 is used.

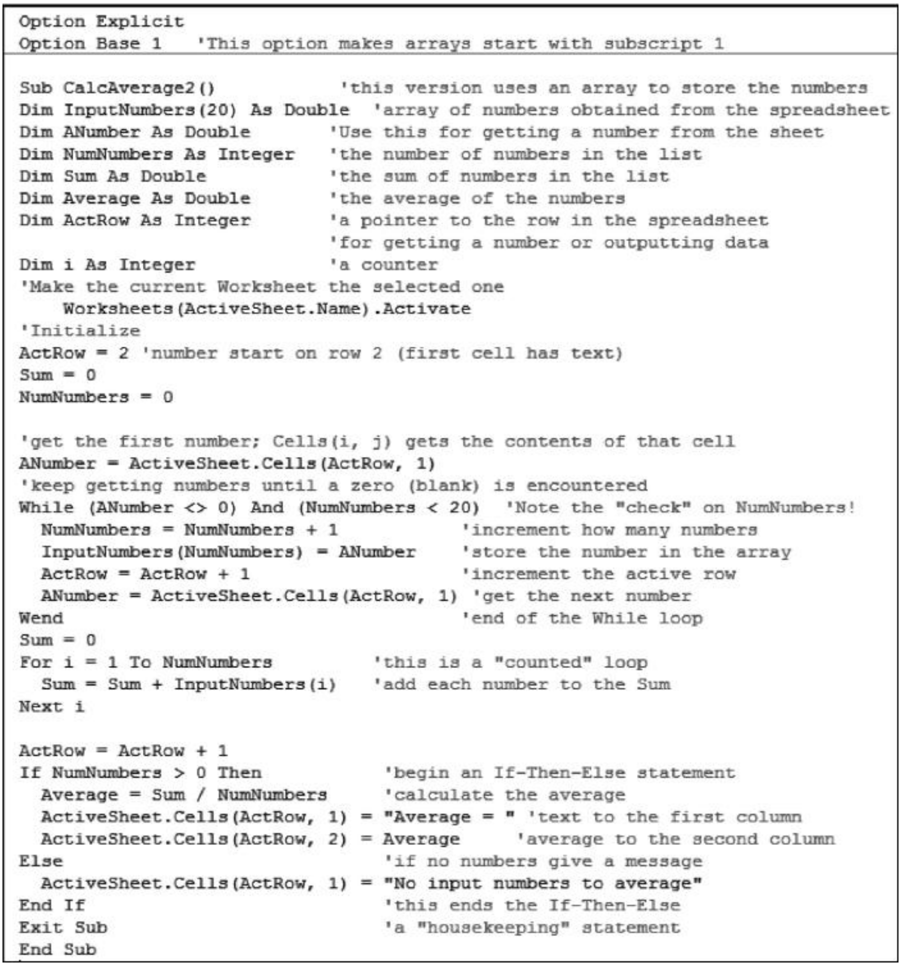
For a two-dimensional array (matrix), an example declaration is as follows:

```
Dim y (3, 4)
```

The matrix y has 3 rows and 4 columns (assuming subscripts start at 1).

Example Program 2.2: Using an Array

The following is a modified version of the program of Example Program 2.1. Here an Array is used to store the numbers to the averaged. A "For" loop is then used to compute the Sum of the numbers. An additional "check" is included to be sure that the number of data items does not exceed 20, which is an arbitrary hard-coded dimension for the array InputNumbers.



```
Option Explicit
Option Base 1   'This option makes arrays start with subscript 1

Sub CalcAverage2()           'this version uses an array to store the numbers
Dim InputNumbers(20) As Double  'array of numbers obtained from the spreadsheet
Dim ANumber As Double        'Use this for getting a number from the sheet
Dim NumNumbers As Integer    'the number of numbers in the list
Dim Sum As Double            'the sum of numbers in the list
Dim Average As Double        'the average of the numbers
Dim ActRow As Integer        'a pointer to the row in the spreadsheet
                             'for getting a number or outputting data
Dim i As Integer             'a counter
'Make the current Worksheet the selected one
    Worksheets(ActiveSheet.Name).Activate
'Initialize
ActRow = 2 'number start on row 2 (first cell has text)
Sum = 0
NumNumbers = 0

'get the first number; Cells(i, j) gets the contents of that cell
ANumber = ActiveSheet.Cells(ActRow, 1)
'keep getting numbers until a zero (blank) is encountered
While (ANumber <> 0) And (NumNumbers < 20)  'Note the "check" on NumNumbers!
  NumNumbers = NumNumbers + 1             'increment how many numbers
  InputNumbers(NumNumbers) = ANumber      'store the number in the array
  ActRow = ActRow + 1                     'increment the active row
  ANumber = ActiveSheet.Cells(ActRow, 1) 'get the next number
Wend                                      'end of the While loop
Sum = 0
For i = 1 To NumNumbers          'this is a "counted" loop
  Sum = Sum + InputNumbers(i)    'add each number to the Sum
Next i

ActRow = ActRow + 1
If NumNumbers > 0 Then            'begin an If-Then-Else statement
  Average = Sum / NumNumbers      'calculate the average
  ActiveSheet.Cells(ActRow, 1) = "Average = " 'text to the first column
  ActiveSheet.Cells(ActRow, 2) = Average     'average to the second column
Else                              'if no numbers give a message
  ActiveSheet.Cells(ActRow, 1) = "No input numbers to average"
End If                            'this ends the If-Then-Else
Exit Sub                          'a "housekeeping" statement
End Sub
```

2.13.1 Array Arguments

Each of the next two versions of the example program uses a subroutine to do some of the work. The subroutine is named `Summer` and is declared as follows:

```
Sub Summer(N, Nums, Avg)
```

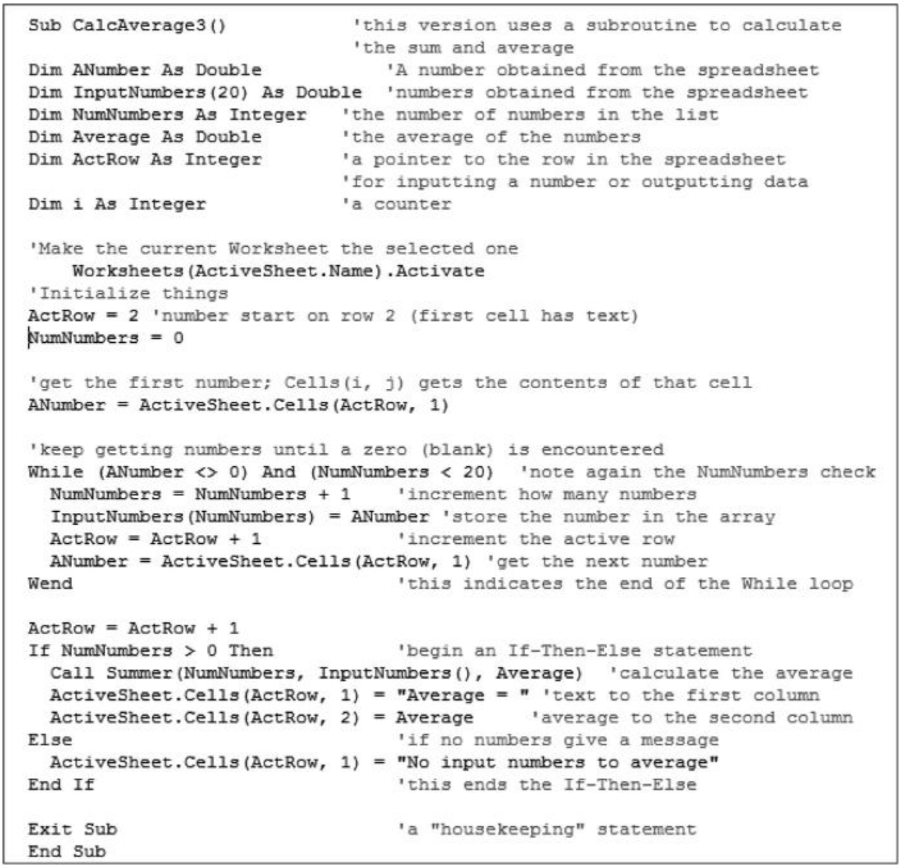
The “dummy” parameters `N`, `Nums`, and `Avg` do not appear in a `Dim` statement. They “inherit” the data type and structure of the actual argument when the subroutine is called. The calling statement in the next two examples is

```
Call Summer(NumNumbers, InputNumbers(), Average)
```

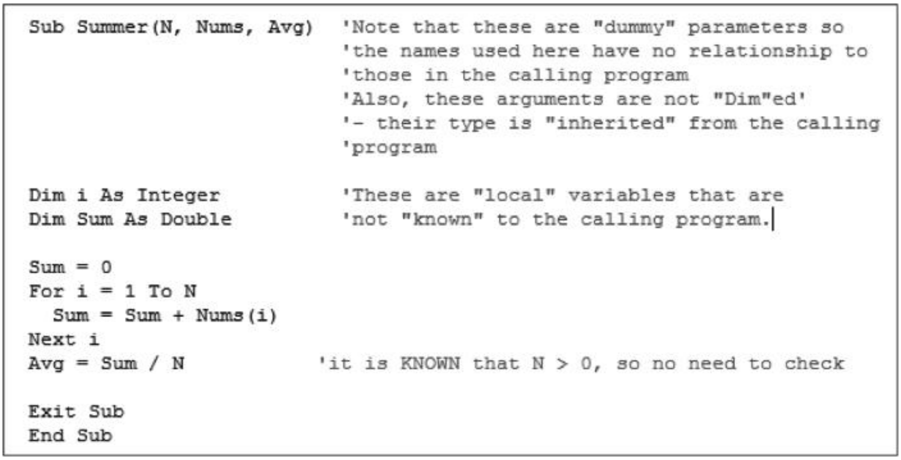
The empty parentheses after `InputNumbers` identify it as an array argument, and the dummy `Nums` becomes an array structure of whatever length `InputNumbers` happens to be. These parentheses are not essential but they help to document the array argument.

Example Program 2.3: Using a Subroutine

The same program is changed again, this time using a Sub with arguments to perform the summing and averaging operations:



```
Sub CalcAverage3()            'this version uses a subroutine to calculate
                              'the sum and average
Dim ANumber As Double           'A number obtained from the spreadsheet
Dim InputNumbers(20) As Double  'numbers obtained from the spreadsheet
Dim NumNumbers As Integer     'the number of numbers in the list
Dim Average As Double         'the average of the numbers
Dim ActRow As Integer         'a pointer to the row in the spreadsheet
                              'for inputting a number or outputting data
Dim i As Integer              'a counter

'Make the current Worksheet the selected one
    Worksheets(ActiveSheet.Name).Activate
'Initialize things
ActRow = 2 'number start on row 2 (first cell has text)
NumNumbers = 0

'get the first number; Cells(i, j) gets the contents of that cell
ANumber = ActiveSheet.Cells(ActRow, 1)

'keep getting numbers until a zero (blank) is encountered
While (ANumber <> 0) And (NumNumbers < 20)  'note again the NumNumbers check
  NumNumbers = NumNumbers + 1     'increment how many numbers
  InputNumbers(NumNumbers) = ANumber 'store the number in the array
  ActRow = ActRow + 1             'increment the active row
  ANumber = ActiveSheet.Cells(ActRow, 1) 'get the next number
Wend                              'this indicates the end of the While loop

ActRow = ActRow + 1
If NumNumbers > 0 Then            'begin an If-Then-Else statement
  Call Summer(NumNumbers, InputNumbers(), Average)  'calculate the average
  ActiveSheet.Cells(ActRow, 1) = "Average = " 'text to the first column
  ActiveSheet.Cells(ActRow, 2) = Average     'average to the second column
Else                              'if no numbers give a message
  ActiveSheet.Cells(ActRow, 1) = "No input numbers to average"
End If                            'this ends the If-Then-Else

Exit Sub                          'a "housekeeping" statement
End Sub
```



```
Sub Summer(N, Nums, Avg)  'Note that these are "dummy" parameters so
                          'the names used here have no relationship to
                          'those in the calling program
                          'Also, these arguments are not "Dim"ed'
                          '- their type is "inherited" from the calling
                          'program

Dim i As Integer          'These are "local" variables that are
Dim Sum As Double         'not "known" to the calling program.

Sum = 0
For i = 1 To N
  Sum = Sum + Nums(i)
Next i
Avg = Sum / N            'it is KNOWN that N > 0, so no need to check

Exit Sub
End Sub
```

2.13.2 Dynamic Arrays in VBA

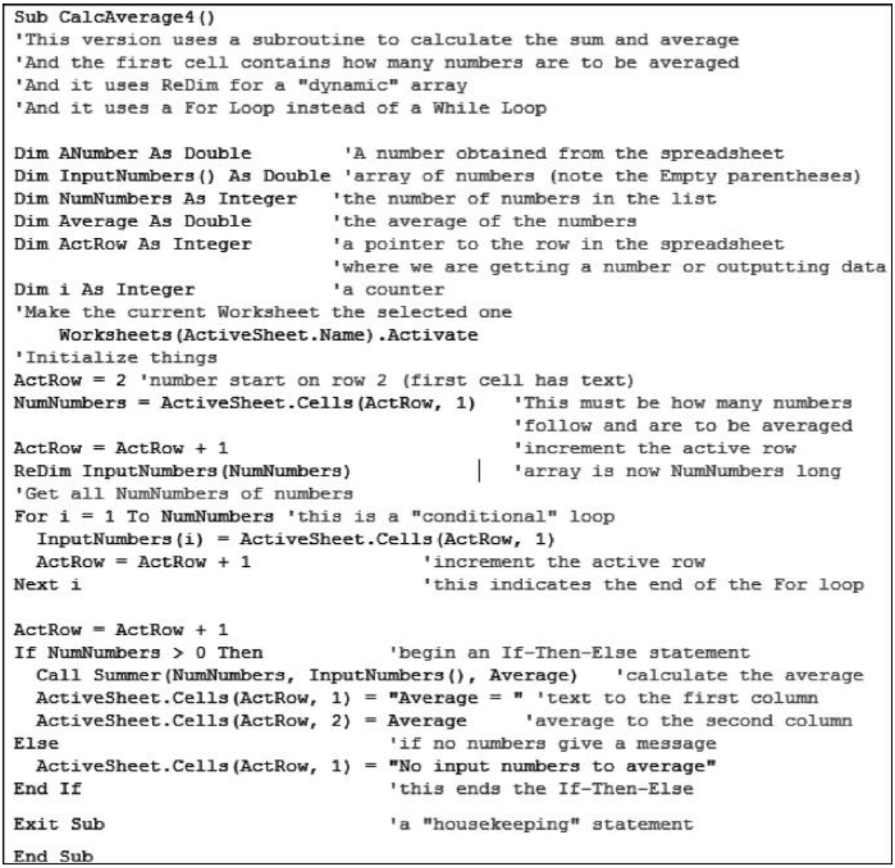
It is often the case that it is not known ahead of time how large an array's dimensions need to be. It is wasteful of memory (and poor programming practice) to simply make the dimensions arbitrarily large. To circumvent this problem, VBA provides the `ReDim` statement so that the dimensions of an array do not have to be "hard coded" in the program.

Example Program 2.4: Use of `ReDim`

To illustrate the use of `ReDim`, one more version of the example program is shown. The input data format is changed so that the very first number in the list is not one of the numbers to be averaged, but it indicates how many numbers there are (and thus the required size of the array that holds the numbers). In this version, the dimension of `InputNumbers` is left *blank* (note the empty parentheses), and then the following statement makes the dimension what it needs to be:

```
ReDim InputNumbers(NumNumbers)
```

Another approach that does not require the number of data items known in advance is to again use a `While` loop to detect a zero or sentinel value and put a `ReDim` statement within the `While` loop. When designing a program, it is wise to think of different ways in which to accomplish the same task.



```
Sub CalcAverage4()
'This version uses a subroutine to calculate the sum and average
'And the first cell contains how many numbers are to be averaged
'And it uses ReDim for a "dynamic" array
'And it uses a For Loop instead of a While Loop

Dim ANumber As Double          'A number obtained from the spreadsheet
Dim InputNumbers() As Double 'array of numbers (note the Empty parentheses)
Dim NumNumbers As Integer    'the number of numbers in the list
Dim Average As Double        'the average of the numbers
Dim ActRow As Integer        'a pointer to the row in the spreadsheet
                             'where we are getting a number or outputting data
Dim i As Integer             'a counter
'Make the current Worksheet the selected one
    Worksheets(ActiveSheet.Name).Activate
'Initialize things
ActRow = 2 'number start on row 2 (first cell has text)
NumNumbers = ActiveSheet.Cells(ActRow, 1)    'This must be how many numbers
                                             'follow and are to be averaged
ActRow = ActRow + 1                          'increment the active row
ReDim InputNumbers(NumNumbers)          |    'array is now NumNumbers long
'Get all NumNumbers of numbers
For i = 1 To NumNumbers 'this is a "conditional" loop
  InputNumbers(i) = ActiveSheet.Cells(ActRow, 1)
  ActRow = ActRow + 1                'increment the active row
Next i                               'this indicates the end of the For loop

ActRow = ActRow + 1
If NumNumbers > 0 Then            'begin an If-Then-Else statement
  Call Summer(NumNumbers, InputNumbers(), Average)   'calculate the average
  ActiveSheet.Cells(ActRow, 1) = "Average = " 'text to the first column
  ActiveSheet.Cells(ActRow, 2) = Average      'average to the second column
Else                              'if no numbers give a message
  ActiveSheet.Cells(ActRow, 1) = "No input numbers to average"
End If                            'this ends the If-Then-Else

Exit Sub                          'a "housekeeping" statement

End Sub
```

2.14 ALTERNATIVE I/O METHODS

Using the `Cells` method is straightforward and the one that is most often used for input and output. There are other methods worthy of mention, however.

2.14.1 Using Range.Select

The `Range.Select` combined with the `ActiveCell` object can accomplish essentially the same thing that `Cells` does. Here is an example:

```
Range("b10").Select
v = ActiveCell.Value
```

This inputs into the variable `v` whatever value is in cell `B10`. The `.Value` method is actually the default for `ActiveCell` and can be omitted. This technique of input is more unwieldy than the `Cells` method.

2.14.2 Using Message Box

The built-in function `MsgBox` can be used to send a message to the user without it appearing in a worksheet cell. Here is an example:

```
MsgBox "This program finds the average of a list of numbers"
```

This pops up a box containing the quoted message overlaid onto the active worksheet.

2.14.3 Using Input Box

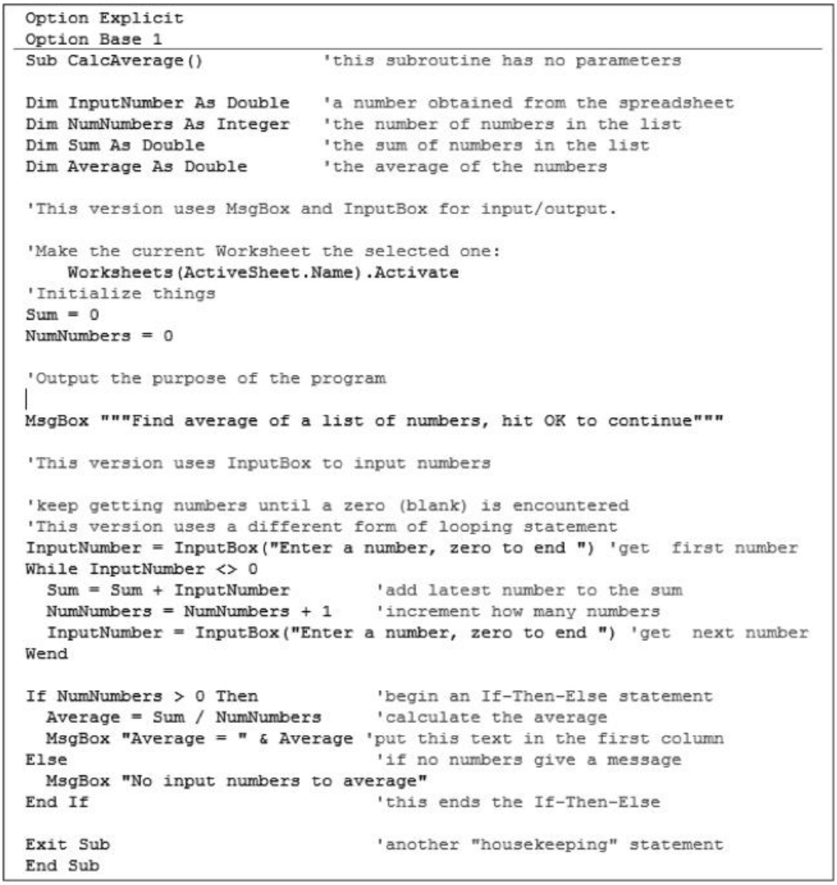
The built-in function `InputBox` can be used to both send a message to the user and to obtain data from the user without it appearing in a worksheet cell. Consider the example:

```
a = InputBox("Please enter the first value to be added")
```

A box pops up on the worksheet with the quoted prompt and a place for the user to type in the value requested (in this case, for the variable `a`).

Example Program 2.5: Using `MsgBox` and `InputBox` for Input/Output

Another version of the example program for finding the average of a list of numbers is shown below. In this case, `MsgBox` and `InputBox` are used for input and output. Note that three consecutive quote marks are required for one quote mark to appear.

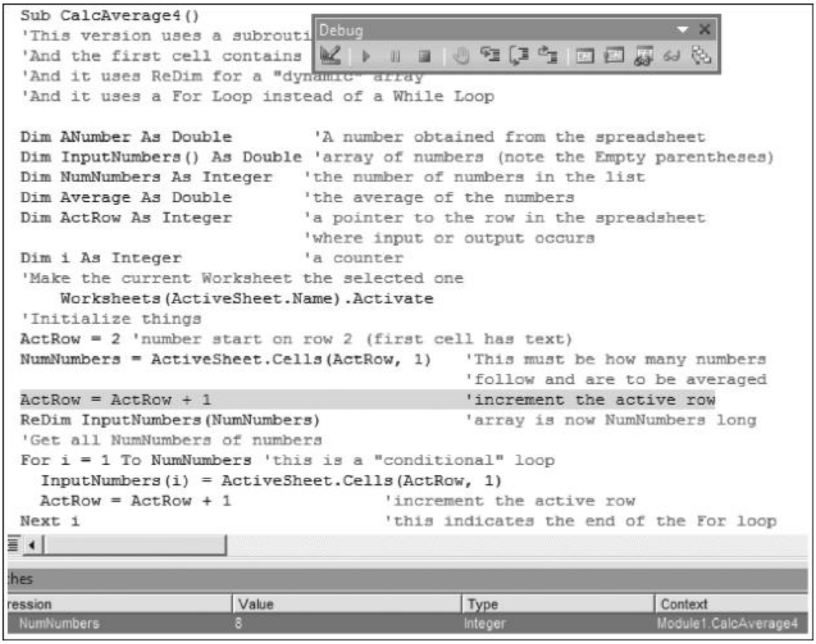


```
Option Explicit
Option Base 1
Sub CalcAverage()            'this subroutine has no parameters

Dim InputNumber As Double    'a number obtained from the spreadsheet
Dim NumNumbers As Integer    'the number of numbers in the list
Dim Sum As Double            'the sum of numbers in the list
Dim Average As Double        'the average of the numbers

'This version uses MsgBox and InputBox for input/output.

'Make the current Worksheet the selected one:
    Worksheets(ActiveSheet.Name).Activate
'Initialize things
Sum = 0
NumNumbers = 0

'Output the purpose of the program

MsgBox """Find average of a list of numbers, hit OK to continue"""

'This version uses InputBox to input numbers

'keep getting numbers until a zero (blank) is encountered
'This version uses a different form of looping statement
InputNumber = InputBox("Enter a number, zero to end ") 'get  first number
While InputNumber <> 0
  Sum = Sum + InputNumber         'add latest number to the sum
  NumNumbers = NumNumbers + 1     'increment how many numbers
  InputNumber = InputBox("Enter a number, zero to end ") 'get  next number
Wend

If NumNumbers > 0 Then            'begin an If-Then-Else statement
  Average = Sum / NumNumbers      'calculate the average
  MsgBox "Average = " & Average 'put this text in the first column
Else                              'if no numbers give a message
  MsgBox "No input numbers to average"
End If                            'this ends the If-Then-Else

Exit Sub                          'another "housekeeping" statement
End Sub
```

The & symbol is the “concatenation” operator. It appends the value of the variable `Average` to the preceeding text.

2.15 USING DEBUGGER

The Debugger allows many operations that assist in finding errors in programs or in the input data. When opening an Excel spreadsheet file that contains a VBA macro then going to Tools/Macros and Edit, the Debug menu is among those that appear. By selecting from the Debug menu `Step Into`, the `Sub` statement is highlighted (in yellow). The program is now running but under *your* control. To view the Debug Toolbar, go to View/Toolbars and select Debug. The following is what the screen looks like for the `CalcAverage4` VBA example:



```
Sub CalcAverage4()
'This version uses a subrouti
'And the first cell contains
'And it uses ReDim for a "dynamic" array
'And it uses a For Loop instead of a While Loop

Dim ANumber As Double        'A number obtained from the spreadsheet
Dim InputNumbers() As Double 'array of numbers (note the Empty parentheses)
Dim NumNumbers As Integer   'the number of numbers in the list
Dim Average As Double       'the average of the numbers
Dim ActRow As Integer       'a pointer to the row in the spreadsheet
                            'where input or output occurs
Dim i As Integer            'a counter
'Make the current Worksheet the selected one
    Worksheets(ActiveSheet.Name).Activate
'Initialize things
ActRow = 2 'number start on row 2 (first cell has text)
NumNumbers = ActiveSheet.Cells(ActRow, 1)   'This must be how many numbers
                                            'follow and are to be averaged
ActRow = ActRow + 1                         'increment the active row
ReDim InputNumbers(NumNumbers)              'array is now NumNumbers long
'Get all NumNumbers of numbers
For i = 1 To NumNumbers 'this is a "conditional" loop
  InputNumbers(i) = ActiveSheet.Cells(ActRow, 1)
  ActRow = ActRow + 1                'increment the active row
Next i                               'this indicates the end of the For loop
```

hes

ression	Value	Type	Context
NumNumbers	8	Integer	Module1.CalcAverage4

To single step through the program, use `F8` or the Step tool button, which is the one just to the right of the "Hand" icon in the Debug Toolbar. Here is what the screen looks like after stepping down to the `ReDim` statement. Here the Debug/Add Watch menu was used to display at the bottom the values of the variables `ActRow` and `NumNumbers`.

From the Debug menu, several useful operations can be invoked to aid in tracking down a programming error. For example, a "breakpoint" can be placed on any statement so that when Run (instead of single stepping) is chosen, execution halts when that statement with a breakpoint is reached. At that time, the value of any variable can be viewed. Note that in addition to using the Set Watch feature, values of a scalar variable can be observed simply by placing the cursor over the variable's name.

Using the Debugger can provide a powerful tool in tracking down programming errors and errors in input data. This discussion has only touched on a few features of the Debugger.

Example Program 2.6: Modified False Position

The method of false position is a "hybrid" of the bisection and secant methods. This method assumes two starting points having opposite function signs. A new point is found by drawing a straight line between the bracketing points. One of the old points is *discarded* depending on the sign of the function at the new point. This is depicted in Figure 2.4.

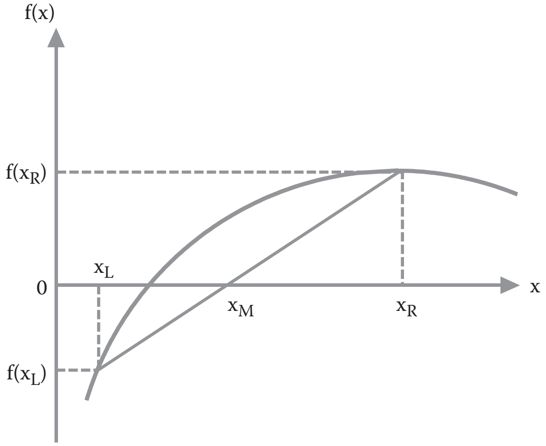


FIGURE 2.4 Modified false position method.

The formula for finding improved values for *x* using the notation of Figure 2.4 is as follows:

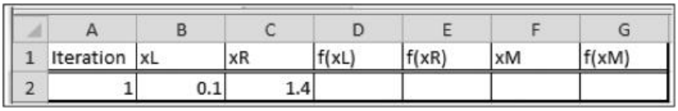
$$x_m = x_L - \frac{(x_R - x_L)f(x_L)}{f(x_R) - f(x_L)}\,. \tag{2.1}$$

An inefficiency of the method of false position is that one of the end points tends to become *stagnant*. That is, the end point remains unchanged for several iterations. A modification to overcome stagnation is to detect when it occurs and to take action to avoid it. An effective modification is to simply *halve* the stagnated *f*(*x*) value in the update formula. While the modified false position method can be implemented directly in Excel, the logic required is sufficiently complex to make using VBA a better choice. Program specification, algorithm design, coding, and testing are now given where the problem is to find the root between 0.4 and 1.2 of the function

$$f(x) = \tan x - x - 0.5 \tag{2.2}$$

SPECIFICATION

It is necessary to place on the spreadsheet user interface the two initial values of x that bracket the root. If the initial values do not bracket the root, the program should terminate with an error message. It is desired that at each iteration the values of x_L, x_R, $f(x_L)$, $f(x_R)$, x_m, and $f(x_m)$ be displayed along with an iteration counter. Shown below is one way the spreadsheet might appear before program execution:



	A	B	C	D	E	F	G
1	Iteration	xL	xR	f(xL)	f(xR)	xM	f(xM)
2	1	0.1	1.4				

The program then computes and displays the remaining values for Iteration 1. The update formula, Equation 2.1, is used to compute x_m. Next, either x_L or x_R is replaced by x_m (based on the sign of $f(x_m)$), and a new iteration is performed. When the absolute value of the difference between x_L and x_R is less than a tolerance (1.E-8), the following message is shown:

```
Solution is within tolerance
```

To prevent continued iterations when convergence is very slow, a maximum number of iterations (100) is invoked, and the following error message is displayed:

```
Maximum iteration reached; Solution might not be valid
```

If the initial values of x_L and x_R do not bracket a root, the following error message is displayed:

```
Root is not bracketed, please try again
```

ALGORITHM DESIGN

An "outline" summary of the logic for this program is as follows:

1. Get input data and fill out the first line of the output.
2. Compute x_m and $f(x_m)$.
3. If $f(x_m)$ has the same sign as $f(x_R)$, then replace x_R with x_m and set a repetition counter for x_R to zero while incrementing a repetition counter for x_L.
4. Otherwise, $f(x_m)$ has the same sign as $f(x_L)$ so replace x_L with x_m and set a repetition counter for x_L to zero while incrementing a repetition counter for x_R.
5. Output a line on the spreadsheet.

This process is repeated until the absolute value of the difference between x_L and x_R is less than the tolerance or the maximum number of iterations has occurred (Figure 2.5).

Coding could be attempted using the brief outline summary of the algorithm, but by using a structure chart, even more detail can be displayed and envisioned unambiguously prior to coding. Such a structure chart appears below, accompanied by a list of variable names used within the chart.

DICTIONARY OF VARIABLES

Name	Definition
Tol	Tolerance to detect small differences between x values
iMax	Maximum iteration before terminating with error message
xL	One of the x values
xR	The other of the x values
fxL	f(xL)
fxR	f(xR)

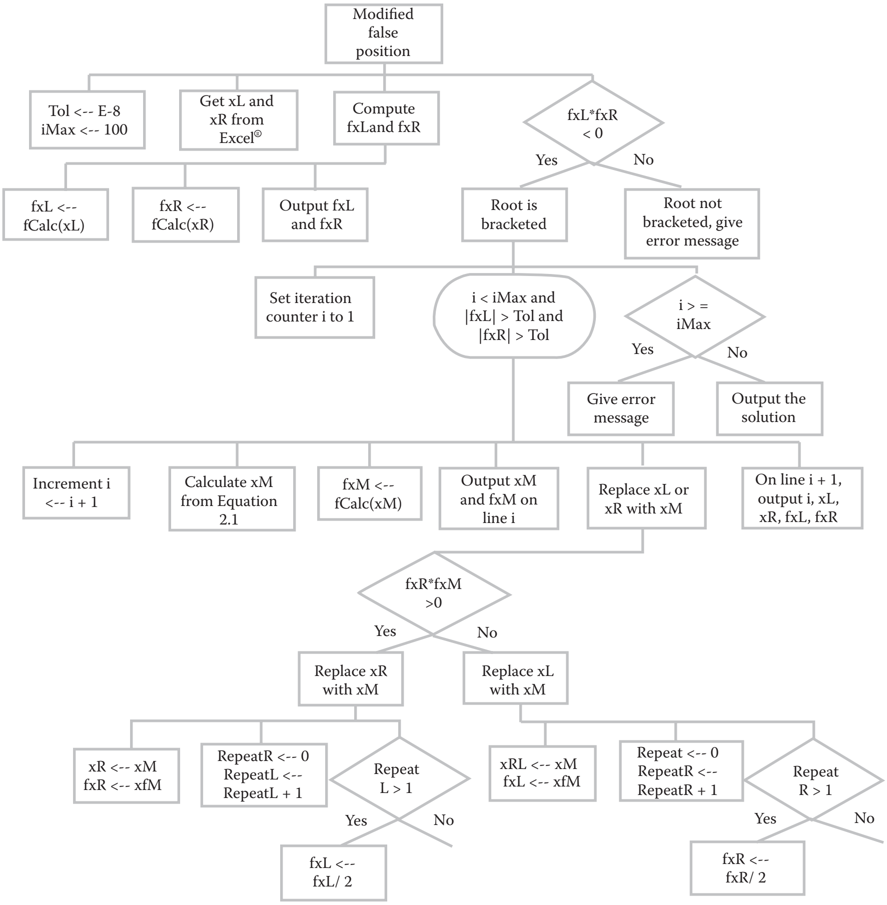


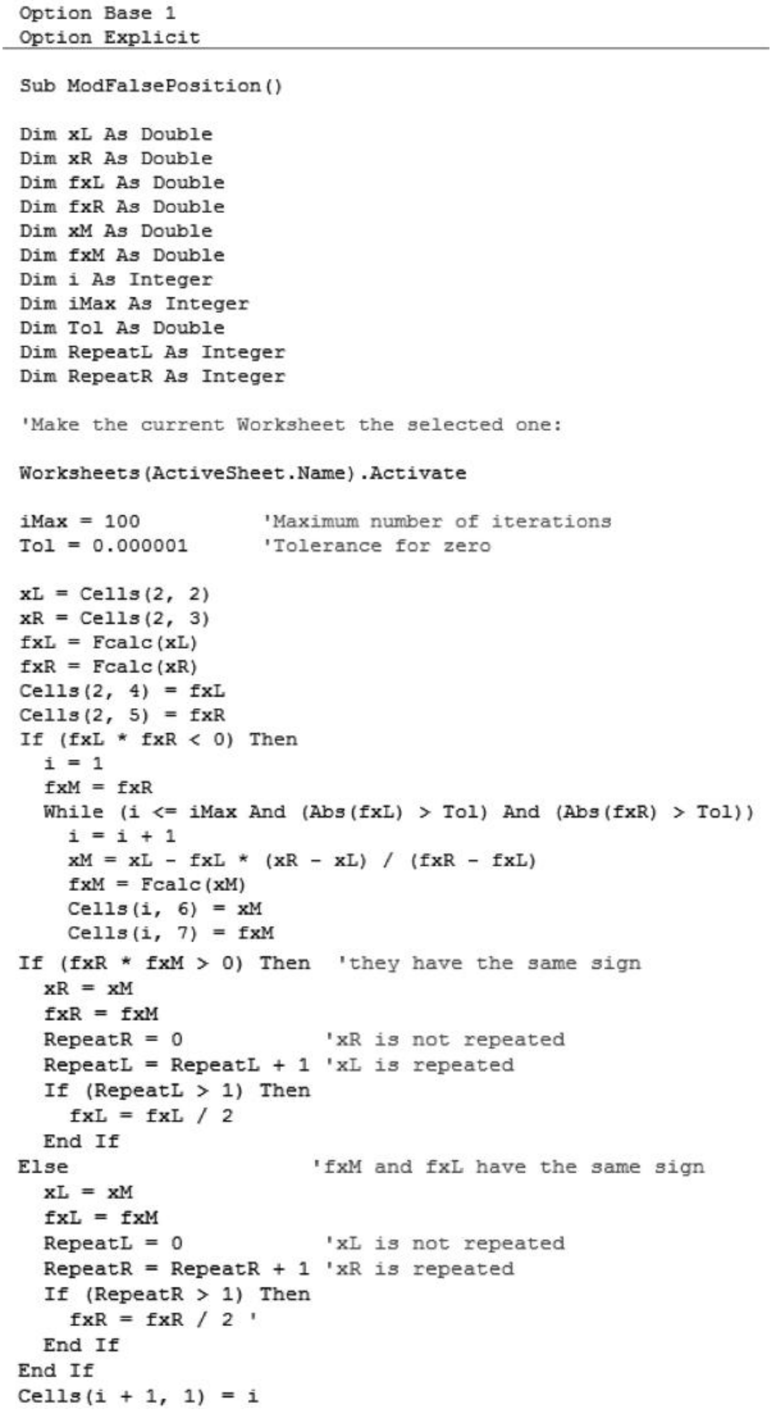
FIGURE 2.5 Algorithm for modified false position.

DICTIONARY OF VARIABLES (cont.)

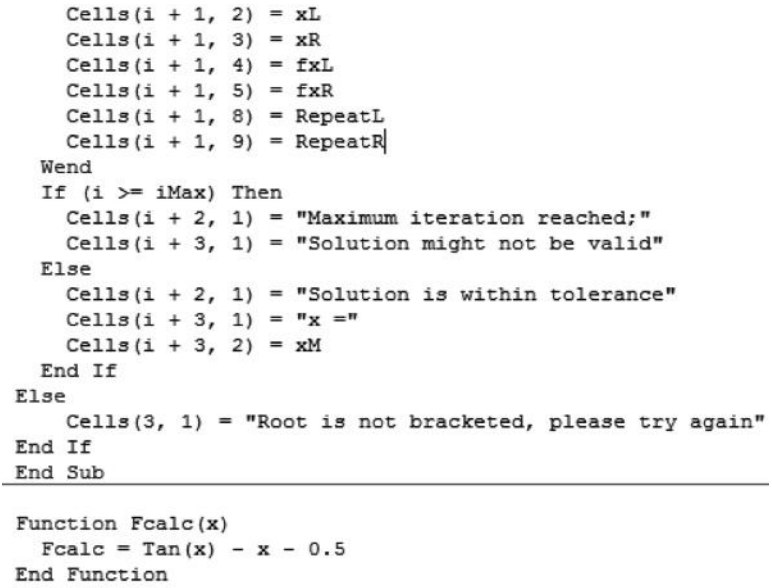
`i`	`Iteration counter and row counter for output`
`fCalc`	`Function subprogram that computes the function to zero`
`xM`	`Updated x value from Equation 2.1`
`fxM`	`f(xM)`
`RepeatR`	`Repetition indicator for xR`
`RepeatL`	`Repetition indicator for xL`

VBA CODE

Shown below is the VBA code that matches the logic of the structure chart. At the end of the code is the Function subprogram `fCalc` that provides the value of the function whose root is desired.



```
Option Base 1
Option Explicit

Sub ModFalsePosition()

Dim xL As Double
Dim xR As Double
Dim fxL As Double
Dim fxR As Double
Dim xM As Double
Dim fxM As Double
Dim i As Integer
Dim iMax As Integer
Dim Tol As Double
Dim RepeatL As Integer
Dim RepeatR As Integer

'Make the current Worksheet the selected one:

Worksheets(ActiveSheet.Name).Activate

iMax = 100         'Maximum number of iterations
Tol = 0.000001     'Tolerance for zero

xL = Cells(2, 2)
xR = Cells(2, 3)
fxL = Fcalc(xL)
fxR = Fcalc(xR)
Cells(2, 4) = fxL
Cells(2, 5) = fxR
If (fxL * fxR < 0) Then
  i = 1
  fxM = fxR
  While (i <= iMax And (Abs(fxL) > Tol) And (Abs(fxR) > Tol))
    i = i + 1
    xM = xL - fxL * (xR - xL) / (fxR - fxL)
    fxM = Fcalc(xM)
    Cells(i, 6) = xM
    Cells(i, 7) = fxM
If (fxR * fxM > 0) Then  'they have the same sign
  xR = xM
  fxR = fxM
  RepeatR = 0           'xR is not repeated
  RepeatL = RepeatL + 1 'xL is repeated
  If (RepeatL > 1) Then
    fxL = fxL / 2
  End If
Else                   'fxM and fxL have the same sign
  xL = xM
  fxL = fxM
  RepeatL = 0           'xL is not repeated
  RepeatR = RepeatR + 1 'xR is repeated
  If (RepeatR > 1) Then
    fxR = fxR / 2 '
  End If
End If
Cells(i + 1, 1) = i
```

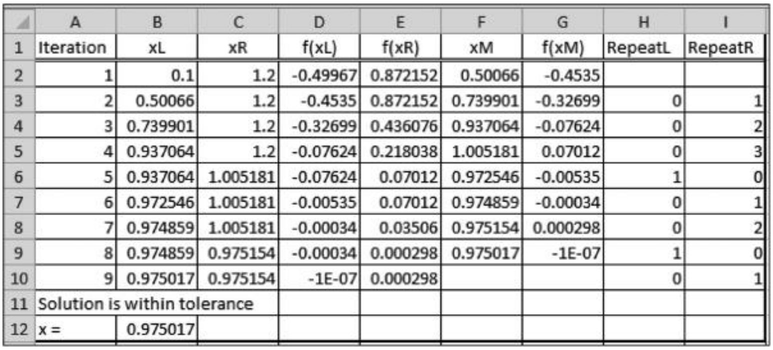


```
        Cells(i + 1, 2) = xL
        Cells(i + 1, 3) = xR
        Cells(i + 1, 4) = fxL
        Cells(i + 1, 5) = fxR
        Cells(i + 1, 8) = RepeatL
        Cells(i + 1, 9) = RepeatR
      Wend
      If (i >= iMax) Then
        Cells(i + 2, 1) = "Maximum iteration reached;"
        Cells(i + 3, 1) = "Solution might not be valid"
      Else
        Cells(i + 2, 1) = "Solution is within tolerance"
        Cells(i + 3, 1) = "x ="
        Cells(i + 3, 2) = xM
      End If
    Else
        Cells(3, 1) = "Root is not bracketed, please try again"
    End If
    End Sub
```

```
    Function Fcalc(x)
      Fcalc = Tan(x) - x - 0.5
    End Function
```

TESTING

The spreadsheet after executing the VBA program appears as follows:



	A	B	C	D	E	F	G	H	I
1	Iteration	xL	xR	f(xL)	f(xR)	xM	f(xM)	RepeatL	RepeatR
2	1	0.1	1.2	-0.49967	0.872152	0.50066	-0.4535		
3	2	0.50066	1.2	-0.4535	0.872152	0.739901	-0.32699	0	1
4	3	0.739901	1.2	-0.32699	0.436076	0.937064	-0.07624	0	2
5	4	0.937064	1.2	-0.07624	0.218038	1.005181	0.07012	0	3
6	5	0.937064	1.005181	-0.07624	0.07012	0.972546	-0.00535	1	0
7	6	0.972546	1.005181	-0.00535	0.07012	0.974859	-0.00034	0	1
8	7	0.974859	1.005181	-0.00034	0.03506	0.975154	0.000298	0	2
9	8	0.974859	0.975154	-0.00034	0.000298	0.975017	-1E-07	1	0
10	9	0.975017	0.975154	-1E-07	0.000298			0	1
11	Solution is within tolerance								
12	x =	0.975017							

It can be detected from the output that `xR` tends to be stagnant. Each time `RepeatR` is greater than 1, the `fxR` value is halved in Equation 2.1. This has a marked effect on the rate of convergence. Exercise 2.1 at the end of this chapter involves programming the unmodified false position method. Upon executing that program for the same test problem as shown in this example, the stagnation of `xR` will be observed.

EXERCISES

Exercise 2.1: Further test the VBA program of Example 2.6 as follows:

a. Put in two initial points that *do not* bracket the roots. This should produce the appropriate error message.

b. Test the program using different initial points. Some suggested ones are (0.1, 1.5) and (0.01, 1.55).

c. Test the program for the van der Waals problem as given in Example 1.1 of Chapter 1. Choose two initial points from the graph of the function shown in Figure 1.1. Shown below is one way to alter the Function Subprogram `Fcalc` for this problem:

```
Function Fcalc (V)
Dim R, T, P,  Tc, PC ,a, b As Double

  R = 0.08206
  T = 250
  P = 10
  Tc = 407.5
  Pc = 111.3
  a = 4.238448175
  b = 0.037555537

  Fcalc = (P + a / V ^ 2) * (V - b) - R * T

End Function
```

Exercise 2.2: In order to appreciate the increased efficiency of the modified false position method, redesign and reprogram Example 2.6 so that the original false position method is implemented.

a. Compare the output of the program for the same test function as Example 2.6 and using the same initial points with that of Example 2.6.

b. Repeat part a using (0.01, 1.55) as the initial point. Does the algorithm converge?

c. Repeat this exercise using the van der Waals function as described in Exercise 2.1c.

Exercise 2.3: Write a VBA program (Macro) to automate the process of finding x_L and x_R that give function values with opposite signs. Use the function of Example 2.6 for testing the program. This can be done in a wide variety of ways, but here is a suggested method:

a. Prepare a spreadsheet that looks like the following:

xL	xR	fxL	fxR
0.001	0.005		

The program "reads" the values of `xL` and `xR` and computes the associated function (that should be zero at a solution). The program displays these two function values. If the functions have opposite signs, the program terminates; otherwise

b. Use the `InputBox` feature to prompt for a new `xR` value. This new value replaces the original one in the spreadsheet and again displays the two function values. This is all displayed on the next line in the spreadsheet. The process continues until the program terminates with function values having opposite signs. Note that `xR` values less than `xL` values are possible.

Exercise 2.4: Design and Code a VBA program to perform the following tasks:

a. Generate a list of 20 numbers between 1 and 100 using the Excel function `RANDBETWEEN`. Select these numbers and copy them (by value) to another column (the reason for this is that `RANDBETWEEN` updates the list each time an operation is performed in the spreadsheet).
b. Input the list of numbers generated in part a. Any method can be used to indicate the "end of data," such as a blank cell as used in Example 1.1.
c. Compute the following "statistics" for the input numbers:
   1. Average
   2. Standard deviation
   3. Largest in absolute value
   4. Smallest in absolute value

   Necessary formulas are as follows:

$$\text{average} = \bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

$$\text{standard deviation} = s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}\left(x_i - \bar{x}\right)^2}$$

Check your results by using the Excel functions `AVERAGE`, `STDEV`, `MAX`, and `MIN`.

Exercise 2.5: A very inefficient way to compute the value of π is as follows:

Consider a circle of radius ½ circumscribed by a unit square (each side is 1 unit). Recall that the equation of a circle is

$$x^2 + y^2 = r^2$$

If N random values are generated for x and y between 0 and ½, the number of times the random coordinates are within the circle ($x^2 + y^2 \le r^2$) can be "tallied." Call this N_{in}. Then the area of the circle can be estimated as

$$A = N_{in}/N$$

It is also known that

$$A = \pi r^2$$

So, π can be estimated as

$$\pi = (N_{in}/N)/r^2 = (N_{in}/N)/0.25 = 4(N_{in}/N)$$

Write a VBA Macro to “read” N from the spreadsheet and output the estimated value of π. Each time the Macro is run, a different estimate of π appears since each time, different random numbers are generated. To “smooth” out the randomness of the estimate, for each N, repeat the execution of the Macro 10 times (the program should be altered to do this automatically) and then output only the average of the 10 estimates.

Repeat this exercise for $N = 10$, 100, 1000, and 10,000. Make a table of N versus the absolute value of the error in the π estimates and show these on a graph using a logarithmic scale for N.

Even though this is a very inefficient way to compute π, it makes a good problem for doing VBA programming.

By the way, the VBA function to generate a random number between 0 and 1 is RND.

Exercise 2.6: Sorting is a frequently used operation in many data processing applications. Excel has a powerful sorting tool, available from the Data/Sort menu. If it is desired to sort data under control of a VBA program, such code might need to be written. The so-called bubble sort algorithm is not a very efficient one, but it is easy to program and it easily will suffice for small data sets. The essence of the bubble sort method is as follows: Repeatedly step through the list to be sorted, comparing two items at a time and swapping them if they are in the wrong order. The process is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm gets its name from the way smaller elements (when sorting in a descending order) “bubble” to the top of the list. The following is a *pseudo-code* description of the process. Pseudo-code looks similar to VBA code, but no particular attention is paid to strict coding rules. The pseudo-code assumes that the array name is A and its length is N.

```
swapped = true          'swapped is a logical (boolean) variable
while swapped
  swapped = false       'temporarily assume the list is sorted
  for i =1 to N - 1
    if A(i) > A(i+1) then
      swap A(i) and A(i+1)
      swapped = true        'a swap indicates an unsorted list
    end if
  end for
end while
```

This logic could just as easily have been illustrated with a structure chart.

Generate a list of 20 numbers between 1 and 100 using the Excel function `RANDBETWEEN`. Select these numbers and copy them (by value) to another column (the reason for this is that `RANDBETWEEN` updates the list each time an operation is performed in the spreadsheet). Write a VBA program to input the copied numbers and sort them into descending order using bubble sort. Output the sorted numbers to the spreadsheet. Be sure to give an error message if no numbers are entered.

REFERENCES

Bowles, K., *Microcomputer Problem Solving Using Pascal*, Springer-Verlag, New York (1979).

Law, V.J., *ANSI Fortran 77: An Introduction to Structured Software Design*, Wm. C. Brown, Dubuque, Iowa (1983).

Law, V.J., *Standard Pascal: An Introduction to Structured Software Design*, Wm. C. Brown, Dubuque, Iowa (1985).

3 Linear Algebra and Systems of Linear Equations

3.1 INTRODUCTION

Linear algebra is a topic that many students of numerical methods will have been exposed to in mathematics classes. In this chapter, a brief review of linear algebra is given along with numerical methods for solving problems that are common in engineering and scientific applications. Linear algebra involves the manipulation of linear relationships and usually involves the use of vectors and matrices. The most common problem class of linear algebra is the solution of a set of linear algebraic equations.

3.2 NOTATION

- Scalars are indicated by a lowercase letter.
- Vectors are also identified by a lowercase letter. The context distinguishes between a scalar and a vector.
- Matrices are designated by a capital letter.
- By default, vectors are column vectors. To show a row vector, the transpose operator (a superscript T) is used (as in x^T).
- When necessary, the dimensions of matrices and vectors are shown by scalar subscripts. For example, $A_{m\times n}$ indicates a matrix with m rows and n columns. Further, $y_{n\times 1}$ designates a column vector of n elements.

Definition: The equation

$$ax + by + cz + dw = h \tag{3.1}$$

where a, b, c, d, and h are known numbers, while x, y, z, and w are unknown numbers (variables), is called a *linear equation*. If $h = 0$, the linear equation is said to be *homogenous*. A *linear system* is a set of linear equations, and a *homogenous linear system* is a set of homogenous linear equations.

For example,

$$\begin{aligned} 2x_1 - 3x_2 &= 1 \\ x_1 + 3x_2 &= -2 \end{aligned} \tag{3.2}$$

is a linear system. But

$$\begin{aligned} 2x_1 - 3x_2^2 &= -1 \\ x_1 + x_2 &= 1 \end{aligned} \tag{3.3}$$

is a nonlinear system (because of x_2^2).

The system

$$\begin{aligned} 2x_1 - 3x_2 - 3x_3 &= 0 \\ x_1 + 3x_2 &= 0 \\ x_1 - x_2 + x_3 &= 0 \end{aligned} \tag{3.4}$$

is a homogenous linear system.

Vectors and matrices offer a convenient, compact way of representing, manipulating, and solving linear systems. These are introduced in the next few sections.

3.3 VECTORS

A vector is an ordered set of numbers arranged as a column (the default). An m-element vector takes the form

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} \tag{3.5}$$

The use of square brackets to enclose the elements is common notation.

Note that lowercase letters (without subscripts) are used to represent the entire vector. Individual elements of a vector are subscripted according to their placement within the vector. For example, x_3 indicates the third element in the column vector x.

Note on vectors of physics and mechanics: Engineers are familiar with vectors as they appear in problems of physics and mechanics. For example, Newton's law of motion takes the form

$$\vec{F} = m\vec{a} \tag{3.6}$$

where $\vec{F}$ is the applied force vector, m is the mass of the object, and $\vec{a}$ is the acceleration vector. The overbar arrow notation is commonly used to distinguish vectors from scalars. The physical vectors are one-, two-, or three-dimensional since they represent quantities in physical space. There is no difference between these physical vectors and those of linear algebra, but the notation is slightly different and linear algebra vectors can be of any size. The vector $\vec{F}$ can be written in component form as

$$\vec{F} = \begin{bmatrix} F_x \\ F_y \\ F_z \end{bmatrix} \tag{3.7}$$

where each component represents the magnitude of the force in each of the three Cartesian coordinates. In summary, physical vectors are a special case (three-dimensional) of the more general mathematical vector concept.

