Excel®
for Scientists and Engineers
Numerical Methods

E. Joseph Billo

WILEY-INTERSCIENCE
A John Wiley & Sons, Inc., Publication

This Page Intentionally Left Blank

Excel®
for Scientists and Engineers

Numerical Methods

THE WILEY BICENTENNIAL–KNOWLEDGE FOR GENERATIONS

Each generation has its unique needs and aspirations. When Charles Wiley first opened his small printing shop in lower Manhattan in 1807, it was a generation of boundless potential searching for an identity. And we were there, helping to define a new American literary tradition. Over half a century later, in the midst of the Second Industrial Revolution, it was a generation focused on building the future. Once again, we were there, supplying the critical scientific, technical, and engineering knowledge that helped frame the world. Throughout the 20th Century, and into the new millennium, nations began to reach out beyond their own borders and a new international community was born. Wiley was there, expanding its operations around the world to enable a global exchange of ideas, opinions, and know-how.

For 200 years, Wiley has been an integral part of each generation's journey, enabling the flow of information and understanding necessary to meet their needs and fulfill their aspirations. Today, bold new technologies are changing the way we live and learn. Wiley will be there, providing you the must-have knowledge you need to imagine new worlds, new possibilities, and new opportunities.

Generations come and go, but you can always count on Wiley to provide you the knowledge you need, when and where you need it!

WILLIAM J. PESCE
PRESIDENT AND CHIEF EXECUTIVE OFFICER

PETER BOOTH WILEY
CHAIRMAN OF THE BOARD

Excel®
for Scientists and Engineers

Numerical Methods

E. Joseph Billo

WILEY-INTERSCIENCE
A John Wiley & Sons, Inc., Publication

Copyright © 2007 by John Wiley & Sons, Inc. All rights reserved.

Published by John Wiley & Sons, Inc., Hoboken, New Jersey.
Published simultaneously in Canada.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 750-4470, or on the web at www.copyright.com. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at http://www.wiley.com/go/permission.

Limit of Liability/Disclaimer of Warranty: While the publisher and author have used their best efforts in preparing this book, they make no representations or warranties with respect to the accuracy or completeness of the contents of this book and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. No warranty may be created or extended by sales representatives or written sales materials. The advice and strategies contained herein may not be suitable for your situation. You should consult with a professional where appropriate. Neither the publisher nor author shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

For general information on our other products and services or for technical support, please contact our Customer Care Department within the United States at (800) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic format. For information about Wiley products, visit our web site at www.wiley.com.

Wiley Bicentennial Logo: Richard J. Pacifico

Library of Congress Cataloging-in-Publication Data is available.

ISBN: 978-0-471-38734-3

Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

Summary of Contents

Detailed Table of Contents vii
Preface xv
Acknowledgments xix
About the Author xix

Chapter 1 Introducing Visual Basic for Applications 1
Chapter 2 Fundamentals of Programming with VBA 15
Chapter 3 Worksheet Functions for Working with Matrices 57
Chapter 4 Number Series 69
Chapter 5 Interpolation 77
Chapter 6 Differentiation 99
Chapter 7 Integration 127
Chapter 8 Roots of Equations 147
Chapter 9 Systems of Simultaneous Equations 189
Chapter 10 Numerical Integration of Ordinary Differential Equations Part I: Initial Conditions 217
Chapter 11 Numerical Integration of Ordinary Differential Equations Part II: Boundary Conditions 245
Chapter 12 Partial Differential Equations 263
Chapter 13 Linear Regression and Curve Fitting 287
Chapter 14 Nonlinear Regression Using the Solver 313
Chapter 15 Random Numbers and the Monte Carlo Method 341

APPENDICES
Appendix 1 Selected VBA Keywords 365
Appendix 2 Shortcut Keys for VBA 387
Appendix 3 Custom Functions Help File 389
Appendix 4 Some Equations for Curve Fitting 409
Appendix 5 Engineering and Other Functions 423
Appendix 6 ASCII Codes 427
Appendix 7 Bibliography 429
Appendix 8 Answers and Comments for End-of-Chapter Problems 431

INDEX 443

This Page Intentionally Left Blank

Contents

Preface xv
Acknowledgments xix
About the Author xix

Chapter 1 Introducing Visual Basic for Applications 1
The Visual Basic Editor 1
Visual Basic Procedures 4
There Are Two Kinds of Macros 4
The Structure of a Sub Procedure 4
The Structure of a Function Procedure 5
Using the Recorder to Create a Sub Procedure 5
The Personal Macro Workbook 7
Running a Sub Procedure 8
Assigning a Shortcut Key to a Sub Procedure 8
Entering VBA Code 9
Creating a Simple Custom Function 10
Using a Function Macro 10
A Shortcut to Enter a Function 12
Some FAQs 13