3.4 VECTOR OPERATIONS

3.4.1 Vector Addition and Subtraction

Only vectors of the same size (same number of elements) and shape (column or row) can be added or subtracted. The shorthand notation

$$c = a + b$$

is equivalent to element by element addition:

$$c_i = a_i + b_i;\ i = 1, n \tag{3.8}$$

3.4.1.1 Multiplication by a Scalar

If a is a vector of length n and σ is a scalar, then

$$b = \sigma a$$

is equivalent to the following where each element of a is multiplied by σ:

$$b_i = \sigma a_i;\ i = 1, n \tag{3.9}$$

3.4.2 Vector Transpose

The transpose operator indicates an interchange of rows with columns and *vice versa*. If u is a (column) vector, then a row vector is indicated by u^T.

3.4.3 Linear Combinations of Vectors

A linear combination of two vectors involves the multiplication of each vector by a scalar and then adding the results. The shorthand notation

$$\alpha u + \beta v = w \tag{3.10}$$

can be summarized as follows:

$$\alpha \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_m \end{bmatrix} + \beta \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_m \end{bmatrix} = \begin{bmatrix} \alpha u_1 + \beta v_1 \\ \alpha u_2 + \beta v_2 \\ \vdots \\ \alpha u_m + \beta v_m \end{bmatrix} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix} \tag{3.11}$$

This concept can be extended to any number of scalars and vectors as long as all of the vectors have the same “shape” (row or column vector) and size.

3.4.4 Vector Inner Product

The vector inner product is the same as the familiar “dot” product of physical vectors. The following are equivalent notations, where x is an n-element vector and y is an n-element vector:

$$\sigma = x \cdot y = x^T y = y^T x = \sum_{i=1}^{n} x_i y_i \tag{3.12}$$

The inner product is always the result of multiplying a row vector by a column vector (the reverse is called the “outer” product and is discussed later). The result of the inner product is a scalar (a single number).

Example: Let $x = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$ and $y = \begin{bmatrix} 2 \\ 1 \\ 3 \end{bmatrix}$. Then

$$x \cdot y = x^T y = y^T x = 1 \cdot 2 + 2 \cdot 1 + 3 \cdot 3 = 13 \tag{3.13}$$

3.4.5 Vector Norm

The usual (L_2) norm of a vector is the square root of the sum of squares of elements. For a vector, v, of n elements,

$$\|v\| = \sqrt{\sum_{i=1}^{n} v_i^2} = \sqrt{v^T v} \tag{3.14}$$

A less useful norm, called the L_1 norm, is the sum of absolute values of the vector elements.

3.4.6 Orthogonal Vectors

The vector inner product has an interesting and useful geometric interpretation. The angle θ between two nonzero vectors *u* and *v* is given by the following (which is easily derived from the law of cosines for the two-dimensional case):

$$\cos\theta = \frac{u^T v}{\|u\|\|v\|} \tag{3.15}$$

If the angle is 90° (π/2 radians), then the inner product is zero, which in two or three dimensions means that the vectors are perpendicular to each other (also called *orthogonal* to each other). The concept of orthogonality can be extended to any number of dimensions. That is, two vectors *u* and *v* are orthogonal if

$$u^T v = 0 \tag{3.16}$$

3.4.7 Orthonormal Vectors

Orthonormal vectors are *unit* vectors (having a magnitude of 1) that are orthogonal to each other. Any vector can be converted to a unit vector by dividing by its L_2 norm:

$$\hat{u} = \frac{u}{\|u\|} \tag{3.17}$$

which is a unit vector in the direction of *u*. Note that since

$$\|u\| = \sqrt{u^T u} \tag{3.18}$$

it follows that $u^T u = 1$ if *u* is a unit vector.

3.5 MATRICES

A matrix is a two-dimensional array of numbers. It can also be viewed as a vector of vectors. Typically, matrix A with m rows and n columns is written as follows:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \tag{3.19}$$

An uppercase letter represents an entire matrix. Among the matrix elements, the first subscript is the row number and the second subscript is the column number.

If u, v, and w are n-element vectors, then a matrix B with three rows and n columns can be constructed from them as follows:

$$B = \begin{bmatrix} u^T \\ v^T \\ w^T \end{bmatrix} \tag{3.20}$$

Likewise, if d, e, and f are m-dimensional vectors, a matrix C with m rows and three columns can be formed from them as follows:

$$C = [d\ e\ f] \tag{3.21}$$

3.6 MATRIX OPERATIONS

3.6.1 Matrix Addition and Subtraction

This is defined only for matrices of exactly the same shape (same number of rows and columns). For addition of two $m \times n$ matrices A and B, the sum C is given by

$$\begin{aligned} &C = A + B \\ &c_{ij} = a_{ij} + b_{ij}; \quad i = 1,\cdots,m; \quad j = 1,\cdots,n \end{aligned} \tag{3.22}$$

3.6.2 Multiplication by a Scalar

If σ is a scalar, then the operation σA involves multiplying every element of A by σ.

3.6.3 Transposition of Matrices

When applied to a matrix, the transposition operator converts each row to a column (and conversely, each column into a row). That is, each row becomes a column in the resulting transposed matrix. This can be represented as follows, where A is an $n \times m$ matrix:

$$\begin{aligned} &B = A^T \\ &b_{ij} = a_{ji}; \quad i = 1, \cdots, m; \quad j = 1, \cdots, n \end{aligned} \tag{3.23}$$

Note that if A is $n \times m$, then B is $m \times n$.

3.6.4 Special Matrices

Square: $m = n$, same number of rows and columns
Diagonal: Square, only elements on the diagonal are nonzero
Identity: Diagonal, diagonal elements are all 1 (rest are 0)
Upper triangular: Usually square, all elements below the diagonal = 0
Lower triangular: Usually square, all elements above the diagonal = 0

3.6.5 Matrix Multiplication

Any two *conformable* matrices A and B can be multiplied in the order AB. A and B are conformable if the number of columns of A is the same as the number of rows of B. This is summarized as follows, where A is $n \times p$ and B is $p \times m$:

$$\begin{aligned} &C = AB \\ &c_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj}; \quad i = 1, \cdots, n; \quad j = 1, \cdots, m \end{aligned} \tag{3.24}$$

Another view is that the i, j-th element of C is the inner product of the ith row of A with the jth column of B.

Example 3.1: Matrix Multiplication

Given the following matrices and vectors:

$$A = \begin{bmatrix} 4 & 3 & 2 \\ -4 & 2 & -2 \\ 0 & -1 & 0 \\ -1 & 2 & 2 \end{bmatrix} \quad B = \begin{bmatrix} 1 & -3 \\ -2 & 2 \\ 3 & 4 \end{bmatrix} \quad x = \begin{bmatrix} -3 \\ 2 \\ 4 \end{bmatrix} \quad y = \begin{bmatrix} 1 & -2 & 3 \end{bmatrix}$$

the following are the results of various valid matrix multiplications:

$$AB = \begin{bmatrix} 4 & 2 \\ -14 & 8 \\ 2 & -2 \\ 1 & 15 \end{bmatrix} \quad Ax = \begin{bmatrix} 2 \\ 8 \\ -2 \\ 15 \end{bmatrix} \quad Ay^T = \begin{bmatrix} 4 \\ -14 \\ 2 \\ 1 \end{bmatrix}$$

$$yB = \begin{bmatrix} 14 & 5 \end{bmatrix} \quad x^T B = \begin{bmatrix} 5 & 29 \end{bmatrix}$$

3.6.6 Matrix Determinant

The determinant of a matrix is a scalar that provides significant information about the matrix. Determinants can be computed only for square matrices. The determinant of a 2 × 2 matrix is easy to compute and is defined as follows:

$$\det(A) = \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = a_{11}a_{22} - a_{12}a_{21} \tag{3.25}$$

Note that this is the product of the diagonal elements minus the product of the off-diagonal elements.

For a 3 × 3 matrix, there is a trick or shortcut for calculating the determinate. This involves duplicating the first two columns and then adding the products "to the right" and subtracting the products "to the left." The trick does not work for higher dimensional matrices. It is much easier to compute the determinant of a matrix by transforming it to an upper or lower triangular matrix (see below).

An important property of determinates is

$$\det(AB) = \det(A)\det(B) \tag{3.26}$$

That is, the determinate of a product is the product of determinates. This property is useful when finding the determinate of a matrix by numerical manipulations.

3.6.7 Matrix Inverse

The inverse is defined only for square matrices (same number of rows and columns). For an $n \times n$ matrix A, the *inverse* of A is designated A^{-1} and has the following properties:

$$A^{-1}A = AA^{-1} = I \quad \text{(the identity matrix)} \tag{3.27}$$

3.6.8 More Special Matrices

Symmetric matrices are square and have the property $A = A^T$. Another way of saying this is that for all i and j, $a_{ij} = a_{ji}$.

Tridiagonal matrices are square and have nonzero entries on the diagonal (elements with indices $i = j$), just above the diagonal (elements with indices $j = i + 1$), and just below the diagonal (elements with indices $j = i - 1$). Here is an example of a tridiagonal matrix:

$$\begin{bmatrix} 2 & -2 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -3 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix}$$

Orthogonal matrices have the powerful property that the transpose is the inverse. That is, for an orthogonal matrix Q, $Q^TQ = I$.

3.7 SOLVING SYSTEMS OF LINEAR ALGEBRAIC EQUATIONS

In matrix-vector form, a system of linear algebraic equations takes the form

$$Ax = b \tag{3.28}$$

where A is an $m \times n$ matrix, x is an n-vector, and b is an m-vector.

When $m = n$, the number of equations is the same as the number of unknowns and this is the usual case. If the inverse of A exists, then the solution can be written as

$$x = A^{-1}b \tag{3.29}$$

If A^{-1} does not exist, then, clearly, an associated linear system cannot be solved using the inverse. Another good reason to not use the inverse is that it is inefficient to compute the inverse and then multiply by the right-hand side vector, b. Later, a very general method is covered using the so-called singular value decomposition (SVD) that always produces a useful solution. A method, called Gaussian elimination, leads to a solution if one exists and also gives insight to those cases where a unique solution does not exist.

3.7.1 Gaussian Elimination

Gaussian elimination is most easily described using a specific example. Consider the linear system

$$\begin{aligned} x_1 + x_2 + x_3 &= 0 \\ x_1 - 2x_2 + 2x_3 &= 4 \\ x_1 + 2x_2 - x_3 &= 2 \end{aligned} \tag{3.30}$$

The idea behind Gaussian elimination is to *kill* (eliminate) one of the unknowns in the second and third equations (using the first) and then to eliminate another unknown from the third equation (using the second). This leaves the third equation with only one unknown. For the example, if the first equation is subtracted from the second and third equations, the result is

$$\begin{aligned} x_1 + x_2 + x_3 &= 0 \\ -3x_2 + x_3 &= 4 \\ x_2 - 2x_3 &= 2 \end{aligned} \tag{3.31}$$

Now, if the second equation is multiplied by 1/3 and the result added to the third, there results

$$\begin{aligned} x_1 + x_2 + x_3 &= 0 \\ -3x_2 + x_3 &= 4 \\ -\frac{5}{3}x_3 &= \frac{10}{3} \end{aligned} \tag{3.32}$$

The last equation now contains only x_3; solving gives $x_3 = -2$. Knowing x_3, from the second equation, $x_2 = -2$, and finally from the first equation, knowing x_3 and x_2 leads to $x_1 = 4$. Therefore, the linear system has one solution

$$x_1 = 4,\ x_2 = -2,\ x_3 = -2$$

Going from the last equation to the first while solving for the unknowns is called *backsolving* or *backsubstitution*. It is important to see that when multiplying one equation by a scalar and adding the result to another one, *equality is maintained.*

Another way to represent the steps of Gaussian elimination is to use an *augmented matrix,* which is designated as $[A|b]$, where *only the coefficients* of the equations are written (the right-most column contains values for the vector b). For the previous example, the augmented matrix is

$$\begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & -2 & 2 & 4 \\ 1 & 2 & -1 & 2 \end{bmatrix}$$

The same elementary row operations can be performed on this matrix as with the original equations. Keeping the first row and subtracting it from the second and third ones gives

$$\begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & -3 & 1 & 4 \\ 0 & 1 & -2 & 2 \end{bmatrix}$$

Then, keep the first and second equations, multiply the second by 1/3, and add the result to the third to get

$$\begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & -3 & 1 & 4 \\ 0 & 0 & -\frac{5}{3} & \frac{10}{3} \end{bmatrix}$$

If the variables were inserted to convert the transformed augmented matrix back into equation form, the last equation would involve only x_3 and can be solved for it. Then, proceed to backsubstitute for x_2 and x_1. Here is a summary of the Gaussian elimination procedure.

Gaussian elimination summary: Consider an $n \times n$ linear system.

1. Construct the augmented matrix for the system.
2. Use elementary row operations to transform the augmented matrix into an upper-triangular one.
3. Solve the last equation for the single variable x_n.
4. Complete the backsubstitution for all other variables.

DID YOU KNOW?

Carl Friedrich Gauss, the great German mathematician, did not discover what is called Gaussian elimination.

The method of Gaussian elimination appears in Chapter 8, *Rectangular Arrays*, of the important Chinese mathematical text *Jiuzhang suanshu* or *The Nine Chapters on the Mathematical Art*. Its use is illustrated in eighteen problems, with two to five equations. The first reference to the book by this title is dated to 179 CE, but parts of it were written as early as approximately 150 BCE. It was commented on by Liu Hui in the 3rd century.

The method in Europe stems from the notes of Isaac Newton. In 1670, he wrote that all the algebra books known to him lacked a lesson for solving simultaneous equations, which Newton then supplied. Cambridge University eventually published the notes as *Arithmetica Universalis* in 1707 long after Newton left academic life. The notes were widely imitated, which made (what is now called) Gaussian elimination a standard lesson in algebra textbooks by the end of the 18th century. Carl Friedrich Gauss in 1810 devised a notation for symmetric elimination that was adopted in the 19th century by professional hand computers to solve the normal equations of least-squares problems. The algorithm that is taught in high school was named for Gauss only in the 1950s as a result of confusion over the history of the subject.

Source: http://meyer.math.ncsu.edu/Meyer/PS_Files/GaussianEliminationHistory.pdf.

3.7.2 Determinant Revisited

Previously, the determinant for a 2 × 2 matrix was defined, and it was stated that it is easy to calculate the determinant for a 3 × 3 matrix. Existence, uniqueness, families of solutions, rank, and even the determinant are all determined via the Gaussian elimination process.

Note that each step in the Gaussian elimination process can be expressed as

$$M_iA \rightarrow B_i \tag{3.33}$$

where M_i is an identity matrix altered only with additional elements either below or above the diagonal (but not both). Recall Example 3.1. The steps of zeroing all elements below the first diagonal can be summarized as follows:

$$M_1A = \begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 1 & -2 & 2 \\ 1 & 2 & -1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & -3 & 1 \\ 0 & 1 & -2 \end{bmatrix} = B_1 \tag{3.34}$$

The matrix, M_1, on the left is a lower triangular transformation matrix and its determinant is 1 (this is easy to verify). One algebraic rule for determinants is that *the determinant of a product is the product of determinants*; therefore, the determinant of the transformed matrix (B_1) is the same as that of A, since $\det(M_1) = 1$.

The next step is to eliminate the remaining nonzero element below the diagonal as follows:

$$M_2M_1A = M_2B_1 \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1/3 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 0 & -3 & 1 \\ 0 & 1 & -2 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & -3 & 1 \\ 0 & 0 & -5/3 \end{bmatrix} = B_2$$

$$\det(A) = \det(B_2) = (1)(-3)(-5/3) = 5 \tag{3.35}$$

Therefore, when, via Gaussian elimination, matrix A has been transformed into an upper (or lower) triangular matrix, the value of the determinant has not changed. The determinant of a triangular matrix is simply the product of the diagonal elements (also easy to prove). So, $\det(A)$ can be computed simply by multiplying the diagonal elements of the final triangular matrix:

$$\det(A) = \det(M_{n-1}M_{n-2}\cdots M_1A) = \det(B) = \prod_{i=1}^{n} b_{ii} \tag{3.36}$$

where n is the number of rows and columns of A.

3.7.3 Gauss–Jordan Elimination

A method closely related to Gauss elimination is called the Gauss–Jordan algorithm. As a "bonus" (but it involves more work), the *inverse* of the matrix is also calculated. The basic idea behind the Gauss–Jordan method is to first form an augmented matrix consisting of the original system matrix and the identity matrix as follows:

$$[A \mid I]$$

Next, A is transformed into an identity matrix using elementary row operations (indicated by the matrix T) resulting in

$$T\,[\mathrm{A}|I] = [I|T] \tag{3.37}$$

This implies that

$$TA = I \tag{3.38}$$

or, in other words,

$$T = A^{-1} \tag{3.39}$$

Example 3.2: Gauss–Jordan Elimination

If the original square matrix, A, is given by the following expression:

$$A = \begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{bmatrix} \tag{3.40}$$

then, after augmentation by the identity matrix, the following is obtained:

$$[AI] = \begin{bmatrix} 2 & -1 & 0 & 1 & 0 & 0 \\ -1 & 2 & -1 & 0 & 1 & 0 \\ 0 & -1 & 2 & 0 & 0 & 1 \end{bmatrix} \tag{3.41}$$

By performing elementary row operations on the $[A|I]$ matrix until it is transformed into the identity matrix, the following form results:

$$[IA^{-1}] = \begin{bmatrix} 1 & 0 & 0 & \frac{3}{4} & \frac{1}{2} & \frac{1}{4} \\ 0 & 1 & 0 & \frac{1}{2} & 1 & \frac{1}{2} \\ 0 & 0 & 1 & \frac{1}{4} & \frac{1}{2} & \frac{3}{4} \end{bmatrix} \tag{3.42}$$

The matrix augmentation can now be *undone*, which gives the following:

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$A^{-1} = \begin{bmatrix} \frac{3}{4} & \frac{1}{2} & \frac{1}{4} \\ \frac{1}{2} & 1 & \frac{1}{2} \\ \frac{1}{4} & \frac{1}{2} & \frac{3}{4} \end{bmatrix} \tag{3.43}$$

To solve the system of equations $Ax = b$, use the same set of operations, indicated as the transformation matrix T, as follows:

$$\begin{aligned} Ax &= b \\ Tax &= Tb \\ Ix &= Tb \end{aligned} \tag{3.44}$$

This is identically the same as

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

Gauss–Jordan is not the most efficient method for solving systems of linear equations, but it is nevertheless popular.

3.7.4 Rank of Matrix

Definition: The rank of a matrix is the number of independent rows or columns in the matrix.

When performing Gaussian elimination, the rank becomes evident as rows are eliminated. The rank can be deduced from the triangular matrix by observing the number of nonzero rows and columns.

Having defined rank and knowing at least one way to compute it, the following statements can be made about the $n \times n$ linear system $Ax = b$:

Consistency: A linear system is consistent if rank(A) = rank([A | b]). That is, both A and the augmented matrix [A | b] have the same rank.

3.7.5 Existence and Uniqueness of Solutions for *Ax* = *b*

Assume that A is an $n \times n$ square matrix. Then the following statements can be made:

For $Ax = b$, there is

1. *No* solution if and only if $\text{rank}(A) \neq \text{rank}([A|\, b])$ (i.e., inconsistent)
2. A *unique* solution if and only if $\text{rank}(A) = \text{rank}([A|\, b]) = n$
3. An $(n - r)$-parameter *family* of solutions if and only if $\text{rank}(A) = \text{rank}([A \mid b]) = r < n$

For the homogeneous case, $Ax = 0$

1. Is consistent
2. Admits the trivial solution $x = 0$
3. Admits the unique trivial solution $x = 0$ if and only if $\text{rank}(A) = n$
4. Admits an $(n - r)$-parameter family of nontrivial solutions, in addition to the trivial solution, if and only if $\text{rank}(A) = r < n$

Stated more succinctly, if A is of full rank and b is independent of any columns of A, then a unique solution exists; otherwise, either no solution exists or an infinite number of solutions exist.

3.8 LINEAR EQUATIONS AND VECTOR/MATRIX OPERATIONS IN EXCEL®

Table 3.1 shows some of the more widely used vector/matrix operations that are native to Excel®.

`MMULT` is the only function that operates on both matrices and vectors. This function requires the two arrays to be conformable.

While the `MINVERSE` and `MMULT` functions are sufficient to solve well-posed linear systems, there are many more functions that can lead to more robust computations. A completely *free* Excel Add-On called *Matrix.xla* can be downloaded from the website http://digilander.libero.it/foxes/SoftwareDownload.htm. In addition to the software, there is a comprehensive manual (in two volumes). Readers are encouraged to install this software on their own computer. Once Matrix.xla is installed, all Excel applications have an enhanced set of functions available for matrix operations

TABLE 3.1
Excel® Matrix Functions

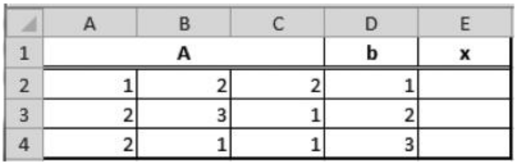
Function Name	Operation
`MDETERM`	Returns the determinant of a matrix
`MINVERSE`	Returns the inverse of a matrix
`MMULT`	Returns the product of two arrays

and linear algebra. Only a few of these are discussed here; refer to the Matrix.xla manual for many more details.

Within Matrix.xla, there are several functions that can lead to solutions of linear systems of the form $Ax = b$. The simplest of these is called `SysLin`, which uses the Gauss–Jordan algorithm. The major advantage of using `SysLin` as opposed to MINVERSE and MMULT is that it is a one-step operation. Also, if the matrix is singular (has no inverse), `SysLin` provides a suitable error message.

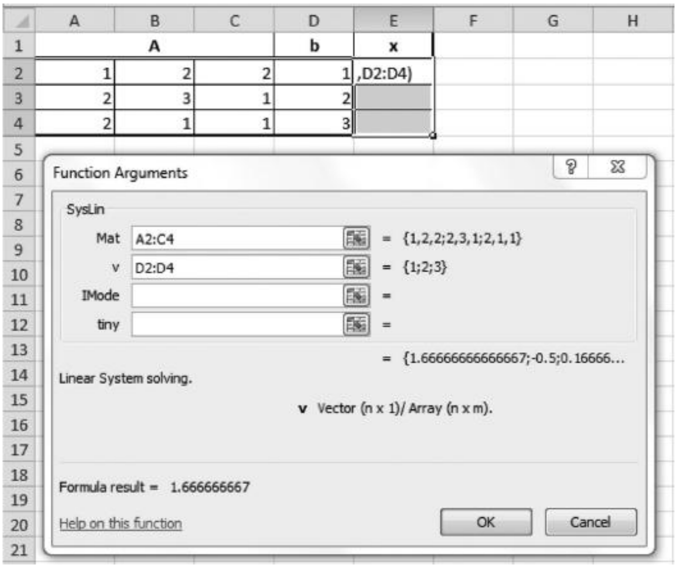
Example 3.3: Use of `SysLin` for Linear Systems

The following window shows an Excel spreadsheet with the matrix **A** and the vector **b** defined for a linear equation set. Space is available for the solution vector **x**.



	A	B	C	D	E
1		A		b	x
2	1	2	2	1	
3	2	3	1	2	
4	2	1	1	3	

If the three cells allocated for the vector **x** are selected and Formulas/InsertFunction/`SysLin` is invoked, a window for `SysLin` appears. In the area for **Mat**, simply drag the cursor across the cells containing the matrix **A**. Then in the area for **v**, drag across the cells containing **b**. At this point, everything appears as follows:



	A	B	C	D	E	F	G	H
1		A		b	x			
2	1	2	2	1	,D2:D4)			
3	2	3	1	2				
4	2	1	1	3				

Function Arguments

SysLin

Mat A2:C4 = {1,2,2;2,3,1;2,1,1}

v D2:D4 = {1;2;3}

IMode =

tiny =

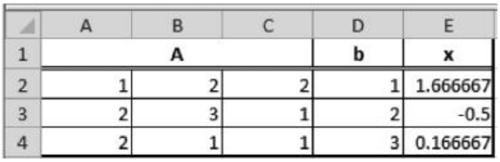
= {1.66666666666667;-0.5;0.16666...

Linear System solving.

v Vector (n x 1)/ Array (n x m).

Formula result = 1.666666667

Now (very important), hold down both Ctrl and Shift and hit Enter. The selected area reserved for **x** is filled in as follows:

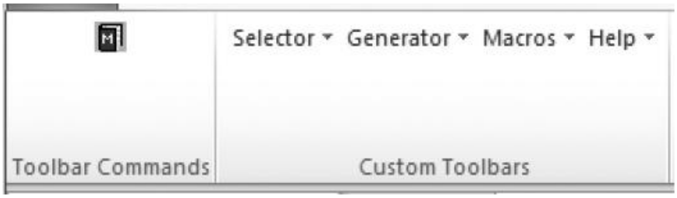


	A	B	C	D	E
1	A			b	x
2	1	2	2	1	1.666667
3	2	3	1	2	-0.5
4	2	1	1	3	0.166667

This example also gives a "feel" for how most of the matrix-related functions work.

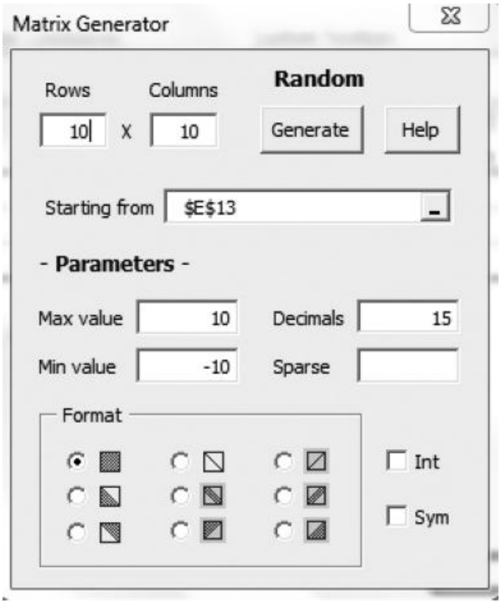
3.9 MORE ABOUT Matrix.xla

Instead of having to go through all of the steps shown in the previous example, in Excel, go to AddIns/ToolBar Commands. A small blue icon with the letter M appears. A click on the M produces a menu that looks like the following:



The number of tools available via Matrix.xla is far beyond what can be covered here. Only the most pertinent options are considered.

By clicking the Generator option and then selecting Random gives a convenient way to generate a matrix of random numbers. The following window appears:



Rows 10 x Columns 10

Starting from $E$13

Max value 10

Decimals 15

Min value -10

Sparse

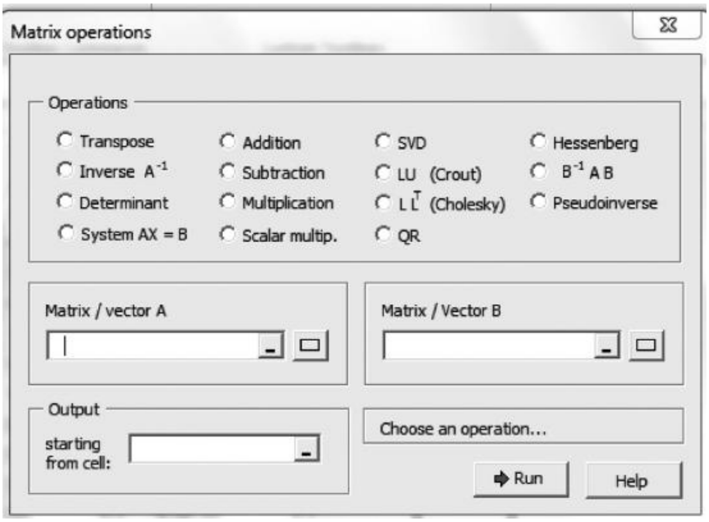
● ▩ ○ ○ ○ ○ ○ ○ ○ ○

□ Int

□ Sym

The Rows × Columns selections allow any (reasonable) size matrix of uniformly distributed random numbers to be generated (the matrix does not have to be square). The *Starting From* space indicates where the first matrix element is to be placed (cell E13 in the example). The user can then choose the maximum and minimum values for the random numbers as well as how many significant digits to include (Decimals). A variety of Formats are available, but usually, the solid icon is chosen indicating a full matrix. Other Formats include lower triangular, upper triangular, diagonal, etc. Pressing the Generate button produces the desired random matrix. This is illustrated in the next example.

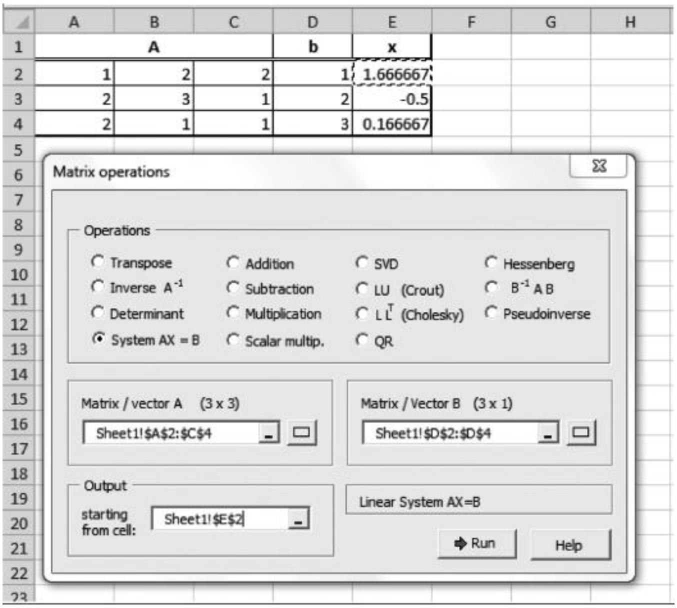
Perhaps the most useful among the Matrix.xla choices is Matrix Operations under the Macro ribbon. This produces a window like the following:



Among the many choices are Transpose, Inverse, Determinant, System AX = B, Multiplication, and Pseudoinverse. The Inverse and Multiplication choices can be used to solve Ax = b instead of using the built-in Excel functions. The System AX = B selection solves a linear system directly and even allows for the right-hand side to be a matrix and, thus, a matrix of solutions.

Example 3.4: Using System AX = B from Matrix.xla

The Excel screen shot shown below indicates how to solve the same problem of the previous example. The button for AX = B is selected. The cell range for the matrix appears in the *Matrix/vector A* selection while the cell range for the right-hand side vector appears in the *Matrix/vector B* selection. The address of the first element of the solution vector is entered into the *Output starting from cell:* selection. When the Run button was clicked, the solution was computed and output to the appropriate cells.



	A	B	C	D	E
1	A			b	x
2	1	2	2	1	1.666667
3	2	3	1	2	-0.5
4	2	1	1	3	0.166667

Matrix operations

Operations

○ Transpose ○ Addition ○ SVD ○ Hessenberg

○ Inverse A^{-1} ○ Subtraction ○ LU (Crout) ○ $B^{-1}AB$

○ Determinant ○ Multiplication ○ LL^T (Cholesky) ○ Pseudoinverse

● System AX = B ○ Scalar multip. ○ QR

Matrix / vector A (3 x 3): Sheet1!$A$2:$C$4

Matrix / Vector B (3 x 1): Sheet1!$D$2:$D$4

Output starting from cell: Sheet1!$E$2

Linear System AX=B

3.10 SVD AND PSEUDO-INVERSE OF A MATRIX

The most general way for solving any linear system (consistent, overdetermined, or underdetermined) is to use the pseudo-inverse of the matrix. A consistent system has a unique solution, an overdetermined system is one with more equations than unknowns, and an underdetermined system has an infinite number of solutions. If the pseudo-inverse is denoted by A^+, then the solution of $Ax = b$ can be written as

$$x = A^+ b \tag{3.45}$$

For a square, nonsingular matrix, A^+ coincides with the inverse, A^{-1}. The pseudo-inverse *always exists*, whether or not the matrix is square or has full rank.

For an $n \times m$ matrix, A, A^+ must satisfy the following four conditions:

$$\begin{aligned} AA^+A &= A \\ A^+AA^+ &= A^+ \\ (AA^+)^T &= (AA^+) \\ (A^+A)^T &= (A^+A) \end{aligned} \tag{3.46}$$

Note that A^+ is an $m \times$ n matrix.

The conditions that the pseudo-inverse must satisfy are not very helpful for computing A^+. Computation is most easily accomplished by using the *singular value decomposition* (SVD) of A. The SVD can be visualized as a factoring of A into three separate matrices as follows:

$$A = UDV^T \tag{3.47}$$

or, for a linear system, $Ax = b$ and

$$UDV^Tx = b \tag{3.48}$$

where, setting $p = \min(n, m)$, U is an $(n \times p)$ orthogonal matrix $(U^TU = I)$, V is an $(m \times p)$ orthogonal matrix, and D is a $(p \times p)$ diagonal matrix (the diagonal elements are called the singular values of A). For simplicity, assume $n > m$, in which case D and V are both square with dimension $(m \times m)$. This assumption does not invalidate the final result.

Multiplying both sides of Equation 3.48 by U^T and remembering that $U^T\ U = I$, there results

$$U^TUDV^Tx = U^Tb \Rightarrow DV^T\,x = U^Tb \tag{3.49}$$

The matrix DV^T is square so taking its inverse gives [recall that for any two matrices X and Y, $(XY)^{-1} = Y^{-1}X^{-1}$]

$$DV^T\,x = U^T\,b \Rightarrow x = (DV^T)^{-1}\,U^Tb \Rightarrow x = (V^T)^{-1}D^{-1}\,U^Tb \tag{3.50}$$

Because $V^T = V^{-1}$, the final result is

$$x = (V^T)^{-1}D^{-1}\,U^Tb = A^+b \tag{3.51}$$

Therefore, the pseudo-inverse can be computed by the following formula:

$$A^+ = V\,D^{-1}U^T \tag{3.52}$$

When using the pseudo-inverse, there are three possible situations:

1. *$m = n$:* The matrix A is square and $A^+ = A^{-1}$.
2. *$m > n$:* There are more equations than unknowns and the system is overdetermined. In this case, x represents the “least squares” solution. That is, x minimizes $||Ax - b||$, which is the sum of squares of “residuals” (the difference between each element of Ax and the corresponding element of b).
3. *$m < n$:* There are fewer equations than unknowns and the system is underdetermined. Therefore, there is no unique solution; in fact, there are an infinite number of solutions. The most convenient way to express these solutions using the pseudo-inverse is

$$x = A^+b + (I - A^+A)z \tag{3.53}$$

where z is an arbitrary vector. Any z can be specified and the corresponding x will satisfy the m equations. The proof of this equation is lengthy and is offered here without proof. Note that $A^+A \neq I$.

The actual algorithm for computing A^+ is complex and is not covered here. Fortunately, in Matrix.xla, this computation is made available by the function `MPseudoinv`. Recall that the pseudo-inverse of an $m \times n$ matrix is an $n \times m$ matrix.

Example 3.5: Solution of Linear Systems Using the Pseudo-Inverse

The Excel spreadsheet shown below illustrates solving

- A consistent system
- An overdetermined system (least-squares solution)
- An underdetermined system (infinite number of solutions)

In each case, the matrix and the right-hand side vector are generated using the random matrix feature of Matrix.xla. For the underdetermined system, an arbitrary *z* vector $[0\ 1\ 1\ 1]^T$ is chosen to show how to produce another of the infinite number of other solutions (see Equation 3.53).

Note the following about the results shown in the spreadsheet:

- For the consistent system, the vector *Ax* is identically equal to the vector *b*. This is a unique solution.
- For the overdetermined system, *Ax* does not equal *b*. This is because the solution minimizes the sum of squares of differences between *Ax* and *b*. The least squares solution is useful in several applications, such as regression, which is covered in Chapter 7.
- For the underdetermined system, the initial (or base) *x* is the product A^+b and is only one of an infinite number of solutions possible. A second solution is generated using Equation 3.53 with the arbitrary vector $z = [0\ 1\ 1\ 1]^T$. In both cases, *Ax* is equal to *b* indicating that indeed the associated *x* is a solution.

Consistent System

	A		**b**	**x**	**Ax**
0.832855	7.902832	–2.18289	–7.67573	–2.73575	–7.67573
3.889161	9.665859	3.044689	3.055222	0.298083	3.055222
–5.28107	8.451514	–5.57466	–2.8325	3.551683	–2.8325

	A^+	
0.387957	–0.12478	–0.22006
–0.0273	0.078798	0.053725
–0.40891	0.23767	0.110541

Overdetermined System

	A		**b**	**x**	**Ax**
6.190156	–0.42224	–5.69814	–9.26793	0.045362	–3.49158
10.65635	–4.55003	1.185262	4.296808	0.484615	–0.97949
2.568451	–6.16077	6.203344	–3.82716	0.626126	1.014982
9.26915	9.534804	10.53612	10.77149		11.63811

Least-squares solution

	A^+		
0.04579	0.04914	–0.00505	0.022212
0.013649	–0.03479	–0.06283	0.048287
–0.0582	–0.00667	0.056641	0.030838

Underdetermined System

A				b	Base x	Ax
–4.90858	–5.57692	0.430979	–4.66382	–6.93659	–0.17947	–6.93659
2.021388	1.200819	7.644887	1.3031	–5.13765	1.595509	–5.13765
–9.45529	0.608641	–6.26193	–9.6764	10.79749	–0.82275	10.79749
					–0.3077	
A$^+$					**Base x = A$^+$b**	
–0.02176	–0.02893	–0.04437				
–0.15092	0.081009	0.089359				
0.030857	0.134928	0.007826				
–0.00819	–0.05395	–0.05943				

I				A$^+$A			
1	0	0	0	0.467865	0.059627	0.04728	0.493129
0	1	0	0	0.059627	0.993319	–0.0053	–0.05526
0	0	1	0	0.04728	–0.0053	0.995799	–0.04381
0	0	0	1	0.493129	–0.05526	–0.04381	0.543017

I – (A$^+$A)				z	Arbitrary Part	Base x	New Solution	New Ax
0.532135	–0.05963	–0.04728	–0.49313	0	–0.60004	–0.17947	–0.77951	–6.93659
–0.05963	0.006681	0.005298	0.055257	1	0.067236	1.595509	1.662745	–5.13765
–0.04728	0.005298	0.004201	0.043815	1	0.053313	–0.82275	–0.76943	10.79749
–0.49313	0.055257	0.043815	0.456983	1	0.556054	–0.3077	0.248349	

New Solution is x = A$^+$ b + (I – A$^+$A)

EXERCISES

Exercise 3.1: Mass balance on a gas absorber

A gas absorber is fed, via stream F_1, 100 mol/min of monoethanolamine (MEA) and CO_2. Stream F_1 is composed of 98% (mol%) MEA and 2% CO_2. Stream F_2 contains CO_2, SO_2, and N_2 (Figure 3.1). Experimental data available for the unit are shown in Table 3.2.

Derive the linear system for the absorber using the supplied information. The three unknowns are the flow rates of P_1, F_2, and P_2. Note that since data are available on only two components, it is necessary to include the overall material balance as one of the equations.

a. Using `SYSLIN` (or the linear system option from Matrix.xla)
b. By calculating A^{-1} followed by multiplication ($A^{-1}b$)
c. By calculating the pseudo-inverse, A^+, followed by matrix multiplication A^+b

Exercise 3.2: Coffee leaching

A "Mr. Coffee" apparatus for brewing a good "cuppa joe" is a chemical extraction unit. Ingredients include water (W), solubles (S), and grounds (G). A schematic diagram of the "system" is shown in Figure 3.2.

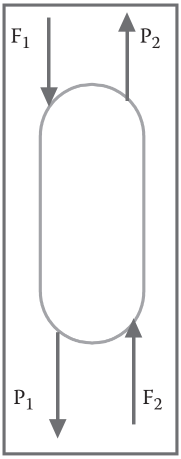


FIGURE 3.1 Gas absorber schematic.

TABLE 3.2
Data for Gas Absorber (Mole Fractions)

	P_1	F_2	P_2
SO_2	0.0170	0.0200	0.0022
N_2	0.0000	0.9000	0.9890

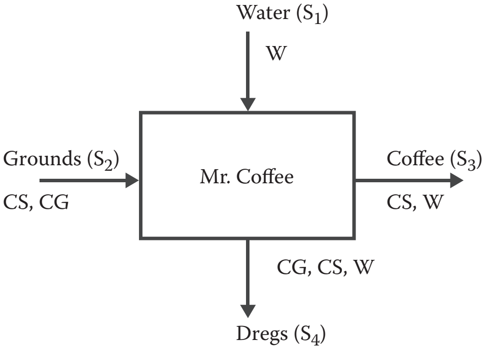


FIGURE 3.2 Automatic coffee maker schematic.

The Grounds input contains components CG and CS. Water input contains only component W. The Coffee stream contains both water (W) and solubles (CS), while the Dregs output has all three components. Other pertinent data are as follows (all percentages are by volume):

- Stream S_1 consists of 1.1 L of pure water.
- Stream S_2 contains 98% solid (CG) and 2% solubles (CS).
- Stream S_3 contains 0.8% CS and 99.2% W.
- Stream S_4 contains 81% CG, 0.5% CS, and 18.5% W.

Write three component balances (these are "volume" balances since percentages are volume based) to give three linear equations in the three unknown flowrates (S_2, S_3, and S_4).

Solve the linear system for the following problem:

a. Using `SYSLIN` (or the linear system option from Matrix.xla)
b. By calculating A^{-1} followed by multiplication ($A^{-1}b$)
c. By calculating the pseudo-inverse, A^+, followed by matrix multiplication A^+b

Exercise 3.3: Flash tanks in series

Shown in Figure 3.3 is a schematic of a separation system consisting of two flash tanks in series. Experimental data are available on streams F, V_1, V_2, and L_2 as shown in Table 3.3. The feed rate, F, is 1000 kg/min.

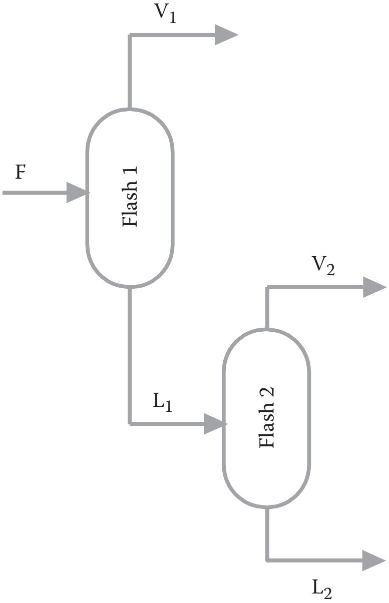


FIGURE 3.3 Two flash tanks in series.

TABLE 3.3
Stream Mass Fractions

	Mass Fraction			
Component	**F**	**V_1**	**V_2**	**L**
Methanol	0.3	0.71	0.44	0.08
Ethanol	0.4	0.27	0.55	0.39
Butanol	0.3	0.02	0.01	0.53

Write mass balances on each of the three components using the supplied data. The result is three linear algebraic equations in the three unknown flowrates, V_1, V_2, and L_2. Use any method to find these three unknowns.

Exercise 3.4: For the following exercises, use the random matrix generator available in Matrix.xla as required.

a. Generate a random matrix of size 3 × 3 and a column vector of length 3. Using these data to represent A and b in the linear system $Ax = b$, solve the system
   1. By taking the matrix inverse followed by matrix-vector multiplication
   2. Use the `SYSLIN` function
   3. Use the function MPseudoinv and matrix multiplication
b. Repeat part a for a 10 × 10 matrix and associated vector.
c. Repeat step 3 of part a for a 4 × 3 matrix.
d. Repeat step 3 of part a for a 3 × 4 matrix.
e. For the 3 × 4 matrix of part d, apply Equation 3.53 to generate a particular solution based on an arbitrary vector, $z = [1\ 1\ 1\ 1]^T$.

Note that for a 3 × 4 matrix, A^+A is not equal to the identity matrix. An infinite number of solutions exist since *any* z gives a proper solution.

Exercise 3.5: A chemical separation system

Benzene, styrene, toluene, and xylene are to be separated with the array of distillation columns shown in Figure 3.4. Experimental data show that the feed rate (stream A) is 100 mole/hr and that the composition of stream A is 10% benzene, 15% styrene, 35% toluene and 40% xylene (compositions are in mole %).

The following information is available for this system:

1. 80% benzene, 60% styrene, 30% toluene and 10% xylene of the feed stream go overhead to stream B. The remainder goes out the bottom to stream C.
2. 70% benzene, 65% styrene, 25% toluene and 5% xylene of stream B go overhead to stream D. The remainder goes out the bottom to stream E.
3. 85% benzene, 65% styrene, 20% toluene and 10% xylene of stream C go overhead to stream F. The remainder goes out the bottom to stream G.

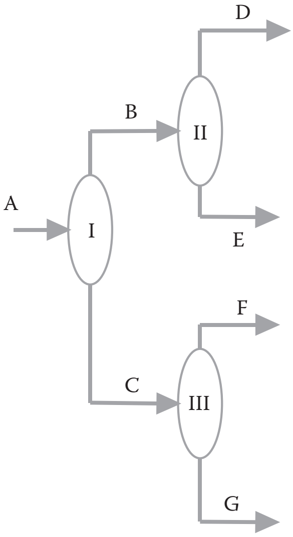


FIGURE 3.4 Distillation column train.

Write four material balances where the unknowns are the flow rates of streams D, E, F, and G. Solve the resulting linear system by any method that has been discussed.

REFERENCE

Gaussian Elimination History; http://meyer.math.ncsu.edu/Meyer/PS_Files/GaussianEliminationHistory.pdf (Oct. 2012).

4 Numerical Differentiation and Integration

4.1 NUMERICAL DIFFERENTIATION

4.1.1 Approximation of a Derivative in One Variable

Recall the definition of the derivative from calculus:

$$\frac{df}{dx} = \lim_{\Delta x \to 0}\left(\frac{f(x+\Delta x) - f(x)}{\Delta x}\right)$$

For a finite Δx, this is an *approximation* of the derivative:

$$\frac{df}{dx} = \left(\frac{f(x+\Delta x) - f(x)}{\Delta x}\right) + E(\Delta x) \tag{4.1}$$

This is the simplest form of "finite difference derivative" and is called the "forward" difference approximation to the derivative. $E(x)$ represents the "error" in the approximation. In order to estimate the size of the error term, consider the Taylor expansion of $f(x + \Delta x)$ in the neighborhood of x:

$$f(x+\Delta x) = f(x) + \Delta x f'(x) + \frac{\Delta x^2}{2!} f''(x) + \frac{\Delta x^3}{3!} f'''(x) + \frac{\Delta x^4}{4!} f^{iv}(x) + \cdots \tag{4.2}$$

Equation 4.1 results from truncating all but the first two terms in the Taylor expansion of Equation 4.2. In order to determine how good the approximation is, consider *temporarily* retaining the term involving f'. This gives

$$f'(x) = \frac{f(x+\Delta x) - f(x)}{\Delta x} - \frac{\Delta x}{2!} f''(x) + \cdots$$

The last term is an estimate of $E(x)$, which is proportional to Δx. A terminology used to describe this is the *Big $\mathcal{O}$ notation* or the order of magnitude notation. The approximation of Equation 4.1 is, therefore, $\mathcal{O}(\Delta x)$, or the error in the approximation is proportional to Δx.

If negative Δx is applied in the Taylor expansion, there results

$$f(x-\Delta x)=f(x)-\Delta x f'(x)+\frac{\Delta x^2}{2!}f''(x)-\frac{\Delta x^3}{3!}f'''(x)+\frac{\Delta x^4}{4!}f^{iv}(x)-\cdots \tag{4.3}$$

Truncating all but the linear terms and solving for $f'(x)$ gives

$$\frac{df}{dx}\cong\left(\frac{f(x)-f(x-\Delta x)}{\Delta x}\right) \tag{4.4}$$

This is called the “backward” difference approximation to the first derivative and is also $\mathcal{O}(\Delta x)$.

Subtracting Equation 4.3 from Equation 4.2 gives

$$f(x+\Delta x)-f(x-\Delta x)=2\Delta x f'(x)+2\frac{\Delta x^3}{3!}f'''(x)+\cdots \tag{4.5}$$

Solving for f' yields

$$\frac{df}{dx}=\frac{f(x+\Delta x)-f(x-\Delta x)}{2\Delta x}-\frac{1}{6}\Delta x^2 f''' \tag{4.6}$$

Note that the error is proportional to Δx^2, or it is $\mathcal{O}(\Delta x^2)$. Equations 4.1 and 4.4 are “first-order correct,” and Equation 4.6 is “second-order correct.” Equation 4.6 (without the error term) is called the central difference approximation to the first derivative.

Adding Equations 4.2 and 4.3 gives the following result:

$$f(x+\Delta x)+f(x-\Delta x)=2f(x)+2\frac{\Delta x^2}{2!}f''(x)+2\frac{\Delta x^4}{4!}f^{iv}(x)+\cdots$$

Solving for $f''(x)$ gives

$$f''(x)\cong\frac{f(x+\Delta x)-2f(x)+f(x-\Delta x)}{\Delta x^2}-\frac{1}{12}\Delta x^2 f^{iv} \tag{4.7}$$

The error term for this approximation to the second derivative is $\mathcal{O}(\Delta x^2)$.

Further manipulations can be performed with the Taylor expansions, such as involving more “points” (e.g., $f(x+2\Delta x)$, $f(x-2\Delta x)$, etc.). This leads to (1) better approximations for the first and second derivatives and (2) approximations for higher-order derivatives. Table 4.1 shows the *first-order correct* formulas for the first and second derivatives, while Table 4.2 depicts the *second-order correct* counterparts. In the tables, $\Delta x = x_{i+1}-x_i = x_i - x_{i-1}$.

TABLE 4.1
First-Order Correct Approximations for Derivatives

Forward Difference	$f'(x_i) = \dfrac{f(x_{i+1}) - f(x_i)}{\Delta x}$
	$f''(x_i) = \dfrac{f(x_i) - 2f(x_{i+1}) + f(x_{i+2})}{\Delta x^2}$
Backward Difference	$f'(x_i) = \dfrac{f(x_i) - f(x_{i-1})}{\Delta x}$
	$f''(x_i) = \dfrac{f(x_i) - 2f(x_{i-1}) + f(x_{i+2})}{\Delta x^2}$
Central Difference	None

TABLE 4.2
Second-Order Correct Approximations for Derivatives

Forward Difference	$f'(x_i) = \dfrac{-3f(x_i) + 4f(x_{i+1}) - f(x_{i+2})}{2\Delta x}$
	$f''(x_i) = \dfrac{2f(x_i) - 5f(x_{i+1}) + 4f(x_{i+2}) - f(x_{i+3})}{\Delta x^2}$
Backward Difference	$f'(x_i) = \dfrac{3f(x_i) - 4f(x_{i-1}) + f(x_{i-2})}{2\Delta x}$
	$f''(x_i) = \dfrac{2f(x_i) - 5f(x_{i-1}) + 4f(x_{i-2}) - f(x_{i-3})}{\Delta x^2}$
Central Difference	$f'(x_i) = \dfrac{f(x_{i+1}) - f(x_{i-1})}{2\Delta x}$
	$f''(x_i) = \dfrac{f(x_{i+1}) - 2f(x_i) + f(x_{i-1})}{\Delta x^2}$

In order to see more clearly where all of these come from, the first one in Table 4.2 is now derived:

$$f(x + \Delta x) = f(x) + \Delta x f'(x) + \frac{\Delta x^2}{2!} f''(x) + \frac{\Delta x^3}{3!} f'''(x) + \frac{\Delta x^4}{4!} f^{iv}(x) + \cdots \tag{4.8}$$

$$f(x + 2\Delta x) = f(x) + 2\Delta x f'(x) + \frac{4\Delta x^2}{2!} f''(x) + \frac{8\Delta x^3}{3!} f'''(x) + \frac{16\Delta x^4}{4!} f^{iv}(x) + \cdots \tag{4.9}$$

Multiplying Equation 4.8 by 4 and subtracting Equation 4.9 from the result gives

$$4f(x + \Delta x) - f(x + 2\Delta x) = 3f(x) + 2\Delta x f'(x) - \frac{4\Delta x^3}{3!} f'''(x) + \cdots$$

Solving for $f'(x)$ yields

$$f'(x) = \frac{-f(x+2\Delta x)+4f(x+\Delta x)-3f(x)}{2\Delta x} + \frac{1}{3}\Delta x^2 f''' \tag{4.10}$$

This is the same as the first formula in Table 4.2 and is a second-order correct formula since the error term is $\mathcal{O}(\Delta x^2)$. This is called an *end-point formula* since it can be applied at a boundary where negative perturbations are not allowed.

4.1.2 Approximation of Partial Derivatives

Consider a two-dimensional function $f(x, y)$. The difference approximation for the partial derivative

$$f_x = \frac{\partial}{\partial x} f(x,y) \text{ at } x = x_0 \text{ and } y = y_0 \tag{4.11}$$

can be derived by fixing y to y_0 and considering $f(x, y_0)$ as a one-dimensional function. The forward, centered, and backward difference approximations for the above partial derivative may be written as follows:

$$\begin{aligned} f_x &= \frac{f(x_0+\Delta x, y_0) - f(x_0, y_0)}{\Delta x} + \mathcal{O}(\Delta x) \\ f_x &= \frac{f(x_0+\Delta x, y_0) - f(x_0-\Delta x, y_0)}{2\Delta x} + \mathcal{O}(\Delta x^2) \\ f_x &= \frac{f(x_0, y_0) - f(x_0-\Delta x, y_0)}{\Delta x} + \mathcal{O}(\Delta x) \end{aligned} \tag{4.12}$$

The centered difference approximations for the second partial derivatives are shown below:

$$\begin{aligned} f_{xx} &= \frac{\partial^2}{\partial x^2} f = \frac{f(x_0+\Delta x, y_0) - 2f(x_0, y_0) + f(x_0-\Delta x, y_0)}{\Delta x^2} \\ f_{yy} &= \frac{\partial^2}{\partial y^2} f = \frac{f(x_0, y_0+\Delta y) - 2f(x_0, y_0) + f(x_0, y_0-\Delta y)}{\Delta y^2} \\ f_{xy} &= \frac{\partial\partial}{\partial \Delta x \partial y} f = \frac{f(x_0+\Delta x, y_0+\Delta y) - f(x_0+\Delta x, y_0-\Delta y)}{4\Delta x\Delta y} + \\ &\qquad \frac{-f(x_0-\Delta x, y_0+\Delta y) + f(x_0-\Delta x, y_0-\Delta y)}{4\Delta x\Delta y} \end{aligned} \tag{4.13}$$

Example 4.1: Solids Volume Fraction in a Fluidized Bed

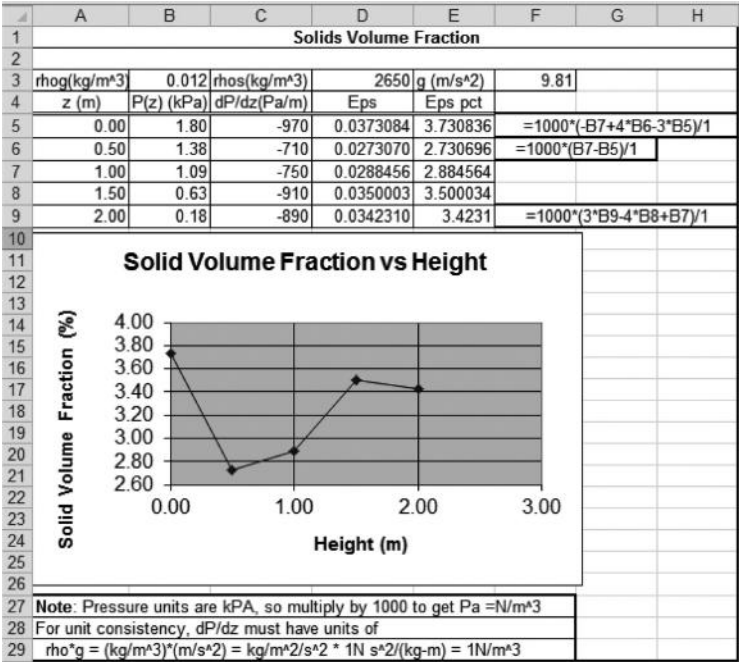
For a fluidized bed with a gas whose density, ρ_g, is 0.012 kg/m^3 and a solid whose density, ρ_s, is 2650 kg/m^3, calculate the percentage solids volume as a function of axial position given the following data:

Axial Position (m)	Pressure (kPa)
0	1.8
0.5	1.38
1	1.09
1.5	0.63
2	0.18

Writing a momentum balance for the two-phase flow leads to the following equation for the fraction solids volume, where z = axial position, P = pressure, and g = 9.81 (the gravitational constant):

$$\varepsilon_s = \frac{(-dP/dz) - \rho_g g}{(\rho_s - \rho_g)g} \tag{4.14}$$

This problem requires numerical differentiation. It is recommended to always apply the second-order correct finite difference formulas. The end-point formulas are applied at $z = 0$ and $z = 2$, while centered difference approximations are used at all other points. The results appear in the following spreadsheet, which includes a graph of percent solids volume versus axial position.



	A	B	C	D	E	F	G	H
1	Solids Volume Fraction							
2								
3	rhog(kg/m^3)	0.012	rhos(kg/m^3)	2650	g (m/s^2)	9.81		
4	z (m)	P(z) (kPa)	dP/dz(Pa/m)	Eps	Eps pct			
5	0.00	1.80	-970	0.0373084	3.730836	=1000*(-B7+4*B6-3*B5)/1		
6	0.50	1.38	-710	0.0273070	2.730696	=1000*(B7-B5)/1		
7	1.00	1.09	-750	0.0288456	2.884564			
8	1.50	0.63	-910	0.0350003	3.500034			
9	2.00	0.18	-890	0.0342310	3.4231	=1000*(3*B9-4*B8+B7)/1		

	A
27	**Note**: Pressure units are kPA, so multiply by 1000 to get Pa =N/m^3
28	For unit consistency, dP/dz must have units of
29	rho*g = (kg/m^3)*(m/s^2) = kg/m^2/s^2 * 1N s^2/(kg-m) = 1N/m^3

The derivative at $z = 0$ is computed as follows:

$$\frac{dP}{dz} = 1000 \cdot \frac{-3(1.80) + 4(1.38) - 1.09}{2(0.5)} = 1000 \cdot \frac{-5.4 + 5.52 - 1.09}{1} = -970 \qquad (4.15)$$

4.2 NUMERICAL INTEGRATION

4.2.1 Trapezoidal Rule

The simplest method for numerical integration is the trapezoidal rule, which is based on joining interval end points with a chord to form a trapezoid, whose area is an approximation to the definite integral over the interval. This is depicted in Figure 4.1 with five nonequally spaced subintervals.

The entire integral of $f(x)$ from x_0 to x_5 can be expressed as

$$\int_{x_0}^{x_5} f(x)\,dx = \sum_{i=1}^{5} \frac{f(x_{i-1}) + f(x_i)}{2}(x_i - x_{i-1}) + E \qquad (4.16)$$

where the error term, E, can be derived using Taylor expansions for $f(x)$ at the interval end points, $f(x_{i-1})$ and $f(x_i)$. This is a long process whose final result is

$$E \cong -\frac{1}{12}(f(x_{i-1}) - f(x_i))h^2\overline{f''} \qquad (4.17)$$

where $\overline{f''}$ is the average second derivative of f over the interval. The bottom line is that the error is proportional to h^2, or the method is $\mathcal{O}(h^2)$, where h is a typical interval width.

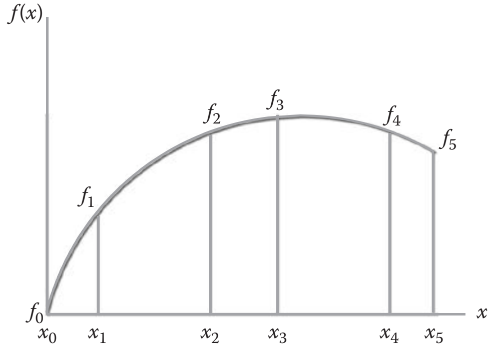


FIGURE 4.1 Schematic of trapezoidal rule.

Unless otherwise stated, all numerical integrations for the remainder of the book will be computed by the trapezoidal rule. It is accurate enough for most engineering applications, does not require equally spaced data, and is easy to implement in Excel® and VBA. For information on more complex and accurate numerical integration methods, Google it!

4.2.2 Simpson's Rule

The trapezoidal rule approximates the function over a subinterval with a straight line. Simpson's rule uses a quadratic function over two successive intervals as shown in Figure 4.2.

By subdividing the interval from *a* to *b* into smaller ones, the following formulas apply for *N* an even number:

$$\begin{aligned}\int_a^b f(x)\,dx &= \frac{h}{3}\left[f(a) + 4\sum_{\substack{i=1\\ i\,odd}}^{N-1} f(a+ih) + 2\sum_{\substack{i=1\\ i\,even}}^{N-2} f(a+ih) + f(b)\right] + E \\ &= \frac{h}{3}[f_0 + 4f_1 + 2f_2 + 4f_3 + 2f_4 + \cdots + +2f_{N-2} + 4f_{N-1} + f_N] + E\end{aligned} \tag{4.18}$$

The error term, *E*, can be shown to be proportional to h^4. Major disadvantages are that *N* must be an *even* number and *h* must be constant. There exists another version of Simpson's rule when *N* is odd, but *h* still must be constant.

There are many more sophisticated methods of quadrature (another name for numerical integration). For reasons already given, the trapezoidal rule is sufficient for most engineering applications. In some specialized cases, Gauss integration (see Section 4.2.3) is the most advantageous method.

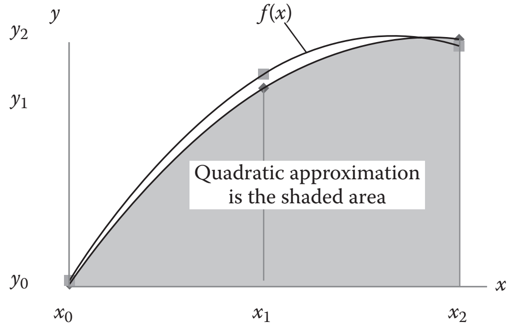


FIGURE 4.2 Schematic of Simpson's rule.

Example 4.2: Using the Trapezoidal Rule

Consider evaluating the integral

$$I = \int_0^2 \pi \left(1 + \left(\frac{x}{2}\right)^2\right)^2 dx$$

The exact answer is 11.7286. The following Excel spreadsheet shows a solution to this using the trapezoidal rule with different values for h:

Trapezoidal Rule Using h = 0.5		
x	$f(x)$	I
0	3.141593	0
0.5	3.546564	1.672039
1	4.908739	3.785865
1.5	7.669904	6.930525
2	12.56637	11.98959

Trapezoidal Rule Using h = 0.2		
x	$f(x)$	I
0	3.141593	0
0.2	3.204739	0.634633
0.4	3.397947	1.294902
0.6	3.732526	2.007949
0.8	4.227327	2.803934
1	4.908739	3.717541
1.2	5.81069	4.789484
1.4	6.97465	6.068018
1.6	8.449628	7.610445
1.8	10.29217	9.484625
2	12.56637	11.77048

It can be seen that the answer using h = 0.5 is rather poor while that for h = 0.2 is close to the correct answer. For most engineering applications, the trapezoidal rule is good enough if a small h is used. In real applications, it is often the case that a good value for h can be estimated. When presented with predetermined data, it is often the case that h is not controllable and might well be variable.

4.2.3 Gauss Quadrature

The mathematician/scientist Gauss developed a particularly unique and interesting method of numerical integration by asking the question, "if I can choose the points on the interval, are there optimal ones to choose?" The answer is a resounding *yes*. The general form of Gauss integration is

$$\int_{-1}^{1} f(x)\,dx = \sum_{i=1}^{n} w_i f(x_i) \tag{4.19}$$

where both w_i and x_i are chosen so that for a given n, the rule is exact for polynomials up to and including degree $2n - 1$. The w_i are called the *weights*. Note the fixed range of integration from –1 to 1. Thus, if the range is a to b, a change of variables is required. That is, if the *original variable* is z and the range is $[a, b]$, then set

$$x = \frac{2z - a - b}{b - a} \tag{4.20}$$

The Gauss formula for $n = 2$ from Equation 4.19 becomes

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

Make this exact for polynomials of degree 0, 1, 2, and 3 ($2n - 1$) as follows:

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

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

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

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

This gives four equations and four unknowns, which can be solved to give

$$x_1 = {}^{-1}\!\big/\!{}_{\sqrt{3}} \quad x_2 = {}^{1}\!\big/\!{}_{\sqrt{3}} \quad w_1 = 1 \quad w_2 = 1$$

Thus,

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

TABLE 4.3
Gauss Points for *n* = 2, 3, …, 6

	Gauss Points and Weights	
n	**±*x***	w_i
2	0.57735	1
3	0	0.88889
	0.7746	0.55556
4	0.33998	0.65215
	0.86114	0.34785
5	0	0.56889
	0.53847	0.47863
	0.90618	0.23693
6	0.23862	0.46791
	0.66121	0.36076
	0.93247	0.17132

Error analysis is not so simple. The above formula is *exact* if *f*(*x*) is a cubic polynomial (or a simpler one). A rule of thumb is that the order of accuracy of Gauss integration is twice that of equally spaced methods using the same number of data points.

Gauss quadrature formulas for higher *n* can be derived in a similar manner, but only the final results are shown here. Table 4.3 shows Gauss points for values of *n* up to 6.

Example 4.3: Using Gauss Integration

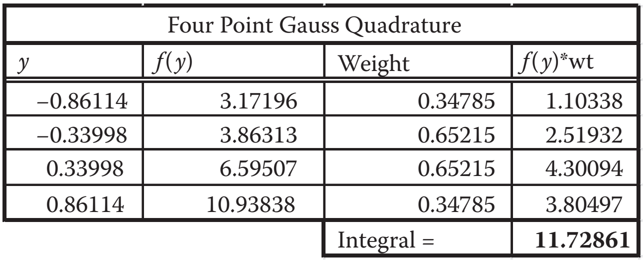
Consider the same problem as in Example 4.2 using four-point Gauss integration. First, perform a change of variables as indicated in Equation 4.20. Let

$$y = \frac{2x-2}{2} = x - 1 \text{or } x = y + 1$$

The restated problem is then

$$I = \int_{-1}^{1} \pi \left(1 + \left(\frac{y+1}{2}\right)^2\right)^2 dy$$

Note that *x* has been replaced with *y* + 1, *dx* with *dy,* and the limits of integration changed accordingly. The four-point Gauss calculations are shown in the following Excel spreadsheet:



Four Point Gauss Quadrature			
y	*f*(*y*)	Weight	*f*(*y*)*wt
−0.86114	3.17196	0.34785	1.10338
−0.33998	3.86313	0.65215	2.51932
0.33998	6.59507	0.65215	4.30094
0.86114	10.93838	0.34785	3.80497
		Integral =	**11.72861**

This example illustrates the power of Gauss integration. A four-point formula gives essentially the exact result. [This is due to the fact that *f*(*x*) is a quartic function, and the four-point formula is exact for polynomials up to degree 7.]

In summary, the trapezoidal rule is easy to implement and usually accurate enough. Furthermore, it can be used with nonequally spaced data (such as real experimental data). When there is a limit to the number of "sample" points, but they can be placed at will, then Gauss integration is often the best choice. For example, suppose a restriction is that only four samples can be obtained on a process during a test whose duration is 1 h. When during the hour should the samples be collected? The answer to this is left as an exercise.

4.3 CURVE FITTING FOR INTEGRATION

Another approach to numerical integration is to "fit" the data to a particular function form and then do the integration analytically. Sophisticated curve fitting methods are covered in Chapter 7. For now, Excel's graphing capability allows the fitting of simple functions to data. Once the fitting function has been determined, it can be integrated analytically. This is demonstrated in Example 4.4 (along with the trapezoidal rule).

Example 4.4: Integrating Fermentation Data

The data shown in the following spreadsheet represent the rate of evolution of CO_2 and the take-up rate of O_2 during a fermentation reaction. It is important in the study of fermentation processes to obtain the net amount of these gases used and released. This is accomplished by *integrating* the rates over time. The fourth and fifth columns in the spreadsheet show the results of integrating the data using the trapezoidal rule, and the results are

Total CO_2 evolution = 168.3450 g
Total O_2 evolution = 145.5200 g

In this case, *h* is fixed by the available experimental data.

Time(h)	CO_2 Rate (g/h)	O_2 Rate (g/h)	Trap CO_2	Trap O_2	CO_2 Evol	O_2 Evol
			Evolution	Evolution	Curve Fit	Curve Fit
140	15.72	15.49	0.00	0.00	0.00	0.00
141	15.53	16.16	15.63	15.83	15.66	16.11
142	15.19	15.35	30.99	31.58	31.58	31.71
143	16.56	15.13	46.86	46.82	47.78	46.90
144	16.21	14.20	63.25	61.49	64.26	61.74
145	17.39	14.23	80.05	75.70	81.04	76.32
146	17.36	14.29	97.42	89.96	98.15	90.70
147	17.42	12.74	114.81	103.48	115.59	104.97
148	17.60	14.74	132.32	117.22	133.39	119.21
149	17.75	13.68	150.00	131.43	151.56	133.49
150	18.95	14.51	168.35	145.52	170.12	147.88

A graph of the experimental data is shown in Figure 4.3 along with a curve fit quadratic polynomial equation. These equations were determined by right clicking on one of the experimental points and choosing Add Trendline, which then displays the window shown in Figure 4.4. The Polynomial button was selected along with the order 2. This produced the following curve fit equations:

$$CO_2 = 0.0082t^2 - 2.0567t + 142.76$$

$$O_2 = 0.0385t^2 - 11.335t + 851.48$$

Upon integrating these equations from t = 140 to t = 150, there results (see columns 6 and 7 of the spreadsheet)

Total CO_2 evolution = 170.12 g
Total O_2 evolution = 147.8 g

These results compare favorably with those obtained from the trapezoidal rule.

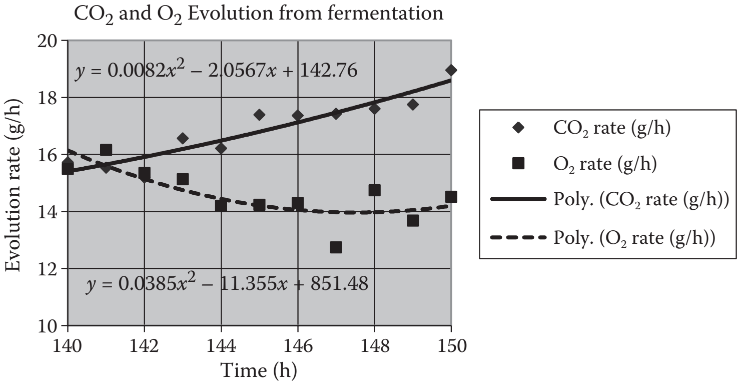
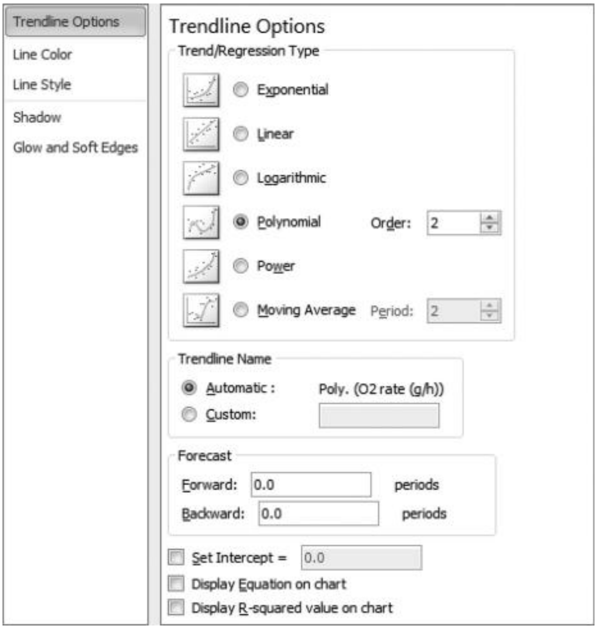


FIGURE 4.3 Graph with data points and trendline with equations displayed.



Trendline Options

Trend/Regression Type

○ Exponential

○ Linear

○ Logarithmic

● Polynomial Order: 2

○ Power

○ Moving Average Period: 2

Trendline Name

● Automatic : Poly. (O2 rate (g/h))

○ Custom:

Forecast

Forward: 0.0 periods

Backward: 0.0 periods

□ Set Intercept = 0.0

□ Display Equation on chart

□ Display R-squared value on chart

FIGURE 4.4 Trendline options window.

EXERCISES

Exercise 4.1: The heat capacity at constant pressure is defined as

$$C_p = \left(\frac{\partial H}{\partial T} \right)_p$$

where C_p is the heat capacity at constant pressure, H is the molar enthalpy, and T is temperature. The following table shows heat capacity versus temperature data for carbon dioxide (http://webbook.nist.gov/chemistry/fluid/).

Temperature (°C)	Enthalpy (kJ/mol)
100	25.186
150	27.254
200	29.408
250	31.640
300	33.942
350	36.307
400	38.732
450	41.209
500	43.736
550	46.307

600	48.919
650	51.568
700	54.252
750	58.966
800	59.710

a. Use finite differences to compute the heat capacity of carbon dioxide at each of the given temperatures. Be sure to use the "end-point" formulas for the first and last entries.

b. Graph the enthalpy data and curve-fit it with a suitable polynomial. Then, calculate the heat capacity at each temperature using analytical differentiation. Compare these results with those of part a.

c. Compare the results for C_p with those from the *nist* database (these values appear in the data table of Exercise 4.2). Be sure to use consistent units for comparison.

Exercise 4.2: The enthalpy required to heat *n* moles of a gas from T_1 to T_2 can be found by integrating the heat capacity at constant pressure over the temperature range. The following table lists heat capacity data for CO_2 (also from the *nist* database):

Temperature (°C)	**Cp (J/mol*K)**
100	40.461
150	42.256
200	43.881
250	45.355
300	46.695
350	47.917
400	49.034
450	50.055
500	50.989
550	51.843
600	52.624
650	53.339
700	53.994
750	54.593
800	55.144

a. Calculate the enthalpy of one mole of CO_2 over the temperature range given in the table using the trapezoidal rule with the following formula (*n* is the number of moles):

$$\Delta H = n\int_{T_1}^{T_2} C_p(T)\,dT$$

b. Curve fit the heat capacity data using an appropriate polynomial. Then find ΔH (800) (with a reference temperature of 100°C) by integrating the resulting function analytically. What is the percentage error between the two methods?
c. Compare your results for ΔH with those from the *nist* database (these values appear in the data table of Exercise 4.1). Be sure to use consistent units and reference temperature when making comparisons.

Exercise 4.3: Evaluate the following integral using three- and four-point Gauss integrations:

$$erf(p) = \frac{2}{\sqrt{\pi}}\int_0^p e^{-x^2}\,dx$$

To get a value for p, generate a random integer between 0 and 300 using the RANDBETWEEN function. Then let p = the random integer divided by 100 (this gives a floating point number between 0 and 3). Be sure to make a "copy" of p because the random number generator will keep changing it every time a mouse or keyboard command is given.

This is the familiar "error" function (Gaussian normal probability distribution), and the exact value when $p = 1$ is 0.84270073517 (this can be verified by using the Excel function `ERF(p)`). Remember to change variables so that the interval of integration is from –1 to 1.

What is the percent error produced by the Gauss method for the value of p generated?

Exercise 4.4: Evaluate the following integral using

a. The trapezoidal rule. Experiment with the Δx increment to produce good results.
b. Gauss quadrature. Use 2-, 3-, and 4-point formulas and compare results.

$$I = \int_0^1 \sqrt{x}\,dx$$

Exercise 4.5: For the function $x^2\cos x$; $0 \le x \le 1$,

a. Calculate the numerical derivative of this function at each point using Δx values of 0.1 and 0.01. Be sure to use end-point formulas at $x = 0$ and $x = 1$. Compare the numerical derivatives at each point with the exact values.

b. Find the integral of this function using the trapezoidal rule over the same range. Calculate the % error of the integral at $x = 1$ for both Δx values.

Exercise 4.6: For the function $f(x) = \sqrt{x}\sin^2(x)$, do the following operations:

1. Make a table of the function between $a = 0.4$ and $b = 1.6$ using a Δx of 0.1.
2. Generate a column of random numbers between 0 and 1 using the Excel function `RAND()`. Make a *copy* of the random numbers in another column (i.e., Paste by Value). This is so the random numbers do not keep changing.
3. In the next column, generate numbers according to the formula (*Rand* – 0.5)/10. In other words, subtract 0.5 from the column of random numbers and divide the result by 10. This constitutes a column of "noise" to be added to the original function values.
4. Generate a column of numbers by adding to $f(x)$ from part 1 the noise values of part 3.
5. Compute the numerical first derivative, $f'(x)$, from the original function values (no noise added).
6. Compute the numerical first derivative, $f'(x)$, from the noisy function values (after noise has been added).
7. From a graph of noisy $f(x)$ values versus x, add a trendline and have the equation displayed on the graph.
8. Differentiate the trendline equation *analytically* and evaluate it for each value of x.
9. Produce a graph of the three columns of derivative calculations (steps 6, 7, and 8) versus x.
10. Comment on the agreement (or disagreement) between the three derivative estimates.

Exercise 4.7: Shown below are compressibility data for nitrogen:

	Compressibility Factor, *z*		
Pressure (atm)	**0°C**	**25°C**	**50°C**
0	1.000	1.000	1.000
10	0.996	0.998	1.000
50	0.985	0.996	1.004
100	0.984	1.004	1.018
200	1.036	1.057	1.072
300	1.134	1.146	1.154
400	1.256	1.254	1.253
600	1.524	1.495	1.471
800	1.798	1.723	1.697

a. The pure component fugacity of a substance can be computed from

$$\ln\left(\frac{f}{P}\right) = \int_0^P \frac{z-1}{P}dP$$

where z is the compressibility factor $\left(P\hat{V}/RT\right)$, P is the pressure, R is the gas constant, T is the absolute temperature, and $\hat{V}$ is the specific volume.

Using the data in the table, compute the pure component fugacity of nitrogen at 0°C, 25°C, and 50°C. Since the data are not equally spaced, use of the trapezoidal rule is suggested to perform the numerical integration.

b. The pure component enthalpy relative to zero enthalpy at 0 atm and 25°C is given by

$$h(P, 25°C) = \int_0^P -\frac{RT^2}{P}\left(\frac{\partial z}{\partial T}\right)_P dP$$

The derivative of z with respect to T at constant P is required within the integrand. Since data are available at 0°C, 25°C, and 50°C, this derivative can be estimated using a centered difference approximation at each pressure. Compute h from this formula for nitrogen at all pressures given. Note that at zero pressure, the integrand is indeterminate (0/0). Using L'Hopital's rule, the value of the integrand is given by

$$-RT^2\left(\frac{\partial^2 z}{\partial T \partial P}\right)_{P=0}$$

The derivative in this expression can be evaluated from the data by first evaluating $\left(\frac{\partial z}{\partial T}\right)_P$ at P = 0, 200, and 400 atm and then using the "left end-point" second-order correct finite difference formula for the first derivative (to get the derivative with respect to P). The resulting value of the integrand is –1.296 cal/gmol-atm (this should be verified).

Exercise 4.8: A process engineer is performing tests on a unit that has been giving problems. A crucial measurement is the *average* concentration of a particular component in the feed stream to the unit. The analytical method available for determining the concentration of this key component is very expensive and time consuming, such that the budget allows only four samples to be drawn for the purpose of determining the average concentration. Further, the experiment will take place over a 2-h period. Suggest *when* the samples should be taken and how the best possible average concentration can be determined.

Hint: One way to compute an average of a function, *f*(*t*), is to integrate over the time interval and divide by the interval width. That is,

$$Average(f(t)) = \frac{1}{t_{\max}} \int_0^{t_{\max}} f(t)\,dt$$

5 Ordinary Differential Equations (Initial Value Problems)

5.1 INTRODUCTION

Many differential equations defy analytical solution. Still others are such that analytical solutions are onerous. In these cases, a numerical solution is usually the best (and sometimes the only) option. In this chapter, several methods are presented for solving single or multiple ordinary differential equation(s) numerically. Here, only initial value problems (IVPs) are considered, where all necessary information is given at the origin of the independent variable (usually time or distance). The coverage of methods is not complete. Only the more popular methods used in engineering problem solving are considered. These include the Euler, backward Euler, trapezoidal, and Runge–Kutta (RK) methods.

5.1.1 General Statement of the Problem

The IVP involving first-order ordinary differential equation(s) can be written as follows:

$$\frac{dy}{dt} = f(y,t), \quad y(0) = y_0 \tag{5.1}$$

where $f(y, t)$ is an n-vector function of the n-vector y. t is the independent variable, and y_0 is an n-vector of the initial condition(s). When $n = 1$, there is a single ordinary differential equation (ODE).

5.2 EULER-TYPE METHODS

5.2.1 Euler's Method for Single ODE

The simplest numerical method for solving one ODE is called the Euler method and is based on approximating the derivative with a forward difference approximation to the derivative as follows:

$$\frac{y_{n+1} - y_n}{h} \cong f(y_n, t_n) \tag{5.2}$$

where

$$\begin{aligned} y_{n+1} &= y(t_n + h) \\ y_n &= y(t_n) \\ h &= \Delta t \end{aligned} \tag{5.3}$$

The Euler approximation can be written in terms of a "recurrence" relation as follows:

$$y_{n+1} = y_n + hf(y_n, t_n) \tag{5.4}$$

Starting with the initial condition, this recurrence formula can be used to "step forward in time" as follows:

$$\begin{aligned} y_1 &= y_0 + hf(y_0, 0) \\ y_2 &= y_1 + hf(y_1, h) \\ &\vdots \\ y_n &= y_{n-1} + hf(y_{n-1}, (n-1)h) \end{aligned} \tag{5.5}$$

This kind of sequential calculation is called an *explicit* method.

Example 5.1: Euler Method for an ODE-IVP

$$\frac{dy}{dt} = -20y + 7\exp(-0.5t), \quad y(0) = 5 \tag{5.6}$$

The analytical solution is obtained easily using an integrating factor to give

$$y = 5e^{-20t} + (7/19.5)(e^{-0.5t} - e^{-20t}) \tag{5.7}$$

Using the Euler method with $h = 0.01$, the first few steps of the calculations are as follows:

$$y_1 = 5 + 0.01(-20 * 5 + 7 * \exp(0)) = 4.07000$$

$$y_2 = 4.07 + 0.01(-20 * 4.07 + 7 * \exp(-0.005)) = 3.32565$$

Figure 5.1 summarizes the calculations for t up to 0.1 using $h = 0.01$. The values from the analytical solution are also shown for comparison.

t	y	y **exact**
0.00	5.00000	5.00000
0.01	4.07000	4.15693
0.02	3.32565	3.46638
0.03	2.72982	2.90068
0.04	2.25282	2.43721
0.05	1.87087	2.05745
0.06	1.56497	1.74622
0.07	1.31990	1.49109
0.08	1.12352	1.28191
0.09	0.96607	1.11033
0.10	0.83977	0.96956

FIGURE 5.1 Euler method results for $h = 0.01$.

From the table, it can be seen that the disagreement between the numerical solution and analytical solution grows with t at a significant rate. This suggests that a smaller value for h is required. The percentage error for $h = 0.01$, 0.001, and 0.0001, respectively, is shown in the graph in Figure 5.2.

It can be observed that the error decreases approximately one order of magnitude for every order of magnitude decrease in h. This demonstrates "empirically" that this method is of $\mathcal{O}(h)$. One example is not the proof of this error behavior for Euler's method, but it is persuasive, and in practice, this is borne out to be the case.

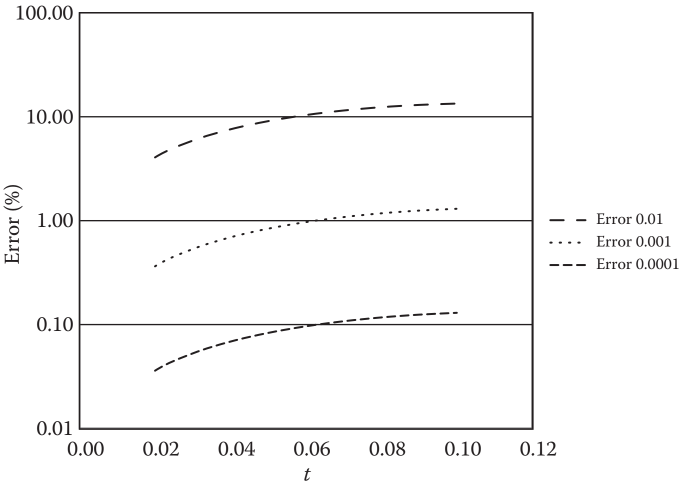


FIGURE 5.2 Error using the Euler method when changing h.

5.2.2 Stability of Numerical Solutions of ODEs

It can be instructive to study the simple ODE-IVP

$$\frac{dy}{dt} = -ay; \quad y(0) = 1 \quad a > 0 \tag{5.8}$$

whose exact solution is $y(t) = \exp(-at)$. $1/a$ is called the system *time constant*. Applying the Euler method, there results

$$y_{n+1} = (1 - ah)y_n \tag{5.9}$$

Since the exact solution decreases exponentially with t, it is clear that the approximate solution should also decrease continuously. Therefore, it is necessary that

$$(1 - ah) < 1 \textit{ or } 0 < ah < 1 \qquad \text{(note: } ah \text{ cannot be negative)}$$

Further, if $1 < ah < 2$, then the solution *alternates sign* at each step. And, if $ah > 2$, the *magnitude* of the solution increases at each step and it *oscillates*—this is called *instability*.

This is a new kind of error. Truncation error exists when h is too large, round-off error occurs when h is too small, and the numerical solution exhibits instability depending on the product of the system time constant and h.

Although this analysis has been performed only for the simplest of ODEs, in practice, this same behavior is often observed. Obviously, some care must be given to the selection of h to avoid unacceptable errors of any kind.

5.2.3 Euler Backward Method

Consider a backward finite difference approximation for the derivative

$$\frac{y_{n+1} - y_n}{h} \cong f(y_{n+1}, t_{n+1}) \tag{5.10}$$

This equation is “implicit” in y_{n+1} since it appears on both sides of the equation. It can be shown that this method is also $\mathcal{O}(h)$.

Applying this approximation to the exponential test problem given by Equation 5.8, the following recurrence results:

$$y_{n+1} = \frac{1}{(1 + ah)} y_n \tag{5.11}$$

This solution decreases monotonically with t for *any* positive value of h. While it is said to be *unconditionally stable*, it still suffers significant truncation error. Also, if $f(y, t)$ is not linear, then a nonlinear equation must be solved at each time step.

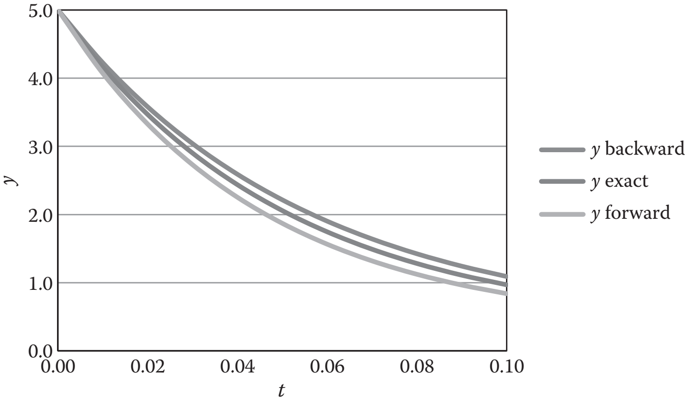


FIGURE 5.3 Comparison of Euler with backward Euler method.

Example 5.2: Backward Euler Method

The following steps show the application of the backward Euler method to the ODE-IVP of Equation 5.6:

$$\frac{y_{n+1}-y_n}{h} = -20y_{n+1} + 7\exp(-0.5t_{n+1})$$

$$(1+20h)y_{n+1} = y_n + 7h\exp(-0.5t_{n+1}) \tag{5.12}$$

$$y_{n+1} = \frac{y_n + 7h\exp(-0.5t_{n+1})}{(1+20h)}$$

Shown in Figure 5.3 is a comparison of the Euler and backward Euler methods for this problem. Only results for $h = 0.01$ are shown. For this example (and it often happens that) one of these methods errs on one side of the solution and the other on the other side.

Here, the Euler method "undershoots" the exact solution, while the backward Euler method "overshoots" it. This suggests that an average of the two methods might give better results, and this is indeed that case (see Section 5.2.4).

5.2.4 Trapezoidal Method (Modified Euler)

This method applies a centered difference approximation for the derivative and an average value for f on the right-hand side. The finite difference analog is "centered about the ½ point." That is, the differential equation is discretized at $t_{n+1/2}$, and $f(y_{n+1/2}, t_{n+1/2})$ is approximated by the average of the end-point values. Therefore, this method is an average of the forward and backward Euler methods, and the discrete approximation appears in the following equation:

$$\frac{y_{n+1}-y_n}{h} = \frac{1}{2}[f(y_{n+1},t_{n+1}) + f(y_n,t_n)] \tag{5.13}$$

This equation is implicit in y_{n+1}. If f is nonlinear, then a nonlinear equation must be solved at each time step. It can be shown that this method is $\mathcal{O}(h^2)$.

Looking again at the exponential test problem given by Equation 5.8 and applying the trapezoidal method, the following recurrence results:

$$y_{n+1} = \frac{(1 - ah/2)}{(1 + ah/2)} y_n \tag{5.14}$$

This equation is stable for $0 < ah/2 < 1$ or $0 < ah < 2$.

Example 5.3: Trapezoidal Method

Applying the trapezoidal method to the ODE-IVP of Equation 5.6 leads to the following:

$$\begin{aligned} \frac{y_{n+1} - y_n}{h} &= \frac{1}{2}[-20y_{n+1} + 7\exp(-0.5t_{n+1}) - 20y_n + 7\exp(-0.5t_n)] \\ (1+10h)y_{n+1} &= y_n + \frac{h}{2}[7\exp(-0.5t_{n+1}) + 7\exp(-0.5t_n) - 20y_n] \\ y_{n+1} &= \frac{y_n + \frac{h}{2}[7\exp(-0.5t_{n+1}) + 7\exp(-0.5t_n) - 20y_n]}{(1+10h)} \end{aligned} \tag{5.15}$$

Figure 5.4 shows results for h = 0.01 and h = 0.001. It can be observed that the error is reduced by two orders of magnitude when h is decreased by only

***y*(0) =**	**5**			**Percent**	
t	***y*(*h* = 0.01)**	***y*(*h* = 0.001)**	***y* exact**	**error(0.01)**	**error(0.001)**
0	5	5	5	0	0
0.01	4.15439	4.15691	4.15693	0.061286	0.000609
0.02	3.46220	3.46633	3.46638	0.120305	0.001197
0.03	2.89556	2.90063	2.90068	0.176501	0.001756
0.04	2.43163	2.43716	2.43721	0.229239	0.002282
0.05	2.05173	2.05739	2.05745	0.277817	0.002766
0.06	1.74060	1.74616	1.74622	0.321490	0.003202
0.07	1.48573	1.49104	1.49109	0.359504	0.003582
0.08	1.27689	1.28186	1.28191	0.391146	0.003899
0.09	1.10572	1.11029	1.11033	0.415804	0.004146
0.1	0.96536	0.96952	0.96956	0.433033	0.004319

FIGURE 5.4 Trapezoidal method results.

one order of magnitude. This empirical observation is evidence that the method is $\mathcal{O}(h^2)$.

5.2.5 Accuracy of Euler-Type Methods

It has been stated that the Euler and backward Euler methods are $\mathcal{O}(h)$ and that the trapezoidal method is $\mathcal{O}(h^2)$. These results can be shown more convincingly for the exponential test problem given by Equation 5.8, the exact solution of which is the negative exponential. This can be written in the following incremental form:

$$y(t) = \exp(-at)$$
$$y_{n+1} = y_n \exp(-ah) = \left(1 - ah + \frac{1}{2}(ah^2) - \frac{1}{6}(ah)^3 + \cdots\right) y_n \tag{5.16}$$

The original Euler method has the recurrence (see Equation 5.9)

$$y_{n+1} = (1 - ah)y_n$$

which represents the first two terms in the series solution. The error term, therefore, is proportional to h^2, but this error is made at *each step*. The global error at the end of many steps is proportional to h itself, so the entire process is $\mathcal{O}(h)$.

For the backward Euler, the recurrence is

$$y_{n+1} = \frac{1}{(1+ah)} y_n = (1 - (ah) + (ah)^2 - (ah)^3 + \cdots)y_n \tag{5.17}$$

This equation, when compared to the exact series expansion, is accurate to the first two terms, and the term involving h^2 is the magnitude of the error. So, once again, each step is $\mathcal{O}(h^2)$ while the global error is $\mathcal{O}(h)$.

For the trapezoidal method,

$$y_{n+1} = \frac{(1 - ah/2)}{(1 + ah/2)} y_n = \left(1 - (ah) + \frac{1}{2}(ah)^2 - \frac{1}{4}(ah)^3 + \cdots)\right) y_n \tag{5.18}$$

This equation is in agreement with the first three terms of the exact expansion, so the step error is $\mathcal{O}(h^3)$, while the global error is $\mathcal{O}(h^2)$.

5.3 RK METHODS

These are among the most popular methods for solving ODE-IVPs. They are *explicit* and are based on the idea of using intermediate points in each major time step (note: even though the independent variable often is time, it can be distance, volume, or some other quantity). Recall that any explicit method can encounter stability issues if the time step is not carefully chosen.

Consider the general ODE-IVP

$$\frac{dy}{dt} = f(y,t), \quad y(0) = y_0 \tag{5.19}$$

To calculate y_{n+1} at $t_{n+1} = t_n + h$ with a known value of y_n, Equation 5.19 can be integrated over the interval $[t_n, t_{n+1}]$ as

$$y_{n+1} = y_n + \int_{t_n}^{t_{n+1}} f(y,t)\,dt \tag{5.20}$$

RK methods are derived by applying a numerical integration method to the integral on the right-hand side.

5.3.1 Second-Order RK Method

It is straightforward to derive the recurrence relations for the second-order RK method. Suppose that the trapezoidal rule is used to evaluate the integral on the right-hand side in Equation 5.20:

$$\int_{t_n}^{t_{n+1}} f(y,t)\,dt = \frac{1}{2}h[f(y_n,t_n) + f(y_{n+1},t_{n+1})] \tag{5.21}$$

Since y_{n+1} is not known, consider approximating it by $f(\bar{y}_{n+1},t_{n+1})$ where $\bar{y}_{n+1}$ is a "first estimate" for y_{n+1} calculated by the forward Euler method:

$$\begin{aligned} \bar{y}_{n+1} &= y_n + hf(y_n,t_n) \\ y_{n+1} &= y_n + \frac{1}{2}h[f(y_n,t_n) + f(\bar{y}_{n+1},t_{n+1})] \end{aligned} \tag{5.22}$$

A standard computational notation is as follows:

$$\begin{aligned} k_1 &= hf(y_n,t_n) \\ k_2 &= hf(y_n + k_1,t_{n+1}) \\ y_{n+1} &= y_n + \frac{1}{2}[k_1 + k_2] \end{aligned} \tag{5.23}$$

Without proof, this method is $\mathcal{O}(h^2)$.

5.3.2 Fourth-Order RK Method

The fourth-order RK method can be derived in a manner similar to that for the second-order method. The basic idea, once again, is to subdivide the interval h and to use successive approximations to y_{n+1}. The final y_{n+1} is a weighted average of the individual approximations. The resulting equations are

$$\begin{aligned}
k_1 &= hf(y_n, t_n) \\
k_2 &= hf\left(y_n + \frac{k_1}{2}, t_n + \frac{h}{2}\right) \\
k_3 &= hf\left(y_n + \frac{k_2}{2}, t_n + \frac{h}{2}\right) \\
k_4 &= hf(y_n + k_3, t_n + h) \\
y_{n+1} &= y_n + \frac{1}{6}[k_1 + 2k_2 + 2k_3 + k_4]
\end{aligned} \tag{5.24}$$

This method can be shown to be $\mathcal{O}(h^4)$ and is perhaps the most popular method for solving ODE-IVPs numerically.

Example 5.4: Second- and Fourth-Order RK Methods

Suppose chemical A is in solution in a perfectly stirred tank and its concentration is $C_A^0(\text{g/L})$. The constant volumetric flow into and out of the tank is F (L/min) and the tank volume is V(L). A mass balance on component A leads to

$$V\frac{dC_A}{dt} = -FC_A \tag{5.25}$$

The analytical solution is

$$C_A = C_A^0 e^{-tF/V} \tag{5.26}$$

Calculations for the second-order RK method are shown in Figure 5.5. Also shown are the analytical (exact) solution and that produced by the Euler method using the same $h = 0.1$ min.

It is obvious that the second-order RK method produces significantly better results than those of the Euler method. Figure 5.5 also illustrates that it is straightforward to implement this method in Excel®. Similar calculations using the fourth-order RK method are shown in Figure 5.6.

From Figure 5.6, it can be seen that the fourth-order RK method produces essentially exact results for this problem. As previously stated, the overall error associated with the fourth-order RK method is $\mathcal{O}(h^4)$. The method is easy to implement in Excel since it is an *explicit* method.

d*t* =	0.1	Tau =	0.5		
Time	RK2	k_1	k_2	Exact	Euler
0	1.0000	−0.2000	−0.1600	1.0000	1.0000
0.1	0.8200	−0.1640	−0.1476	0.8187	0.8000
0.2	0.6642	−0.1328	−0.1196	0.6703	0.6400
0.3	0.5380	−0.1076	−0.0968	0.5488	0.5120
0.4	0.4358	−0.0872	−0.0784	0.4493	0.4096
0.5	0.3530	−0.0706	−0.0635	0.3679	0.3277
0.6	0.2859	−0.0572	−0.0515	0.3012	0.2621
0.7	0.2316	−0.0463	−0.0417	0.2466	0.2097
0.8	0.1876	−0.0375	−0.0338	0.2019	0.1678
0.9	0.1519	−0.0304	−0.0274	0.1653	0.1342
1	0.1231	−0.0246	−0.0222	0.1353	0.1074

FIGURE 5.5 Second-order RK results for mixing tank.

d*t* =	0.1	Tau =	0.5				
Time	RK4	k_1	k_2	k_3	k_4	Exact	Euler
0	1.0000	−0.2000	−0.1800	−0.1820	−0.1636	1.0000	1.0000
0.1	0.8187	−0.1637	−0.1474	−0.1490	−0.1339	0.8187	0.8000
0.2	0.6703	−0.1341	−0.1207	−0.1220	−0.1097	0.6703	0.6400
0.3	0.5488	−0.1098	−0.0988	−0.0999	−0.0898	0.5488	0.5120
0.4	0.4493	−0.0899	−0.0809	−0.0818	−0.0735	0.4493	0.4096
0.5	0.3679	−0.0736	−0.0662	−0.0670	−0.0602	0.3679	0.3277
0.6	0.3012	−0.0602	−0.0542	−0.0548	−0.0493	0.3012	0.2621
0.7	0.2466	−0.0493	−0.0444	−0.0449	−0.0403	0.2466	0.2097
0.8	0.2019	−0.0404	−0.0363	−0.0367	−0.0330	0.2019	0.1678
0.9	0.1653	−0.0331	−0.0298	−0.0301	−0.0270	0.1653	0.1342
1	0.1353	−0.0271	−0.0244	−0.0246	−0.0221	0.1353	0.1074

FIGURE 5.6 Fourth-order RK results for mixing tank.

5.4 STIFF ODEs

Stiffness often refers to a system with a very short time constant. For the system model

$$y' = -ay + s(t);\ y(0) = y_0 \tag{5.27}$$

the time constant is $1/|a|$; if a is large, the time constant is small. Note that the units of a are time^{-1}.

The solution to this equation can be written as

$$y(t) = y_0 e^{-at} + e^{-at} \int_0^t s(x) e^{ax}\, dx \tag{5.28}$$

Even if a is very large, if $s(t)$ is a slowly varying function, the system responds slowly. To follow this slow response numerically is problematic since the stability of the method depends only on a and has nothing to do with $s(t)$. A very small time step might be required, even though the overall system response is a slowly varying function. This is one example of a phenomenon called *stiffness*.

Another example of stiffness occurs with a system (two or more) of ODEs each with greatly different time constants. For example, consider the system

$$\begin{aligned} y' &= -y + z + 3 \\ z' &= -10^7 z + y \end{aligned} \tag{5.29}$$

The very short time constant in the second equation requires a very small h to follow both y and z in time. Problems of instability can easily arise. To solve stiff problems, the most typical approach is to use an *implicit* method, which is known to exhibit excellent stability properties (recall the backward Euler method). Special software packages are available for solving stiff systems. Fortunately, many of the straightforward chemical engineering problems encountered in practice do not yield stiff systems, but when difficulties arise, stiffness might well be the culprit.

5.5 SOLVING SYSTEMS OF ODE-IVPs

The following is a system of ODE-IVPs:

$$\begin{aligned} \frac{dy_1}{dt} &= f_1(y_1, y_2, \cdots, y_n, t); \qquad y_1(0) = y_{10} \\ \frac{dy_2}{dt} &= f_2(y_1, y_2, \cdots, y_n, t); \qquad y_2(0) = y_{20} \\ &\vdots \\ \frac{dy_n}{dt} &= f_n(y_1, y_2, \cdots, y_n, t); \qquad y_n(0) = y_{n0} \end{aligned} \tag{5.30}$$

The only difference between solving a single ODE-IVP and solving a system of them is that all variables and functions become vectors. This is illustrated in Example 5.5.

Example 5.5: Solving a System of ODE-IVPs

Suppose the following chemical reactions take place in a continuous stirred tank reactor (CSTR):

$$A \underset{k_2}{\overset{k_1}{\Leftrightarrow}} B \underset{k_4}{\overset{k_3}{\Leftrightarrow}} C \tag{5.31}$$

where the rate constants are as follows:

$$k_1 = 1 \text{ min}^{-1},\ k_2 = 0 \text{ min}^{-1},\ k_3 = 2 \text{ min}^{-1},\ k_4 = 3 \text{ min}^{-1}$$

The initial charge to the reactor is all A, so the initial conditions are (in mol/L)

$$C_{A_0} = 1 \quad C_{B_0} = 0 \quad C_{C_0} = 0$$

An unsteady-state mass balance on each component leads to the following set of ODEs:

$$\begin{aligned}
\frac{dC_A}{dt} &= -k_1C_A + k_2C_B \\
\frac{dC_B}{dt} &= k_1C_A - k_2C_B - k_3C_B + k_4C_C \\
\frac{dC_C}{dt} &= k_3C_B - k_4C_C
\end{aligned} \tag{5.32}$$

The following spreadsheet displays a solution to this system using the Euler method for time only up to 0.13 min to conserve space:

***k*₁ =**	**1**	***k*₂ =**	**0**
***k*₃ =**	**2**	***k*₄ =**	**3**
***h* =**	**0.01**		
Time	**CA**	**CB**	**CC**
0.00	1.0000	0.0000	0.0000
0.01	0.9900	0.0100	0.0000
0.02	0.9801	0.0197	0.0002
0.03	0.9703	0.0291	0.0006
0.04	0.9606	0.0382	0.0012
0.05	0.9510	0.0471	0.0019
0.06	0.9415	0.0556	0.0028
0.07	0.9321	0.0639	0.0038
0.08	0.9227	0.0720	0.0050
0.09	0.9135	0.0798	0.0063
0.10	0.9044	0.0873	0.0077
0.11	0.8953	0.0946	0.0092
0.12	0.8864	0.1017	0.0108
0.13	0.8775	0.1085	0.0125

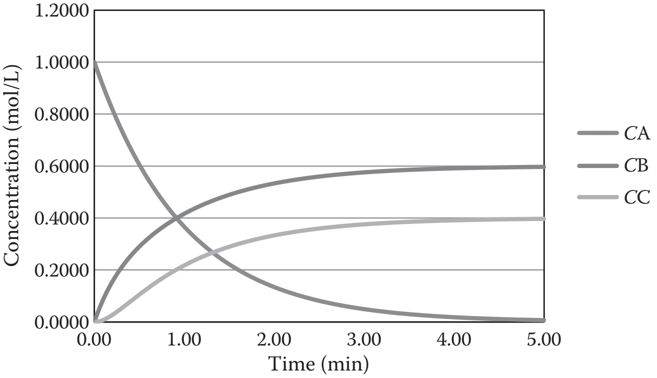


FIGURE 5.7 Euler solution for three simultaneous ODE-IVPs.

Shown in Figure 5.7 is the solution for time up to 5 min at which time a steady state has been essentially reached. It can be observed that component *A* decreases monotonically while *B* and *C* increase.

5.6 HIGHER-ORDER ODEs

Consider a general second-order ODE-IVP of the form

$$\frac{d^2y}{dt^2} = f(y, y', t) \quad y(0) = y_0 \quad y'(0) = y'_0 \tag{5.33}$$

Since all of the methods discussed apply only to first-order equations, one way to solve those of the form of Equation 5.33 is to convert them into two simultaneous first-order equations. This is easily accomplished by defining a *new variable z* as follows:

$$\frac{dy}{dt} = z; \quad y(0) = y_0 \tag{5.34}$$

Equation 5.34 is a first-order ODE-IVP involving both y and z. If this equation is differentiated with respect to t and substituted into Equation 5.33, there results

$$\frac{dz}{dt} = f(y, z, t); \quad z(0) = y'(0) \tag{5.35}$$

Equations 5.34 and 5.35 constitute a system of *coupled* ODE-IVPs. This system can be solved numerically by any of the methods previously discussed.

Example 5.6: Second-Order System Response

A linear second-order dynamic system is defined by the ODE-IVP

$$\tau^2 \frac{d^2y}{d\theta^2} + 2\tau\zeta \frac{dy}{d\theta} + y = f(\theta) \tag{5.36}$$

where

τ = the system *time constant*
ζ = the damping factor
$f(\theta)$ = the system *input* or *forcing* function

and the initial value of y and y' are given.

By defining a dimensionless time as $t = \theta/\tau$ and assuming a constant input of 1 (a unit step function occurring at time zero), the model equation becomes

$$\frac{d^2y}{dt^2} + 2\zeta \frac{dy}{dt} + y = 1 \tag{5.37}$$

The only parameter of this model is the damping factor. The damping factor determines if the system is overdamped ($\zeta > 1$), underdamped or oscillatory ($\zeta < 1$), or critically damped ($\zeta = 1$).

In order to solve this model equation numerically with, for example, the Euler method, it must be converted into two first-order equations as follows:

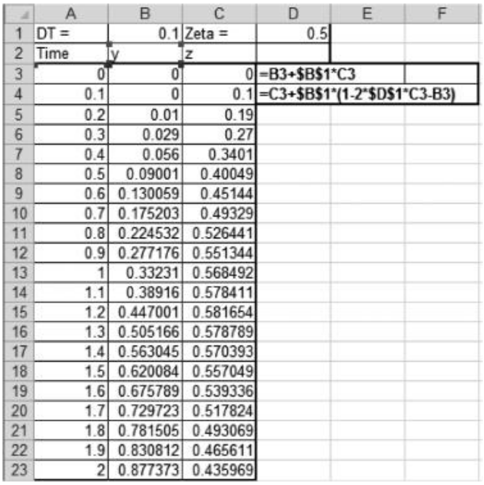
Define a new variable z such that

$$\frac{dy}{dt} = z;\ y(0) = y_0 \tag{5.38}$$

Differentiating this expression with respect to t and substituting into the original model equation, there results

$$\frac{dz}{dt} = 1 - 2\zeta z - y;\ z(0) = y'(0) \tag{5.39}$$

Equations 5.38 and 5.39 are two first-order ODE-IVPs that can be solved by any of the methods previously described. Calculations using the Euler method with $h = 0.1$ and $\zeta = 0.5$ are shown in the following spreadsheet for t up to 2. The formulas for the right-hand side of the equations for y and z are shown in bold type.



	A	B	C	D	E	F
1	DT =	0.1	Zeta =	0.5		
2	Time	y	z			
3	0	0	0	=B3+$B$1*C3		
4	0.1	0	0.1	=C3+$B$1*(1-2*$D$1*C3-B3)		
5	0.2	0.01	0.19			
6	0.3	0.029	0.27			
7	0.4	0.056	0.3401			
8	0.5	0.09001	0.40049			
9	0.6	0.130059	0.45144			
10	0.7	0.175203	0.49329			
11	0.8	0.224532	0.526441			
12	0.9	0.277176	0.551344			
13	1	0.33231	0.568492			
14	1.1	0.38916	0.578411			
15	1.2	0.447001	0.581654			
16	1.3	0.505166	0.578789			
17	1.4	0.563045	0.570393			
18	1.5	0.620084	0.557049			
19	1.6	0.675789	0.539336			
20	1.7	0.729723	0.517824			
21	1.8	0.781505	0.493069			
22	1.9	0.830812	0.465611			
23	2	0.877373	0.435969			

A graph of the system output appears in Figure 5.8. Note that the system response is oscillatory and *overshoots* before settling to the final steady-state value of 1.

Example 5.7: A VBA Program for the Euler Method

In certain circumstances, it can be more efficient to write a VBA program to solve ODE-IVPs numerically. Such cases include times when it is convenient to simply alter a subroutine subprogram that defines the right-hand side function and also when there is a large system of ODE-IVPs involved. This example shows the development of a VBA program to implement Euler's method for any number of simultaneous ODEs. First, a program Specification is given that defines the Excel interface to the program followed by an Algorithm Design, Coding in VBA, and Testing using a problem previously solved using only Excel.

SPECIFICATION

The following "mock-up" of an Excel spreadsheet for solving any number of ODE-IVPs using the Euler method represents a program specification:

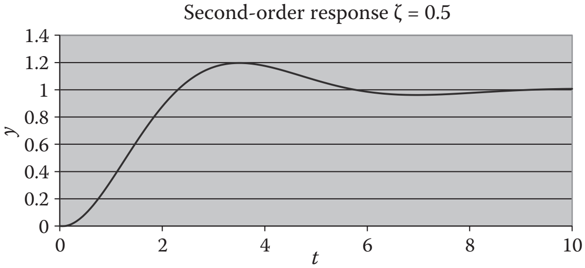
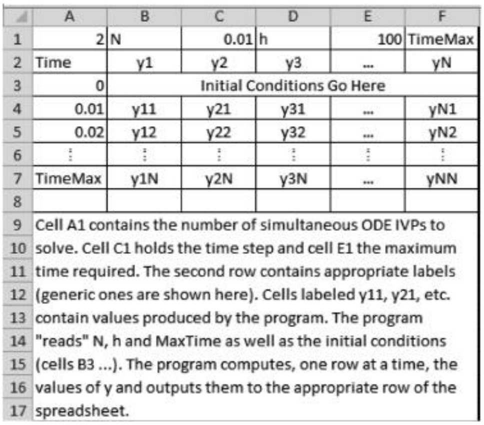


FIGURE 5.8 Response of a second-order system to a step response.



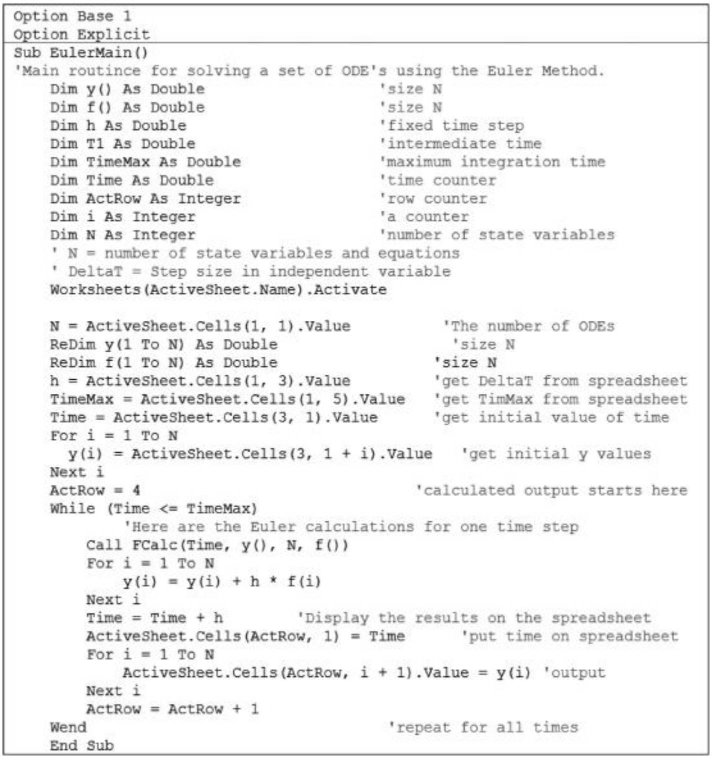
	A	B	C	D	E	F
1	2	N	0.01	h	100	TimeMax
2	Time	y1	y2	y3	...	yN
3	0	Initial Conditions Go Here				
4	0.01	y11	y21	y31	...	yN1
5	0.02	y12	y22	y32	...	yN2
6	⋮	⋮	⋮	⋮	⋮	⋮
7	TimeMax	y1N	y2N	y3N	...	yNN
8						
9	Cell A1 contains the number of simultaneous ODE IVPs to					
10	solve. Cell C1 holds the time step and cell E1 the maximum					
11	time required. The second row contains appropriate labels					
12	(generic ones are shown here). Cells labeled y11, y21, etc.					
13	contain values produced by the program. The program					
14	"reads" N, h and MaxTime as well as the initial conditions					
15	(cells B3 ...). The program computes, one row at a time, the					
16	values of y and outputs them to the appropriate row of the					
17	spreadsheet.					

ALGORITHM DESIGN

The structure chart of Figure 5.9 shows an algorithm design for the main program and for a subroutine to calculate the right-hand side functions. A dictionary of variables is also given.

VBA CODE

A listing of the VBA code for this program is shown below:



```
Option Base 1
Option Explicit
Sub EulerMain()
'Main routince for solving a set of ODE's using the Euler Method.
    Dim y() As Double                        'size N
    Dim f() As Double                        'size N
    Dim h As Double                          'fixed time step
    Dim T1 As Double                         'intermediate time
    Dim TimeMax As Double                    'maximum integration time
    Dim Time As Double                       'time counter
    Dim ActRow As Integer                    'row counter
    Dim i As Integer                         'a counter
    Dim N As Integer                         'number of state variables
    ' N = number of state variables and equations
    ' DeltaT = Step size in independent variable
    Worksheets(ActiveSheet.Name).Activate

    N = ActiveSheet.Cells(1, 1).Value               'The number of ODEs
    ReDim y(1 To N) As Double                        'size N
    ReDim f(1 To N) As Double                      'size N
    h = ActiveSheet.Cells(1, 3).Value              'get DeltaT from spreadsheet
    TimeMax = ActiveSheet.Cells(1, 5).Value        'get TimMax from spreadsheet
    Time = ActiveSheet.Cells(3, 1).Value           'get initial value of time
    For i = 1 To N
      y(i) = ActiveSheet.Cells(3, 1 + i).Value     'get initial y values
    Next i
    ActRow = 4                                   'calculated output starts here
    While (Time <= TimeMax)
            'Here are the Euler calculations for one time step
        Call FCalc(Time, y(), N, f())
        For i = 1 To N
            y(i) = y(i) + h * f(i)
        Next i
        Time = Time + h        'Display the results on the spreadsheet
        ActiveSheet.Cells(ActRow, 1) = Time       'put time on spreadsheet
        For i = 1 To N
            ActiveSheet.Cells(ActRow, i + 1).Value = y(i) 'output
        Next i
        ActRow = ActRow + 1
    Wend                                     'repeat for all times
    End Sub
```

Name	Usage
N	The number of simultaneous ODEs to solve
h	The time step (or step for any independent variable)
TimeMax	The maximum time over which the solution is sought
y	N-length vector of dependent variables
ActRow	Pointer to current row in spreadsheet
Time	Current value of time
f	N-length vector of right side functions
Func	Subroutine to calculate f given y and N
Zeta	Damping factor for second order system

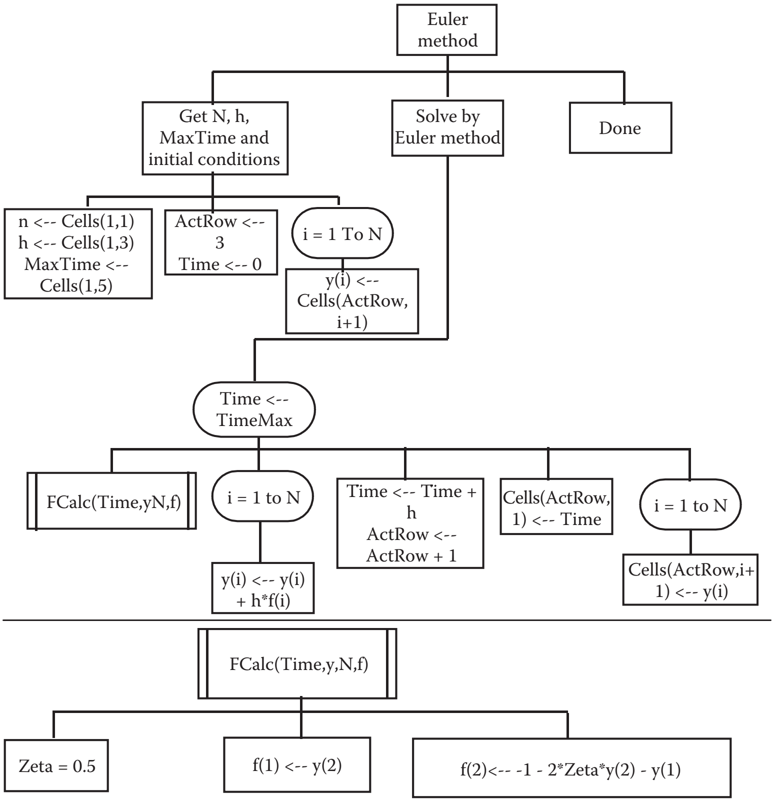
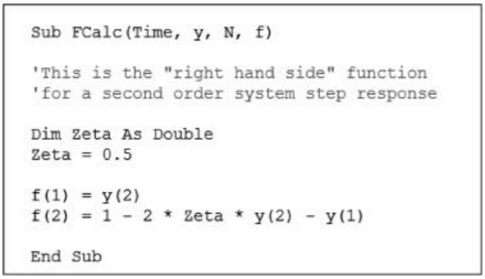


FIGURE 5.9 Algorithm design for Euler method.



```
Sub FCalc(Time, y, N, f)

'This is the "right hand side" function
'for a second order system step response

Dim Zeta As Double
Zeta = 0.5

f(1) = y(2)
f(2) = 1 - 2 * Zeta * y(2) - y(1)

End Sub
```

PROGRAM EXECUTION

The following spreadsheet shows program execution up to Time = 1. These results are identical to those of Example 5.6.

2	**= N**	**0.1**	**= h**	**10**	**= TimeMax**
T	*y*	*z*			
0	0.0000	0.0000			
0.1	0.0000	0.1000			
0.2	0.0100	0.1900			
0.3	0.0290	0.2700			
0.4	0.0560	0.3401			
0.5	0.0900	0.4005			
0.6	0.1301	0.4514			
0.7	0.1752	0.4933			
0.8	0.2245	0.5264			
0.9	0.2772	0.5513			
1	0.3323	0.5685			

EXERCISES

Exercise 5.1: Dynamic flow in a tank can be modeled by making a mass balance on the fluid in the tank (Figure 5.10). The nature of the resulting ODE-IVP depends on the model used for the outlet valve. If a linear valve is

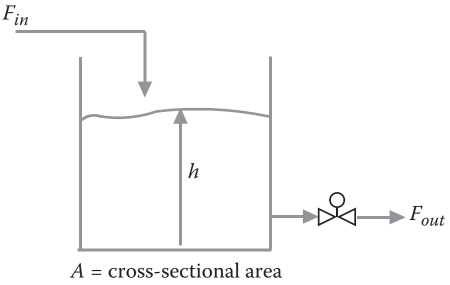


FIGURE 5.10 Dynamic flow in a tank schematic.

assumed, then the model ODE is linear. A more accurate valve description makes the outlet flow a nonlinear function of the height. Both of these cases are considered in that which follows.

Mass balance:

d(mass in tank at any time)/dt = Input – Output + Generation

Assume a pure (single) component of constant density fluid:

$$\frac{d\rho V}{dt} = F_{in} - F_{out} \tag{5.40}$$

where

ρ = density
V = volume
F = mass flow rate

In terms of cross-sectional area and height,

$$\frac{d\rho Ah}{dt} = F_{in} - F_{out} \tag{5.41}$$

Since density and area are constant,

$$\frac{dh}{dt} = \frac{F_{in} - F_{out}}{\rho A} \tag{5.42}$$

F_{out} can be

a. Directly proportional to h: $F_{out} = C_v\ h$ in which case the model equation is

$$\frac{dh}{dt} = \frac{F_{in} - C_v h}{\rho A} \tag{5.43}$$

b. A function of h, such as $F_{out} = C_v\ h^{1/2}$ and the model becomes

$$\frac{dh}{dt} = \frac{F_{in} - C_v\sqrt{h}}{\rho A} \tag{5.44}$$

Additional pertinent data are as follows:

- Tank diameter = 6 ft
- Density of liquid = 62.5 lb/ft^3
- Valve constant C_v = 250 lb/h-ft for the linear case

- Valve constant C_v = 500 lb/h-ft$^{0.5}$ for the nonlinear case
- Initial height = 4 ft
- F_{in} = 1000 lb/h (initial steady-state inlet flow)
- t = time (h)

F_{in} undergoes a change at time 0+ to 1400 lb/h until time 10 h, after which it returns to 1000 lb/h.

a. Solve the linear ODE (F_{out} proportional to h) using Euler's method. Start with a Δt of 0.5 and again with $\Delta t = 0.05$ to determine an acceptable Δt. Use an `IF` function to generate the proper values for F_{in}.
b. Repeat part a for the nonlinear case (F_{out} proportional to $h^{1/2}$).

Exercise 5.2: Solve the ODEs of Exercise 5.1 using the second-order RK method. Find acceptable values of h when using the second-order RK method.

Exercise 5.3: Solve the ODEs of Exercise 5.1 using the Euler VBA program as given in Example 5.7.

Exercise 5.4: Suppose the following chemical reactions take place in a continuous stirred tank reactor (CSTR):

$$A \underset{k_2}{\overset{k_1}{\Leftrightarrow}} B$$

where the rate constants are as follows:

$$k_1 = 1\ \text{min}^{-1},\ k_2 = 0.5\ \text{min}^{-1}$$

The charge to the reactor is all component A, so initially, the concentrations within the reactor are

$$C_{A_0} = 1 \quad C_{B_0} = 0\ \text{(gmol/L)}$$

An unsteady-state mass balance on each component leads to the following set of ODEs:

$$\frac{dC_A}{dt} = -k_1 C_A + k_2 C_B$$
$$\frac{dC_B}{dt} = k_1 C_A - k_2 C_B \tag{5.45}$$

Solve this ODE-IVP system using the Euler method in Excel with a time step of 0.1 min and repeat with a time step of 0.01 min. If a significant difference results (compare values at a specific time to see if they are different), repeat with a time step of 0.001, etc. until a satisfactory time step is found. Integrate out to time = 2 min. Plot the solutions in each case and compare them. Report the percent difference in the solutions at time = 1 min.

Perform experiments by changing the rate constants in one of the solutions to see how Excel immediately recalculates the solution and updates the graph.

Exercise 5.5: Solve the ODEs of Exercise 5.4 using the second-order RK method. Compare results for $\Delta t = 0.1$ min and for $\Delta t = 0.01$ min.

Exercise 5.6: Solve the ODEs of Exercise 5.4 using the Euler VBA program as given in Example 5.7. Use time steps of 0.1, 0.01, and 0.001 min, respectively, and deduce which is the most appropriate for an acceptable solution.

Exercise 5.7: Solve the ODEs of Exercise 5.7 using the Euler VBA program as given in Example 5.7. Use time steps of 0.1, 0.01, and 0.001 min and compare the results to find an acceptable solution.

Exercise 5.8: Implement the second-order RK method for *any number* of simultaneous ODE-IVPs in VBA. Use Example 5.7 as a model for designing, coding, and testing the program. Solve the problem of Exercise 5.7 to test the program.

Exercise 5.9: Penicillin Fermentation

A model for a batch reactor in which penicillin is produced by fermentation has been derived as follows (Constantantinides et al. 1970) for cell production and penicillin synthesis, respectively:

$$\begin{aligned} \frac{dy_1}{dt} &= b_1 y_1 - \frac{b_1}{b_2} y_1^2 \qquad y_1(0) = 0.03 \\ \frac{dy_2}{dt} &= b_3 y_1 \qquad\qquad\quad y_2(0) = 0.0 \end{aligned} \tag{5.46}$$

where

y_1 = dimensionless concentration of cell mass
y_2 = dimensionless concentration of penicillin
t = dimensionless time, $0 \le t \le 1$

Experiments have determined that

$$b_1 = 13.1$$

$$b_2 = 0.94$$

$$b_3 = 1.71$$

a. Solve this ODE-IVP using the Euler method and Excel (not VBA).
b. Solve this ODE-IVP using the second-order RK method and Excel (not VBA).
c. Solve this ODE-IVP using the Euler VBA program as given in Example 5.7.
d. Solve this ODE-IVP using the second-order RK VBA program from Exercise 5.8.

In all cases, experiment with the time step to assure an accurate solution. Also, graph y_1 and y_2 versus time with appropriate annotations.

Exercise 5.10: Bioreaction Kinetics

The "Monod" model for bioreaction kinetics can be expressed as

$$\begin{aligned} \frac{ds}{dt} &= -\frac{ksx}{k_s + s} \\ \frac{dx}{dt} &= y\frac{ksx}{k_s + s} - bx \end{aligned} \qquad s(0) = s_o, x(0) = x_o \tag{5.47}$$

where

s = Growth limiting substrate concentration (ML^{-3})
x = Biomass concentration (ML^{-3})
k = Maximum specific uptake rate of the substrate (T^{-1}) = 5
k_s = Half saturation constant for growth (ML^{-3}) = 20
y = Yield coefficient (MM^{-1}) = 0.05
b = Decay coefficient (T^{-1}) = 0.01

Initial conditions are s_o = 1000 and x_o = 100.

a. Solve this ODE-IVP using the Euler method and Excel (not VBA).
b. Solve this ODE-IVP using the second-order RK method and Excel (not VBA).
c. Solve this ODE-IVP using the Euler VBA program as given in Example 5.7.
d. Solve this ODE-IVP using the second-order RK VBA program from Exercise 5.8.

An appropriate maximum time over which to integrate these equations must be determined by experimentation. Also, experiment with the time step to guarantee good results. Graph s and x versus time with appropriate annotations (do this with data for only an appropriate time step).

Exercise 5.11: Implement the fourth-order RK method for *any number* of simultaneous ODE-IVPs in VBA.

The only thing a user must change to use the program is the Subroutine to calculate the "right-hand side" functions for the ODEs. The user must also specify the required input data. A *suggested* user interface associated with the Excel spreadsheet that interacts with the VBA program is as shown in Example 5.7.

In what follows, some hints are given to assist in preparing the VBA program.

Recall the algorithm for using the fourth-order RK method with only one ODE:

$$
\begin{aligned}
k_1 &= hf(y_n, t_n) \\
k_2 &= hf\left(y_n + \frac{k_1}{2}, t_n + \frac{h}{2}\right) \\
k_3 &= hf\left(y_n + \frac{k_2}{2}, t_n + \frac{h}{2}\right) \\
k_4 &= hf(y_n + k_3, t_n + h) \\
y_{n+1} &= y_n + \frac{1}{6}[k_1 + 2k_2 + 2k_3 + k_4]
\end{aligned}
\tag{5.48}
$$

To generalize this to N simultaneous ODEs, make each k, y, and f an array with subscripts from 1 to N. Here are suggested ReDim statements:

```
ReDim k1(N) as double
ReDim k2(N) as double
ReDim k3(N) as double
ReDim k4(N) as double
ReDim y(N) as double
ReDim f(N) as double
ReDim z(N) as double
```

Use a subroutine (such as `Sub FCalc` of Example 5.7) that has as input the current time, the `y` array, and the number of equations (N), and it returns N right-hand-side functions (`f`). This subroutine must be called *four times per time step*. Note that y, k_1, k_2, k_3, and k_4 are N-length vectors.

1. Once at the base point $f(y_n, t_n)$
2. A first time at the half-way point $f\left(y_n + \frac{k_1}{2}, t_n + \frac{h}{2}\right)$
3. A second time at the half-way point $f\left(y_n + \frac{k_2}{2}, t_n + \frac{h}{2}\right)$
4. And finally at the end point $f(y_n + k_3, t_n + h)$

It is suggested to define another array (call it z) and use this for the first argument in the subroutine calls at the half-way and end points. Also, define a second time variable (call it `Ttemp`). For example, for the first call at the half-way point, use the sequence

```
z(i) = y(i) + $k_1$(i)/2       i = 1, 2,..., N
Ttemp = Time + h/2
Call FCalc(Ttemp, z(), N, f())
$k_2$(i) = h*f(i)              i = 1, 2,..., N
```

It is strongly urged to sketch out the logic of the code and to “run through” this logic carefully before actually starting to write the code.

Finally, test the resulting VBA program by solving the ODE-IVP of Exercise 5.10. Experiment with the time step to be sure that an acceptable solution is being generated. Beware of the results for a time step of 0.1.

6 Ordinary Differential Equations (Boundary Value Problems)

6.1 INTRODUCTION

So-called boundary value problems (BVPs) occur most often when the system model is a second-order ODE and the known information is available for two different values of the independent variable. For example, consider the following ODE problem:

$$\frac{d^2y}{dx^2} = f(x, y) \quad y(x_1) = y_1 \quad y(x_2) = y_2 \tag{6.1}$$

This problem differs from ODE-IVP since two initial conditions are not given. The two values of the independent variable where information is available are usually at a physical boundary of the system, and the problem is referred to as a *boundary value problem*. Here is a more specific example:

Example 6.1: Heat Conduction in a Rod

A copper rod of length 1 m is placed between two tanks, one containing boiling water and the other containing ice. The rod is exposed to the air. The mathematical model for this system can be expressed as follows:

$$\frac{d^2T}{dx^2} = \frac{4h}{Dk}(T - T_a) \tag{6.2}$$

where

h = Heat transfer coefficient between rod and air = 50 W/(m^2 K)
D = Diameter of the rod = 4 cm = 0.04 m
k = Thermal conductivity of the rod = 390 W/(m K)
T_a = Air temperature = 25°C

The *boundary* conditions are

$$T(0) = 100°\text{C}$$
$$T(1) = 0°\text{C}$$

The methods covered in Chapter 5 cannot be used directly for this problem. The standard procedure would be to convert Equation 6.2 into two first-order ODEs, which would require two initial conditions. In the present example, $0 \le x \le 1$, and only one condition is available at $x = 0$. Therefore, some strategy must be used so that available methods can be applied.

6.2 SHOOTING METHOD

The *shooting method*, as the name implies, makes a guess at a second initial condition and then applies one of the numerical methods covered in Chapter 5 to "shoot" at the far boundary. Based on the error in the result from the known second boundary condition, the guessed initial condition is adjusted until the condition is satisfied. This idea is now applied to the problem of Example 6.2.

Example 6.2: Shooting Method for Heat Conduction in a Rod

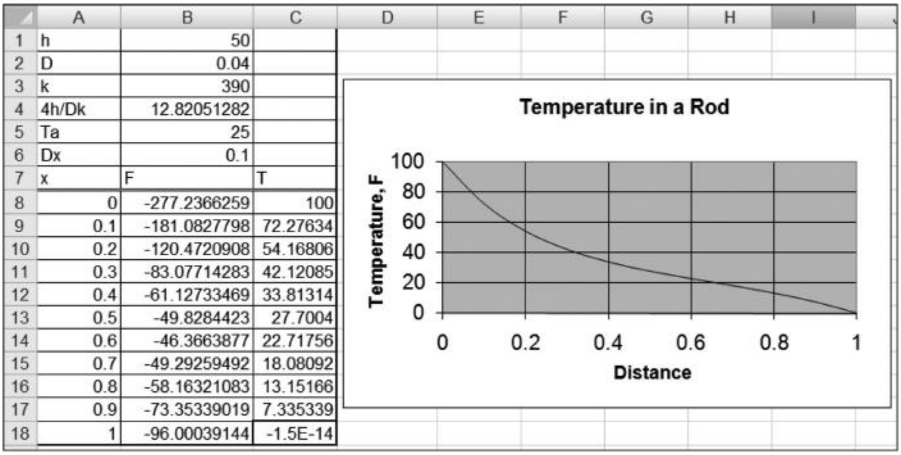
First, the second-order ODE is transformed into two first-order ODEs. Define $F = \dfrac{dT}{dx}$.

Then, the converted problem becomes

$$\begin{aligned}
\frac{dF}{dx} &= \frac{4h}{Dk}(T - T_a) \\
\frac{dT}{dx} &= F \\
T(0) &= 100 \\
F(0) &= ? \\
T(1) &= 0
\end{aligned} \tag{6.3}$$

Since $F(0)$ is not known, a guess is made and the problem is solved as an IVP. The value of T at $x = 1$ is then checked; if it is too low, $F(0)$ is increased (aimed too low); if too high, $F(0)$ is decreased (aimed too high).

At the Excel® level, the Goal Seek tool can be used to converge the second boundary condition. This process is illustrated in the following spreadsheet where the initial guess was $F(0) = -100$ (the derivative must be negative if the temperature is to decrease).



	A	B	C
1	h	50	
2	D	0.04	
3	k	390	
4	4h/Dk	12.82051282	
5	Ta	25	
6	Dx	0.1	
7	x	F	T
8	0	-277.2366259	100
9	0.1	-181.0827798	72.27634
10	0.2	-120.4720908	54.16806
11	0.3	-83.07714283	42.12085
12	0.4	-61.12733469	33.81314
13	0.5	-49.8284423	27.7004
14	0.6	-46.3663877	22.71756
15	0.7	-49.29259492	18.08092
16	0.8	-58.16321083	13.15166
17	0.9	-73.35339019	7.335339
18	1	-96.00039144	-1.5E-14

The lower right-hand cell contains the value of $T(1)$, which should be zero. It was driven to (nearly) zero using Goal Seek by varying $F(0)$. The initial "guess" was –100 for $F(0)$ and Goal Seek found the value –277.237.

6.3 SPLIT BVPs USING FINITE DIFFERENCES

Another approach to solving BVPs is to discretize the ODEs using finite differences. The boundary conditions are applied directly, and the resulting set of equations are solved simultaneously. This approach is well suited to *linear* problems (ones that lead to a set of linear algebraic equations). When the problem leads to nonlinear equations, a tool (such as Solver) must be used. There are also other methods that have been developed for nonlinear problems, but these are beyond the scope of the present discussion (an excellent discussion is given in Riggs 1994). Note that the shooting method can be used without undue difficulty on nonlinear problems.

Example 6.3: Finite Difference Solution for Heat Conduction in a Rod

Consider the same problem as in Example 6.2, which used the shooting method. Dividing the line from 0 to 1 into equal increments of Δx and writing the finite difference form of the equation using a central difference approximation, at the ith node, there results

$$\frac{T_{i-1} - 2T_i + T_{i+1}}{\Delta x^2} = \frac{4h}{Dk}(T_i - T_a) \tag{6.4}$$

with the conditions

$$T_0 = 100$$
$$T_{n+1} = 0$$

There are n interior points. The relationship between n and Δx is

$$\Delta x = \frac{1}{n+1} \tag{6.5}$$

Writing the equation at $i = 1$ gives (let $4h/Dk = a$, and recall that $T_0 = 100$)

$$\begin{aligned} \frac{T_0 - 2T_1 + T_2}{\Delta x^2} &= a(T_1 - T_a) \\ -2T_1 + T_2 &= a(T_1 - T_a)\Delta x^2 - 100 \\ -(2 + a\Delta x^2)T_1 + T_2 &= -a\Delta x^2 T_a - 100 \\ (2 + a\Delta x^2)T_1 - T_2 &= a\Delta x^2 T_a + 100 \end{aligned} \tag{6.6}$$

Let

$$\begin{aligned} b &= (2 + a\Delta x^2) \\ c &= a\Delta x^2 T_a \end{aligned} \tag{6.7}$$

Then the equation for $i = 1$ becomes

$$bT_1 - T_2 = c + 100 \tag{6.8}$$

Writing the equation for $i = 2$ gives

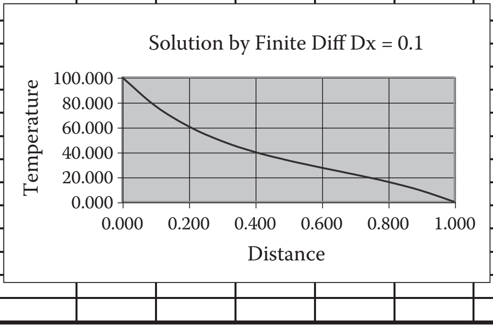
$$\begin{aligned} \frac{T_1 - 2T_2 + T_3}{\Delta x^2} &= a(T_2 - T_a) \\ T_1 - 2T_2 + T_3 &= a(T_2 - T_a)\Delta x^2 \\ T_1 - (2 + a\Delta x^2)T_2 + T_3 &= -a\Delta x^2 T_a \\ -T_1 + (2 + a\Delta x^2)T_2 - T_3 &= a\Delta x^2 T_a \\ -T_1 + bT_2 - T_3 &= c \end{aligned} \tag{6.9}$$

Each succeeding equation is like this one, until $i = n$, in which case (recall that $T_{n+1} = 0$) the following results:

$$\begin{aligned} \frac{T_{n-1} - 2T_n + T_{n+1}}{\Delta x^2} &= a(T_n - T_a) \\ T_{n-1} - 2T_n + 0 &= a(T_n - T_a)\Delta x^2 \\ T_{n-1} - (2 + a\Delta x^2)T_n &= -a\Delta x^2 T_a \\ -T_{n-1} + (2 + a\Delta x^2)T_n &= a\Delta x^2 T_a \\ -T_{n-1} + bT_n &= c \end{aligned} \tag{6.10}$$

Writing all of the equations simultaneously gives a system of *linear* algebraic equations, which can be solved in Excel using the SYSLIN function (or any other method). Note that the matrix of the linear system for problems like this is tridiagonal (has nonzero elements on the diagonal and just above and below the diagonal). These can be solved by SYSLIN, but it is much more efficient to use special numerical methods that take advantage of the "sparseness" of the matrix. The function SYSLIN3 does exactly this and was used to produce the results shown in the following spreadsheet:

h	50.000									
D	0.040									
k	390.000									
4h/Dk	12.821									
Ta	25.000									
n	9.000	Dx =	0.100							
a =	12.821									
b	2.128									
c	3.205									
					A =				c	T
2.128	−1.000								103.205	76.966
−1.000	2.128	−1.000							3.205	60.594
	−1.000	2.128	−1.000						3.205	48.785
		−1.000	2.128	−1.000					3.205	40.025
			−1.000	2.128	−1.000				3.205	33.192
				−1.000	2.128	−1.000			3.205	27.409
					−1.000	2.128	−1.000		3.205	21.935
						−1.000	2.128	−1.000	3.205	16.068
							−1.000	2.128	3.205	9.056
x	T									
0.000	100.000									
0.100	76.966									
0.200	60.594									
0.300	48.785									
0.400	40.025									
0.500	33.192									
0.600	27.409									
0.700	21.935									
0.800	16.068									
0.900	9.056									
1.000	0.000									



6.4 MORE COMPLEX BOUNDARY CONDITIONS WITH ODE-BVPs

Consider again the problem of heat transfer in a rod, where the ODE is

$$\frac{d^2T}{dx^2} = \frac{4h}{Dk}(T - T_a) \tag{6.11}$$

However, now assume that the left boundary is still exposed to boiling water, but that the right side is insulated (no heat flow). From Fourier's law of heat conduction,

$$q = -hA\frac{dT}{dx} \tag{6.12}$$

If $q = 0$, this implies that $\frac{dT}{dx} = 0$. So, now the boundary conditions for the problem are

$$\begin{aligned} &T(0) = 100 \\ &\frac{dT(1)}{dx} = 0 \end{aligned} \tag{6.13}$$

The shooting method can be used to solve this problem with no difficulty by transforming the second-order ODE into two first-order ones with one of the variables being the temperature gradient. That is, define F such that

$$\begin{aligned} &\frac{dT}{dx} = F \\ &\frac{dF}{dt} = \frac{4h}{Dk}(T - T_a) \end{aligned} \tag{6.14}$$

In the shooting method, the procedure is to assume a value for $F(0)$ and "shoot" for a value $F(1) = 0$.

The finite difference method can also be applied. The only thing that changes is the "last" equation (Equation 6.10). Recall the finite difference equation when $i = n$:

$$\frac{T_{n-1} - 2T_n + T_{n+1}}{\Delta x^2} = \alpha(T_n - T_a) \tag{6.15}$$

where

$$a = \frac{4h}{Dk}$$

The right-hand boundary condition can be written using the right-hand second-order correct finite difference formula (the third equation in Table 4.2). This is the second-order correct backward difference formula applied at the right boundary:

$$\begin{aligned} &\frac{3T_{n+1} - 4T_n + T_{n-1}}{2\Delta x} = 0 \\ &T_{n+1} = \frac{4T_n - T_{n-1}}{3} = \frac{4}{3}T_n - \frac{1}{3}T_{n-1} \end{aligned} \tag{6.16}$$

Substituting for T_{n+1} results in

$$\begin{aligned} \frac{T_{n-1} - 2T_n + T_{n+1}}{\Delta x^2} &= a(T_n - T_a) \\ T_{n-1} - 2T_n + \frac{4}{3}T_n - \frac{1}{3}T_{n-1} &= a\Delta x^2(T_n - T_a) \\ \frac{2}{3}T_{n-1} - \left(\frac{2}{3} + a\Delta x^2\right)T_n &= -a\Delta x^2 T_a \end{aligned} \tag{6.17}$$

So, the last of the n equations is *special* and accounts for the insulated boundary condition.

Another more complex boundary condition involves what is often called (incorrectly) the *radiation* boundary condition. For example, at the end of the rod, there might be heat transfer to the surroundings as follows:

$$\begin{aligned} -kA\frac{dT(1)}{dx} &= hA[T(1) - T_a] \text{ or} \\ \frac{dT(1)}{dx} &= -\frac{h}{k}[T(1) - T_a] \end{aligned} \tag{6.18}$$

As with the simpler insulated boundary condition, this more complex one presents no particular difficulty when solving using the shooting method together with Goal Seek. When applying finite differences, the equation representing the boundary ($i = n$) must be altered.

EXERCISES

Exercise 6.1: Consider the problem of heat transfer in a fin with variable thermal conductivity. For a rectangular fin of thickness $2B$ and length L, it can be shown (with suitable assumptions) that the governing ODE is

$$\frac{d}{dx}\left(k\frac{dT}{dx}\right) = \frac{h}{B}(T - T_a) \tag{6.19}$$

where

$T =$ $T(x)$ is the temperature in the fin (°F)

$h =$ Heat transfer coefficient between fin and air [Btu/(h ft^2 °F)]

$B =$ The half thickness of the fin, 0.02 in. (note: inches)

L = Length of the fin, 1.5 in. (note: inches)
k = Thermal conductivity of the fin [Btu/(h ft °F)]
T_a = Air temperature = 90°F
T_w = Temperature of the wall to which the fin is affixed = 450°F
h = 40 Btu/(h ft^2 °F)

The thermal conductivity of the fin varies with distance as follows:

$$k = k_0\,(1 + x) \tag{6.20}$$

$$k_0 = 60 \text{ Btu/(h ft °F)}$$

The boundary conditions are

$$T(0) = T_w$$
$$\frac{dT(L)}{dy} = 0 \text{ (insulated)}$$

Defining a dimensionless distance, $y = x/L$, and differentiating the first term in the ODE (using the equation for k as a function of x), the result is (should be verified)

$$k\frac{d^2T}{dy^2} + k_0 L\frac{dT}{dy} = \frac{hL^2}{B}(T - T_a) \tag{6.21}$$

with boundary conditions

$$T(0) = T_w$$
$$\frac{dT(1)}{dy} = 0 \text{ (insulated)}$$

a. Convert the ODE to two first-order ODEs and solve this problem using the Euler method together with the shooting method.
b. Change the second boundary condition to $T(L) = T_a$ (90°F) and repeat the solution.
c. Change the right-hand boundary condition (B.C.) to the "radiation," which can be expressed as

$$\frac{dT(1)}{dy} = -\frac{hL}{k}[T(1) - T_a]$$

Exercise 6.2: Consider the problem of diffusion and reaction in a cylindrical pore (e.g., in a solid catalyst) where component A reacts at the walls of the cylinder according to

$$A \xrightarrow{r} B \tag{6.22}$$

where

$$r = kC_A^2 \text{ (second-order reaction)} \tag{6.23}$$

In this system, component A diffuses into the pore due to lower concentration of A inside the pore than at the pore mouth. Since B is produced by the reaction, the concentration of B inside the pore is larger than at the inlet, causing diffusion of B out of the pore. At the inlet of the pore ($x = 0$), the concentration is C_{A0}. The end of the pore ($x = L$) is assumed to be sealed, so there is no flux of A at $x = L$. The mathematical model for this system can be expressed as follows:

$$\begin{aligned} D_A \frac{d^2C_A}{dx^2} &= kC_A^2 \\ C_A(0) &= C_{A0} \\ \frac{dC_A(L)}{dx} &= 0 \end{aligned} \tag{6.24}$$

The second boundary condition is the "no flux at $x = L$" condition. This is a split BVP that can be solved using the shooting method. Here are some data for the problem:

k = 0.01 L/(gmol s) (rate constant)
C_{A0} = 1.0 gmol/L (inlet concentration)
D_A = 1×10^{-3} cm^2/s (diffusivity)
L = 1 cm (length of pore)

Prepare an Excel spreadsheet to solve this problem using the Euler method. Use Goal Seek to implement the shooting method. Be sure to get a reasonable value of the unknown boundary condition (by experimentation) before invoking Goal Seek.

Exercise 6.3: Consider the same problem as in Exercise 6.2, but with a first-order reaction:

$$A \xrightarrow{r} B \tag{6.25}$$

where

$$r = kC_A \text{ (first-order reaction)} \tag{6.26}$$

The model equations then become

$$\begin{aligned} D_A \frac{d^2C_A}{dx^2} &= kC_A \\ C_A(0) &= C_{A0} \\ \frac{dC_A(L)}{dx} &= 0 \end{aligned} \tag{6.27}$$

Apply the finite difference method to solve this problem (first introduce a dimensionless distance $y = x/L$). Use the same data as for the second-order reaction case. For the right-hand boundary condition, do not forget to use a second-order correct finite difference form.

Exercise 6.4: Consider heat transfer in a counter-current heat exchanger as depicted in Figure 6.1.

When the flow direction of the two streams is opposite, counter-current flow exists and the system equations take the form

$$\begin{aligned} \frac{dT'}{dx} &= -\frac{U_i \pi D_i}{m'C'_p}(T' - T) \\ \frac{dT}{dx} &= \frac{U_i \pi D_i}{mC_p}(T' - T) \end{aligned} \quad \text{with } T'(0) \text{ and } T(L) \text{ known} \tag{6.28}$$

where

T = Temperature on the shell (outer) side
T' = Temperature on the tube (inner) side
U_i = Overall heat transfer coefficient based on D_i
D_i = Inside diameter of the inner pipe (tube side)
m = Mass flow rate on the shell side
m' = Mass flow rate on the tube side

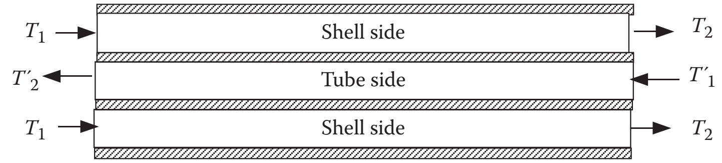


FIGURE 6.1 Double pipe heat exchanger schematic.

C_p = Fluid heat capacity on the shell side
C'_p = Fluid heat capacity on the tube side
L = Length of the heat exchanger

Pertinent data are as follows:

$$\frac{U_i \pi D_i}{m' C'_p} = 1.2$$
$$\frac{U_i \pi D_i}{m C_p} = 0.5 \tag{6.29}$$

Assume that the length of the exchanger = 1 and then consider the boundary conditions

$$T'(0) = 180; T(1) = 70$$

In order to integrate the two ODEs from $x = 0$ to $x = 1$, $T(0)$ must be guessed and the integration performed to see if the target of 70 is hit. In effect, the single nonlinear equation $T(1) - 70 = 0$ must be solved.

Use the Euler method along with Goal Seek to solve this problem (at the Excel level). Begin using $\Delta x = 0.1$; find $T(0)$ that satisfies the right-hand boundary condition. Then reduce Δx until comparable temperature profiles result. Finally, plot the temperature profiles with appropriate annotations on the graph. Assume that temperatures are in degrees Celsius.

Exercise 6.5: The transient BVP for a rectangular fin can be stated as follows:

$$\frac{\partial^2 T}{\partial x^2} - \beta^2 (T - T_a) = \frac{1}{\alpha} \frac{\partial T}{\partial t}$$

$$0 \le x \le 1$$
$$T(0,t) = T_1$$
$$T(1,t) = T_2$$
$$T(x,0) = 0$$

α and β are constants.

Define a dimensionless temperature and time as follows:

$$\theta = \frac{T - T_a}{T_1 - T_a} \qquad \tau = \alpha t$$

The BVP then becomes

$$\frac{\partial^2\theta}{\partial x^2} - \beta^2\theta = \frac{\partial\theta}{\partial\tau}$$

The new boundary conditions are

$$\theta(0,t) = 1$$

$$\theta(1,t) = \frac{T_2 - T_a}{T_1 - T_a}$$

$$\theta(x,0) = \frac{-T_a}{T_1 - T_a}$$

The steady-state model results when the time derivative is zero:

$$\frac{\partial^2\theta}{\partial x^2} - \beta^2\theta = 0 \quad \theta(0) = 1 \quad \theta(1) = \frac{T_2 - T_a}{T_1 - T_a}$$

The steady-state solution can be shown to be

$$\theta_{ss} = \frac{\sinh[\beta(1-x)]}{\sinh\beta} + \left(\frac{T_2 - T_a}{T_1 - T_a}\right)\frac{\sinh\beta x}{\sinh\beta}$$

For simplicity, take $T_a = 0$, $T_1 = 1$, $T_2 = 0$, and $\beta = 4$.

a. Solve the steady-state dimensionless problem using the shooting method and the Euler method. Check the results with the analytical solution.
b. Solve the steady-state dimensionless problem using finite differences. Again, compare the results with the analytical solution.

Exercise 6.6: (*Note: This problem involves significant VBA programming and complex logic.*) Solve the ODE-BVP of Equation 6.11 with boundary conditions given by Equation 6.13 using the VBA program of Example 5.7 (Euler method). The requisite ODE and boundary conditions are repeated below:

$$\frac{d^2T}{dx^2} = \alpha(T - T_a) \tag{6.30}$$

$$T(0) = 100$$

$$\frac{dT(1)}{dx} = 0 \tag{6.31}$$

where

$$\alpha = 12.8$$
$$T_a = 25$$

While Goal Seek can be invoked from VBA, it is not possible to use it in the context of the Euler program. Therefore, it is necessary to include VBA code to implement one of the methods described in Chapter 1 to solve a single nonlinear equation. In particular, use the *secant method* to do this. The logic changes required in the Example 5.4 VBA program must be carefully thought out, but in general, the steps required are as follows:

a. Execute the current VBA program logic for one initial guess for $T'(0)$.
b. Run the current VBA program logic for a second initial guess for $T'(0)$.
c. Use the results of steps a and b to produce a new guess for $T'(0)$ and use the logic of the secant method to proceed (replace one of the initial guesses and repeat this step until convergence is achieved). The process is converged when $T'(1)$ is close to zero.

Exercise 6.7: Solve the problem of Exercise 6.2 using the instructions given in Exercise 6.6. That is, use the Euler VBA program of Example 5.4 together with the secant method. As with Exercise 6.6, this problem requires significant logic planning and VBA programming.

REFERENCE

Riggs, J.B., *An Introduction to Numerical Methods for Chemical Engineers*, 2nd ed., Texas Tech University Press, Lubbock, TX. pp. 241–282 (1994).

7 Regression Analysis and Parameter Estimation

7.1 INTRODUCTION AND THE GENERAL METHOD OF LEAST SQUARES

The problem of regression analysis is to find coefficients (parameters) of a function that are believed to properly represent a set of experimental data *and* to perform statistical analyses to confirm (or deny) that the function gives a good fit to the data. Without the additional statistical analysis, just finding the parameters of some candidate function is called *curve fitting*. Curve fitting, while useful in certain circumstances, is not as powerful as regression analysis. Consider data as represented in Figure 7.1.

It is assumed that x, the independent variable, is error free (this might be time or temperature, for example), and y, the dependent variable, contains experimental error. In chemical and biomolecular engineering applications, theoretical knowledge often exists of the function that should "fit" the data. If the deviations (errors) between the data and the fitting function are statistically distributed with a normal distribution with zero mean and *constant variance*, then it can be shown that a proper way to find the unknown coefficients of the function is to minimize the sum of squares of the errors. It is common nomenclature to call the errors "residuals," which are defined as follows:

$$r_i = y_i^{calc} - y_i^{data};\ i = 1, 2, \ldots, n_d \tag{7.1}$$

where n_d is the number of data points, y_i^{calc} is the value of y calculated from the fitting function, and y_i^{data} is the associated data value. The function to be minimized is then

$$q = \frac{1}{2}\sum_{i=1}^{n_d} r_i^2 \tag{7.2}$$

This is, therefore, called the *method of least squares*. If the fitting function can be represented as

$$y_i^{calc} = f_i(c_1, c_2, \ldots, c_n;\ x_1, x_2, \ldots, x_m)\ i = 1, 2, \ldots, n_d \tag{7.3}$$

where the x_i are the "independent variables" (e.g., time, temperature, distance, etc. in which there are no significant errors) and y is the "dependent variable" that contains

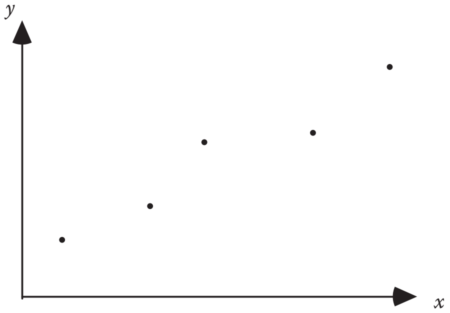


FIGURE 7.1 Data with experimental error in the dependent variable.

random errors (perhaps due to measurement error or some other error source). The c_i are unknown model parameters to be determined. Note that the unknowns to be determined are not represented by x. It takes some time to change the usual mind-set that x is the unknown.

To minimize Equation 7.2, differentiate with respect to the unknown parameters and set the result to zero (seeking a stationary point):

$$\frac{\partial q}{\partial c_k} = \sum_{i=1}^{n_d} r_i \frac{\partial r_i}{\partial c_j} = 0 \quad j = 1, 2, \ldots, n \tag{7.4}$$

If r_i is expanded into a Taylor's series about a point c^0 (this is an "initial guess" of the c's), then

$$r_i = r_i(c^0) + \sum_{k=1}^{n} \frac{\partial r_i(c^0)}{\partial c_k} \cdot \Delta c_k \tag{7.5}$$

where

$$\Delta c_k = c_k - c_k^0 \tag{7.6}$$

Substituting Equation 7.5 into Equation 7.4 gives

$$\sum_{i=1}^{n_d} \left(r_i + \sum_{k=1}^{n} \frac{\partial r_i}{\partial c_k} \right) \frac{\partial r_i}{\partial c_j} \cdot \Delta c_k = 0 \quad j = 1, 2, \ldots, n \tag{7.7}$$

In Equation 7.7, it is understood that the residuals and their derivatives are evaluated at c^0 (the initial guess).

By expanding Equation 7.7 and defining

$$Z_k = \frac{\partial f(c^0)}{\partial c_k} \tag{7.8}$$

the following set of equations in matrix–vector form result where the *summations are over all data points*:

$$\begin{bmatrix} \sum Z_1Z_1 & \sum Z_1Z_2 & \cdots & \sum Z_1Z_n \\ \sum Z_2Z_1 & \sum Z_2Z_2 & \cdots & \sum Z_2Z_n \\ \vdots & \vdots & \ddots & \vdots \\ \sum Z_nZ_1 & \sum Z_nZ_2 & \cdots & \sum Z_nZ_n \end{bmatrix} \begin{bmatrix} \Delta c_1 \\ \Delta c_2 \\ \vdots \\ \Delta c_n \end{bmatrix} = - \begin{bmatrix} \sum r_iZ_1 \\ \sum r_iZ_2 \\ \vdots \\ \sum r_iZ_n \end{bmatrix} \tag{7.9}$$

A shorthand representation of Equation 7.9 is

$$G\Delta c = -b \tag{7.10}$$

where G is the left-hand side matrix, Δc is the vector of unknowns, and b is the right-hand side vector. Since Equations 7.9 and 7.10 are applicable to any fitting function, these are often called the normal equations of the general method of least squares. The symmetric matrix G is referred to here as the *Gauss–Newton matrix*, but it goes by several other names in the literature.

7.2 LINEAR REGRESSION ANALYSIS

If the Zs are constant (not dependent on the cs), Equation 7.10 is a linear system and the unknown coefficients can be easily determined. Note that a "linear regression" does not require the fitting function to be linear (a straight line); the only requirement is that the Zs are constants (the function is linear in its parameters).

Another way of getting the coefficients for *a limited number of fitting functions* is to use *X*–*Y* scatter graphs in Excel® and add a trendline.

7.2.1 Straight Line Regression

Consider "straight line" regression where theory dictates that the data should lie on a straight line (or in the absence of a theoretical justification, a plot of the data suggests that a straight line should suffice). The fitting function is then

$$y_i^{calc} = c_1 + c_2 x_i \tag{7.11}$$

To calculate the regression coefficients using Equation 7.9, proceed as follows:

$$y_i^{calc} = c_1 + c_2 x_i$$
$$Z_1 = \frac{\partial y^{calc}}{\partial c_1} = 1 \quad Z_2 = \frac{\partial y^{calc}}{\partial c_2} = x_i \tag{7.12}$$

$$G = \begin{bmatrix} \sum_{i=1}^{10} (1)(1) & \sum_{i=1}^{10} (1)(x_i) \\ \sum_{i=1}^{10} (x_i)(1) & \sum_{i=1}^{10} (x_i)(x_i) \end{bmatrix} = \begin{bmatrix} 10 & \sum_{i=1}^{10} x_i \\ \sum_{i=1}^{10} x_i & \sum_{i=1}^{10} (x_i)^2 \end{bmatrix}$$

If the initial guess is taken to be zero (this is always suggested for linear problems), then

$$\Delta c_k = c_k \tag{7.13}$$

and

$$r_i = -y_i^{data} (\text{since } y_i^{calc} = 0 \text{ when } c_1 = c_2 = 0) \tag{7.14}$$

The equations to solve for the unknown coefficients become

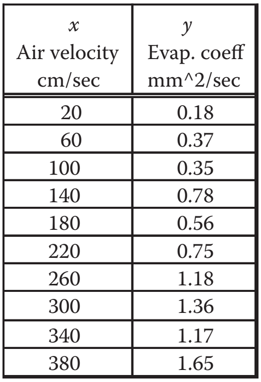
$$\begin{bmatrix} 10 & \sum_{i=1}^{10} x_i \\ \sum_{i=1}^{10} x_i & \sum_{i=1}^{10} (x_i)^2 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = + \begin{bmatrix} \sum_{i=1}^{10} y_i^{data} \\ \sum_{i=1}^{10} y_i^{data} x_i \end{bmatrix} \tag{7.15}$$

The following Excel spreadsheet shows the calculation of all quantities required in these equations:

Example 7.1: Evaporation Coefficient Correlation

Figure 7.2 displays data for an evaporation coefficient for different air velocities over a pool of liquid.

Shown in Figure 7.3 is a plot of the original data produced with Excel. Note the regression equation that is displayed on the graph, which was calculated by Excel by adding a "trendline." To add the trendline, after having plotted the data as an *X–Y* scatter plot, right click the mouse on any data point. This brings up a menu of choices, one of which is Add trendline. A number of choices appear for the fitting function. After selecting the fitting function, click on the Options tab; some check boxes appear at the bottom. One of these says "Display Equation on Chart," and the other says "Display R-squared Value on Chart." Check both of these boxes, and the result should look like that shown in Figure 7.2. The quantity R^2 is covered in more detail later.



x Air velocity cm/sec	*y* Evap. coeff mm^2/sec
20	0.18
60	0.37
100	0.35
140	0.78
180	0.56
220	0.75
260	1.18
300	1.36
340	1.17
380	1.65

FIGURE 7.2 Evaporation coefficient data.

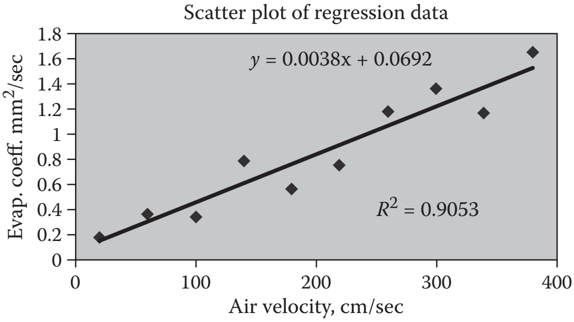


FIGURE 7.3 Scatter plot of evaporation coefficient data.

	x Air velocity cm/sec	*y* Evap. coeff mm^2/sec	*x*^2	*x y*	*y*^2
	20	0.18	400	3.6	0.0324
	60	0.37	3600	22.2	0.1369
	100	0.35	10000	35	0.1225
	140	0.78	19600	109.2	0.6084
	180	0.56	32400	100.8	0.3136
	220	0.75	48400	165	0.5625
	260	1.18	67600	306.8	1.3924
	300	1.36	90000	408	1.8496
	340	1.17	115600	397.8	1.3689
	380	1.65	144400	627	2.7225
sums	2000	8.35	532000	2175.4	9.1097
Equations:	10	2000	8.35		
	2000	532000	2175.4		
G Inv.	0.40303	–0.001515	c1=	0.069242	
	–0.001515	7.576E-06	c2=	0.003829	

Note the following in the spreadsheet:

$$G = \begin{bmatrix} 10 & 2000 \\ 2000 & 532000 \end{bmatrix} \qquad -b = \begin{bmatrix} 8.35 \\ 21.75 \end{bmatrix} \qquad c = \begin{bmatrix} 0.069242 \\ 0.003829 \end{bmatrix} \tag{7.16}$$

The *c*s are identical to those found by Excel when a trendline was added to the plot of the data. The *c*s were determined using the inverse of the matrix *G*. The *G* matrix and its inverse have other powerful uses as well, as will be covered in later discussions.

One might (justifiably) ask why learn all of these details when Excel produced the desired results so easily. The reasons are twofold. First, it is always useful to know the details behind any automatic computations. Second, the computational details are required when the problem is nonlinear (they are not done automatically by Excel).

7.2.2 Curvilinear Regression

The next example considers data that obviously cannot be represented by a straight line. The fitting function is often chosen as a polynomial or other function that can "bend" to better represent the data. It is important that although the fitting function is nonlinear in the dependent variable (*x*), it remains linear in the unknown coefficients (*c*).

Example 7.2: Curvilinear Regression

Consider the data given in Figure 7.4. A scatter plot of the data is shown in Figure 7.5. From the plot, it does not appear that a straight line can represent the data adequately. In the absence of a physical system model (conservation of mass, energy, or momentum principles), the usual procedure is to simply seek a function that will represent the data in an appropriate manner (this can be highly subjective—even very approximate models are better than a pure guess about the function).

The function next up the ladder of complexity from a straight line is a quadratic, which can be written as follows:

$$y^{calc} = c_1 + c_2 x + c_3 x^2 \tag{7.17}$$

Varnish additive, g	Drying time, hours
0	12.0
1	10.5
2	10.0
3	8.0
4	7.0
5	8.0
6	7.5
7	8.5
8	9.0

FIGURE 7.4 Drying time data.

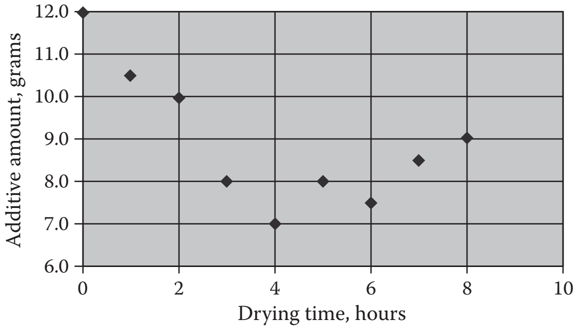


FIGURE 7.5 Effect of additive on drying time.

To set up the least-squares (normal) equations, proceed by finding the *Z*s (the derivatives of y^{calc} with respect to the unknown coefficients):

$$\begin{aligned} Z_1 &= 1 \\ Z_2 &= x \\ Z_3 &= x^2 \end{aligned} \tag{7.18}$$

Since the *Z*s are not dependent on the *c*s, this is still a linear regression problem.

Taking $c^0 = 0$ (i.e., all initial guesses of the *c*s = 0), the normal equations result from Equation 7.9 as follows:

$$r_i = -y_i^{data} (\text{since } y^{calc}(x^0) = 0) \tag{7.19}$$

$$\begin{bmatrix} \sum 1 & \sum x & \sum x^2 \\ \sum x & \sum x^2 & \sum x^3 \\ \sum x^2 & \sum x^3 & \sum x^4 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} \sum y^{data} \\ \sum x \cdot y^{data} \\ \sum x^2 \cdot y^{data} \end{bmatrix} \tag{7.20}$$

Shown below is an Excel spreadsheet that performs the indicated calculations:

x	y	x^2	x^3	x^4	yx	yx^2
0	12.0	0	0	0	0	0
1	10.5	1	1	1	10.5	10.5
2	10.0	4	8	16	20	40
3	8.0	9	27	81	24	72
4	7.0	16	64	256	28	112
5	8.0	25	125	625	40	200
6	7.5	36	216	1296	45	270
7	8.5	49	343	2401	59.5	416.5
8	9.0	64	512	4096	72	576
36	80.5	204	1296	8772	299	1697

So, the normal equations become

$$\begin{bmatrix} 9 & 36 & 204 \\ 36 & 204 & 1296 \\ 204 & 1296 & 8772 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} 80.5 \\ 299 \\ 1697 \end{bmatrix} \tag{7.21}$$

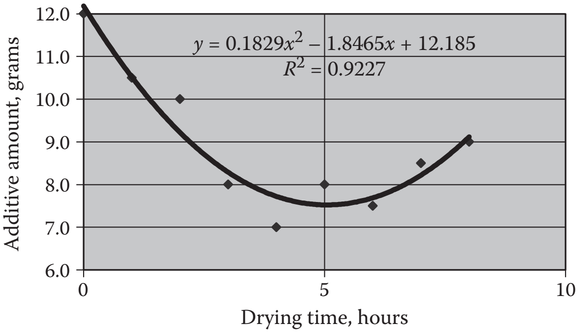


FIGURE 7.6 Effect of additive on drying time with trendline.

These linear algebraic equations can be solved using any of the methods previously discussed. The solution is shown below in terms of the approximating function:

$$y = 12.2 - 1.85x + 0.183x^2 \tag{7.22}$$

This equation together with the correlation coefficient can be found using Excel's graphing capabilities. Simply plot the data (points only) and add a trendline (Figure 7.6). Use the appropriate Option to show the equation and R^2. The coefficients for the quadratic trendline are identical to those calculated previously.

This example will be amplified upon discussion after having introduced the "statistical part" of regression analysis.

7.3 HOW GOOD IS THE FIT FROM A STATISTICAL PERSPECTIVE?

There are several tools to help visualize if the chosen function is a good one. Five of these will be covered:

- Residual plots
- Correlation coefficient
- Parameter standard deviations
- Parameter confidence intervals
- Parameter *t*-ratios

7.3.1 Residual Plots

When the residuals ($y^{calc} - y^{data}$) versus x are plotted, these values should distribute themselves somewhat evenly about zero, and their magnitude should be approximately constant (these were the basic assumptions for the least-squares method).

7.3.2 Correlation Coefficient

A quantitative measure of the "goodness of fit" is provided by the correlation coefficient:

$$R^2 = 1 - \frac{\sum_{i=1}^{n_d} r_i^2}{\sum_{i=1}^{n_d} \left(y_i - \bar{y}\right)^2} \tag{7.23}$$

where

$$\bar{y} = \frac{1}{n_d}\sum_{i=1}^{n_d} y_i$$

An R^2 value near 1 indicates that all residuals are small and that the fit is "good." An elaborate treatment regarding R^2 and the analysis of variance for regression analysis is beyond the scope of the present discussion.

7.3.3 Parameter Standard Deviations

The parameter (regression coefficient) variances (and thus standard deviations) can be estimated from the matrix G of Equation 7.10. The following is stated without proof:

$$\mathrm{var}(c) = S = s_e^2 G^{-1} \tag{7.24}$$

where

$$s_e^2 = \frac{\sum_{i=1}^{n_d} r_i^2}{\nu} \tag{7.25}$$

and ν is the number of degrees of freedom (see Equation 7.26).

Variance [var(c)] represent the parameter variances. S is sometimes called the variance–covariance matrix. Variance is a measure of the spread of expected values of random variables belonging to a specific probability distribution. As has been mentioned previously, the validity of the least-squares method for determining regression parameters is based on errors in the data having a normal (Gaussian) distribution (the familiar bell-shaped curve) with zero mean and constant variance. The values of the parameters determined from data with such normal errors are, in a

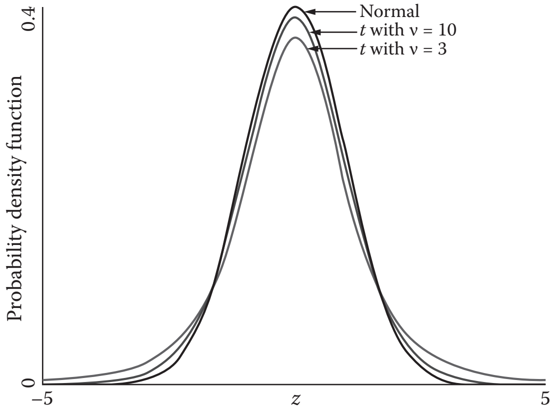


FIGURE 7.7 Plot of the *t*-distribution.

sense, average values—they too are subject to errors. If the variance of the errors in the data were known, then the parameters themselves would be normally distributed. However, it is rarely the case that this variance is known with certainty. Therefore, the variance must be estimated. The matrix S of Equation 7.24 provides an estimate of the variances of the individual parameters (the diagonal elements of S) as well as the off-diagonal covariances between pairs of parameters (no further treatment of the covariances is given here).

When the variances must be estimated, the probability distribution of the parameters is the *t*-distribution. The *t*-distribution is similar to the normal distribution, but it has an additional argument called the degrees of freedom, which is the difference between the number of data points and the number of parameters:

$$\text{Degrees of freedom} = \nu = n_d - n \tag{7.26}$$

The *t*-distribution probability density function looks very much like that of the normal distribution, but it is "shorter" and has a longer tail. Figure 7.7 displays the *t*-distribution with 3 and 10 degrees of freedom as well as the normal distribution. The *t*-distribution with an infinite number of degrees of freedom is coincident with the normal distribution.

7.3.4 Parameter Confidence Intervals

As previously stated, the regression parameters have the *t*-distribution. Again, without proof, the following inequality can be written:

$$c_i - t_{1-\alpha/2} s_e \sqrt{S_{ii}} < \bar{c}_i < c_i + t_{1-\alpha/2} s_e \sqrt{S_{ii}} \tag{7.27}$$

where $\bar{c}_i$ is the true value of c_i.

7.3.5 Using *t*-Ratios (*t*-Statistics) for Individual Parameter Significance

When choosing an arbitrary function to fit a set of data (such as the quadratic in Example 7.2), it might be asked if all three terms in the equation are needed or if it can be simplified by dropping perhaps the linear term (the one involving $c_2 x$). To answer this question, statistics gives the answer. It can be shown that the ratio of the optimal parameter values divided by their standard deviations (as determined by Equation 7.7) has a *t*-distribution with $n_d - n - 1$ degrees of freedom. That is,

$$t_i = \frac{c_i}{s_i} \tag{7.28}$$

Looking at a table of the *t*-distribution (Google it), it will be noticed that for $\alpha = 0.025$ (two-sided confidence of 95%), for more than 3 degrees of freedom, the tabulated values are all near 2. So, for convenience, it can be stated as an informal test of the null hypothesis that a parameter's contribution is insignificant (i.e., its value is zero) if the calculated value $|t_i| < 2$ and rejected if the calculated value $|t_i| \geq 2$. This can be done more formally by using the precise value of the *t*-distribution with the proper number of degrees of freedom. In Excel, this can be found using the function `TINV(α, ν)`. Note that α and not α/2 is used with the `TINV` function. This is because the function returns what is called the two-tailed *t*-value; α is split between the negative tail and the positive tail of the *t*-distribution. Only a much more detailed study of the statistics associated with these arguments would explain these concepts fully, but this is beyond the present discussion. Suffice it to say that if a parameter *t*-ratio (Equation 7.28) is less than about 2, it can be concluded that the parameter is no different than zero and can be removed from the correlating equation.

Example 7.3: Applying Statistics to the Problem of Example 7.2

Consider again the quadratic function fit to the drying time data of Example 7.2. In what follows, each of the statistical tools for determining the quality of the "fit" is demonstrated.

RESIDUAL PLOT

The residual plot for Example 7.2 (quadratic curve fit) appears in Figure 7.8.

With only 9 data points, the data distribute nicely about zero, and the magnitude does not (visually) appear to be a function of *x*. It can be concluded that the fit is adequate. Note that a single point with a very large residual is a candidate "outlier" and might be omitted if this can be justified (poor experimental procedure, other extenuating circumstances, etc.). However, the arbitrary exclusion of outliers must be avoided.

CORRELATION COEFFICIENT

From Equation 7.23, the R^2 value for Example 7.2 can be calculated as 0.9227. This value is identical to that shown in the trendline graph in Figure 7.4.

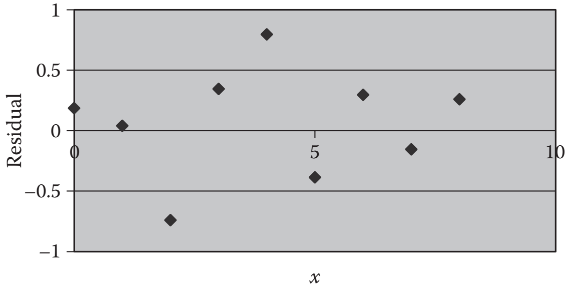


FIGURE 7.8 Residual plot for Example 7.2.

PARAMETER STANDARD DEVIATIONS

The *G* matrix for Example 7.2 was

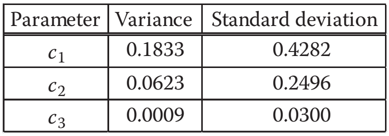
9	36	204
36	204	1296
204	1296	8772

The associated inverse matrix is

0.660606	–0.30909	0.030303
–0.30909	0.224459	–0.02597
0.030303	–0.02597	0.003247

x	residual	resid^2
0	0.185	0.034225
1	0.0389	0.001513
2	–0.7414	0.549674
3	0.3441	0.118405
4	0.7954	0.632661
5	–0.3875	0.150156
6	0.2954	0.087261
7	–0.1559	0.024305
8	0.2586	0.066874
	sum	1.665074
	se^2	0.277512

0.183326	–0.085777	0.008409
–0.085777	0.06229	–0.007208
0.008409	–0.007208	0.000901



Parameter	Variance	Standard deviation
c_1	0.1833	0.4282
c_2	0.0623	0.2496
c_3	0.0009	0.0300

FIGURE 7.9 Parameter standard deviations.

The final results are shown in Figure 7.9.

To obtain the parameter variances, the diagonal elements must be multiplied by s_e^2 (see Equation 7.25). s_e^2 is the sum of squares of residuals divided by the degrees of freedom (number of data points – number of parameters; 6 in the present case). Shown below is the calculation of s_e^2 followed by the matrix whose diagonal elements are the parameter variances:

PARAMETER CONFIDENCE INTERVALS (95%)

From Equation 7.27, the 95% parameter confidence intervals are shown in the following inequalities:

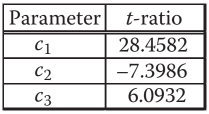
$$\begin{aligned} &c_1\text{: } 11.633 < 12.185 < 12.737 \\ &c_2\text{: } -2.168 < -1.847 < -1.525 \\ &c_3\text{: } 0.1442 < 0.1829 < 0.2216 \end{aligned} \tag{7.29}$$

An examination of these confidence intervals reveals that the uncertainty in all three parameters is not unreasonably large.

PARAMETER *t*-RATIOS

From Equation 7.28, the *t*-ratios for the three parameters are shown in Figure 7.10.

The *t*-ratios for c_1, c_2, and c_3 have absolute values considerably greater than 2. Therefore, it can be concluded that all parameters are significant and must be retained in the correlating equation.



Parameter	*t*-ratio
c_1	28.4582
c_2	–7.3986
c_3	6.0932

FIGURE 7.10 Parameter *t*-ratios.

Example 7.4: Another Curvilinear Regression Problem

Consider fitting a polynomial to the data in Figure 7.11.

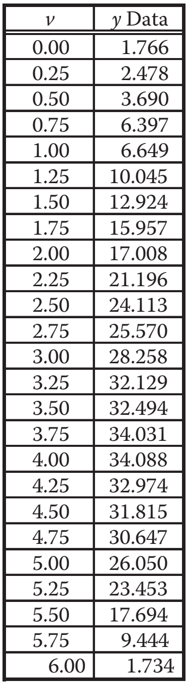
A typical first step when choosing a fitting function (in the absence of one based on theory) is to prepare a graph of the data. When using Excel, a variety of trendline types can also be helpful in arriving at a suitable functional form. Shown in Figure 7.12 is a plot of the data together with both quadratic and cubic polynomial fitting functions.

The graph in Figure 7.12 suggests clearly that a cubic fitting function is superior to a quadratic one. A cubic fitting function is as shown by the following equation:

$$y_{calc} = c_1 + c_2v + c_3v^2 + c_4v^3 \tag{7.30}$$

The derivative of Equation 7.29 with respect to each of the four parameters gives

$$Z_1 = 1,\ Z_2 = v,\ Z_3 = v^2,\ Z_4 = v^3 \tag{7.31}$$



v	*y* Data
0.00	1.766
0.25	2.478
0.50	3.690
0.75	6.397
1.00	6.649
1.25	10.045
1.50	12.924
1.75	15.957
2.00	17.008
2.25	21.196
2.50	24.113
2.75	25.570
3.00	28.258
3.25	32.129
3.50	32.494
3.75	34.031
4.00	34.088
4.25	32.974
4.50	31.815
4.75	30.647
5.00	26.050
5.25	23.453
5.50	17.694
5.75	9.444
6.00	1.734

FIGURE 7.11 Data for curvilinear regression.

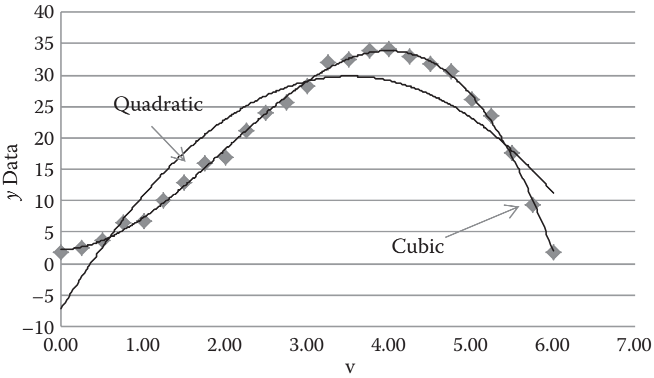


FIGURE 7.12 Curvilinear data showing quadratic and cubic fitting functions.

The normal equations [see Equation 7.10] are as shown in Equation 7.32:

$$\begin{bmatrix} 25.00 & 75.00 & 306.25 & 1406.25 \\ 75.00 & 306.25 & 1406.25 & 6886.80 \\ 306.21 & 406.25 & 6886.80 & 35126.95 \\ 1406.25 & 6886.80 & 35126.95 & 184262.87 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} = \begin{bmatrix} 482.60 \\ 1697.26 \\ 6779.00 \\ 29158.68 \end{bmatrix} \tag{7.32}$$

Solving these equations gives

c_1	2.2241
c_2	0.1829
c_3	5.8748
c_4	−0.9855

The gross sum of squares of residuals = 0.5191.

The inverse of *G* is as follows:

0.4810	−0.5869	0.1915	−0.0182
−0.5869	1.0442	−0.3946	0.0407
0.1915	−0.3946	0.1604	−0.0173
−0.0182	0.0407	−0.0173	0.0019

The parameter standard deviations and *t*-ratios are shown in Figure 7.13.

	Base value	Std. dev	*t*-ratios
c_1	2.2241	0.4997	4.451
c_2	0.1829	0.7363	0.248
c_3	5.8747	0.2886	20.357
c_4	−0.9855	0.0316	−31.2

FIGURE 7.13 Parameter values, standard deviations, and *t*-ratios.

	Base value	Std. dev.	t-ratios
c_1	2.327	0.274	8.489
c_2	5.944	0.075	79.209
c_3	–0.993	0.013	–76.610

FIGURE 7.14 Revised parameter values, standard deviations, and t-ratios.

From Figure 7.13, it can be deduced that c_2 is poorly determined. Upon removing the term involving c_2, the new model equation is given by

$$y = c_1 + c_2x^2 + c_3x^3 \tag{7.33}$$

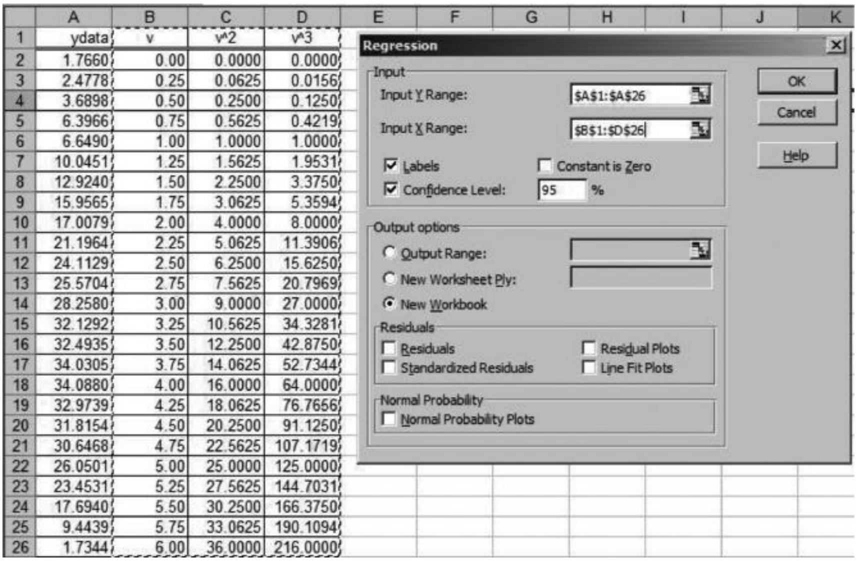
When the computations are repeated using the revised fitting function, the optimal coefficients, standard deviations, and t-ratios are as shown in Figure 7.14.

Now, all t-ratios are well above 2, and Equation 7.33 gives an adequate representation of the data.

As a note of warning, when turned out that two of the original parameters had t-ratios less than 2, it was appropriate to remove only the one with the smallest (in absolute value) t-ratio. Once the *worst actor* removed, the revised t-ratio of the *other offender* was greater than 2.

7.4 REGRESSION USING EXCEL'S® REGRESSION ADD-IN

Now that all of the computations for regression analysis the "hard way" (e.g., do it yourself) have been covered, a *little secret* can be revealed: Excel will do almost everything that has been discussed as long as the problem is one of linear regression. To invoke this package, go to Data/Data Analysis/Regression. The following spreadsheet displays the original data and columns for other terms to be included in the original fitting function (Equation 7.30). Also shown is the window that is presented by the Regression Add-In:



	A	B	C	D
1	ydata	v	v^2	v^3
2	1.7660	0.00	0.0000	0.0000
3	2.4778	0.25	0.0625	0.0156
4	3.6898	0.50	0.2500	0.1250
5	6.3966	0.75	0.5625	0.4219
6	6.6490	1.00	1.0000	1.0000
7	10.0451	1.25	1.5625	1.9531
8	12.9240	1.50	2.2500	3.3750
9	15.9565	1.75	3.0625	5.3594
10	17.0079	2.00	4.0000	8.0000
11	21.1964	2.25	5.0625	11.3906
12	24.1129	2.50	6.2500	15.6250
13	25.5704	2.75	7.5625	20.7969
14	28.2580	3.00	9.0000	27.0000
15	32.1292	3.25	10.5625	34.3281
16	32.4935	3.50	12.2500	42.8750
17	34.0305	3.75	14.0625	52.7344
18	34.0880	4.00	16.0000	64.0000
19	32.9739	4.25	18.0625	76.7656
20	31.8154	4.50	20.2500	91.1250
21	30.6468	4.75	22.5625	107.1719
22	26.0501	5.00	25.0000	125.0000
23	23.4531	5.25	27.5625	144.7031
24	17.6940	5.50	30.2500	166.3750
25	9.4439	5.75	33.0625	190.1094
26	1.7344	6.00	36.0000	216.0000

Regression

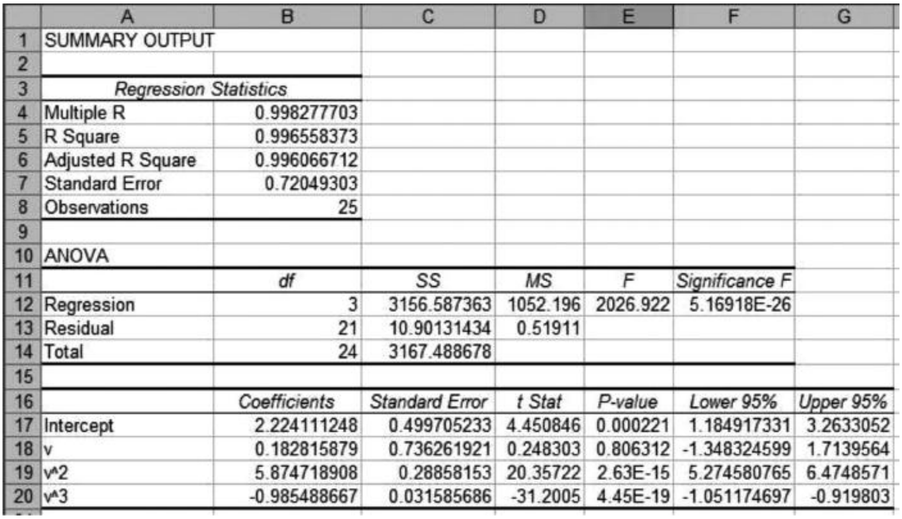
Input
Input Y Range: $A$1:$A$26
Input X Range: $B$1:$D$26
☑ Labels
□ Constant is Zero
☑ Confidence Level: 95 %

Output options
○ Output Range:
○ New Worksheet Ply:
● New Workbook

Residuals
□ Residuals
□ Residual Plots
□ Standardized Residuals
□ Line Fit Plots

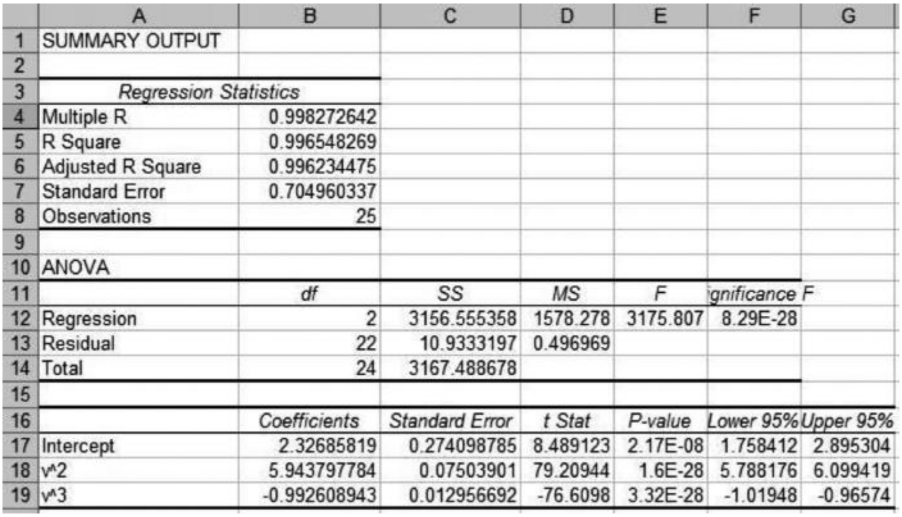
Normal Probability
□ Normal Probability Plots

For the Input Y Range, the first column was selected, and for the Input X Range, the next three columns were identified. Since the first row contains labels, the “Labels” box was checked. Also checked was the Confidence Level box and 95% for the confidence level. For the Output options, the New Workbook radio button was checked, and the following is the output from the Regression Add-In:



	A	B	C	D	E	F	G
1	SUMMARY OUTPUT						
2							
3	*Regression Statistics*						
4	Multiple R	0.998277703					
5	R Square	0.996558373					
6	Adjusted R Square	0.996066712					
7	Standard Error	0.72049303					
8	Observations	25					
9							
10	ANOVA						
11		*df*	*SS*	*MS*	*F*	*Significance F*	
12	Regression	3	3156.587363	1052.196	2026.922	5.16918E-26	
13	Residual	21	10.90131434	0.51911			
14	Total	24	3167.488678				
15							
16		*Coefficients*	*Standard Error*	*t Stat*	*P-value*	*Lower 95%*	*Upper 95%*
17	Intercept	2.224111248	0.499705233	4.450846	0.000221	1.184917331	3.2633052
18	v	0.182815879	0.736261921	0.248303	0.806312	-1.348324599	1.7139564
19	v^2	5.874718908	0.28858153	20.35722	2.63E-15	5.274580765	6.4748571
20	v^3	-0.985488667	0.031585686	-31.2005	4.45E-19	-1.051174697	-0.919803

The analysis of variance (ANOVA) portion is not covered here. The other results are identical to those that were performed “by hand.” Note that the *t* Stat (same as *t*-ratio) for *v* is less than 2, so the associated term (involving *v*) was deleted and the analysis was repeated. To do this, the second data column was not included in the X-Range in the Regression Add-In window. Results when doing that are as follows:



	A	B	C	D	E	F	G
1	SUMMARY OUTPUT						
2							
3	*Regression Statistics*						
4	Multiple R	0.998272642					
5	R Square	0.996548269					
6	Adjusted R Square	0.996234475					
7	Standard Error	0.704960337					
8	Observations	25					
9							
10	ANOVA						
11		*df*	*SS*	*MS*	*F*	*gnificance F*	
12	Regression	2	3156.555358	1578.278	3175.807	8.29E-28	
13	Residual	22	10.9333197	0.496969			
14	Total	24	3167.488678				
15							
16		*Coefficients*	*Standard Error*	*t Stat*	*P-value*	*Lower 95%*	*Upper 95%*
17	Intercept	2.32685819	0.274098785	8.489123	2.17E-08	1.758412	2.895304
18	v^2	5.943797784	0.07503901	79.20944	1.6E-28	5.788176	6.099419
19	v^3	-0.992608943	0.012956692	-76.6098	3.32E-28	-1.01948	-0.96574

Once again, the results are identical with the "hand calculations." One might ask why the hand calculations were explained when Excel Add-In would perform all of the requisite computations? The reasons are twofold: (1) so the power and results from the Regression Add-In can be fully appreciated, and (2) the Regression Add-In works only for *linear* regression, where the *Z*s in Equation 7.9 do not involve the unknown parameters. For nonlinear regression, the calculations must be done "by hand." Nonlinear regression is covered in Chapter 9.

7.5 NUMERICAL DIFFERENTIATION AND INTEGRATION REVISITED

When data to be differentiated or integrated contain random errors (usually experimental errors), it can be advantageous to first fit the data to an appropriate function using regression techniques. Then, the fitted function can be differentiated or integrated analytically. Example 7.5 illustrates this approach.

Example 7.5: Mean Heat Capacity of Gaseous Propane

The mean heat capacity of a gas is determined by the relationship

$$\bar{C}_p = \frac{\int_{T_{ref}}^{T} C_p\, dT}{T - T_{ref}} \tag{7.34}$$

The data of Figure 7.15 are for gaseous propane.

Point number	Temperature (K)	Heat capacity (kJ/kg-mol-K)
1	50	34.16
2	100	41.3
3	150	48.79
4	200	56.07
5	273.15	68.74
6	300	73.93
7	400	94.01
8	500	112.59
9	600	128.7
10	700	142.67
11	800	154.77
12	900	163.35
13	1000	174.6
14	1100	182.67
15	1200	189.74
16	1300	195.85
17	1400	201.21
18	1500	205.89

FIGURE 7.15 Heat capacity data for propane gas (reference temperature = 50 K).

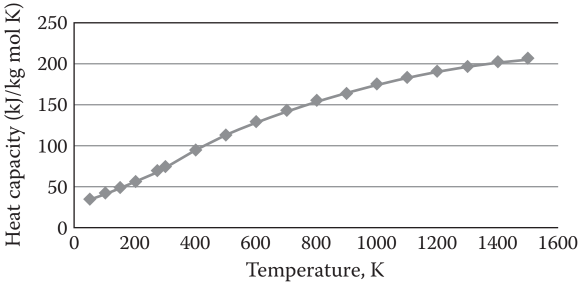


FIGURE 7.16 Plot of heat capacity of propane (reference temperature = 50 K).

A plot of the data is shown in Figure 7.16.

Because of the curvature, initially a cubic polynomial fit to the data was tried. Two parameters had *t*-ratios less than 2, and the coefficient of the cubic term was the *worst actor*. This led to a quadratic fit with the following result:

$$C_p = -6.17 \times 10^{-5}T^2 + 0.2175T + 18.1968 \tag{7.35}$$

Integrating this function between limits of 50 and 1500 K and dividing by the temperature range gives a mean heat capacity of 138.79 kJ/(kg mol K). Performing the integration by the trapezoidal rule on the same data gives a result of 139.70. Since the data are quite smooth (small experimental error), it is to be expected that these results will be similar.

EXERCISES

Exercise 7.1: For the data shown below:

v	*y*
0	1.766
0.25	2.478
0.5	3.690
0.75	6.397
1	6.649

a. Fit the data to a quadratic polynomial. As a first step, plot the data and add a quadratic polynomial trendline—this will indicate what the coefficients will be when they are calculated. Do these calculations *by hand*; determine c_1, c_2, c_3, the parameter standard deviations, the *t*-ratios, and the R^2 value (which can also be checked from the trendline).

b. Repeat the calculations of part a using the Excel Data Analysis/Regression Add-In.

c. Based on the results of parts a and b, if any of the *t*-ratios suggest an alteration in the fitting function, change the function and recalculate all necessary quantities using the Excel Data Analysis/Regression Add-In.

d. Fit the data to a fourth-degree polynomial using an Excel trendline *only*. Comment on the results (Is it good, and if so why? Is it bad, if so why?).

Exercise 7.2: Consider the following data:

x	y
0	0.998
0.1	1.061
0.2	1.050
0.3	1.111
0.4	1.298
0.5	1.482
0.6	1.751
0.7	2.211
0.8	2.658
0.9	3.262
1	3.965

a. Plot the data in the usual way.
b. Plot the data again, but with no connecting line.
c. Add a trendline to the data; display the equation and the R^2 value.
d. Use the Excel Regression Add-In to fit the data to a cubic polynomial: $y = c_1 + c_2x + c_3x^2 + c_4x^3$. Based on the calculated t-ratios (called t-stat in the results table), should all four constants be retained? If not, use the Excel Regression Add-In to redo the regression with the most significant terms in the fitting polynomial. Continue this until all t-ratios are satisfactory.
e. (Optional) Now that the answers are known, redo step d *by hand* using the appropriate equations and formulas. Formulate the linear equations to be solved, and calculate the value of R^2, the parameter standard deviations (called the standard error by the Excel Add-In), and the t-ratios. Again, repeat this step until all t-ratios are satisfactory.

Exercise 7.3: The Clausius–Clapyron equation relates the latent heat of vaporization to temperature and vapor pressure according to the following equation:

$$\ln(p) = -\frac{\Delta H_v}{RT} + k \tag{7.36}$$

Data for water are shown below (units are K for T and mmHg for p). Use linear regression to find the heat of vaporization of water; the published value is –540 cal/g. The heat of vaporization (/*R*) is the slope of the line described by Equation 7.36.

1/T	ln(P)
0.002755	6.2649
0.002740	6.3404
0.002725	6.4149
0.002710	6.4886
0.002695	6.5615
0.002681	6.6333
0.002667	6.7043
0.002653	6.7743
0.002639	6.8436
0.002625	6.9121
0.002611	6.9797

Exercise 7.4: The data shown below are to be fitted to the function

$$y^{calc} = c_1 + c_2 \frac{\ln x}{x^2} + c_3 e^{-x} \tag{7.37}$$

Consider the following data (these are "artificial" and were generated to suit this problem):

x	*y* Data
1	–0.3680
1.1	–0.2540
1.2	–0.1740
1.3	–0.1170
1.4	–0.0749
1.5	–0.0429
1.6	–0.0183
1.7	0.0009
1.8	0.0161
1.9	0.0282
2	0.0380
2.1	0.0458
2.2	0.0521
2.3	0.0572
2.4	0.0612
2.5	0.0645
2.6	0.0671
2.7	0.0690
2.8	0.0705
2.9	0.0716
3	0.0723

a. Find the three parameters c_1, c_2, and c_3 using the Excel Data Analysis Regression Add-In. If any of the parameters are insignificant based on the t-ratios, repeat the calculations using the reduced model.

b. Do the calculations for the three parameter models *by hand*. Your results should be the same as those from part a.

Exercise 7.5: The following are vapor–liquid equilibrium data for the binary system SO_2–water at 20°C.

x (mole fr. SO_2 in liquid)	y (mole fr. SO_2 in vapor)
0	0
5.62E–05	0.000685
0.00014	0.00158
0.00028	0.00421
0.000422	0.00763
0.000564	0.0112
0.000842	0.01855
0.001403	0.0342
0.001965	0.0513
0.00279	0.0775
0.0042	0.121
0.00698	0.212
0.01385	0.443
0.0206	0.682
0.0273	0.917

Fit these data to an appropriate polynomial form. First, plot the data and successively fit a linear, quadratic, cubic, and quartic *trendline*, noting the R^2 value each time; when the R^2 no longer improves, use the Excel Regression Add-On to find the parameters and statistical indicators. Verify that the polynomial chosen does indeed satisfy the usual statistical criteria. In all cases, the constant term must be zero since the (0, 0) data point is without error.

8 Partial Differential Equations

8.1 INTRODUCTION

The most frequently occurring partial differential equations (PDEs) involving two independent variables are as follows:

Parabolic:

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} \quad \text{one-dimensional unsteady state} \tag{8.1}$$

Elliptic:

$$\frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial x^2} = 0 \quad \text{two-dimensional steady state} \tag{8.2}$$

Hyperbolic:

$$\frac{\partial^2 u}{\partial t^2} + \frac{\partial^2 u}{\partial x^2} = 0 \quad \text{vibration of a string} \tag{8.3}$$

In this chapter, only parabolic and elliptic PDEs are considered.

8.2 PARABOLIC PDEs

The equation

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2} \tag{8.4}$$

describes many phenomena in chemical and biomolecular engineering. Examples are one-dimensional, unsteady-state heat conduction (where α is the thermal diffusivity), fluid flow (where α is kinematic viscosity), and molecular diffusion (where α is the molecular diffusivity). The heat conduction examples are most widely used since they are easy to visualize.

Recall Example 6.1 involving heat conduction in a thin rod of length 1 (when the perimeter is insulated). For transient operation (unsteady state), this system can be described by

$$\frac{\partial u}{\partial \theta} = \frac{\partial^2 u}{\partial x^2} \tag{8.5}$$

where α is the thermal diffusivity of the rod material and θ is a dimensionless time defined as

$$\theta = \alpha t \tag{8.6}$$

This kind of problem is a PDE-IVP. It is a boundary value problem in x and an initial value problem in θ. It remains to specify initial and boundary conditions. For simplicity, consider the following:

$$u(0,\theta) = 0;\ u(1,\theta) = 1;\ u(x,0) = 1 \quad \text{(1 everywhere except } x = 0) \tag{8.7}$$

Physically, this represents heat transfer in a one-dimensional object (a rod or slab) with unit initial temperature. At time 0, the temperature at the left boundary undergoes a step change to zero while the right boundary is maintained at a temperature of 1. This problem can be solved analytically to give

$$u = x + \frac{2}{\pi}\sum_{n=1}^{\infty}\frac{1}{n}\exp(-n^2\pi^2\theta)\sin(n\pi x) \tag{8.8}$$

The following spreadsheet shows the calculation of *one* temperature at the centerline ($x = 0.5$) and for θ = 0.1:

n	**n^2 Pi^2 Theta**	**n Pi x**	**exp(–B)**	**sin C**	**Term**	**Sum**
1	0.9869605	1.57080	0.372708	1	0.23727	0.23727
2	3.9478419	3.14159	0.019296	–5E–08	0.00000	0.23727
3	8.8826442	4.71239	0.000139	–1	–0.00003	0.23724
4	15.7913675	6.28319	1.39E–07	9E–08	0.00000	0.23724
5	24.6740117	7.85398	1.92E–11	1	0.00000	0.23724

Since $x = 0.5$, the solution at this point is $u = 0.73724$. This exercise gives a *point* that can be compared to that produced by any numerical solution.

To solve such problems numerically, a popular approach is to substitute finite difference analogs for the derivatives and solve the resulting algebraic equations. There are many ways in which this can be done, a few of which are summarized below:

1. Centered difference in x, forward difference in θ
2. Centered difference in x, backward difference in θ
3. Centered difference in x, centered difference in θ

8.2.1 Explicit Method (Centered Difference in x, Forward Difference in θ)

Solution by centered difference in x, forward difference in θ, is now illustrated. This approach suffers the same shortcomings as the Euler method for ODEs (the truncation error is $\mathcal{O}(\Delta x)$). By replacing the derivatives with the appropriate finite difference analogs (see Table 4.2) as follows:

$$\frac{u_{i,j+1}-u_{i,j}}{\Delta\theta}=\frac{u_{i-1,j}-2u_{i,j}+u_{i+1,j}}{\Delta x^2};\ j=0,1,\ldots,M;\ \mathrm{i}=0,1,2,\ldots,N+1 \tag{8.9}$$

M should be large enough to reach the steady-state solution.

This corresponds to a two-dimensional "grid" with θ on the vertical axis and x on the horizontal axis. Subscript j is an index for the θ axis while i is an index for the x-axis. When $i = 0$, the left boundary condition applies; when $i = N + 1$, the right boundary condition applies. The process begins with *u known* when $j = 0$ ($\theta = 0$ or time = zero) for all i; these are the initial conditions (zero at all points—Equation 8.7). The only *unknown* in this equation is $u_{i,j+1}$. Solving for this unknown gives

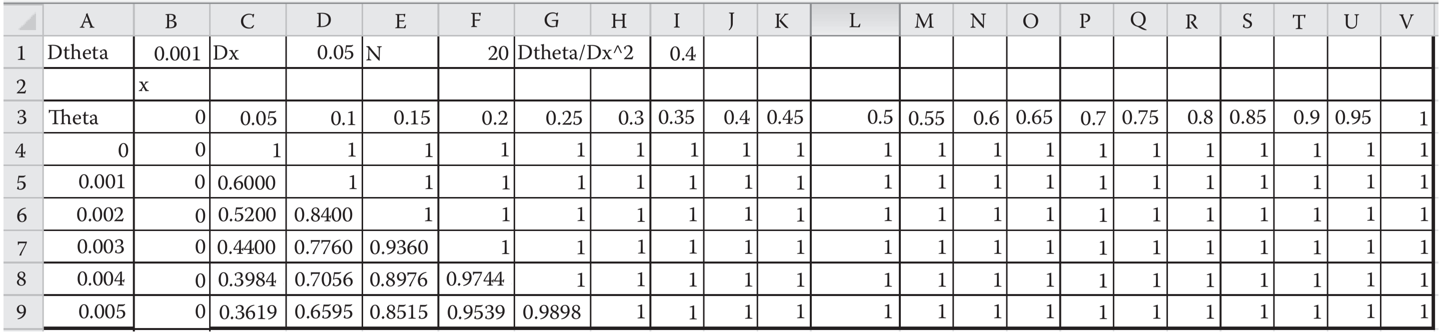
$$u_{i,j+1}=u_{i,j}+\frac{\Delta\theta}{\Delta x^2}(u_{i-1,j}-2u_{i,j}+u_{i+1,j}) \tag{8.10}$$

$\Delta\theta$ and Δx must be determined so that reasonably accurate results occur (if possible). A typical value for N is 19, but a larger value might be required. This corresponds to Δx of 0.05.

This method is the simplest approach to solving the problem numerically. It can be implemented directly in Excel®, or a VBA Macro can be written.

Example 8.1: Transient Heat Conduction in a Rod

Shown below are the first few rows of an Excel program to solve Equation 8.10 together with the initial and boundary conditions given in Equation 8.7:



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
1	Dtheta	0.001	Dx	0.05	N	20	Dtheta/Dx^2		0.4													
2		x																				
3	Theta	0	0.05	0.1	0.15	0.2	0.25	0.3	0.35	0.4	0.45	0.5	0.55	0.6	0.65	0.7	0.75	0.8	0.85	0.9	0.95	1
4	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
5	0.001	0	0.6000	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
6	0.002	0	0.5200	0.8400	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
7	0.003	0	0.4400	0.7760	0.9360	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
8	0.004	0	0.3984	0.7056	0.8976	0.9744	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
9	0.005	0	0.3619	0.6595	0.8515	0.9539	0.9898	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Values of Δθ of 0.01, 0.001, and 0.0001 were tried. Δθ = 0.001 gave nearly the same results as Δθ = 0.0001. The value of u corresponding to θ = 0.1 and *x* = 0.5 was 0.7361. This compares favorably with the analytically computed value of 0. 0.7372. When Δθ = 0.001, the numerically computed value was 0.7371. Given the simplicity of this method, these results are remarkably good. It must be pointed out, however, that when Δθ = 0.01, the solution was unstable (wildly fluctuating temperatures were the result).

8.2.2 Centered Difference in *x*, Backward Difference in θ

In the interest of keeping this discussion brief, this method will not be implemented. The required algorithms are similar to what is discussed next. A potential significant advantage of this approach is that it has excellent stability attributes. For more information, Google it.

8.2.3 Crank–Nicholson Method (Centered Difference in *x*, Centered Difference in θ)

The third method (called the Crank–Nicholson method) applies some innovation in that the finite difference analog is "centered about a fictitious half-way point" as shown in Figure 8.1.

Here are the finite difference analogs for the terms in the PDE; the unknowns are at level *j* + 1 for any *i*:

$$\frac{\partial u}{\partial t} = \frac{u_{i,j+1} - u_{i,j}}{\Delta\theta} + \mathcal{O}(\Delta\theta^2) \quad \text{(centered about } j+1/2) \tag{8.11}$$

$$\frac{\partial^2 u}{\partial x^2} = \frac{u_{i-1,j+1/2} - 2u_{i,j+1/2} + u_{i+1,j+1/2}}{\Delta x^2} + \mathcal{O}(\Delta x^2) \quad \text{(centered about } i) \tag{8.12}$$

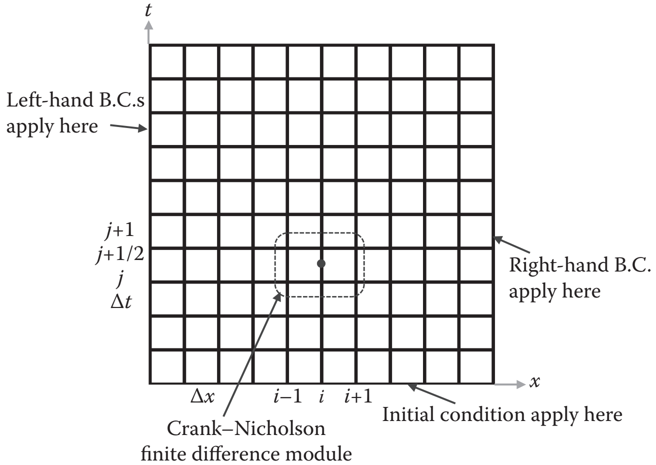

t
Left-hand B.C.s apply here
j+1
j+1/2
j
Δt
Right-hand B.C. apply here
x
Δx
i–1
i
i+1
Initial condition apply here
Crank–Nicholson finite difference module

FIGURE 8.1 Crank–Nicholson finite difference nomenclature.

The j + 1/2 terms are replaced with its *average at the j and j + 1 points*:

$$u_{i,j+1/2} = \frac{1}{2}(u_{i,j} + u_{i,j+1}) \tag{8.13}$$

With these substitutions, Equation 8.5 is replaced by the following finite difference equations:

$$-u_{i-1,j+1} + 2\left(1 + \frac{\Delta x^2}{\Delta\theta}\right)u_{i,j+1} - u_{i+1,j+1} = u_{i-1,j} - 2\left(1 - \frac{\Delta x^2}{\Delta\theta}\right)u_{i,j} + u_{i+1,j} \tag{8.14}$$

Note that the unknowns are at level j + 1, so all terms on the right-hand side are known. When the finite difference equation is applied for all i, a tridiagonal system of equations results. Such systems can be solved very efficiently (as compared to full-matrix linear systems) using a method called the Thomas algorithm (see below). The Matrix.xla function SYSLINT can also be used.

Example 8.2: Crank–Nicholson Method

The algorithm for solving the example problem can be summarized as follows:

1. Start with all *u*s at the initial conditions.
2. Set up the tridiagonal set of equations for time Δθ. Apply the boundary conditions for i = 0 and $i = N + 1$.
3. Use the Thomas algorithm or SYSLINT to find all *u*s at Δθ.
4. Repeat the process for 2 Δθ, 3 Δθ, and so forth.

The initial and boundary conditions of Equation 8.7 must be introduced. Let the vector v represent the present (known or level j) temperatures and u be the unknown (level j + 1) temperatures. The finite difference equations are as follows (note that the initial values for v are all 1). Recall that the left boundary value is 0 while the right boundary value is 1.

$$\begin{aligned}
&-0 + 2\left(1 + \frac{\Delta x^2}{\Delta\theta}\right)u_1 - u_2 = v_0 - 2\left(1 - \frac{\Delta x^2}{\Delta\theta}\right)v_1 + v_2 \quad \text{(left boundary)}\\
&-u_1 + 2\left(1 + \frac{\Delta x^2}{\Delta\theta}\right)u_2 - u_3 = v_1 - 2\left(1 - \frac{\Delta x^2}{\Delta\theta}\right)v_2 + v_3\\
&-u_2 + 2\left(1 + \frac{\Delta x^2}{\Delta\theta}\right)u_3 - u_4 = v_2 - 2\left(1 - \frac{\Delta x^2}{\Delta\theta}\right)v_3 + v_4\\
&\qquad\vdots\\
&-u_{N-1} + 2\left(1 + \frac{\Delta x^2}{\Delta\theta}\right)u_N = v_{N-1} - 2\left(1 - \frac{\Delta x^2}{\Delta\theta}\right)v_N + 1 \quad \text{(right boundary)}
\end{aligned} \tag{8.15}$$

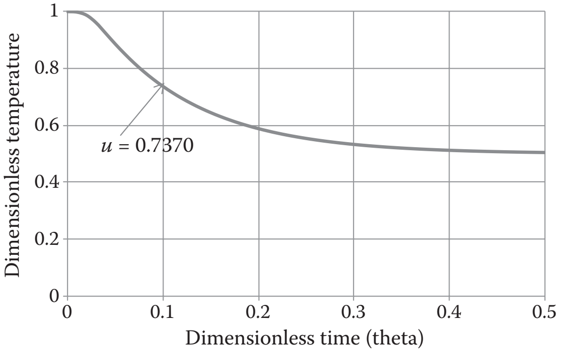


FIGURE 8.2 Centerline temperature ($\Delta\theta = 0.1$, $\Delta x = 0.05$).

This is a tridiagonal system. Once solved for all *u*s, replace *v* with *u* and repeat for any desired number of time steps.

Figure 8.2 is a graph of the temperature when $x = 0.5$ (the centerline). The numerical solution at the centerline (when $\Delta\theta = 0.1$ and $\Delta x = 0.05$) for $\theta = 0.1$ is 0.7370, which is the same as the infinite series solution to three significant figures. Accuracy to additional significant figures is easily obtained by reducing $\Delta\theta$. It is highly significant that a value of $\Delta\theta = 0.01$ is one or two orders of magnitude larger than was required with the Euler in time method.

8.3 THOMAS ALGORITHM FOR TRIDIAGONAL SYSTEMS

The Thomas algorithm is well known in numerical mathematics as an implementation of Gaussian elimination applied specifically to tridiagonal systems. If each equation is of the form

$$c_i x_{i-1} + d_i x_i + e_i x_{i+1} = b_i \tag{8.16}$$

applying Gaussian elimination to the system results in the following algorithm:

$$\begin{aligned}
&1.\ \beta_1 = d_1 \\
&2.\ \gamma_1 = \frac{b_1}{\beta_1} \\
&3.\ \beta_i = d_i - \frac{c_i \cdot e_{i-1}}{\beta_{i-1}} \quad i = 2,3,\cdots,n \\
&4.\ \gamma_i = \frac{b_i - c_i \cdot \gamma_{i-1}}{\beta_i} \quad i = 2,3,\cdots,n \\
&5.\ x_n = \gamma_n \\
&6.\ x_i = \gamma_i - \frac{e_i \cdot x_{i+1}}{\beta_i} \quad i = n-1, n-2, \cdots, 1
\end{aligned} \tag{8.17}$$

Example 8.3: Illustration of the Thomas Algorithm

A three-equation, three-unknown example is as follows:

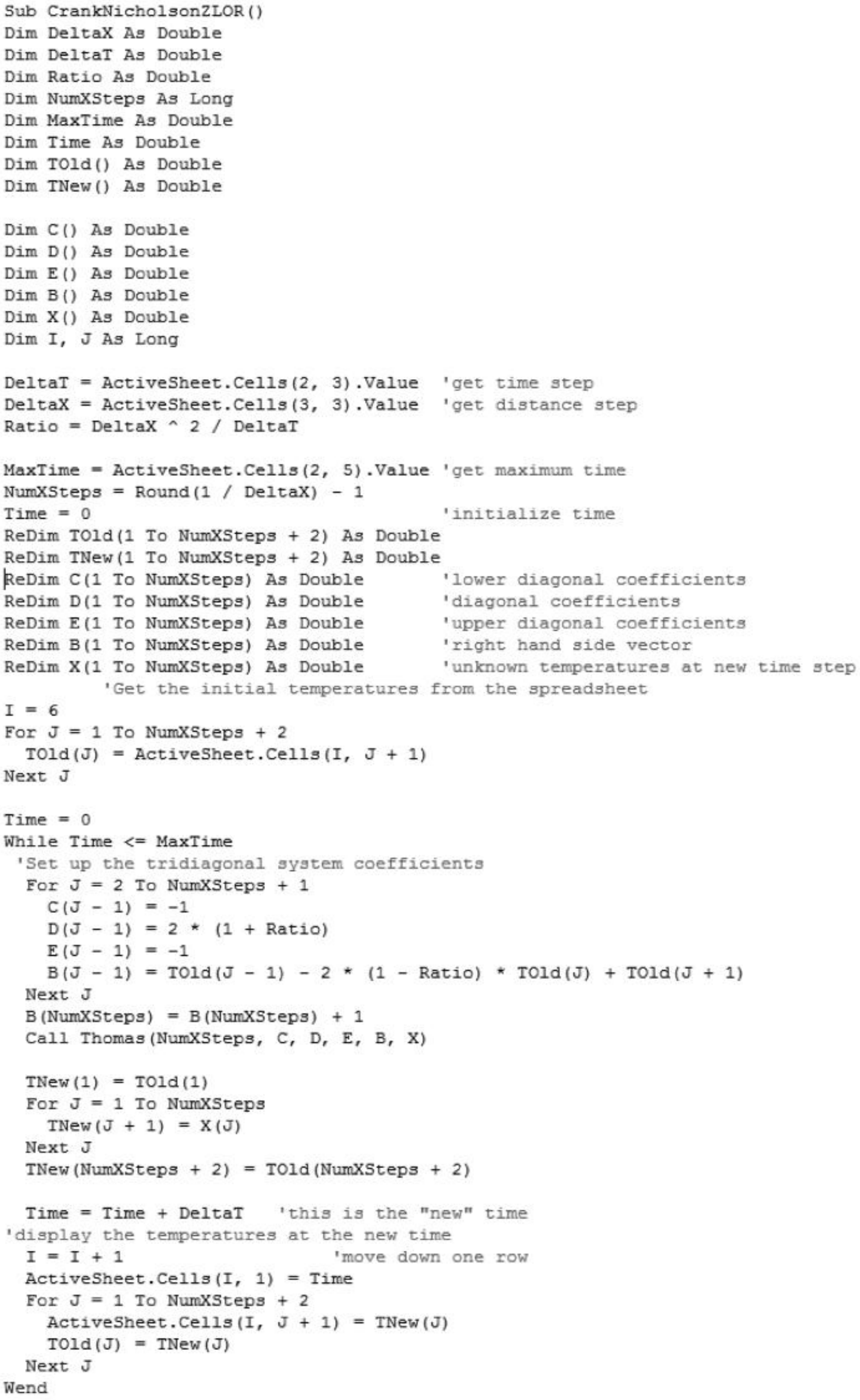
$$\begin{bmatrix} 2 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \\ 3 \end{bmatrix} \quad \text{(note that the matrix is tridiagonal)}$$

The steps of the Thomas algorithm are shown below:

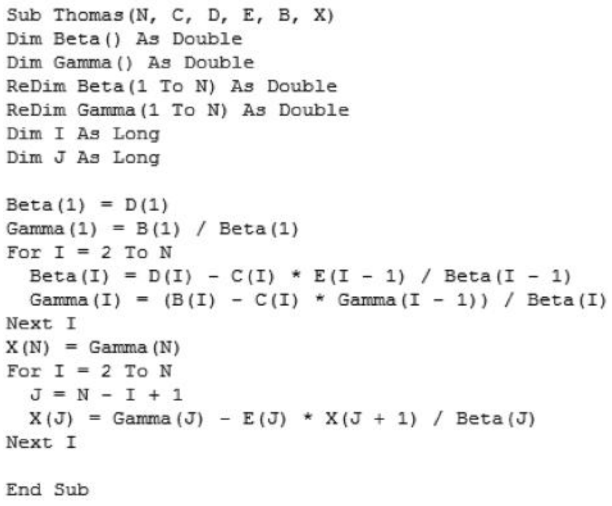
$$\begin{aligned}
& \beta_1 = 2 \quad \gamma_1 = 3/2 \\
i = 2: \quad & \beta_2 = 2 - \frac{1 \cdot 1}{2} = 3/2 \\
& \gamma_2 = \frac{4 - 1(3/2)}{3/2} = \frac{8/2 - 3/2}{3/2} = \frac{5/2}{3/2} = 5/3 \\
i = 3: \quad & \beta_3 = 2 - \frac{1 \cdot 1}{3/2} = 6/3 - 2/3 = 4/3 \\
& \gamma_3 = \frac{3 - 1(5/3)}{4/3} = \frac{9/3 - 5/3}{4/3} = \frac{4/3}{4/3} = 1 \\
& x_3 = 1 \\
i = 2: \quad & x_2 = \frac{5}{3} - \frac{1 \cdot 1}{3/2} = \frac{5}{3} - \frac{2}{3} = 1 \\
i = 1: \quad & x_1 = \frac{3}{2} - \frac{1 \cdot 1}{2} = \frac{3}{2} - \frac{1}{2} = 1
\end{aligned}$$

Example 8.4: VBA Program for the Crank–Nicholson Method

A VBA program that implements the Crank–Nicholson method for the problem of Equation 8.5 with the initial and boundary conditions of Equation 8.7 is shown below. Equation 8.15 forms the basis for the tridiagonal system to solve at each time step.



```
Sub CrankNicholsonZLOR()
Dim DeltaX As Double
Dim DeltaT As Double
Dim Ratio As Double
Dim NumXSteps As Long
Dim MaxTime As Double
Dim Time As Double
Dim TOld() As Double
Dim TNew() As Double

Dim C() As Double
Dim D() As Double
Dim E() As Double
Dim B() As Double
Dim X() As Double
Dim I, J As Long

DeltaT = ActiveSheet.Cells(2, 3).Value  'get time step
DeltaX = ActiveSheet.Cells(3, 3).Value  'get distance step
Ratio = DeltaX ^ 2 / DeltaT

MaxTime = ActiveSheet.Cells(2, 5).Value 'get maximum time
NumXSteps = Round(1 / DeltaX) - 1
Time = 0                                'initialize time
ReDim TOld(1 To NumXSteps + 2) As Double
ReDim TNew(1 To NumXSteps + 2) As Double
ReDim C(1 To NumXSteps) As Double       'lower diagonal coefficients
ReDim D(1 To NumXSteps) As Double       'diagonal coefficients
ReDim E(1 To NumXSteps) As Double       'upper diagonal coefficients
ReDim B(1 To NumXSteps) As Double       'right hand side vector
ReDim X(1 To NumXSteps) As Double       'unknown temperatures at new time step
         'Get the initial temperatures from the spreadsheet
I = 6
For J = 1 To NumXSteps + 2
  TOld(J) = ActiveSheet.Cells(I, J + 1)
Next J

Time = 0
While Time <= MaxTime
 'Set up the tridiagonal system coefficients
  For J = 2 To NumXSteps + 1
    C(J - 1) = -1
    D(J - 1) = 2 * (1 + Ratio)
    E(J - 1) = -1
    B(J - 1) = TOld(J - 1) - 2 * (1 - Ratio) * TOld(J) + TOld(J + 1)
  Next J
  B(NumXSteps) = B(NumXSteps) + 1
  Call Thomas(NumXSteps, C, D, E, B, X)

  TNew(1) = TOld(1)
  For J = 1 To NumXSteps
    TNew(J + 1) = X(J)
  Next J
  TNew(NumXSteps + 2) = TOld(NumXSteps + 2)

  Time = Time + DeltaT   'this is the "new" time
'display the temperatures at the new time
  I = I + 1                   'move down one row
  ActiveSheet.Cells(I, 1) = Time
  For J = 1 To NumXSteps + 2
    ActiveSheet.Cells(I, J + 1) = TNew(J)
    TOld(J) = TNew(J)
  Next J
Wend
```



```
Sub Thomas(N, C, D, E, B, X)
Dim Beta() As Double
Dim Gamma() As Double
ReDim Beta(1 To N) As Double
ReDim Gamma(1 To N) As Double
Dim I As Long
Dim J As Long

Beta(1) = D(1)
Gamma(1) = B(1) / Beta(1)
For I = 2 To N
  Beta(I) = D(I) - C(I) * E(I - 1) / Beta(I - 1)
  Gamma(I) = (B(I) - C(I) * Gamma(I - 1)) / Beta(I)
Next I
X(N) = Gamma(N)
For I = 2 To N
  J = N - I + 1
  X(J) = Gamma(J) - E(J) * X(J + 1) / Beta(J)
Next I

End Sub
```

The spreadsheet shown on the next page illustrates the program user interface and output. Note, particularly, the value for $\theta = 0.1$ and $x = 0.5$. To three significant figures, this value is the same as that provided by the analytical solution.

To review, the Crank–Nicholson method involves writing second-order correct finite difference approximations by using a fictitious "halfway point" in time. To resolve the unknowns at the halfway point, the average between the old time points and the new time points is used. This leads to a tridiagonal system of linear equations to be solved at each time step. The resulting algorithm is second-order correct in both distance and time. Using the same example, the time step required by the Crank–Nicholson method was two orders of magnitude greater than when the Euler in time method was used.

Heat transfer in a slab						Crank–Nicholson				Left boundary zero				Right boundary at one				IC = 1			
	Dt =	0.01	Tmax	1						0.25						0.125	0.25				
	Dx=	0.05																			
	Distance, dimensionless																				
Time	0	0.05	0.1	0.15	0.2	0.25	0.3	0.35	0.4	0.45	0.5	0.55	0.6	0.65	0.7	0.75	0.8	0.85	0.9	0.95	1
0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0.01	0	3E-12	0.5	0.75	0.875	0.938	0.969	0.984	0.992	0.996	0.998	0.999	1	1	1	1	1	1	1	1	1
0.02	0	0.333	0.333	0.5	0.667	0.792	0.875	0.927	0.958	0.977	0.987	0.993	0.996	0.998	0.999	0.999	1	1	1	1	1
0.03	0	0.074	0.352	0.472	0.579	0.683	0.774	0.847	0.9	0.936	0.96	0.976	0.986	0.992	0.995	0.997	0.998	0.999	1	1	1
0.04	0	0.193	0.243	0.395	0.523	0.624	0.71	0.782	0.842	0.888	0.923	0.949	0.967	0.979	0.987	0.992	0.995	0.997	0.998	0.999	1
0.05	0	0.09	0.271	0.365	0.467	0.568	0.656	0.731	0.794	0.845	0.886	0.918	0.942	0.96	0.973	0.982	0.989	0.993	0.996	0.998	1
0.06	0	0.138	0.208	0.335	0.438	0.529	0.612	0.686	0.751	0.806	0.851	0.888	0.917	0.94	0.957	0.97	0.98	0.987	0.992	0.996	1
0.07	0	0.09	0.224	0.309	0.404	0.495	0.577	0.65	0.714	0.77	0.818	0.858	0.891	0.918	0.939	0.956	0.969	0.979	0.987	0.994	1
0.08	0	0.11	0.187	0.294	0.384	0.467	0.546	0.618	0.682	0.739	0.789	0.831	0.867	0.897	0.921	0.941	0.957	0.971	0.982	0.991	1
0.09	0	0.086	0.194	0.274	0.362	0.444	0.52	0.59	0.654	0.711	0.762	0.806	0.844	0.876	0.903	0.926	0.945	0.962	0.976	0.988	1
0.1	0	0.094	0.171	0.264	0.345	0.423	0.497	0.566	0.629	0.686	**0.737**	0.782	0.822	0.856	0.886	0.912	0.934	0.953	0.97	0.985	1
0.11	0	0.081	0.173	0.249	0.33	0.406	0.478	0.545	0.607	0.663	0.715	0.761	0.802	0.838	0.87	0.898	0.922	0.944	0.964	0.982	1

8.4 METHOD OF LINES

When using finite differences to solve the unsteady heat conduction problem, another approach involves writing finite difference equations at each grid point (node) only for the spatial variables while leaving the time derivative intact. This leads, generally, to a large number of simultaneous ODEs, which can be solved by, for example, a Runge–Kutta method. However, one must be careful since this set of ODEs can be *stiff*. Consider the same one-dimensional, unsteady state heat conduction problem as solved in Examples 8.1 and 8.2. This problem is solved by the method of lines in the next example.

Example 8.5: Method of Lines

$$\begin{gathered}\frac{\partial u}{\partial \theta}=\frac{\partial^2 u}{\partial x^2}\\ u(0,t)=0;\quad u(1,t)=1\quad u(x,0)=1\end{gathered}\tag{8.18}$$

Applying a second-order correct finite difference analog for the spatial derivative term at "line" *i* gives

$$\frac{du_i}{dt}=\frac{u_{i-1}-2u_i+u_{i+1}}{\Delta x^2};\quad i=1,2,\cdots n\tag{8.19}$$

At the left boundary, u_{i-1} = 0 because of the left boundary condition. So, the ODE when *i* = 1 becomes

$$\frac{du_1}{dt}=\frac{-2u_1+u_2}{\Delta x^2}\tag{8.20}$$

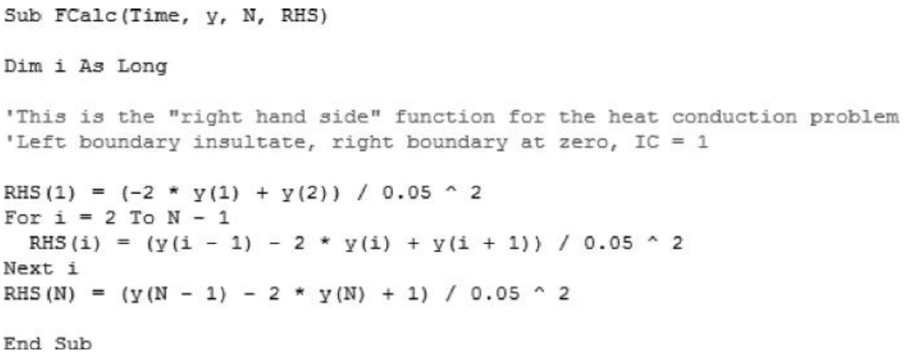
For 2 ≤ *i* ≤ *n* − 1, the ODEs are

$$\frac{du_i}{dt}=\frac{u_{i-1}-2u_i+u_{i+1}}{\Delta x^2};\quad i=2,3,\cdots n-1\tag{8.21}$$

And the last equation (*i* = *n* − 1) is as follows because the right-hand boundary condition is 1:

$$\frac{du_n}{dt}=\frac{u_{n-1}-2u_n+1}{\Delta x^2}\tag{8.22}$$

Shown below is a VBA subprogram `FCalc` that would be called by an ODE solver such as a Runge–Kutta method. The subprogram implements the right-hand-side functions of Equations 8.20 through 8.22.



```
Sub FCalc(Time, y, N, RHS)

Dim i As Long

'This is the "right hand side" function for the heat conduction problem
'Left boundary insultate, right boundary at zero, IC = 1

RHS(1) = (-2 * y(1) + y(2)) / 0.05 ^ 2
For i = 2 To N - 1
  RHS(i) = (y(i - 1) - 2 * y(i) + y(i + 1)) / 0.05 ^ 2
Next i
RHS(N) = (y(N - 1) - 2 * y(N) + 1) / 0.05 ^ 2

End Sub
```

The results for this implementation are essentially identical to the results shown in Example 8.2.

8.5 SUCCESSIVE OVERRELAXATION FOR ELLIPTIC PDEs

Consider Laplace's equation in two dimensions (this is an elliptic equation):

$$\frac{\partial^2\phi}{\partial x^2}+\frac{\partial^2\phi}{\partial y^2}=0 \tag{8.23}$$

This equation represents many steady-state phenomena in two dimensions, such as temperature distributions, laminar flow distributions, and voltage distributions.

Example 8.6: Relaxation Method for an Elliptic Equation

Consider a rectangular (thin) flat plate as shown in Figure 8.3 with the given conditions along each edge.

Substituting second-order correct finite difference analogs into Equation 8.23, there results

$$\frac{\phi_{i-1,j}-2\phi_{i,j}+\phi_{i+1,j}}{\Delta x^2}+\frac{\phi_{i,j-1}-2\phi_{i,j}+\phi_{i,j+1}}{\Delta y^2}=0 \tag{8.24}$$

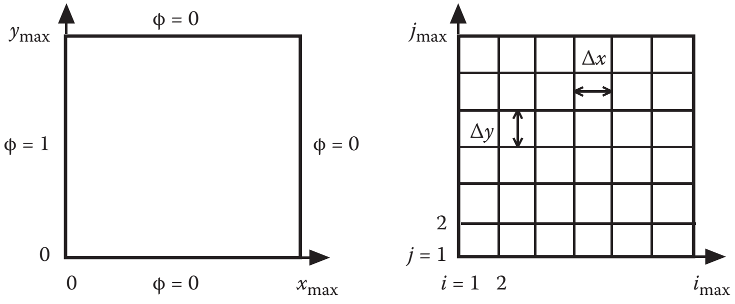


FIGURE 8.3 Nomenclature: finite difference representation of heat transfer in a flat plate.

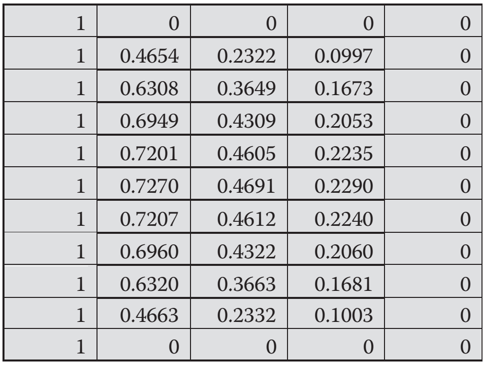
Solving for $\phi_{i,j}$ gives

$$\phi_{i,j} = \frac{\Delta x^2(\phi_{i,j-1} + \phi_{i,j+1}) + \Delta y^2(\phi_{i-1,j} + \phi_{i+1,j})}{2\Delta x^2 + 2\Delta y^2} \tag{8.25}$$

If $\Delta x = \Delta y$, this can be written as the average of the four surrounding temperatures as in Equation 8.26.

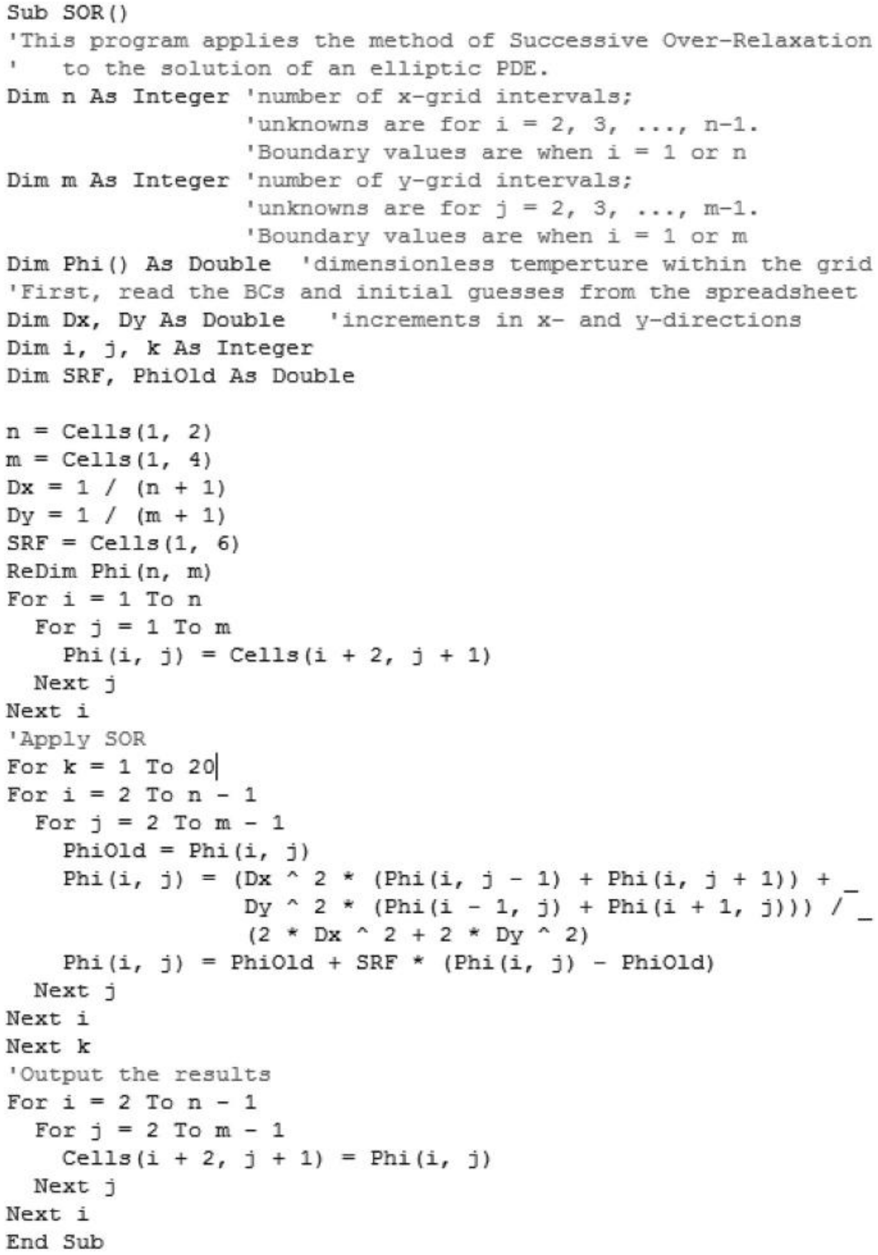
$$\phi_{i,j} = \frac{\phi_{i,j-1} + \phi_{i,j+1} + \phi_{i-1,j} + \phi_{i+1,j}}{4} \tag{8.26}$$

To apply this *recurrence* relation, start by guessing the φ values at all nodes and then sweep through all *i* and *j* updating with the above "average of 4" formula. This process is repeated until there is only a small change from one sweep (iteration) to the next or for a fixed number of iterations (easier). The process is called *relaxation*. The relaxation method is a procedure for solving simultaneous equations by guessing a solution and then reducing the errors that result by *successive approximations* until all the errors are less than some specified amount. The VBA program shown below implements the relaxation method of Equation 8.26 and for the boundary conditions shown in Figure 8.3. The program is set to 20 iterations; this is much easier testing to see if the temperatures cease to change. Results are shown in Figure 8.4.



1	0	0	0	0
1	0.4654	0.2322	0.0997	0
1	0.6308	0.3649	0.1673	0
1	0.6949	0.4309	0.2053	0
1	0.7201	0.4605	0.2235	0
1	0.7270	0.4691	0.2290	0
1	0.7207	0.4612	0.2240	0
1	0.6960	0.4322	0.2060	0
1	0.6320	0.3663	0.1681	0
1	0.4663	0.2332	0.1003	0
1	0	0	0	0

FIGURE 8.4 Relaxation results after 20 iterations.



```
Sub SOR()
'This program applies the method of Successive Over-Relaxation
'   to the solution of an elliptic PDE.
Dim n As Integer 'number of x-grid intervals;
                 'unknowns are for i = 2, 3, ..., n-1.
                 'Boundary values are when i = 1 or n
Dim m As Integer 'number of y-grid intervals;
                 'unknowns are for j = 2, 3, ..., m-1.
                 'Boundary values are when i = 1 or m
Dim Phi() As Double  'dimensionless temperture within the grid
'First, read the BCs and initial guesses from the spreadsheet
Dim Dx, Dy As Double   'increments in x- and y-directions
Dim i, j, k As Integer
Dim SRF, PhiOld As Double

n = Cells(1, 2)
m = Cells(1, 4)
Dx = 1 / (n + 1)
Dy = 1 / (m + 1)
SRF = Cells(1, 6)
ReDim Phi(n, m)
For i = 1 To n
  For j = 1 To m
    Phi(i, j) = Cells(i + 2, j + 1)
  Next j
Next i
'Apply SOR
For k = 1 To 20
For i = 2 To n - 1
  For j = 2 To m - 1
    PhiOld = Phi(i, j)
    Phi(i, j) = (Dx ^ 2 * (Phi(i, j - 1) + Phi(i, j + 1)) + _
                 Dy ^ 2 * (Phi(i - 1, j) + Phi(i + 1, j))) / _
                 (2 * Dx ^ 2 + 2 * Dy ^ 2)
    Phi(i, j) = PhiOld + SRF * (Phi(i, j) - PhiOld)
  Next j
Next i
Next k
'Output the results
For i = 2 To n - 1
  For j = 2 To m - 1
    Cells(i + 2, j + 1) = Phi(i, j)
  Next j
Next i
End Sub
```

Because of symmetry, it is known that the values 0.2883 and 0.2924 should be the same, but even after 15 iterations, they are relatively far apart. To improve this, there is the method of successive overrelaxation or the SOR method. The idea here is that on each sweep, the newly calculated value is not used directly; instead an interpolation/extrapolation formula as shown by the following equation is used:

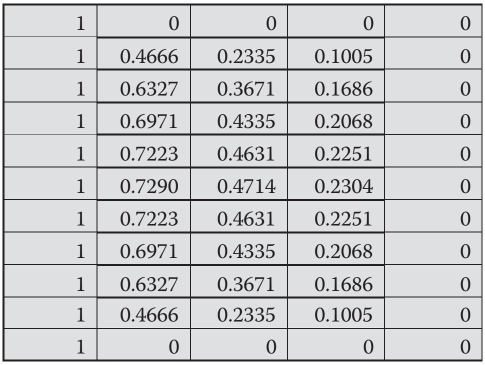
$$\phi_{New} = \phi_{Old} + SRF(\phi_{Calc} - \phi_{Old}) \tag{8.27}$$

where *SRF* is the *successive relaxation factor.* If *SRF* = 1, the new value is equal to the calculated one. If *SRF* < 1, the formula interpolates, and if *SRF* > 1, it extrapolates. Typically "good" values for *SRF* are between 1.2 and 1.5.

Example 8.7: Successive Overrelaxation

When Equation 8.27 is used with an SRF of 1.5 and 20 iterations, the results are as shown in Figure 8.5. Note that these data are symmetric to the precision shown.

Much more time could be spent on PDEs of various kinds and with a variety of boundary conditions. Time does not allow this, but with the background gained so far, the reader should be able to comprehend more advanced numerical analysis textbooks and research papers to learn about solving other kinds of problems. Want to know more? Google it!



1	0	0	0	0
1	0.4666	0.2335	0.1005	0
1	0.6327	0.3671	0.1686	0
1	0.6971	0.4335	0.2068	0
1	0.7223	0.4631	0.2251	0
1	0.7290	0.4714	0.2304	0
1	0.7223	0.4631	0.2251	0
1	0.6971	0.4335	0.2068	0
1	0.6327	0.3671	0.1686	0
1	0.4666	0.2335	0.1005	0
1	0	0	0	0

FIGURE 8.5 Successive overrelaxation results for 20 iterations.

EXERCISES

Exercise 8.1: Rework the problem of Example 8.1 using the simple explicit method and the following initial and boundary conditions:

a. $u\,(x, 0) = 1$; $u\,(0, t) = u\,(1, t) = 0$
b. $u\,(x, 0) = 1$; $u(0, t) = 0$; $du(1, t)/dx = 0$ (right side insulated)
c. $u\,(x, 0) = x$ (linear initial temperature profile); $u\,(0, t) = 1$; $u\,(1, t) = 0$
d. $u\,(x, 0) = \sin(\pi x)$; $u\,(0, t) = u\,(1, t) = 0$
e. $u\,(x, 0) = x$; $du\,(0, t)/dx = du\,(1, t)/dx = 0$

Exercise 8.2: Rework Exercise 8.1 using the Crank–Nicholson method.

Exercise 8.3: Rework Exercise 8.1 using the method of lines.

Exercise 8.4: The boundary value problem for a rectangular fin is as follows:

$$\frac{\partial^2 T}{\partial x^2} - \beta^2 (T - T_a) = \frac{1}{\alpha}\frac{\partial T}{\partial t} \tag{8.28}$$

$$\begin{aligned} &0 \le x \le 1 \\ &T(0,t) = T_1 \\ &T(1,t) = T_2 \\ &T(x,0) = 0 \end{aligned} \tag{8.29}$$

Define a dimensionless temperature and time as follows:

$$u = \frac{T - T_a}{T_1 - T_a} \qquad \tau = \alpha t \tag{8.30}$$

The boundary value problem then becomes

$$\frac{\partial^2 u}{\partial x^2} - \beta^2 u = \frac{\partial u}{\partial \tau} \tag{8.31}$$

The BCs become

$$\begin{aligned} &u(0,t) = 1 \\ &u(1,t) = \frac{T_2 - T_a}{T_1 - T_a} \\ &u(x,0) = \frac{-T_a}{T_1 - T_a} \end{aligned} \tag{8.32}$$

The steady-state solution can be shown to be

$$u_{ss} = \frac{\sinh[\beta(1-x)]}{\sinh\beta} + \left(\frac{T_2 - T_a}{T_1 - T_a}\right)\frac{\sinh\beta x}{\sinh\beta} \tag{8.33}$$

For simplicity, take $T_a = 0$, $T_1 = 1$, $T_2 = 0$, and $\beta = 4$.

The steady-state problem was described in Exercise 6.5. The present exercise involves solving the transient problem by the methods presented in this chapter. Be sure that in each case, the eventual profile corresponds to the steady-state solution.

a. Solve the transient (PDE) dimensionless problem using the explicit method.
b. Solve the transient (PDE) dimensionless problem using the Crank–Nicholson method.
c. Solve the transient (PDE) dimensionless problem using the method of lines.

Exercise 8.5: Resolve the example elliptic PDE of Equation 8.23 using the following boundary conditions and with an SOR factor of 1.4. Use a grid such that $\Delta x = \Delta y$.

$d\varphi(0, y)/dy = 0$	Insulated left boundary
$\varphi(1, y) = 1$	Right side at 1
$d\varphi(x, 0)/dx = 0$	Insulated bottom boundary
$\varphi(x, 1) = 0$	Top side at 0

Experiment with different SOR factors and compare the results.

9 Linear Programming, Nonlinear Programming, Nonlinear Equations, and Nonlinear Regression Using Solver

9.1 INTRODUCTION

Solver is a powerful tool that is a standard Excel® Add-In. It can produce solutions to many different kinds of problems, among which are the following:

- Linear programming
- Nonlinear programming
- Nonlinear regression
- Nonlinear sets of equations

Solver can also be used for simpler problems such as solving one nonlinear equation (such as those discussed in Chapter 1). It is a much more powerful tool than Goal Seek and might be used in instances where Goal Seek fails. For most problem types (except most notably linear programming), there is no guarantee that Solver will find a solution. An initial guess must be provided, and then Solver attempts to find better and better estimates for the unknowns until it finds conditions that indicate that a solution has been found, or it gives an error message stating failure.

9.2 LINEAR PROGRAMMING

Linear programming (LP) involves problems with an objective function (often profit or loss) and constraints (equality and inequality) that typically specify the availability (or lack thereof) of resources. The problem is to maximize (or minimize) the objective function while satisfying all of the constraints. As the name implies, the objective function and constraints are *linear* functions of the unknown variables. In this context, the word *programming* does not refer to a computer program but to the action or process of scheduling something such as assigning people to jobs or how to allocate resources. Many important engineering (and other) problems can be formulated as linear programs.

Example 9.1: A Simple LP Problem

Consider the following LP in two variables:

$$\begin{aligned}&\text{Maximize } y = 2x_1 + 4x_2\\&\text{subject to}\\&x_2 - x_1 \le 4\\&x_1 + x_2 \le 8\end{aligned} \tag{9.1}$$

It is typical to also impose so-called nonnegativity constraints. These are not essential but "traditional." This can always be made the case by change of variables, but some lower limit is most often imposed. The nonnegativity constraints are expressed as

$$\begin{aligned}x_1 &\ge 0\\x_2 &\ge 0\end{aligned} \tag{9.2}$$

A graphical representation of this LP problem is shown in Figure 9.1. The solid lines represent the two constraints, and the arrows point into the feasible region (where both constraints are satisfied). Lines representing the objective function (y) are dashed. It is obvious from the figure that the maximum value of y on (or within) the feasible region is 28. This same result can be found by simply solving the two constraints as equalities.

In the late 1940s, George Dantzig perfected the so-called *simplex algorithm* for solving LP problems. This effort was initiated during World War II but was kept secret until 1946. The simplex method starts with an initial guess of the origin (called the

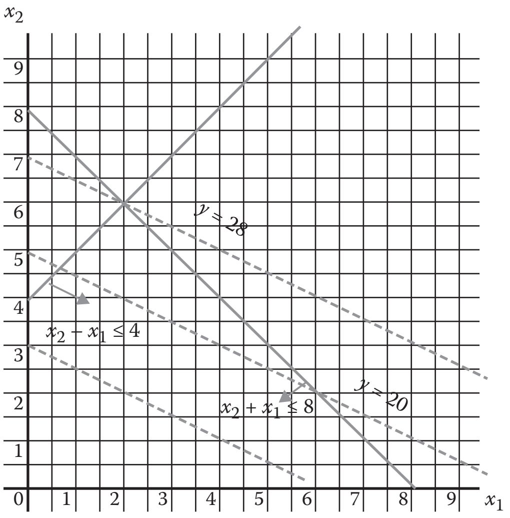
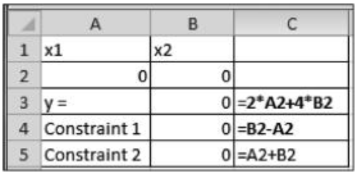


FIGURE 9.1 Graphical interpretation of a linear programming problem.

basic solution since it always satisfies all of the constraints). It then *visits* vertices based on those that give the most improvement in the objective function. It is actually a bit more complex than that, but this is the general idea. For a long time, researchers were surprised by the repeated successes of the simplex algorithm. It was finally proven that any LP problem can be solved in what is called *polynomial time* (Google it).

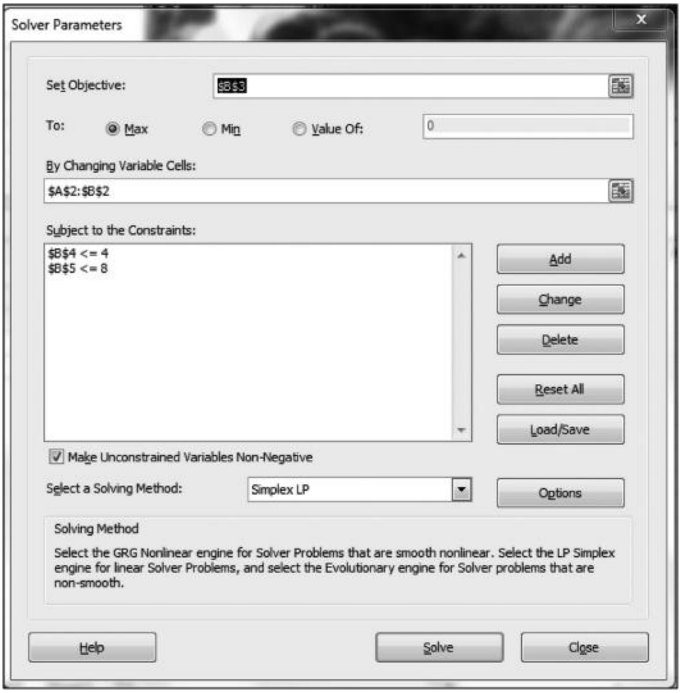
Solver can handle LPs very robustly. Before invoking Solver, a spreadsheet must be set up to represent the objective function and constraints. A typical spreadsheet setup for the example problem appears below:



	A	B	C
1	x1	x2	
2	0	0	
3	y =	0	**=2*A2+4*B2**
4	Constraint 1	0	**=B2-A2**
5	Constraint 2	0	**=A2+B2**

The actual formulas for the cells in column B are shown in bold in column C. The initial values for x_1 and x_2 have been set to zero; other values could be used, but selecting the origin as initial values is the usual procedure.

To start Solver, use the menu Data/Solver. The following window shows the Solver Parameters filled in for the example problem. Note that cell B3 contains the value of the objective function, and the Max button is selected. The By Changing Variable Cells has the range indicator for x_1 and x_2. The constraints were included by using the Add button and entering the appropriate values. Note that the box next to Make Unconstrained Variables Non-Negative is checked.



Solver Parameters

Set Objective: $B$3

To: ● Max ○ Min ○ Value Of: 0

By Changing Variable Cells: $A$2:$B$2

Subject to the Constraints:

$B$4 <= 4
$B$5 <= 8

☑ Make Unconstrained Variables Non-Negative

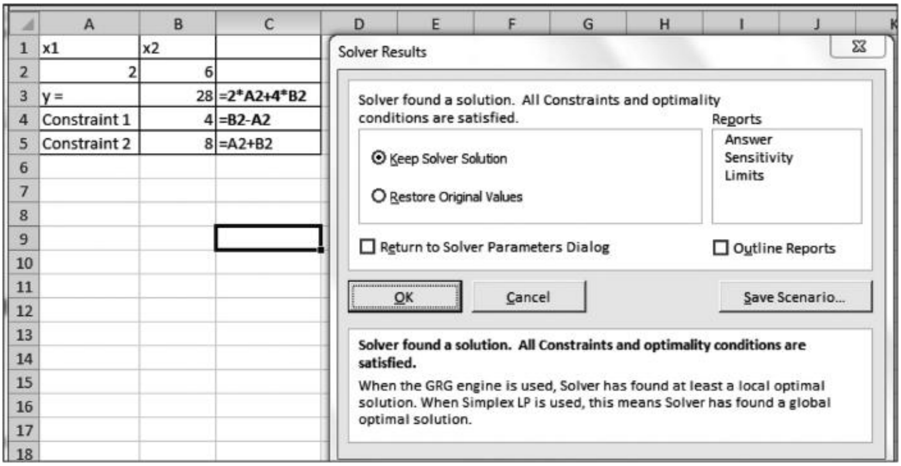
Select a Solving Method: Simplex LP

Solving Method

Select the GRG Nonlinear engine for Solver Problems that are smooth nonlinear. Select the LP Simplex engine for linear Solver Problems, and select the Evolutionary engine for Solver problems that are non-smooth.

Importantly, the Select a Solving Method shows that the Simplex LP method has been chosen.

When the Solve button is selected, the spreadsheet changes to the following:



	A	B	C
1	x1	x2	
2	2	6	
3	y =	28	=2*A2+4*B2
4	Constraint 1	4	=B2-A2
5	Constraint 2	8	=A2+B2

Solver Results

Solver found a solution. All Constraints and optimality conditions are satisfied.

● Keep Solver Solution

○ Restore Original Values

□ Return to Solver Parameters Dialog

Reports: Answer, Sensitivity, Limits

□ Outline Reports

Solver found a solution. All Constraints and optimality conditions are satisfied.

When the GRG engine is used, Solver has found at least a local optimal solution. When Simplex LP is used, this means Solver has found a global optimal solution.

The Solver Results window indicates if a solution has been found that satisfies all constraints and optimality conditions. By selecting OK, the Solver Results window disappears, and the optimal values of x_1 and x_2 remain displayed on the spreadsheet (hitting cancel causes all spreadsheet values to revert to the initial ones).

The next example is a more realistic one of interest to chemical engineers. It is a very simplified version of what might be used by oil refinery management in order to optimize plant operation.

Example 9.2: Refinery Linear Program

Suppose that a refinery has four different types of crude oil (in a tank farm) available. In order to optimize plant operation (maximize profit), the plant management can process different amounts of the four crudes. Perhaps the hardest part of any LP problem is collecting the data and constructing the objective function and constraints. Figure 9.2 shows the product fractions that result from refining each

Crude number		1	2	3	4
Fractional product	Gasoline	0.15	0.55	0.3	0.25
	Diesel	0.25	0.2	0.3	0.35
	Aviation fuel	0.25	0.15	0.25	0.15
	Lube oil	0.25	0.05	0	0.1
	Losses	0.01	0	0.05	0.15
Availability 1000 Bbl/day		20	25	35	30

FIGURE 9.2 Crude oil product fractions and availability.

crude. Also shown is the availability of each crude oil. Figure 9.3 contains data regarding processing costs, sales price, and market demand.

Let x_1, x_2, x_3, and x_4 represent the amount of each crude to be processed (1000 Bbl/day). The objective function is the profit generated by processing the four crude oils in the optimal proportions. Profit for each crude is calculated as the amount of the crude (in barrels/day) times the profit margin (selling price – processing cost) times the fraction yield for each product. For example, for crude no. 1, the net profit is calculated as

$$\begin{aligned} &x_1[(56 - 52)(0.15) + (49 - 45)(0.25) + (62 - 55)(0.25) \\ &\quad + (70 - 60)(0.25)] * 1000 = 5850x_1 \end{aligned} \tag{9.3}$$

Using similar calculations for the other crudes, the objective function becomes

$$y = 5850x_1 + 5050x_2 + 5150x_3 + 4450x_4 \tag{9.4}$$

There are three kinds of constraints: crude availability, market demands, and nonnegativity.

Crude availability constraints:

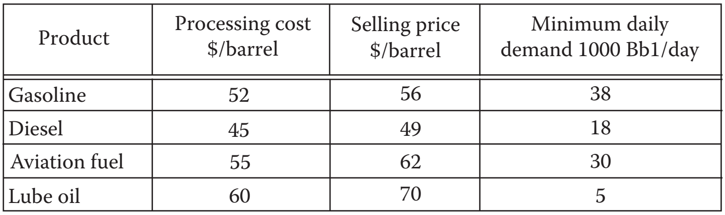
$$x_1 \le 20 \; x_2 \le 25 \; x_3 \le 35 \; x_4 \le 30 \tag{9.5}$$

Demand constraints:

$$\begin{aligned} 0.15x_1 + 0.55x_2 + 0.30x_3 + 0.25x_4 &\le 38 \\ 0.25x_1 + 0.20x_2 + 0.30x_3 + 0.35x_4 &\le 18 \\ 0.25x_1 + 0.15x_2 + 0.25x_3 + 0.15x_4 &\le 30 \\ 0.25x_1 + 0.05x_2 + 0.00x_3 + 0.10x_4 &\le 5 \end{aligned} \tag{9.6}$$

Nonnegativity constraints:

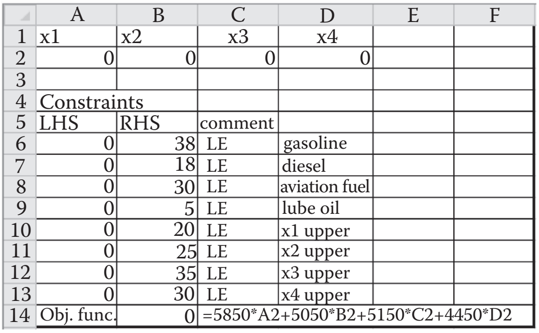
$$x_1 \ge 0 \; x_2 \ge 0 \; x_3 \ge 0 \; x_4 \ge 0 \tag{9.7}$$



Product	Processing cost $/barrel	Selling price $/barrel	Minimum daily demand 1000 Bb1/day
Gasoline	52	56	38
Diesel	45	49	18
Aviation fuel	55	62	30
Lube oil	60	70	5

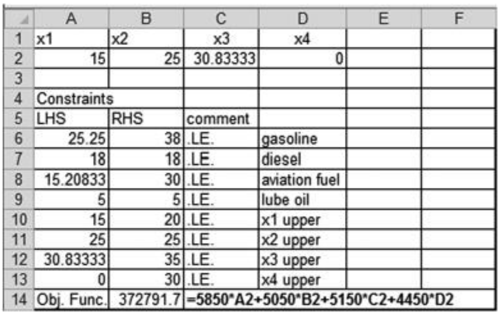
FIGURE 9.3 Financial and daily demand data for products.

The following spreadsheet shows the setup for this LP problem. Note that the “initial guess” for all variables is zero.



	A	B	C	D	E	F
1	x1	x2	x3	x4		
2	0	0	0	0		
3						
4	Constraints					
5	LHS	RHS	comment			
6	0	38	LE	gasoline		
7	0	18	LE	diesel		
8	0	30	LE	aviation fuel		
9	0	5	LE	lube oil		
10	0	20	LE	x1 upper		
11	0	25	LE	x2 upper		
12	0	35	LE	x3 upper		
13	0	30	LE	x4 upper		
14	Obj. func.	0	=5850*A2+5050*B2+5150*C2+4450*D2			

When Solver is invoked, the spreadsheet changes to the following:



	A	B	C	D	E	F
1	x1	x2	x3	x4		
2	15	25	30.83333	0		
3						
4	Constraints					
5	LHS	RHS	comment			
6	25.25	38	.LE.	gasoline		
7	18	18	.LE.	diesel		
8	15.20833	30	.LE.	aviation fuel		
9	5	5	.LE.	lube oil		
10	15	20	.LE.	x1 upper		
11	25	25	.LE.	x2 upper		
12	30.83333	35	.LE.	x3 upper		
13	0	30	.LE.	x4 upper		
14	Obj. Func.	372791.7	=5850*A2+5050*B2+5150*C2+4450*D2			

Therefore, it is optimal to process 15,000 Bbl/day of crude 1, 25,000 Bbl/day of crude 2, 30,833 Bbl/day of crude 3, and *none* of crude 4. In typical refinery operations, the constraining data change often, so when the LP is run on another day, the optimal values are apt to change.

9.3 NONLINEAR PROGRAMMING

Nonlinear programming (NLP), as the name implies, is similar to LP, but the objective function or constraints can be nonlinear functions. There are no algorithms (like the simplex method) that guarantee a solution for NLP problems. Many methods have been developed, and Solver has one of these built in (called Generalized Reduced Gradient). The subject of NLP is quite complex and far beyond what can be covered here. NLP is introduced by way of a simple example. Even the simplest of chemical and biomolecular engineering NLP problems can be too complex to warrant coverage here.

Example 9.3: An NLP Problem

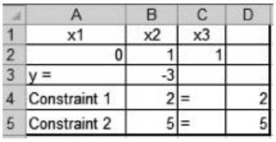
Consider the following NLP:

$$\text{Minimize } y = 2x_1^2 + 2x_2^2 + x_3^2 - 2x_1x_2 - 4x_1 - 6x_2 \tag{9.8}$$

subject to

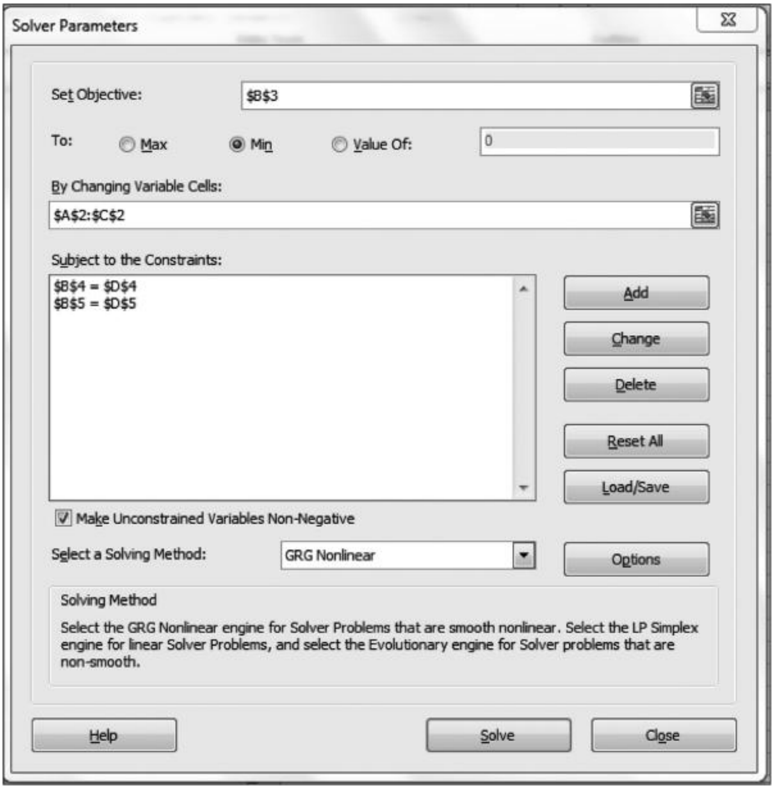
$$
\begin{aligned}
&x_1 + x_2 + x_3 = 2\\
&x_1^2 + 5x_2 = 5\\
&x_i \geq 0, \quad i = 1,2,3
\end{aligned}
\tag{9.9}
$$

Clearly, the objective function and the second (equality) constraint are nonlinear since they involve quadratic terms. For such problems, a good initial guess is often necessary if Solver is to find a solution. It is sometimes necessary to try several initial guesses before a proper solution can be found. In this case, the initial guess of [0 1 1] was tried. The following spreadsheet shows a setup for this problem:



	A	B	C	D
1	x1	x2	x3	
2	0	1	1	
3	y =	-3		
4	Constraint 1	2	=	2
5	Constraint 2	5	=	5

The associated Solver window is as shown below–note that the Select a Solving Method displays GRG Nonlinear:



Solver Parameters

Set Objective: $B$3

To: ○ Max ● Min ○ Value Of: 0

By Changing Variable Cells:

$A$2:$C$2

Subject to the Constraints:

$B$4 = $D$4
$B$5 = $D$5

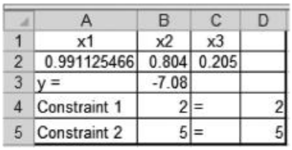
☑ Make Unconstrained Variables Non-Negative

Select a Solving Method: GRG Nonlinear

Solving Method

Select the GRG Nonlinear engine for Solver Problems that are smooth nonlinear. Select the LP Simplex engine for linear Solver Problems, and select the Evolutionary engine for Solver problems that are non-smooth.

After selecting Solve, the spreadsheet then appears as follows:



	A	B	C	D
1	x1	x2	x3	
2	0.991125466	0.804	0.205	
3	y =	-7.08		
4	Constraint 1	2	=	2
5	Constraint 2	5	=	5

The solution shown may or may not be the global optimum. Also, there could be more than one solution (even a simple quadratic equation usually has two solutions). The only way to discover this is to try several other initial guesses. This is left as an additional exercise.

9.4 NONLINEAR EQUATIONS

The same algorithms that solve NLP problems can be applied to solving sets of nonlinear equation problems (NEPs). Material and energy balances applied to chemical and biomolecular engineering problems often lead to NEPs. The following example is typical of such problems.

Example 9.4: Continuous Stirred Tank Reactor (CSTR)

Consider a CSTR as depicted in Figure 9.4.

Q is the volumetric flowrate (L/s), V is the reactor volume (L), and C_i is the concentration of each of the four components (gmol/L).

Also consider the following hypothetical reactions taking place in the CSTR:

$$\begin{gathered} A \xrightarrow{r_1} 2B \\ A \underset{r_3}{\overset{r_2}{\rightleftarrows}} C \\ B \xrightarrow{r_4} D + C \end{gathered} \tag{9.10}$$

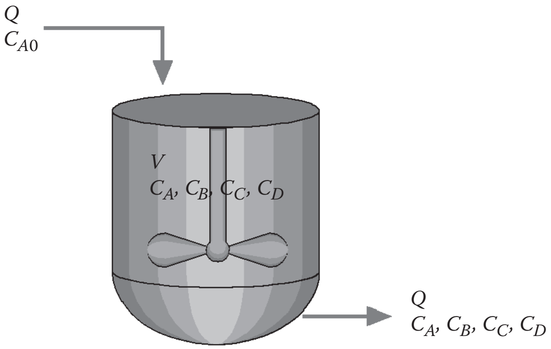


FIGURE 9.4 Continuous stirred tank reactor.

where

$$
\begin{aligned}
r_1 &= k_1 C_A \\
r_2 &= k_2 C_A^{3/2} \\
r_3 &= k_3 C_C^2 \\
r_4 &= k_4 C_B^2
\end{aligned}
\tag{9.11}
$$

k_1, k_2, k_3, and k_4 are rate constants with the proper units. Typical values are as follows:

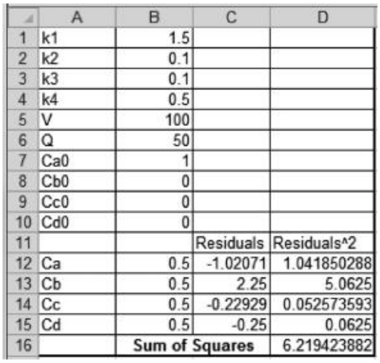
$$
\begin{aligned}
k_1 &= 1.5\ \mathrm{s}^{-1} \\
k_2 &= 0.1\ \mathrm{L}^{1/2}/\mathrm{gmol}^{1/2} - \mathrm{s} \\
k_3 &= 0.1\ \mathrm{L/gmol} - \mathrm{s} \\
k_4 &= 0.5\ \mathrm{L/gmol} - \mathrm{s}
\end{aligned}
\tag{9.12}
$$

The r_i have units of gmol/L-s.

A mass balance on each of the four components leads to the following set of nonlinear equations:

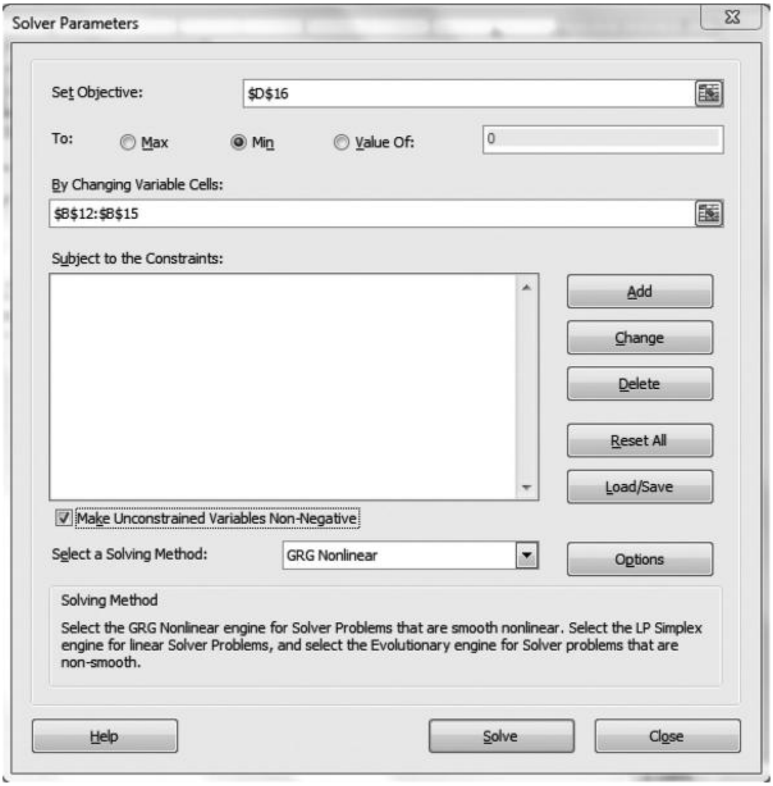
$$
\begin{aligned}
C_A &= C_{A0} + V\left(-k_1 C_A - k_2 C_A^{3/2} + k_3 C_C^2\right)/Q \\
C_B &= C_{B0} + V\left(2k_1 C_A - k_4 C_B^2\right)/Q \\
C_C &= C_{C0} + V\left(k_2 C_A^{3/2} - k_3 C_C^2 + k_4 C_B^2\right)/Q \\
C_D &= C_{D0} + V\left(k_4 C_B^2\right)/Q
\end{aligned}
\tag{9.13}
$$

There are two ways in which to use Solver for nonlinear equations. The *direct* way is to set up the nonlinear equations as *constraints* with *no objective function*. The other way is to set up the spreadsheet to compute the *sum of squares of residuals* and use Solver to minimize this (without any constraints). The latter method is used in the following spreadsheet, where the feed consists only of component A with C_{A0} = 1. The volumetric flow rate is 50 gmol/s, and the reactor volume is 100 L/s. The equations are rearranged in the form $f(x) = 0$ so that the left-hand sides are *residuals* whose value at a solution is zero (within tolerance). The initial guess for all concentrations is 0.5 gmol/L.



	A	B	C	D
1	k1	1.5		
2	k2	0.1		
3	k3	0.1		
4	k4	0.5		
5	V	100		
6	Q	50		
7	Ca0	1		
8	Cb0	0		
9	Cc0	0		
10	Cd0	0		
11			Residuals	Residuals^2
12	Ca	0.5	-1.02071	1.041850288
13	Cb	0.5	2.25	5.0625
14	Cc	0.5	-0.22929	0.052573593
15	Cd	0.5	-0.25	0.0625
16		**Sum of Squares**		6.219423882

The Solver setup is as follows:



Solver Parameters

Set Objective: $D$16

To: ○ Max ● Min ○ Value Of: 0

By Changing Variable Cells: $B$12:$B$15

Subject to the Constraints:

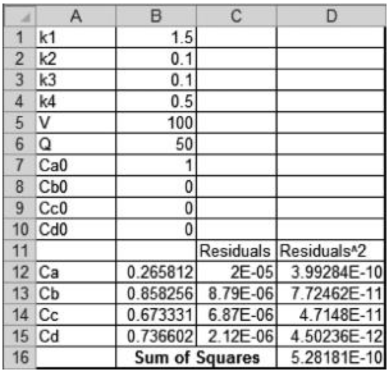
☑ Make Unconstrained Variables Non-Negative

Select a Solving Method: GRG Nonlinear

Solving Method

Select the GRG Nonlinear engine for Solver Problems that are smooth nonlinear. Select the LP Simplex engine for linear Solver Problems, and select the Evolutionary engine for Solver problems that are non-smooth.

When Solver is invoked, the spreadsheet changes as shown below:



	A	B	C	D
1	k1	1.5		
2	k2	0.1		
3	k3	0.1		
4	k4	0.5		
5	V	100		
6	Q	50		
7	Ca0	1		
8	Cb0	0		
9	Cc0	0		
10	Cd0	0		
11			Residuals	Residuals^2
12	Ca	0.265812	2E-05	3.99284E-10
13	Cb	0.858256	8.79E-06	7.72462E-11
14	Cc	0.673331	6.87E-06	4.7148E-11
15	Cd	0.736602	2.12E-06	4.50236E-12
16		Sum of Squares		5.28181E-10

As with all nonlinear problems, it is always good practice to try several initial guesses to see if the same solution results. This is left as an exercise.

9.5 NONLINEAR REGRESSION ANALYSIS

Recall Equations 7.9 and 7.10. *G* is a matrix of constants for linear regression. For nonlinear regression (NLR), *G* is a function of the unknown parameters, and Equation 7.10 becomes a set of nonlinear equations; therefore, there are two possible approaches to solving NLR problems. One method involves treating Equation 7.10 as a set of constraints (with no objective function), and the other is to minimize the sum of squares of residuals (no constraints). The latter approach (to minimize the sum of squares of residuals) is very much the more straightforward of the two approaches.

A common NLR problem in chemical and biomolecular engineering involves finding model coefficients (parameters) for models in which the parameters occur nonlinearly. A typical problem is solved in Example 9.5.

Example 9.5: NLR in Reaction Kinetics

Consider the simple decomposition reaction of compounds *A* to *B*:

$$A \rightarrow B \tag{9.14}$$

Assuming an elementary reaction, the rate of disappearance of *A* is given by

$$\frac{dC_A}{dt} = -kC_A \tag{9.15}$$

where C_A is the molar concentration of *A*. Assuming an initial condition of

$$C_A(0) = 1 \text{ mol/L} \tag{9.16}$$

the solution to the separable differential Equation 9.15 can be obtained as follows:

1. Rearrange the differential equation to the form

$$\frac{dC_A}{C_A} = -kdt \tag{9.17}$$

2. Integrate both sides

$$\ln(C_A) = -kt + \text{constant}$$

3. From the initial condition,

$$\ln(1) = 0 = \text{constant}$$

4. So, finally, the solution is

$$C_A = e^{-kt} \tag{9.18}$$

Further assume that the rate constant, *k*, is a function of temperature according to the Arrhenius form:

$$k = c_1 e^{-c_2/T} \tag{9.19}$$

where *T* is the absolute temperature. The overall mathematical model for this system then becomes

$$C_A = \exp(-c_1 t)\exp(-c_2/T) \tag{9.20}$$

So, C_A is the dependent variable, *t* and *T* are the two independent variables, and c_1 and c_2 are two parameters to be determined. A typical set of data appears in Table 9.1.

To show clearly that this is an NLR problem, the derivatives of the dependent variable with respect to the unknown parameters (Equation 7.9) lead to

$$\begin{aligned} Z_1 &= -t\exp(-c_2/T)\exp(-c_1 t)\exp(-c_2/T) \\ Z_2 &= -\frac{c_1 t}{T}\exp(-c_1 t)\exp(-c_2/T) \end{aligned} \tag{9.21}$$

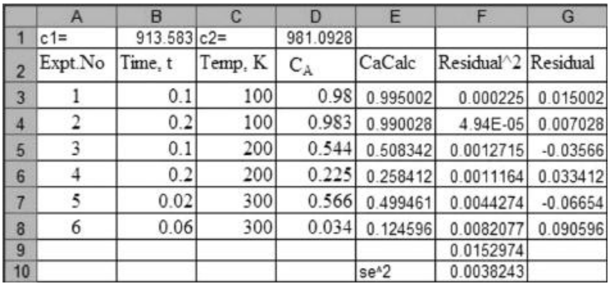
Clearly, the matrix *G* of Equation 7.10 involving the sum of products of the *Z*'s is dependent on the unknown coefficients c_1 and c_2, and the equations are *nonlinear*. Because the analytical derivatives for NLR problems are often complex (such as those of Equation 9.21), it is often simpler to determine these derivatives numerically.

To solve the nonlinear equations associated with NLR, Solver can be used. However, in doing so, the statistical nature of the regression problem is ignored. It is necessary to calculate the *G* matrix *at the solution*. Once the parameters are known, the *G* matrix again becomes one of constants and can be inverted. Once the matrix *G* is known, all of the statistical aspects of the problem (Chapter 7) can be computed.

TABLE 9.1
Data for Kinetics NLR

Expt. No	Time, s	Temp, K	C_A
1	0.1	100	0.98
2	0.2	100	0.983
3	0.1	200	0.544
4	0.5	200	0.225
5	0.02	300	0.566
6	0.06	300	0.034

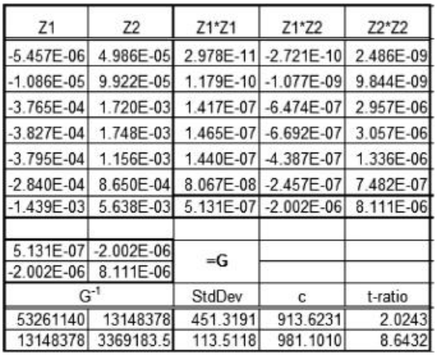
Shown below is a spreadsheet for solving this problem using Solver:



	A	B	C	D	E	F	G
1	c1=	913.583	c2=	981.0928			
2	Expt.No	Time, t	Temp, K	C_A	CaCalc	Residual^2	Residual
3	1	0.1	100	0.98	0.995002	0.000225	0.015002
4	2	0.2	100	0.983	0.990028	4.94E-05	0.007028
5	3	0.1	200	0.544	0.508342	0.0012715	-0.03566
6	4	0.2	200	0.225	0.258412	0.0011164	0.033412
7	5	0.02	300	0.566	0.499461	0.0044274	-0.06654
8	6	0.06	300	0.034	0.124596	0.0082077	0.090596
9						0.0152974	
10					se^2	0.0038243	

S_e^2 was minimized using Solver. The initial guesses for c_1 and c_2 were both 1000.

The following spreadsheet segment shows the *Z*s calculated using finite differences (second-order correct), the requisite *Z* products, and the sum of *Z*s required for Equation 9.10. Also shown are G^{-1}, the parameter standard deviations, the optimal c values, and the t-ratios.



Z1	Z2	Z1*Z1	Z1*Z2	Z2*Z2
-5.457E-06	4.986E-05	2.978E-11	-2.721E-10	2.486E-09
-1.086E-05	9.922E-05	1.179E-10	-1.077E-09	9.844E-09
-3.765E-04	1.720E-03	1.417E-07	-6.474E-07	2.957E-06
-3.827E-04	1.748E-03	1.465E-07	-6.692E-07	3.057E-06
-3.795E-04	1.156E-03	1.440E-07	-4.387E-07	1.336E-06
-2.840E-04	8.650E-04	8.067E-08	-2.457E-07	7.482E-07
-1.439E-03	5.638E-03	5.131E-07	-2.002E-06	8.111E-06
5.131E-07	-2.002E-06	=G		
-2.002E-06	8.111E-06			
G^{-1}		StdDev	c	t-ratio
53261140	13148378	451.3191	913.6231	2.0243
13148378	3369183.5	113.5118	981.1010	8.6432

It can be seen from the t-ratios that c_1 is "borderline" well determined, while c_2 is more well determined. Since these results are based on very few data points, it is likely that the parameter behavior would improve with a much larger data set.

EXERCISES

Exercise 9.1: Linear programming.

It is required to produce one pound of an alloy that has at least 30% Pb and at least 30% Zn by mixing a number of available Pb–Zn–Sn alloys. Find the cheapest blend using the following data:

	Analysis (%)			
Available Alloy	**Pb**	**Zn**	**Sn**	**Cost ($/lb)**
1	20	20	60	6.0
2	10	40	50	6.3
3	40	50	10	7.5
4	50	30	20	8.0

Use Solver for this LP problem.

Exercise 9.2: Nonlinear regression.

A heterogeneous reaction is known to occur at a rate described by the following Langmuir–Hinshelwood expression:

$$r = \frac{k_1 P_A}{(1 + K_A P_A + K_R P_R)^2} \tag{9.22}$$

From initial rate measurements, k_1 has been determined as 0.015 gmol/s-gcat-atm at 400 K. Using the following rate data at 400 K, estimate the values of K_A and K_R:

P_A (atm)	1	0.9	0.8	0.7	0.6
P_R (atm)	0	0.1	0.2	0.3	0.4
r	3.4e-5	3.6e-5	3.7e-5	3.9e-5	4.0e-5

This is an NLR problem. Solve this by minimizing the sum of squares of residuals using Solver. After you have determined the optimal values for K_A and K_R, calculate numerically (using second-order correct formulas) the derivatives Z_1 and Z_2 at each data point, form the G matrix, calculate the parameter standard deviations, and calculate the t-ratios for each parameter.

Exercise 9.3: Nonlinear programming.

Consider the following NLP:

$$\text{minimize } x_2^2 - x_1^2$$

$$\text{subject to } x_1^2 + x_2^2 = 4$$

First, solve this problem *analytically* by solving the constraint for x_1^2 and substituting this into the objective function. Then differentiate the objective function (the only remaining variable is x_2), set the derivative to zero, and find x_2. Use the value for x_2 to find the value(s) for x_1. Next use Solver to find the solution(s). Use a starting point of [1, 1] and then [–1, –1] and see what solutions Solver finds from these starting points.

Exercise 9.4: Nonlinear equations.

The calculation of the equilibrium concentration when we have several reactions and components usually results in nonlinear algebraic equations. Consider the following three reactions involving seven components:

$$\begin{aligned}
&A + B \overset{x_1}{\Leftrightarrow} C + D \qquad && K_1 = \frac{C_C C_D}{C_A C_B} \\
&B + C \overset{x_2}{\Leftrightarrow} E + F \qquad && K_2 = \frac{C_E C_F}{C_B C_C} \\
&A + E \overset{x_3}{\Leftrightarrow} G \qquad && K_3 = \frac{C_G}{C_A C_E}
\end{aligned} \tag{9.23}$$

$$K_1 = 1, K_2 = 2, K_3 = 4$$

Here, x_1, x_2, and x_3 (the unknowns) are the *extents of reaction* at equilibrium, and the Cs are molar concentrations. Note that the extent of reaction is a number between 0 and 1. Zero indicates no production of products, while a value of 1 means that the reaction goes to completion (no reactants remain). Given the extents of reaction, the following mass balances can be written:

$$\begin{aligned}
C_A &= C_{A0} - x_1 C_{B0} - x_3 C_{A0} \\
C_B &= C_{B0} - x_1 C_{B0} - x_2 C_{B0} \\
C_C &= C_{C0} + x_1 C_{B0} - x_2 C_{B0} \\
C_D &= C_{D0} + x_1 C_{B0} \\
C_E &= C_{E0} + x_2 C_{B0} - x_3 C_{A0} \\
C_F &= C_{F0} + x_2 C_{B0} \\
C_G &= C_{G0} + x_3 C_{A0}
\end{aligned} \tag{9.24}$$

Initial conditions are $C_{A0} = C_{B0} = 1$; all others are zero. The nonlinear equations 9.23 can be expressed as

$$\begin{aligned}
C_C * C_D - K_1 * C_A * C_B &= 0 \\
C_E * C_F - K_2 * C_B * C_C &= 0 \\
C_G - K_3 * C_A * C_E &= 0
\end{aligned} \tag{9.25}$$

When formulating an initial guess, make note of the following: the optimal values for the variables *must be between 0 and 1.* Use Solver for this problem.

Exercise 9.5: Solve the following linear program:

$$\begin{aligned}
&\text{maximize } q = 80x_1 + 100x_2 \\
&\text{subject to} \\
&0.5x_1 + 0.5x_2 \le 25 \\
&0.2x_1 + 0.6x_2 \le 10 \\
&0.8x_1 + 0.4x_2 \le 14 \\
&x_1, x_2 \ge 0
\end{aligned}$$

Exercise 9.6: Solve the following nonlinear program:

$$\text{minimize } q = (x_1 - 0.5)^2 + (x_2 - 2.5)^2$$

$$\text{subject to}$$

$$(x_1 - 2)^2 + x_2^2 \le 4$$

$$x_1 \ge 0$$

$$0 \le x_2 \le 2$$

Exercise 9.7: Solve the following nonlinear algebraic equations:

$$(x_1 - 1)^3 + x_2^2 = 0$$

$$x_1 + x_2 = 1$$

Try to find more than one solution.

Exercise 9.8: Solve the following NLR problem:

Fit the function $y = c_1\exp(c_2/T)$ to the data shown below. After having determined the optimal values for c_1 and c_2, calculate analytically or numerically (using second order correct formulas) the derivatives Z_1 and Z_2 at each data point, form the G matrix, calculate the parameter standard deviations and calculate the t-ratios for each parameter. Comment on the significance of the two parameters c_1 and c_2.

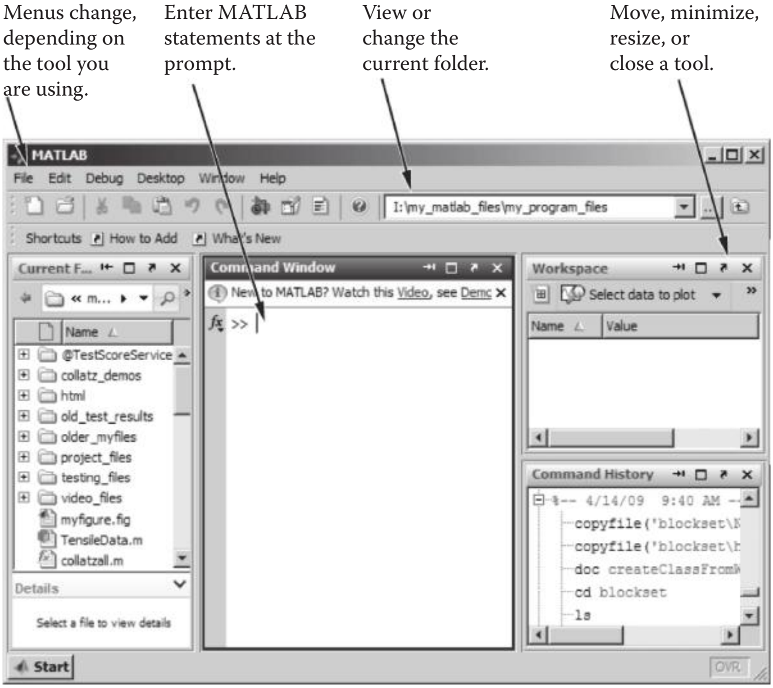
T	*y* Data
100	0.63
110	0.60
120	0.57
130	0.53
140	0.51

10 Introduction to MATLAB®

10.1 INTRODUCTION

The name MATLAB® stands for Matrix Laboratory and was first published before graphical user interfaces were popular. It has evolved through many versions and is usually updated every 6 months or so. MATLAB is a popular computing environment in universities and research institutions. It is not, however, used often in industrial settings because of somewhat expensive licensing fees. Inasmuch as many students of chemical and biomolecular engineering undertake postgraduate or professional studies where MATLAB can be popular, this programming environment is introduced here with the assumption that the reader is familiar with Excel® and VBA. This brief introduction is intended only to present the rudiments of MATLAB. Full documentation is available at http://www.mathworks.com/help/techdoc/learn_matlab/bqr_2pl.html. Several of the instructions in this chapter are taken from this reference.

When MATLAB starts up (in either Windows or Macintosh environments), the MATLAB Command Window and subsidiary windows appear as follows:



Menus change, depending on the tool you are using.

Enter MATLAB statements at the prompt.

View or change the current folder.

Move, minimize, resize, or close a tool.

```
%-- 4/14/09  9:40 AM --
copyfile('blockset\K
copyfile('blockset\h
doc createClassFromW
cd blockset
ls
```

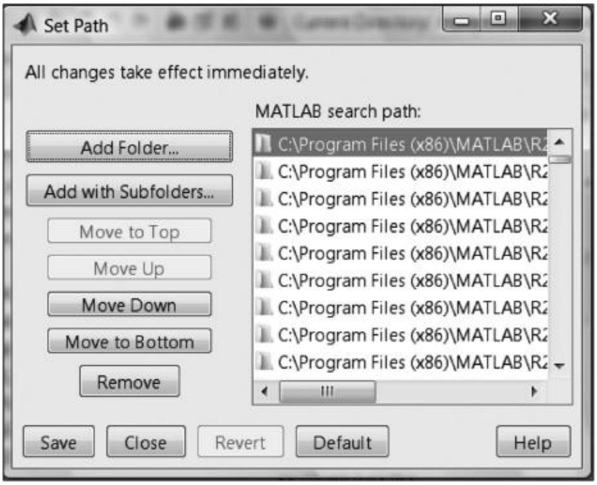
The >> icon is the command prompt. Anything entered after this is a MATLAB command. The Current Directory window displays folders and files associated with the current directory (see Path discussion below). The Workspace Window shows the names of all variables that have been created, and the Command History shows a record of the most recent commands given at the Command Prompt.

The actual arrangement of the sub-windows might be different than shown. The Command Window is somewhat akin to the Excel Spreadsheet; it is through this window that MATLAB commands are given. While many useful things can be accomplished directly in the Command Window, for present purposes, it acts as the interface to MATLAB's programming language. Most input and output are accomplished through the Command Window. For Excel/VBA users, it is convenient to think of the Command Window as comparable to the spreadsheet and MATLAB programs as similar to VBA macros. This is not a perfect metaphor but is sufficient for present purposes.

If a previously issued command is needed again, the Command History can be recalled using the ↑ key. The last command issued is shown first. Each time ↑ is pressed, the previous commands appear in reverse order of having been typed.

10.2 MATLAB BASICS

Perhaps the first thing to do when first using MATLAB is to set the *Path*. The Path is a list of directories that MATLAB searches for files. The default Path is where users usually want to store files and recover them later. A usual place for file storage might be on a thumb drive. Assume that the directory of interest is `F:\MyDocuments\MATLAB`. To put this directory into the Path and to make it the default directory, go to the File menu and click on Set Path—the following window appears:



All changes take effect immediately.

MATLAB search path:

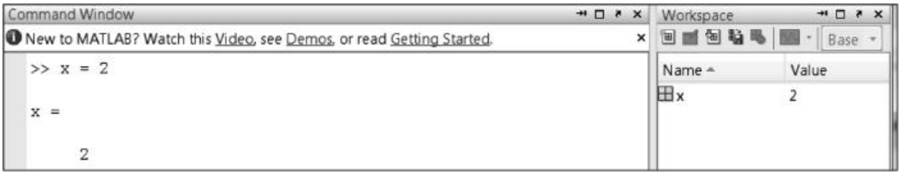
C:\Program Files (x86)\MATLAB\R
C:\Program Files (x86)\MATLAB\R
C:\Program Files (x86)\MATLAB\R
C:\Program Files (x86)\MATLAB\R
C:\Program Files (x86)\MATLAB\R
C:\Program Files (x86)\MATLAB\R
C:\Program Files (x86)\MATLAB\R
C:\Program Files (x86)\MATLAB\R
C:\Program Files (x86)\MATLAB\R

Click on `Add Folder`, `Save`, and then `Close`. Next, at the Command prompt, issue the following command (`cd` stands for change directory).

```
>> cd 'F:\My Documents\MATLAB'
```

From this point forward, any file that is saved is deposited in the selected directory, and when opening a file, this directory will be searched first.

A unique thing about MATLAB is that all variables are *matrices*. For example, the command shown in the Command Window below creates a variable named x (see that name having been added to the Workspace). Following the command, the current value of x is listed. To avoid having the value of x printed following the command, simply add a semicolon at the end of the command. It is important to note that all MATLAB identifiers are *case sensitive*.


Command Window
New to MATLAB? Watch this Video, see Demos, or read Getting Started.
Workspace
Base

```
>> x = 2

x =

     2
```

Name	Value
x	2

In the interest of keeping this discussion brief, emphasis is given to the *differences* between Excel/VBA and MATLAB. When the MATLAB syntax is the same as that of Excel/VBA, no explanation is given. The useful command

```
>> clc
```

clears the Command Window. It is a good idea to always begin with a blank window.

To get the feel of MATLAB, some annotated Command Window sessions are now shown.

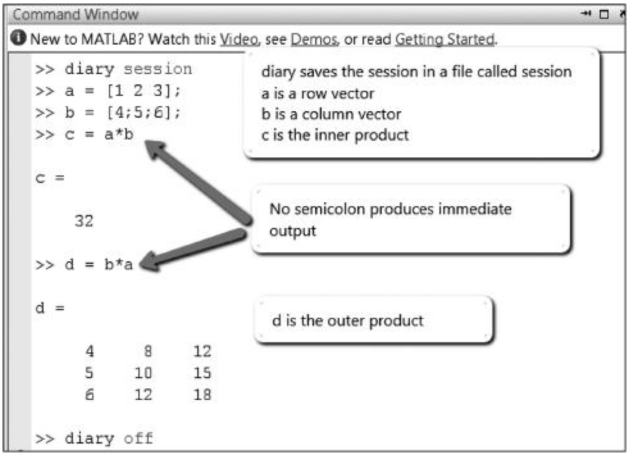
In the following example, the command diary is used. This command has the following syntax:

diary filename

where filename is any legal MATLAB file name. Everything that appears in the Command Window following this command is recorded in the file. To terminate recording, the command

```
diary off
```

is entered


Command Window
New to MATLAB? Watch this Video, see Demos, or read Getting Started.

```
>> diary session
>> a = [1 2 3];
>> b = [4;5;6];
>> c = a*b

c =

    32

>> d = b*a

d =

     4     8    12
     5    10    15
     6    12    18

>> diary off
```

diary saves the session in a file called session
a is a row vector
b is a column vector
c is the inner product

No semicolon produces immediate output

d is the outer product

Special attention should be given to the command

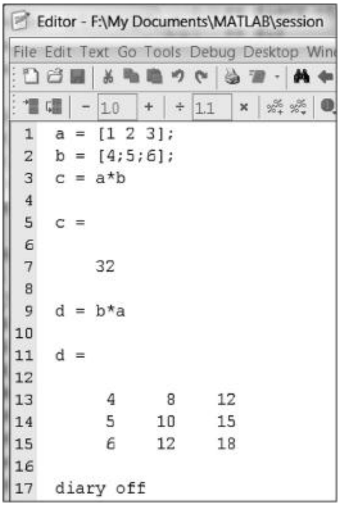
```
b = [4;5;6];
```

The interior semicolons indicate “start of a new row.” So, b is a column vector while a is a row vector.

Issuing the command

```
Dir
```

produces a list of the files in the current directory. Among those will be a file called session. (Note that the name session is arbitrary—any legal file name can be used.) This is a text file that can be opened within MATLAB or by any word processor. To see the file in MATLAB, go to File/Open. At the bottom of the window that appears, change the File of Type to All Files (the default is to show only MATLAB type files). Then click on the file name session and then Open. The following is displayed:


Editor - F:\My Documents\MATLAB\session
File Edit Text Go Tools Debug Desktop Win
- 1.0 + ÷ 1.1 ×

```
a = [1 2 3];
b = [4;5;6];
c = a*b

c =

    32

d = b*a

d =

     4     8    12
     5    10    15
     6    12    18

diary off
```

This is an exact duplicate of the Command Window session and includes everything from the diary session command until diary off. The contents of the file are displayed in the Editor window. This window is similar to the VBA editor window in that this is where MATLAB programs are coded.

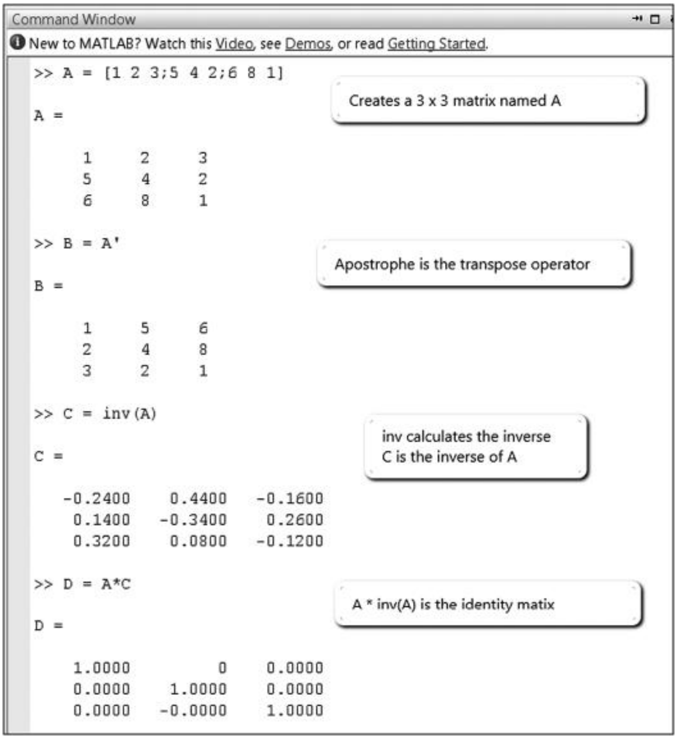
If a command (at the Command Window or in a MATLAB program) is very long, a continuation indicator is three (or more) consecutive periods ....

Table 10.1 enumerates the MATLAB operators. Most of these are the same as for Excel and VBA. A notable difference is the backslash operator \ (called left division), which is used primarily when solving sets of linear algebraic equations. Examples using this operator appear in the sequel.

The following MATLAB session shows a variety of matrix operations. It can be seen again that the semicolon is used to suppress printing when placed at the end of a command, and it also indicates the end of a row (and the beginning of a new one) in a matrix. The apostrophe is the transposition operator, and the built-in function inv takes the inverse of a matrix. If the inverse does not exist, an error message appears.

TABLE 10.1
MATLAB Operators

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
\	Left division
^	Power
`	Transpose
( )	Specify evaluation order
=	Assignment
>	Greater than
<	Less than
> =	Greater than or equal to
< =	Less than or equal to
= =	Equal to (logical)
~ =	Not equal to
&	Logical and
|	Logical or
~	Logical not


Command Window
New to MATLAB? Watch this Video, see Demos, or read Getting Started.

```
>> A = [1 2 3;5 4 2;6 8 1]

A =

     1     2     3
     5     4     2
     6     8     1

>> B = A'

B =

     1     5     6
     2     4     8
     3     2     1

>> C = inv(A)

C =

   -0.2400    0.4400   -0.1600
    0.1400   -0.3400    0.2600
    0.3200    0.0800   -0.1200

>> D = A*C

D =

    1.0000         0    0.0000
    0.0000    1.0000    0.0000
    0.0000   -0.0000    1.0000
```

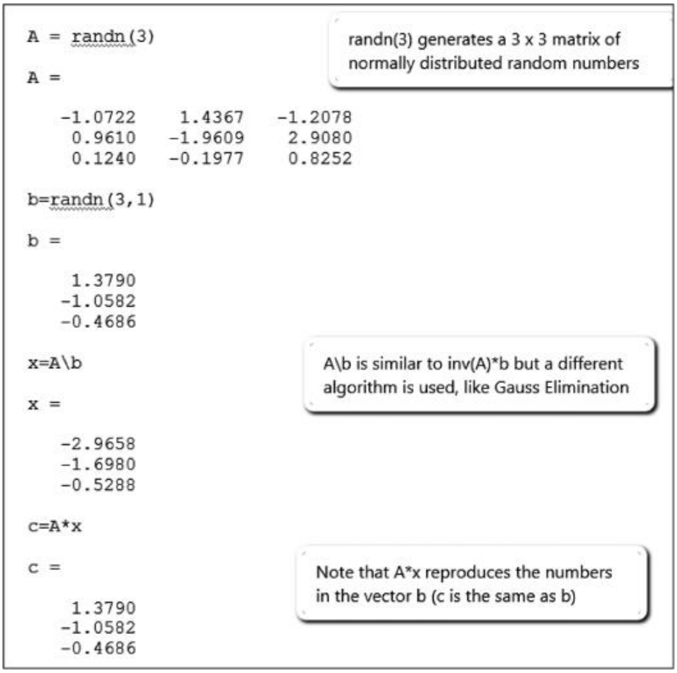
Creates a 3 x 3 matrix named A

Apostrophe is the transpose operator

inv calculates the inverse
C is the inverse of A

A * inv(A) is the identity matix

The following is a listing of a diary file where the randn function and the backslash operator are used:



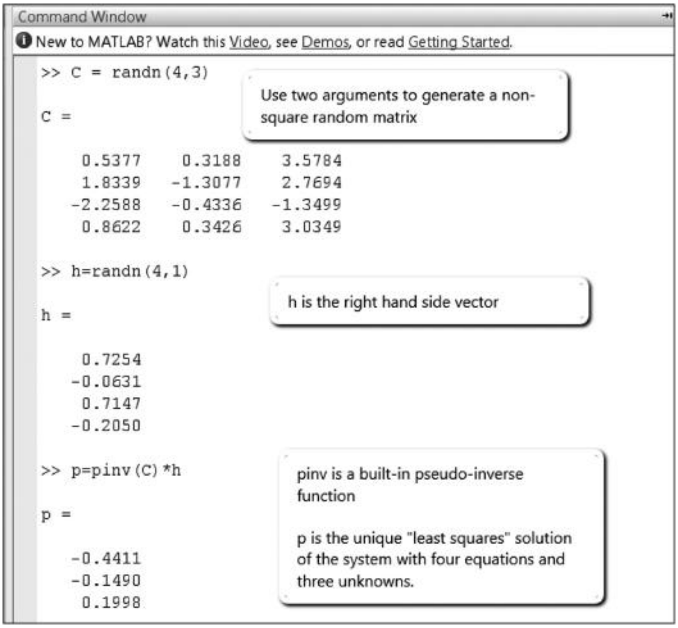
```
A = randn(3)

A =

   -1.0722    1.4367   -1.2078
    0.9610   -1.9609    2.9080
    0.1240   -0.1977    0.8252

b=randn(3,1)

b =

    1.3790
   -1.0582
   -0.4686

x=A\b

x =

   -2.9658
   -1.6980
   -0.5288

c=A*x

c =

    1.3790
   -1.0582
   -0.4686
```

randn(3) generates a 3 x 3 matrix of normally distributed random numbers

A\b is similar to inv(A)*b but a different algorithm is used, like Gauss Elimination

Note that A*x reproduces the numbers in the vector b (c is the same as b)

The next MATLAB session shows the use of the pinv (pseudo-inverse function).


Command Window
New to MATLAB? Watch this Video, see Demos, or read Getting Started.

```
>> C = randn(4,3)

C =

    0.5377    0.3188    3.5784
    1.8339   -1.3077    2.7694
   -2.2588   -0.4336   -1.3499
    0.8622    0.3426    3.0349

>> h=randn(4,1)

h =

    0.7254
   -0.0631
    0.7147
   -0.2050

>> p=pinv(C)*h

p =

   -0.4411
   -0.1490
    0.1998
```

Use two arguments to generate a non-square random matrix

h is the right hand side vector

pinv is a built-in pseudo-inverse function

p is the unique "least squares" solution of the system with four equations and three unknowns.

Access to any element of an array is the same as in VBA. For example, using variables from the previous MATLAB session:

```
h(2) = -0.0631
p(3) = 0.1998
C(2, 4) = 0.3426
```

10.2.1 MATLAB Colon Operator

The colon (:) is an important MATLAB operator. It occurs in several different contexts. The expression

```
1:10
```

produces a row vector containing the integers from 1 to 10:

```
1 2 3 4 5 6 7 8 9 10
```

To obtain nonunit spacing, specify an increment. For example,

```
100:-7:50
```

generates

```
100 93 86 79 72 65 58 51
```

and

```
0:pi/4:pi
```

produces

```
0 0.7854 1.5708 2.3562 3.1416
```

Subscript expressions involving colons refer to portions of a matrix:

```
A(1:k,j)
```

is the first `k` elements of the `jth` column of `A`. Therefore,

```
sum(A(1:4,4))
```

computes the sum of the fourth column assuming a 4×4 matrix. However, there is another way to perform this computation. The colon by itself refers to *all* the elements in a row or column of a matrix, and the keyword `end` refers to the *last* row or column. Therefore,

```
sum(A(:,end))
```

computes the sum of the elements in the last column of `A`.

10.2.2 MATLAB, M-Files, and Input from Command Window

While a sequence of commands in the Command Window can implement many algorithms, it is awkward if selection (if-then-else) or repetition (e.g., while) logic is involved. The best way to do programming in MATLAB is to construct an M-file (comparable to a VBA Macro). These are called M-files since the automatic file type is `.m`.

Before writing a first M-file program, input/output with the command window must be covered. For MATLAB programs with small amounts of input, the `input` statement is used. For output to the Command Window, the usual method involves the `fprintf` statement. The syntax of the input statement is

```
Variable = input('prompt')
```

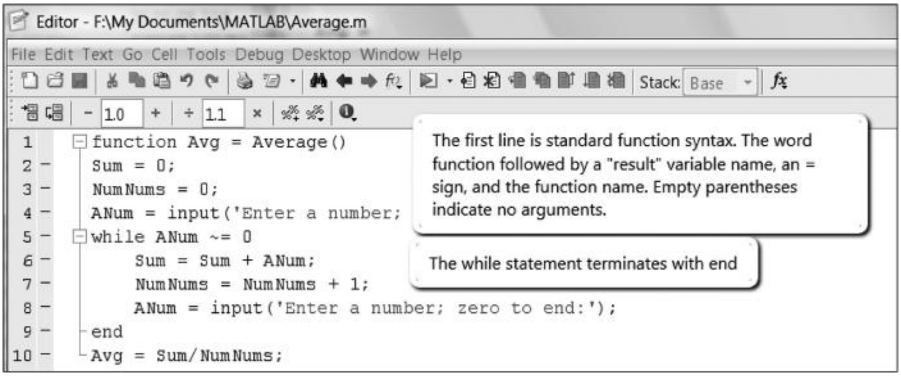
For example,

```
A = input('Enter a number:')
```

Note that strings are delimited by astrophes (recall that quote marks are used in VBA).

When executed, the prompt appears in the Command Window. A number is entered and stored in the variable `A`.

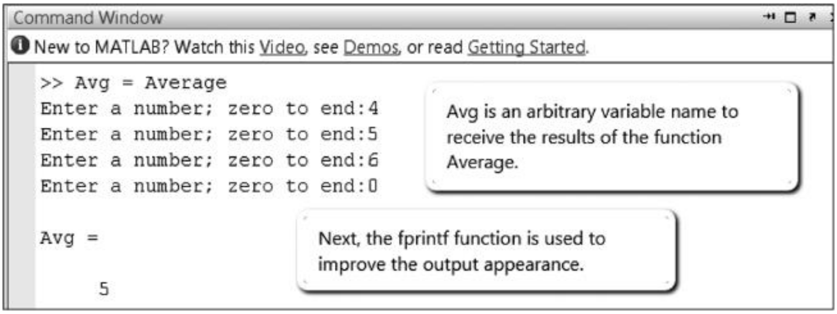
Shown next is an M-file that reads several numbers and computes the average of the numbers. After the M-file is a Command Window session that invokes the program.


Editor - F:\My Documents\MATLAB\Average.m
File Edit Text Go Cell Tools Debug Desktop Window Help
Stack: Base
- 1.0 + ÷ 1.1 ×

```
function Avg = Average()
 Sum = 0;
 NumNums = 0;
 ANum = input('Enter a number;
while ANum ~= 0
     Sum = Sum + ANum;
     NumNums = NumNums + 1;
     ANum = input('Enter a number; zero to end:');
end
Avg = Sum/NumNums;
```

The first line is standard function syntax. The word function followed by a "result" variable name, an = sign, and the function name. Empty parentheses indicate no arguments.

The while statement terminates with end


Command Window
New to MATLAB? Watch this Video, see Demos, or read Getting Started.

```
>> Avg = Average
Enter a number; zero to end:4
Enter a number; zero to end:5
Enter a number; zero to end:6
Enter a number; zero to end:0

Avg =

     5
```

Avg is an arbitrary variable name to receive the results of the function Average.

Next, the fprintf function is used to improve the output appearance.

The fprintf function is somewhat complicated since it requires a cryptic formatting string. The format of the fprintf statement is

```
fprintf (<format string>, <variables>)
```

Here is an example:

```
fprintf ('The sum is%5.3f\n', Sum)
```

If the variable sum has a value of 16.12365, the output produced by the statement is

```
The sum is 16.124
```

Note that the value of sum has been truncated to three digits after the decimal, and the last digit is rounded up. The \n at the end of the format string is the new line control character—any further output appears on a new line.

There are a large number of format string data type specifiers available for use with the fprintf function. Some of these are enumerated in Table 10.2.

Table 10.3 displays some of the available *control* characters for formatting.

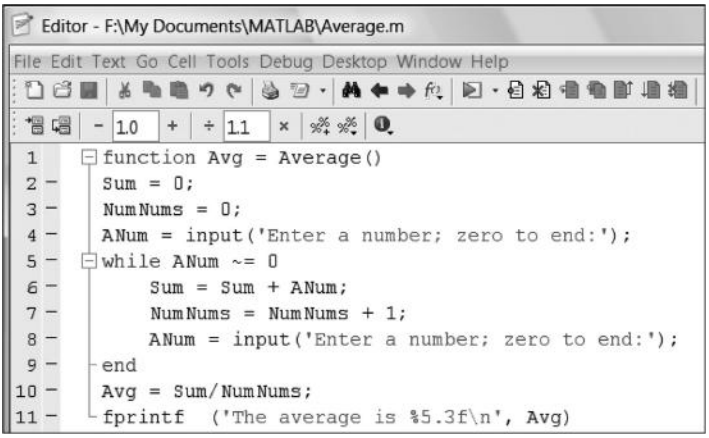
Shown next is a revised version of the Average file. In this case, output is accomplished using the fprintf function. Following the program listing is the associated Command Window where the function file is called without putting the result into a variable since the result has already been output.

TABLE 10.2
Format Data Type Specifiers

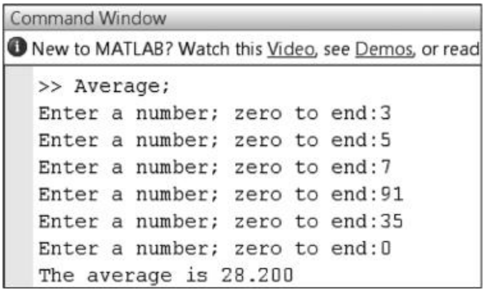
Specifier	Display
%d	Integer/whole number
%f	Floating point
%e	Exponential
%g	General (shortest format possible)
%c	Character
%s	Character string

TABLE 10.3
Format Control Characters

Control Character	Description
\n	New line
\t	Tab
''	Two apostrophes prints one apostrophe


Editor - F:\My Documents\MATLAB\Average.m
File Edit Text Go Cell Tools Debug Desktop Window Help
1.0
1.1

```
function Avg = Average()
Sum = 0;
NumNums = 0;
ANum = input('Enter a number; zero to end:');
while ANum ~= 0
    Sum = Sum + ANum;
    NumNums = NumNums + 1;
    ANum = input('Enter a number; zero to end:');
end
Avg = Sum/NumNums;
fprintf  ('The average is %5.3f\n', Avg)
```



```
>> Average;
Enter a number; zero to end:3
Enter a number; zero to end:5
Enter a number; zero to end:7
Enter a number; zero to end:91
Enter a number; zero to end:35
Enter a number; zero to end:0
The average is 28.200
```

10.3 MATLAB PROGRAMMING LANGUAGE STATEMENTS

MATLAB's programming language is similar to that of VBA. The assignment statement has already been used and is indicated by the = sign. Some of the other statements are discussed next.

10.3.1 If-Then-Else Statements

The syntax of the MATLAB If-Then-Else statement is as follows:

```
if condition1
  Statements1
else
  Statements2
end
```

Note that the words `if`, `else`, and `end` are all lowercase. `Statements1` and `Statements2` can be any other MATLAB statements. The `else` clause is optional.

10.3.2 Looping Statements (For, While)

The syntax of the `for` statement is

```
for variable = initial_value:increment/decrement:final_value
  Statements
end
```

The `increment/decrement` is optional, and if omitted, the increment is 1.

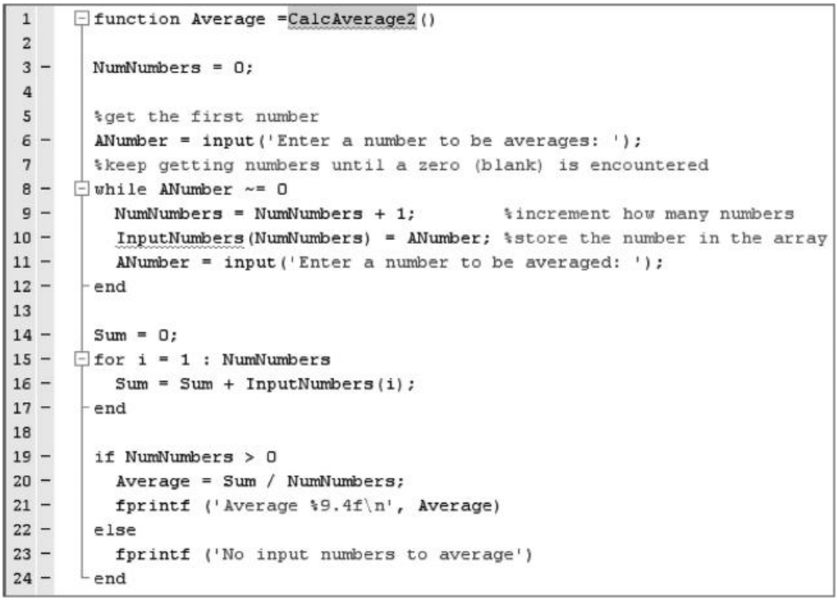
The syntax of the `while` statement is

```
while condition
  Statements
end
```

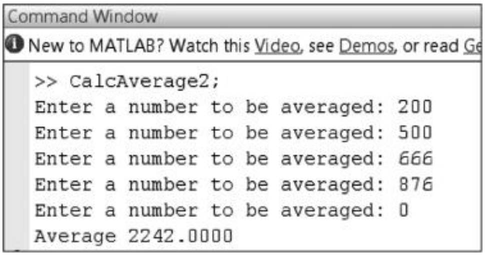
Sufficient MATLAB programming background is now available so that programs previously written in VBA can be demonstrated in MATLAB.

Example 10.1: MATLAB Program for Averaging Numbers

The MATLAB program listing below reimplements the one of Example 2.2. In that example, numbers were input from a spreadsheet and stored in an array, the average of the numbers calculated, and the average output to the spreadsheet. Note that the % sign is used to indicate a *comment*. Following the MATLAB program listing is a Command Window session that executes the program and inputs a set of numbers, and the result is output to the Command Window. There is great similarity between this program and that of Example 2.2 with minor syntax differences. The most significant difference is in the input/output portion. A direct comparison with the VBA program of Example 2.2 is advised.



```
function Average =CalcAverage2()

NumNumbers = 0;

%get the first number
ANumber = input('Enter a number to be averages: ');
%keep getting numbers until a zero (blank) is encountered
while ANumber ~= 0
  NumNumbers = NumNumbers + 1;         %increment how many numbers
  InputNumbers(NumNumbers) = ANumber; %store the number in the array
  ANumber = input('Enter a number to be averaged: ');
end

Sum = 0;
for i = 1 : NumNumbers
  Sum = Sum + InputNumbers(i);
end

if NumNumbers > 0
  Average = Sum / NumNumbers;
  fprintf ('Average %9.4f\n', Average)
else
  fprintf ('No input numbers to average')
end
```



```
>> CalcAverage2;
Enter a number to be averaged: 200
Enter a number to be averaged: 500
Enter a number to be averaged: 666
Enter a number to be averaged: 876
Enter a number to be averaged: 0
Average 2242.0000
```

10.4 MATLAB FUNCTION ARGUMENTS

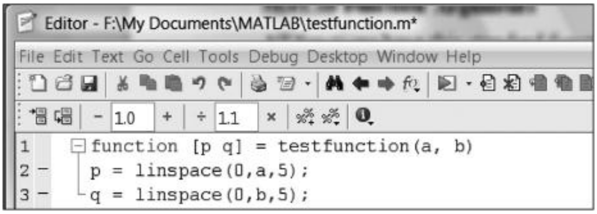
All functions have this standard *function syntax*:

```
function[output1,..., outputM] = functionName(input1,...,
inputN)
```

Input arguments are passed by *value*, while output arguments are passed by *reference*. Even if an input argument is changed by the function, the altered value is not returned after the function call. (Recall the argument passing descriptions given in Chapter 2.) Typically, output arguments are not defined at the time of calling, and the values returned are set by the function. As with VBA, the argument names used in the function definition are dummy arguments; the actual arguments are those used when the function is called.

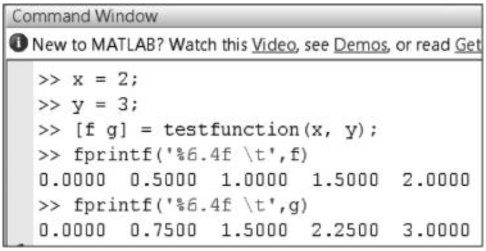
Example 10.2: Argument Passing to and from a Function

The function shown below generates two vectors using the MATLAB function `linspace`. This built-in MATLAB function can be handy for generating equally spaced data. It generates the number of vector elements given by the third argument. The numbers start with its first argument and are equally spaced up to the value of the second argument. Dummy input arguments are named `a` and `b`, while dummy output arguments are called `p` and `q`.


Editor - F:\My Documents\MATLAB\testfunction.m*
File Edit Text Go Cell Tools Debug Desktop Window Help
- 1.0 + ÷ 1.1 ×

```
function [p q] = testfunction(a, b)
p = linspace(0,a,5);
q = linspace(0,b,5);
```

Shown next is a Command Line session that sets the first two arguments (actual argument names `x` and `y`) and then calls the `testfunction`. The output arguments (actual arguments `f` and `g`) are printed using the `fprint` command.



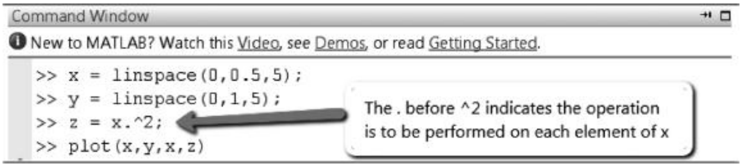
```
>> x = 2;
>> y = 3;
>> [f g] = testfunction(x, y);
>> fprintf('%6.4f \t',f)
0.0000  0.5000  1.0000  1.5000  2.0000
>> fprintf('%6.4f \t',g)
0.0000  0.7500  1.5000  2.2500  3.0000
```

10.5 PLOTTING IN MATLAB

MATLAB's plotting capabilities are such that professional quality graphs can be generated. The commands for producing graphs vary from the very simple to the quite complex. In this coverage, only relatively simple plot commands are discussed, but more complete discussions are readily available.

10.5.1 Plotting Two Functions on the Same Graph

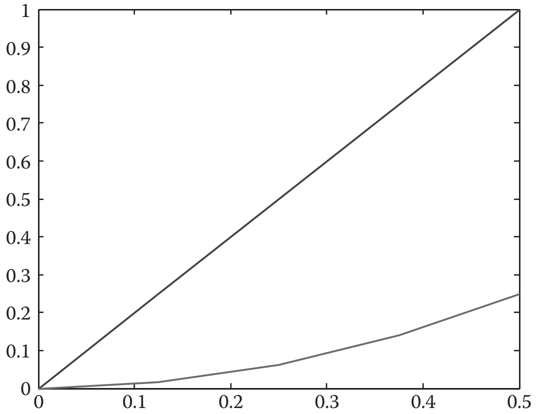
Consider the following MATLAB Command Window session:


Command Window
New to MATLAB? Watch this Video, see Demos, or read Getting Started.

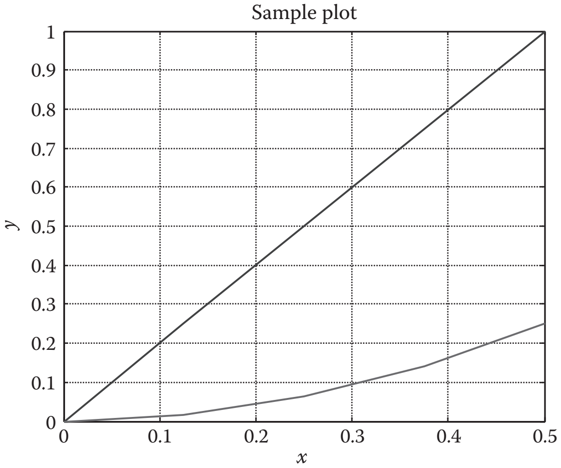
```
>> x = linspace(0,0.5,5);
>> y = linspace(0,1,5);
>> z = x.^2;
>> plot(x,y,x,z)
```

The . before ^2 indicates the operation is to be performed on each element of x

The graph produced by the plot command is shown below:



While labeling and changing the graph properties can be done under program control, it is much easier to use the plot editor. When in the plot window, go to Edit/Axis Properties. Axis labels and a plot title can be added. Also, line types and colors can be changed along with a myriad other things. Shown below is an edited plot with some of these changes:



10.6 EXAMPLE MATLAB PROGRAMS

In this section, several MATLAB M-files are presented that perform operations previously visited using VBA. These include

- Solving a single nonlinear equation using `fzero`
- Solving ordinary differential equations using `ode45`
- Solving a boundary value problem using the `ode45` and the shooting method
- Nonlinear equations using `fsolve`
- Nonlinear regression using `minsearch`

Example 10.3: Solving a Single Nonlinear Equation Using `fzero`

Consider finding a zero of the function

$$f(x) = x^4 - e^{-x} + 1 \qquad (10.1)$$

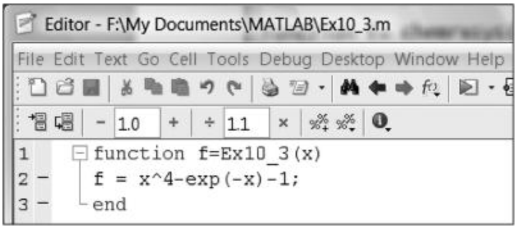
The syntax for the `fzero` function is

```
var = fzero('equation', init_guess)
```

where

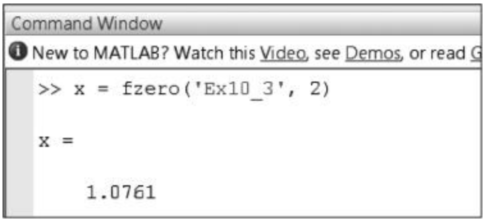
```
var          = the final value of x
equation     = name of the function representing the function to zero
init _ guess = the initial guess of x
```

A listing of the M-file for the function is as follows:


Editor - F:\My Documents\MATLAB\Ex10_3.m
File Edit Text Go Cell Tools Debug Desktop Window Help
1.0
11

```
function f=Ex10_3(x)
f = x^4-exp(-x)-1;
end
```

Next is a Command Window session that invokes `fzero` and finds a value of `x` that is a root of the function:



```
>> x = fzero('Ex10_3', 2)

x =

    1.0761
```

The function `fzero` uses a combination of the methods discussed in Chapter 1. Depending on the initial guess, it might find different roots than the one shown.

Example 10.4: Solving Ordinary Differential Equations Using `ode45`

The built-in MATLAB function `ode45` uses a Runge–Kutta method and a *variable time step*. Based on how rapidly the solution functions are changing, the time step is altered to improve accuracy. The user need not be aware of the details of the algorithm, but when it is necessary to know the number of time steps, it can be useful to call the function `length`, which is illustrated in the example problem below.

Recall the problem of Example 5.5. Suppose the following chemical reactions take place in a continuous stirred tank reactor (CSTR):

$$A \underset{k_2}{\overset{k_1}{\Leftrightarrow}} B \underset{k_4}{\overset{k_3}{\Leftrightarrow}} C \tag{10.2}$$

where the rate constants are as follows:

$$k_1 = 1\ \text{min}^{-1},\ k_2 = 0\ \text{min}^{-1},\ k_3 = 2\ \text{min}^{-1},\ k_4 = 3\ \text{min}^{-1}$$

The initial charge to the reactor is all *A*, so the initial conditions are (in mol/L)

$$C_{A_0} = 1 \quad C_{B_0} = 0 \quad C_{C_0} = 0$$

An unsteady-state mass balance on each component leads to the following set of ODEs:

$$\begin{aligned}\frac{dC_A}{dt} &= -k_1C_A + k_2C_B\\ \frac{dC_B}{dt} &= k_1C_A - k_2C_B - k_3C_B + k_4C_C\\ \frac{dC_C}{dt} &= k_3C_B - k_4C_C\end{aligned} \tag{10.3}$$

The syntax of the `ode45` function is

```
[t, y] = ode45(@rhs_function, tspan, initial_conditions)
```

where

`t` = the independent variable vector.

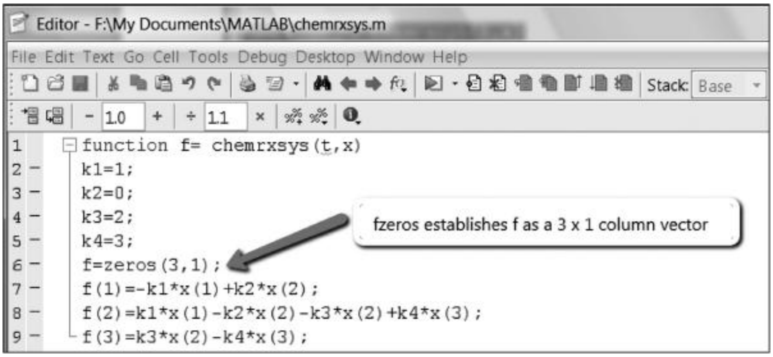
`y` = the dependent variable matrix (first column is the first dependent variable, second column is the second, etc.).

`rhs _ function` = an M-file function defining the right-hand sides of first-order ODEs. The @ sign signifies this as a function name.

`tspan` = a vector of initial and final values of t.

`initial _ conditions` = a vector of initial conditions.

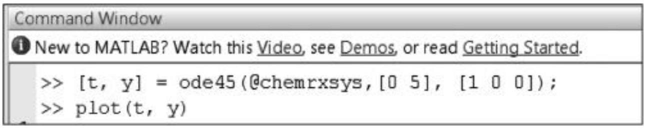
The following is an M-file listing of a function called chemrxsys. Within the function, the rate constants are fixed, and the right-hand-side functions are identified.


Editor - F:\My Documents\MATLAB\chemrxsys.m
File Edit Text Go Cell Tools Debug Desktop Window Help
Stack: Base
- 1.0 + ÷ 1.1 ×

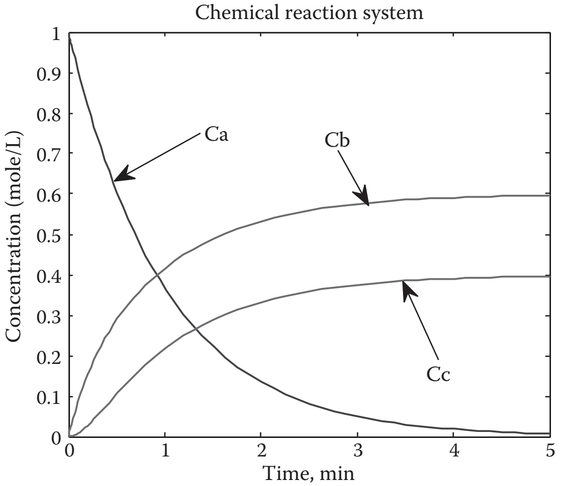
```
function f= chemrxsys(t,x)
k1=1;
k2=0;
k3=2;
k4=3;
f=zeros(3,1);
f(1)=-k1*x(1)+k2*x(2);
f(2)=k1*x(1)-k2*x(2)-k3*x(2)+k4*x(3);
f(3)=k3*x(2)-k4*x(3);
```

fzeros establishes f as a 3 x 1 column vector

Shown next is a Command Window session in which `ode45` is called to solve this problem. Also, a graph is produced (the graph shown was enhanced by editing it).


Command Window
New to MATLAB? Watch this Video, see Demos, or read Getting Started.

```
>> [t, y] = ode45(@chemrxsys,[0 5], [1 0 0]);
>> plot(t, y)
```



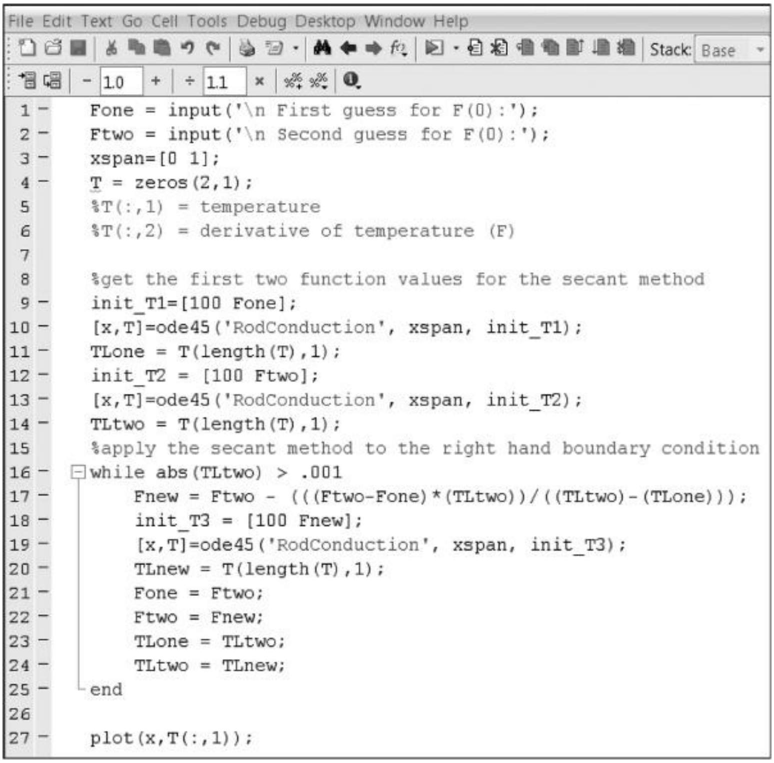
When comparing this graph to that of Example 5.5, the two plots are essentially identical.

Example 10.5: Solving a Boundary Value Problem Using `ode45` and the Shooting Method

Recall the problem of Example 6.2 involving heat conduction in a rod. The requisite ODEs and boundary conditions are

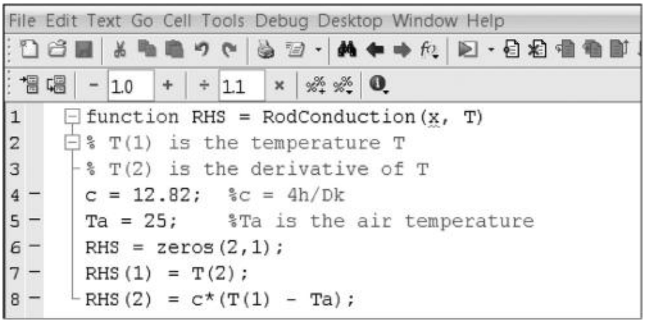
$$
\begin{aligned}
&\frac{dT}{dx} = F \\
&\frac{dF}{dx} = \frac{4h}{Dk}(T - T_a) \\
&T(0) = 100 \\
&F(0) = ? \\
&T(1) = 0
\end{aligned}
\tag{10.4}
$$

Shown below is a MATLAB script (no function heading) file that prompts the user for two initial guesses for *F*(0). These guesses are used by the *secant method* to converge the right and boundary conditions, which is *T*(1) = 0.



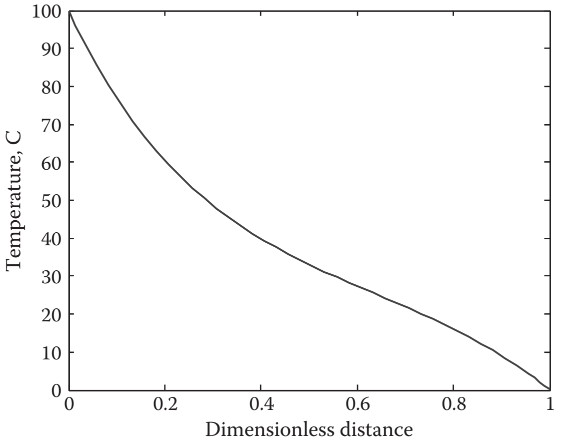
```
Fone = input('\n First guess for F(0):');
Ftwo = input('\n Second guess for F(0):');
xspan=[0 1];
T = zeros(2,1);
%T(:,1) = temperature
%T(:,2) = derivative of temperature (F)

%get the first two function values for the secant method
init_T1=[100 Fone];
[x,T]=ode45('RodConduction', xspan, init_T1);
TLone = T(length(T),1);
init_T2 = [100 Ftwo];
[x,T]=ode45('RodConduction', xspan, init_T2);
TLtwo = T(length(T),1);
%apply the secant method to the right hand boundary condition
while abs(TLtwo) > .001
    Fnew = Ftwo - (((Ftwo-Fone)*(TLtwo))/((TLtwo)-(TLone)));
    init_T3 = [100 Fnew];
    [x,T]=ode45('RodConduction', xspan, init_T3);
    TLnew = T(length(T),1);
    Fone = Ftwo;
    Ftwo = Fnew;
    TLone = TLtwo;
    TLtwo = TLnew;
end

plot(x,T(:,1));
```

The function `RodConduction`, which defines the two right-hand-side functions for the two ODEs, appears below:



```
function RHS = RodConduction(x, T)
% T(1) is the temperature T
% T(2) is the derivative of T
c = 12.82;  %c = 4h/Dk
Ta = 25;    %Ta is the air temperature
RHS = zeros(2,1);
RHS(1) = T(2);
RHS(2) = c*(T(1) - Ta);
```

The plot generated by the script (with initial guesses of –100 and –150, respectively) is shown below:



This plot is essentially identical to that of Example 6.2.

Example 10.6: Nonlinear Equations Using `fsolve`

Recall the problem of Example 9.4 involving a CSTR. The appropriate equations and data are as follows:

$$\begin{aligned}
k_1 &= 1.5\ \mathrm{s}^{-1} \\
k_2 &= 0.1\ \mathrm{L}^{1/2}/\mathrm{gmol}^{1/2} - \mathrm{s} \\
k_3 &= 0.1\ \mathrm{L/gmol} - \mathrm{s} \\
k_4 &= 0.5\ \mathrm{L/gmol} - \mathrm{s} \\
Q &= 50\ \mathrm{gmol/s} \\
V &= 100\ \mathrm{L/s}
\end{aligned} \tag{10.5}$$

$$\begin{aligned}
C_A &= C_{A0} + V\left(-k_1 C_A - k_2 C_A^{3/2} + k_3 C_C^2\right)/Q \\
C_B &= C_{B0} + V\left(2k_1 C_A - k_4 C_B^2\right)/Q \\
C_C &= C_{C0} + V\left(k_2 C_A^{3/2} - k_3 C_C^2 + k_4 C_B^2\right)/Q \\
C_D &= C_{D0} + V\left(k_4 C_B^2\right)/Q
\end{aligned} \tag{10.6}$$

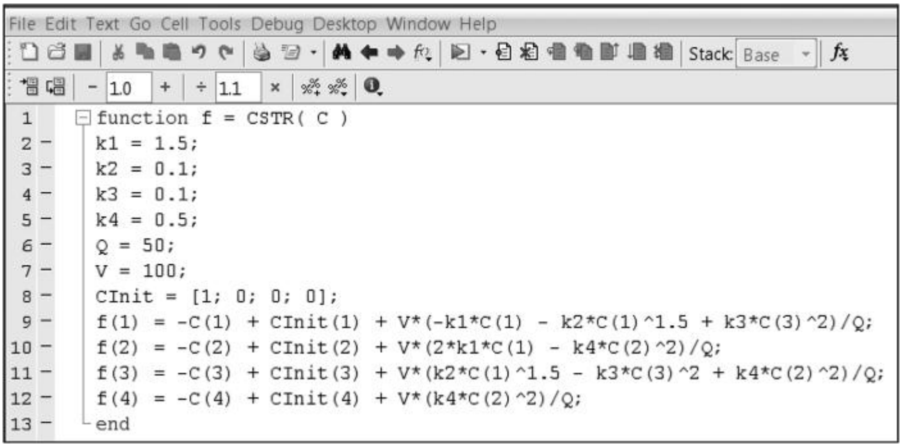
The MATLAB function `fsolve` is used to solve sets of nonlinear equations. The syntax for `fsolve` is as follows:

```
x = fsolve(func, x0)
```

where

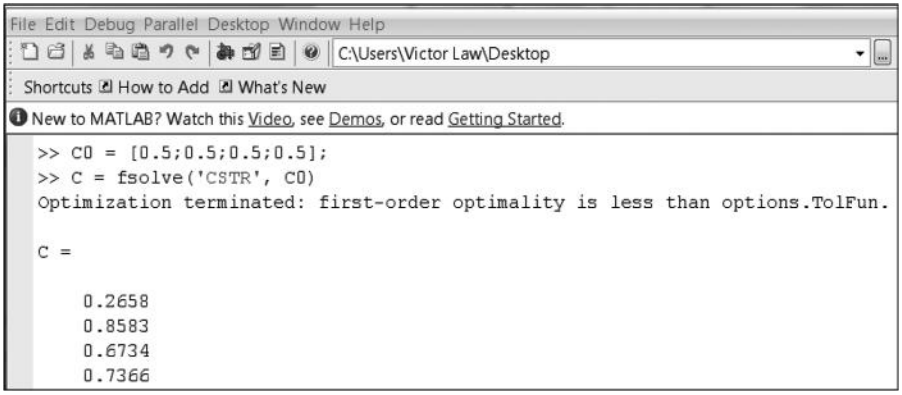
`x` = a vector of unknowns
`func` = a function M-file that evaluates the right-hand side of *f*(x) = 0
`x(0)` = a vector of initial guesses for `x`

The following is a listing of a function CSTR, which codes Equation 10.6 in the form *f*(x) = 0:



```
function f = CSTR( C )
k1 = 1.5;
k2 = 0.1;
k3 = 0.1;
k4 = 0.5;
Q = 50;
V = 100;
CInit = [1; 0; 0; 0];
f(1) = -C(1) + CInit(1) + V*(-k1*C(1) - k2*C(1)^1.5 + k3*C(3)^2)/Q;
f(2) = -C(2) + CInit(2) + V*(2*k1*C(1) - k4*C(2)^2)/Q;
f(3) = -C(3) + CInit(3) + V*(k2*C(1)^1.5 - k3*C(3)^2 + k4*C(2)^2)/Q;
f(4) = -C(4) + CInit(4) + V*(k4*C(2)^2)/Q;
end
```

A MATLAB Command Window session where the initial guess for the concentrations is given and `fsolve` is called appears below. The solution vector for the concentrations is essentially identical to that of Example 9.4.


File Edit Debug Parallel Desktop Window Help
C:\Users\Victor Law\Desktop
Shortcuts How to Add What's New
New to MATLAB? Watch this Video, see Demos, or read Getting Started.

```
>> C0 = [0.5;0.5;0.5;0.5];
>> C = fsolve('CSTR', C0)
Optimization terminated: first-order optimality is less than options.TolFun.

C =

    0.2658
    0.8583
    0.6734
    0.7366
```

Example 10.7: Nonlinear Regression Using `minsearch`

The built-in function `minsearch` is based on a rather unsophisticated algorithm. There are more robust unconstrained minimization functions available in some of the MATLAB Toolboxes, but unfortunately, these are not standard. For simple problems, `minsearch` often works well enough. It is used here to minimize the sum of squares between fictitious data (program generated data) and a function in which the regression coefficients appear nonlinearly.

The syntax of `minsearch` is as follows:

```
params = fminsearch(@function, initial_guess,[], xdata,
ydata)
```

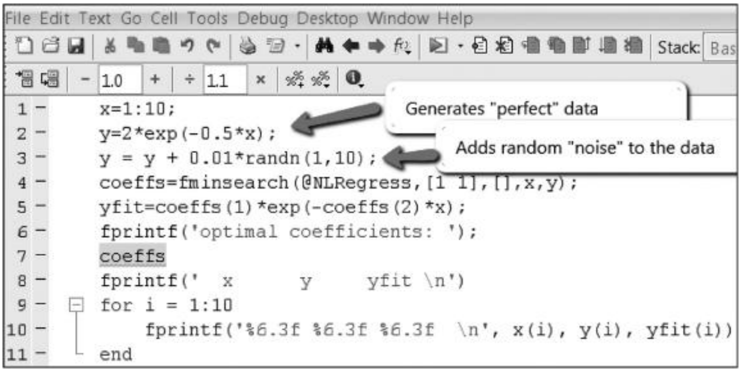
where

```
params          = The regression coefficients
function        = The name of the function that calculates the sum of squares
initial_guess   = Vector of initial guesses for coefficients
[]              = An “empty” argument that is not needed for present purposes
xdata, ydata    = Vectors holding the experimental data
```

The specific nonlinear regression problem to be considered is to find the coefficients, *c*(1) and *c*(2), in the function of Equation 10.7.

$$ycalc = c(1)e^{c(2)x} \tag{10.7}$$

A MATLAB script file saved as `NIRMAIN.M` that generates data using `c(1) = 2` and `c(2) = 0.5`, adds Gaussian random noise to these data (to make things a bit more realistic), calls `fminsearch`, and prints results is shown below:

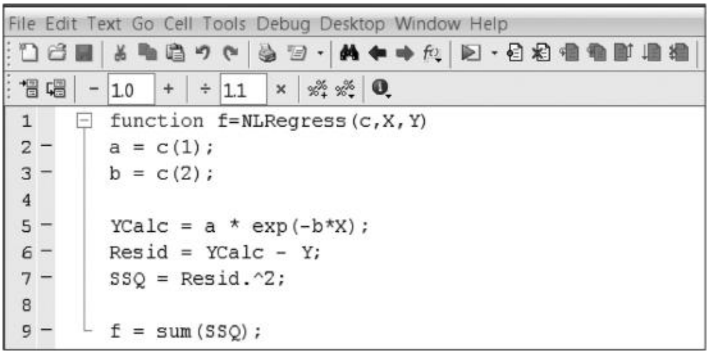


```
x=1:10;
y=2*exp(-0.5*x);
y = y + 0.01*randn(1,10);
coeffs=fminsearch(@NLRegress,[1 1],[],x,y);
yfit=coeffs(1)*exp(-coeffs(2)*x);
fprintf('optimal coefficients: ');
coeffs
fprintf('  x      y     yfit \n')
for i = 1:10
    fprintf('%6.3f %6.3f %6.3f  \n', x(i), y(i), yfit(i))
end
```

Generates "perfect" data

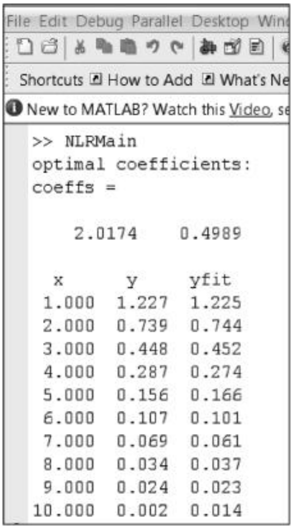
Adds random "noise" to the data

Shown next is a listing of the function `NLRegress`, which provides `fminsearch` with the objective function to minimize. In this case, it is the sum of squares of residuals between data and calculated values.



```
function f=NLRegress(c,X,Y)
a = c(1);
b = c(2);

YCalc = a * exp(-b*X);
Resid = YCalc - Y;
SSQ = Resid.^2;

f = sum(SSQ);
```

Finally, the following shows a MATLAB Command Window session that calls the script file. The results are displayed in the Command Window.


File Edit Debug Parallel Desktop Win
Shortcuts How to Add What's Ne
New to MATLAB? Watch this Video, se

```
>> NLRMain
optimal coefficients:
coeffs =

    2.0174    0.4989

  x       y      yfit
 1.000  1.227  1.225
 2.000  0.739  0.744
 3.000  0.448  0.452
 4.000  0.287  0.274
 5.000  0.156  0.166
 6.000  0.107  0.101
 7.000  0.069  0.061
 8.000  0.034  0.037
 9.000  0.024  0.023
10.000  0.002  0.014
```

The coefficients used to generate the data were 2 and 0.5, respectively. The values of 2.0174 and 0.4989 are optimal for the data with random noise added.

Note: Each time this program is run, the results will be slightly different. This is because a different set of random numbers is generated on each run.

10.7 CLOSING COMMENT REGARDING MATLAB

As with the coverage of VBA in this text, this chapter has only touched the “tip of the iceberg” with respect to MATLAB. It is intended that with the background of the introductory material present here, a student can explore the vastness of available MATLAB features and functions. For example, the nonlinear regression example (Example 10.7) used the function `minsearch`, which is not a highly robust minimization algorithm. Another MATLAB function that is particularly suited to nonlinear regression is `nlfit` and its companion `nlparci`, which provides confidence intervals for each parameter. A MATLAB add-on Optimization Toolbox provides several algorithms for nonlinear programming. There are many other Toolboxes available for specialized areas. If one searches diligently, a MATLAB function set can be found for any of a vast number of application areas.

EXERCISES

Exercise 10.1: Solve the problem described in Exercise 5.1 using MATLAB.

Exercise 10.2: Solve the problem described in Exercise 5.4 using MATLAB.

Exercise 10.3: Solve the problem described in Exercise 5.7 using MATLAB.

Exercise 10.4: Solve the problem described in Exercise 5.10 using MATLAB.

Exercise 10.5: Solve the problem described in Exercise 5.11 using MATLAB.

Exercise 10.6: Solve the problem described in Exercise 6.1 using MATLAB. Implement the secant method, as in Example 10.5, to converge the right-hand boundary condition. Use `ode45` to solve the ODEs.

Exercise 10.7: Solve the problem described in Exercise 6.2 using MATLAB. Implement the secant method, as in Example 10.5, to converge the right-hand boundary condition. Use `ode45` to solve the ODEs.

Exercise 10.8: Solve the problem described in Exercise 6.4 using MATLAB. Implement the secant method, as in Example 10.5, to converge the right-hand boundary condition. Use `ode45` to solve the ODEs.

Exercise 10.9: Solve the problem described in Exercise 6.5 using MATLAB. Implement the secant method, as in Example 10.5, to converge the right-hand boundary condition. Ignore part b and use `ode45` to solve the ODEs.

Exercise 10.10: Solve the problem described in Exercise 6.6 using MATLAB. Implement the secant method, as in Example 10.5, to converge the right-hand boundary condition. Use `ode45` to solve the ODEs.

Exercise 10.11: Solve the problem described in Exercise 9.2 using MATLAB. Use the `fminsearch` function to minimize the sum of squares of residuals.

Exercise 10.12: Solve the problem described in Exercise 9.4 using MATLAB. Use the `fsolve` function to minimize the sum of squares of residuals.

Exercise 10.13: Solve the problem described in Exercise 9.7 using MATLAB. Use the `fsolve` function to minimize the sum of squares of residuals.

Exercise 10.14: Solve the problem described in Exercise 9.8 using MATLAB. Use the `fminsearch` function to minimize the sum of squares of residuals.

Appendix: Additional Features of VBA

A.1 INTRODUCTION

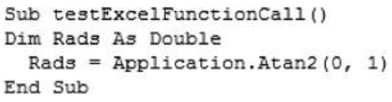
VBA is a mega system of programming language and objects. It is probably impossible for any one person to be familiar with all of the documented features of VBA. However, the object-oriented nature of the system makes it extensible both by "official" Microsoft® documented items as well as those added by third party developers and individual programmers. In this appendix, a few additional features of VBA are presented that might be useful to chemical and biomolecular engineering students and practitioners. The following items are covered:

1. How to call both built-in functions and Add-In functions from VBA Macros
2. How to include user-defined functions as Add-Ins that can be accessed by other VBA subs and functions
3. How to return arrays from functions, which can in turn be included as Add-Ins

Warning: The author is not a VBA expert. The methods presented in this appendix are ones that have been found to work. No claim is made for their uniqueness or efficiency. When viewed by a true VBA "guru," these techniques might be considered naïve. Engineers often settle for things that work as opposed to ones that are perfect.

A.2 CALLING EXCEL® BUILT-IN FUNCTIONS IN VBA MACROS

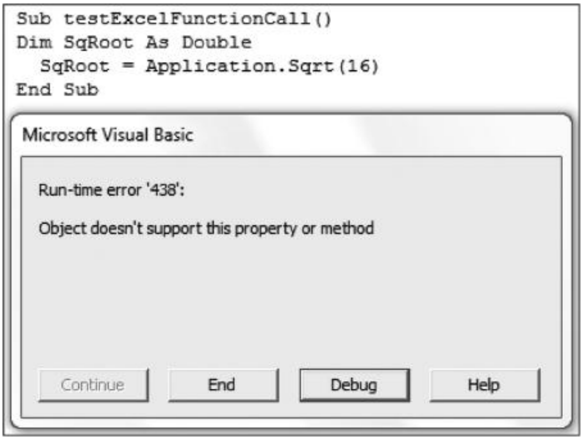
To use Excel functions for which there is no VBA counterpart (e.g., `ATAN2`), the `Application` object can be used as shown in Chapter 2. Given the code



```
Sub testExcelFunctionCall()
Dim Rads As Double
  Rads = Application.Atan2(0, 1)
End Sub
```

the variable `Rads` is assigned the value π/2.

Note that using the `Application` object does not work with functions for which there is a VBA counterpart (even if the names are different). For example, the code shown below produces the error message that appears after the code:



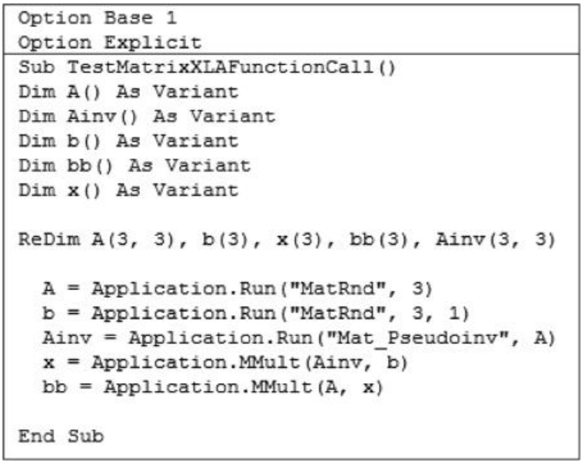
```
Sub testExcelFunctionCall()
Dim SqRoot As Double
  SqRoot = Application.Sqrt(16)
End Sub
```

Run-time error '438':

Object doesn't support this property or method

A.3 CALLING EXCEL® ADD-IN FUNCTIONS IN VBA MACROS

Consider the following VBA sub:



```
Option Base 1
Option Explicit
Sub TestMatrixXLAFunctionCall()
Dim A() As Variant
Dim Ainv() As Variant
Dim b() As Variant
Dim bb() As Variant
Dim x() As Variant

ReDim A(3, 3), b(3), x(3), bb(3), Ainv(3, 3)

  A = Application.Run("MatRnd", 3)
  b = Application.Run("MatRnd", 3, 1)
  Ainv = Application.Run("Mat_Pseudoinv", A)
  x = Application.MMult(Ainv, b)
  bb = Application.MMult(A, x)

End Sub
```

Since the sub involves matrices and vectors, the usual `Option Base 1` is used. Five variables are declared as dynamic arrays whose elements are of type `Variant`. This data type has been avoided in purely numerical computations but is necessary here to allow assignment to array variables. The statement

```
        A = Application.Run(“MatRnd”, 3)
```

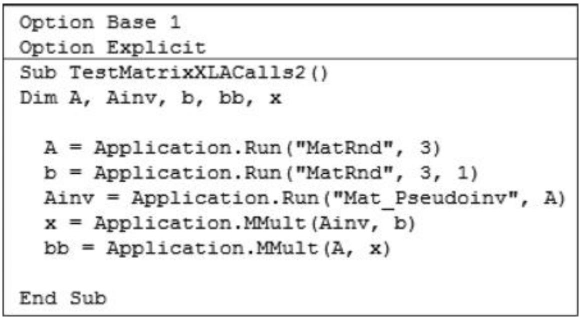
executes the `MatRnd` function from the `Matrix.XLA` add-in and produces a 3 × 3 matrix of random integers (the default range is –10 to 10), and this matrix is assigned to the variable `A`. The next statement creates a 3 × 1 vector (*b*) of random integers.

The statement

```
Ainv = Application.Run(“Mat_Pseuoinv”, A)
```

calls the `Matrix.XLA` add-in function to calculate the pseudoinverse of `A`, which is then stored in the variable `Ainv`. The next statement uses the Excel function `MMULT` to produce the solution to the system *Ax* = *b*. The last statement again uses `MMULT`; in this case, the variable `bb` should have the same values as the original right-hand side, `b`. To see these results, it is best to run the Macro in Debug mode and use the Set Watch feature to display the values stored in each variable. This is the first use of the Variant data type in this text. See the next section for more details on this type.

Shown below is another VBA sub that performs the same operations as the last example.



```
Option Base 1
Option Explicit
Sub TestMatrixXLACalls2()
Dim A, Ainv, b, bb, x

  A = Application.Run("MatRnd", 3)
  b = Application.Run("MatRnd", 3, 1)
  Ainv = Application.Run("Mat_Pseudoinv", A)
  x = Application.MMult(Ainv, b)
  bb = Application.MMult(A, x)

End Sub
```

Here, an “anonymous” Dim statement is used for all variables (no data type is indicated). The structure and data type of these variables are established when they are assigned something. For example, in the case of the variable A, the assignment statement stores a 3 × 3 matrix of random integers. Although this example is more compact than the previous one, it is not as explicit. From an engineering perspective, since they both work the same, there is no reason to prefer one over the other.

A.4 VARIANT DATA TYPE

The following is an excerpt from “The Power of Variants” (http://www.tushar-mehta.com/publish_train/book_vba/08_variants.htm#_ftn1):

> A variable declared as type *Variant* can contain any type of data. Unlike a variable that declared on a specific type, say, String or Integer, which can only contain a text string or a specific range of integers respectively, a variant can contain any data—text or an integer value or a real, i.e., a floating point, value. It can even behave like an array or refer to an object, either built into Excel or a user defined type. Essentially, there are almost no rules on what a developer can do with a variant. For example, with `aVar` declared as a variant each of the assignment statements is legitimate.

```
Dim aVar as Variant
aVar = "a"
aVar = 1
aVar = Array (1,22,333)
set aVar = ActiveSheet
aVar = 3.1415927
```

The common wisdom is that one should stay away from variants. By and large, that is true. If one knows the data type of a variable it is best to declare it correctly. There are many benefits to doing so, the most significant being that the VBA compiler can ensure data and program integrity. With a variant one could accidentally assign a text string to what might be intended to be a number. Essentially, the developer gets the flexibility of a variable that can take any type of data together with the responsibility of ensuring proper data type use. That's a steep burden and one best avoided whenever possible. Consequently, in those cases where the data type is pre-determined and will not change, it is indeed best to declare the variable of the particular type.

However, *there are many instances where a variant allows one to do things that otherwise would be impossible.* The power of a variant comes from the fact that it is a simple data element and yet can contain ***any***—and that means ***any***—type of data. It can be a string or a Boolean or an integer or a real number. Hence, when the data type returned by a function can vary, one is obligated to use a variant for the returned value.

As we will see in a later section of this chapter, the ability to create an array in a variant makes it possible to create functions that would otherwise be impossible. For example, a function can return either an error condition or an array of values. It also allows a developer to write a User Defined Function (UDF) that returns multiple values in a single call to the function. A variant is also one way to pass an array as a 'by value' argument to a procedure. One Excel-specific reason to use a variant is that it provides a very efficient way to exchange information between Excel and VBA.

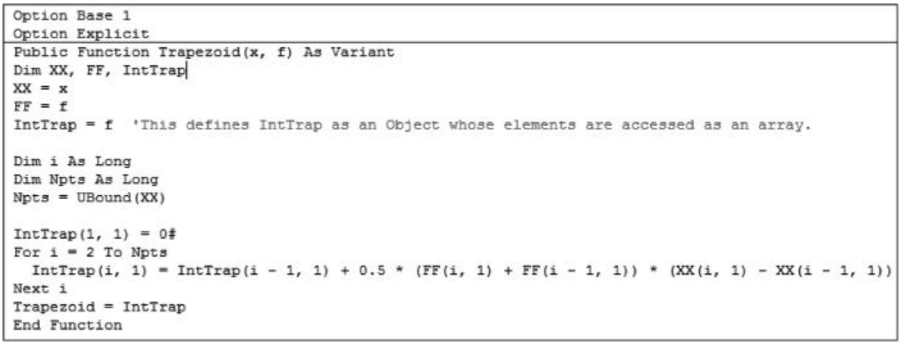
Finally, in the advanced section of the chapter, we will use the variant data type to create *an array of arrays.* This makes it possible to create, and work with, data structures that would otherwise be impossible. It also allows one to operate on an entire row of an array.

In the hands of a creative—and defensive—developer, the power of a variant can be nearly limitless.

With this background, it can be seen why the Variant data type was used in the prior example when it was desired to assign an entire array to a variable.

A.5 VBA FUNCTION THAT RETURNS AN ARRAY

Consider the following VBA Function Macro.



```
Option Base 1
Option Explicit
Public Function Trapezoid(x, f) As Variant
Dim XX, FF, IntTrap
XX = x
FF = f
IntTrap = f  'This defines IntTrap as an Object whose elements are accessed as an array.

Dim i As Long
Dim Npts As Long
Npts = UBound(XX)

IntTrap(1, 1) = 0#
For i = 2 To Npts
  IntTrap(i, 1) = IntTrap(i - 1, 1) + 0.5 * (FF(i, 1) + FF(i - 1, 1)) * (XX(i, 1) - XX(i - 1, 1))
Next i
Trapezoid = IntTrap
End Function
```

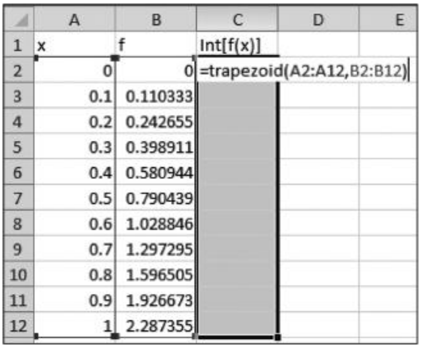
Before going into any detail about this code, it is important to see how one enters a stand-alone function into the VBA system. Recall that when writing a Sub Macro, the Macros option is chosen from the Developer tab, in which case the VBA editor appears with a blank `Sub`. It is not possible to change the word `Sub` to `Function` and proceed as usual. Instead, the Visual Basic® option must be chosen from the Developer tab, which gives a blank screen in the VBA editor. Look for the name of the associated VBA project (something like Book 2, for example); right click on the project name and select Insert/Module. This gives a blank editor page where the code for the function is to be entered. When the associated Excel Worksheet is saved, the function Macro is saved with it (the Worksheet must be saved as a macro-enabled one). Another oddity of function Macros occurs when they are to be edited. The function name does not appear when visiting Developer/Macros. However, if the name of the function is typed on the appropriate line, then the Edit button activates and the Macro can be edited as though it were a Sub Macro.

Referring now to the code for the function Trapezoid shown above, there are several things to note:

1. The function has two arguments called `x` and `f`, which are arrays of independent variable and associated function value whose definite integral is required. The function uses the trapezoidal rule to calculate the integral, which is returned to the calling spreadsheet.
2. The `Public` declaration might not be required, but it guarantees that anyone can use the function without permission.
3. The `Variant` type of the function allows an *array* to be returned from the function.
4. The anonymous `Dim` statement allows *anything* to be assigned to the associated variables.

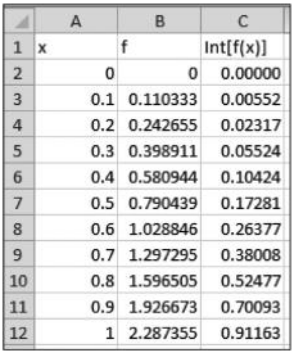
5. The three assignment statements create Objects whose elements can be accessed with subscripts. They are actually accessed as two-dimensional arrays with a second subscript of 1—that is, they are addressed as Npts × 1 arrays instead of vectors of length Npts. The items to the right of the = sign are function arguments.
6. The variable Npts takes on the upper bound of subscripts of XX, whose length is that of the input arrays.
7. The initialization IntTrap(1, 1) = 0# sets the first value of the integral to floating point zero (that is what the #means).
8. The For loop computes the integral at each x-value using the trapezoidal rule.
9. The final assignment Trapezoid = IntTrap returns the array of integral values in the function name.

Shown next is a spreadsheet in which the x and y columns contain an independent variable varying between 0 and 1 and associated function values for the simple function e^{x}sin x. The third column was selected and the text = trapezoid(was typed. At this point, the data values for x were selected followed by a comma, then the selection of the second column of values (for f), and finally a close parenthesis.



	A	B	C	D	E
1	x	f	Int[f(x)]		
2	0	0	=trapezoid(A2:A12,B2:B12)		
3	0.1	0.110333			
4	0.2	0.242655			
5	0.3	0.398911			
6	0.4	0.580944			
7	0.5	0.790439			
8	0.6	1.028846			
9	0.7	1.297295			
10	0.8	1.596505			
11	0.9	1.926673			
12	1	2.287355			

Note that the cell ranges for x and f appear within the parentheses. To view the entire column of results, it is necessary to strike Shift/Ctrl/Enter. The results appear in the spreadsheet shown below.



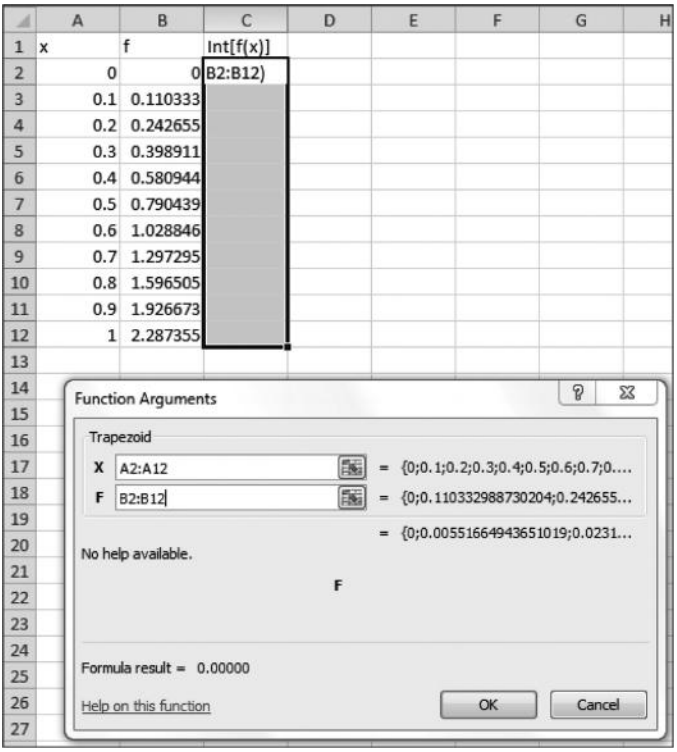
	A	B	C
1	x	f	Int[f(x)]
2	0	0	0.00000
3	0.1	0.110333	0.00552
4	0.2	0.242655	0.02317
5	0.3	0.398911	0.05524
6	0.4	0.580944	0.10424
7	0.5	0.790439	0.17281
8	0.6	1.028846	0.26377
9	0.7	1.297295	0.38008
10	0.8	1.596505	0.52477
11	0.9	1.926673	0.70093
12	1	2.287355	0.91163

A.6 CREATING EXCEL® ADD-INS

Suppose that the function for calculating an integral using the trapezoidal rule is to be used in more than one spreadsheet and is to be used frequently. It would be useful to have this function available as an Excel Add-In (like the Matrix.xla functions). To do this, once the function has been thoroughly tested, simply save the spreadsheet as an Excel Add-In. When saving, change the file type accordingly. The file is saved in a special directory reserved for Add-Ins (this can be overridden if desired). To activate the Add-In, choose the File/Options/Add-Ins menu. At the bottom of the screen is an option to manage Add-Ins—choose Go. This brings up a screen with Add-In names and a check box in front of each. Be sure the box is checked for the macro to be activated. If the name of the Add-In does not appear, choose Browse to locate the file for the Add-In.

Shown below is a spreadsheet in which the Trapezoid Add-In is used. The column where results are to appear is selected and then Formulas/Insert Function. Choose User Defined from the "Or Select a Category" menu. Find and select Trapezoid and click OK.

Note that the user is automatically prompted to select the range for `x` and `f`. This feature is added when the Add-In is created and activated—no special programming is required.



	A	B	C	D	E	F	G	H
1	x	f	Int[f(x)]					
2	0	0	B2:B12)					
3	0.1	0.110333						
4	0.2	0.242655						
5	0.3	0.398911						
6	0.4	0.580944						
7	0.5	0.790439						
8	0.6	1.028846						
9	0.7	1.297295						
10	0.8	1.596505						
11	0.9	1.926673						
12	1	2.287355						

Function Arguments

Trapezoid

X A2:A12 = {0;0.1;0.2;0.3;0.4;0.5;0.6;0.7;0....

F B2:B12 = {0;0.110332988730204;0.242655...

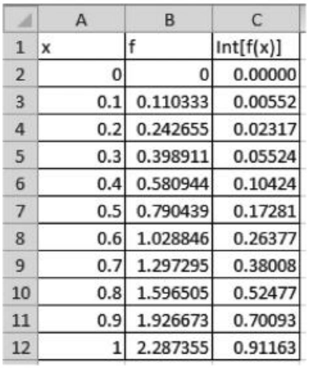
= {0;0.00551664943651019;0.0231...

No help available.

F

Formula result = 0.00000

After selecting the range for x and f, hit Shift/Ctrl/Enter to display the results as follows:



	A	B	C
1	x	f	Int[f(x)]
2	0	0	0.00000
3	0.1	0.110333	0.00552
4	0.2	0.242655	0.02317
5	0.3	0.398911	0.05524
6	0.4	0.580944	0.10424
7	0.5	0.790439	0.17281
8	0.6	1.028846	0.26377
9	0.7	1.297295	0.38008
10	0.8	1.596505	0.52477
11	0.9	1.926673	0.70093
12	1	2.287355	0.91163

CHEMICAL ENGINEERING

Numerical Methods for Chemical Engineers
Using Excel®, VBA, and MATLAB®

VICTOR J. LAW

While teaching the Numerical Methods for Engineers course over the last 15 years, the author found a need for a new textbook, one that was less elementary, provided applications and problems better suited for chemical engineers, and contained instruction in Visual Basic® for Applications (VBA). This led to six years of developing teaching notes that have been enhanced to create the current textbook, **Numerical Methods for Chemical Engineers Using Excel®, VBA, and MATLAB®**.

Focusing on Excel gives the advantage of it being generally available, since it is present on every computer—PC and Mac—that has Microsoft Office® installed. The VBA programming environment comes with Excel and greatly enhances the capabilities of Excel spreadsheets. While there is no perfect programming system, teaching this combination offers knowledge in a widely available program that is commonly used (Excel) as well as a popular academic software package (MATLAB). Chapters cover nonlinear equations, Visual Basic, linear algebra, ordinary differential equations, regression analysis, partial differential equations, and mathematical programming methods.

Each chapter contains examples that show in detail how a particular numerical method or programming methodology can be implemented in Excel and/or VBA (or MATLAB in Chapter 10). Most of the examples and problems presented in the text are related to chemical and biomolecular engineering and cover a broad range of application areas including thermodynamics, fluid flow, heat transfer, mass transfer, reaction kinetics, reactor design, process design, and process control. The chapters feature "Did You Know" boxes, used to remind readers of Excel features. They also contain end-of-chapter exercises, with solutions provided.

CRC Press
Taylor & Francis Group
an informa business

www.taylorandfrancisgroup.com

6000 Broken Sound Parkway, NW
Suite 300, Boca Raton, FL 33487
711 Third Avenue
New York, NY 10017
2 Park Square, Milton Park
Abingdon, Oxon OX14 4RN, UK

K16702

ISBN: 978-1-4665-7534-9

90000

9 781466 575349

www.crcpress.com