Chapter 2 Fundamentals of Programming with VBA 15
Components of Visual Basic Statements 15
Operators 16
Variables 16
Objects, Properties, and Methods 17
Objects 17
Properties 17
Using Properties 19
Functions 20
Using Worksheet Functions with VBA 22
Some Useful Methods 22
Other Keywords 23
Program Control 23
Branching 23
Logical Operators 24
Select Case 24
Looping 24
For...Next Loop 25
Do While... Loop 25

For Each...Next Loop ... 25
Nested Loops ... 26
Exiting from a Loop or from a Procedure ... 26
VBA Data Types ... 27
The Variant Data Type ... 28
Subroutines ... 28
Scoping a Subroutine ... 29
VBA Code for Command Macros ... 29
Objects and Collections of Objects ... 29
"Objects" That Are Really Properties ... 30
You Can Define Your Own Objects ... 30
Methods ... 31
Some Useful Methods ... 31
Two Ways to Specify Arguments of Methods ... 32
Arguments with or without Parentheses ... 33
Making a Reference to a Cell or a Range ... 33
A Reference to the Active Cell or a Selected Range ... 33
A Reference to a Cell Other than the Active Cell ... 34
References Using the Union or Intersect Method ... 35
Examples of Expressions to Refer to a Cell or Range ... 35
Getting Values from a Worksheet ... 36
Sending Values to a Worksheet ... 37
Interacting with the User ... 37
MsgBox ... 37
MsgBox Return Values ... 39
InputBox ... 39
Visual Basic Arrays ... 41
Dimensioning an Array ... 41
Use the Name of the Array Variable to Specify the Whole Array ... 42
Multidimensional Arrays ... 42
Declaring the Variable Type of an Array ... 42
Returning the Size of an Array ... 42
Dynamic Arrays ... 43
Preserving Values in Dynamic Arrays ... 43
Working with Arrays in Sub Procedures: Passing Values from Worksheet to VBA Module ... 44
A Range Specified in a Sub Procedure Can Be Used as an Array ... 44
Some Worksheet Functions Used Within VBA Create an Array Automatically ... 45
Some Worksheet Functions Used Within VBA Create an Array Automatically ... 45
An Array of Object Variables ... 45

Working with Arrays in Sub Procedures:
Passing Values from a VBA Module to a Worksheet 45
A One-Dimensional Array Assigned to a Worksheet Range
Can Cause Problems 46
Custom Functions 47
Specifying the Data Type of an Argument 47
Specifying the Data Type Returned by a Function Procedure 47
Returning an Error Value from a Function Procedure 48
A Custom Function that Takes an Optional Argument 48
Arrays in Function Procedures 48
A Range Passed to a Function Procedure Can Be Used as an Array 48
Passing an Indefinite Number of Arguments:
Using the ParamArray Keyword 49
Returning an Array of Values as a Result 49
Creating Add-In Function Macros 50
How to Create an Add-In Macro 51
Testing and Debugging 51
Tracing Execution 52
Stepping Through Code 52
Adding a Breakpoint 52
Examining the Values of Variables While in Break Mode 53
Examining the Values of Variables During Execution 54

Chapter 3 Worksheet Functions for Working with Matrices **57**
Arrays, Matrices and Determinants 57
Some Types of Matrices 57
An Introduction to Matrix Mathematics 58
Excel's Built-in Matrix Functions 60
Some Additional Matrix Functions 63
Problems 66

Chapter 4 Number Series **69**
Evaluating Series Formulas 70
Using Array Constants to Create Series Formulas 70
Using the ROW Worksheet Function to Create Series Formulas 71
The INDIRECT Worksheet Function 71
Using the INDIRECT Worksheet Function
with the ROW Worksheet Function to Create Series Formulas 72
The Taylor Series 72
The Taylor Series: An Example 73
Problems 75

Chapter 5 Interpolation 77
Obtaining Values from a Table 77
Using Excel's Lookup Functions to Obtain Values from a Table 77
Using VLOOKUP to Obtain Values from a Table 78
Using the LOOKUP Function to Obtain Values from a Table 79
Creating a Custom Lookup Formula to Obtain Values from a Table 80
Using Excel's Lookup Functions to Obtain Values from a Two-Way Table 81
Interpolation 83
Linear Interpolation in a Table by Means of Worksheet Formulas 83
Linear Interpolation in a Table by Using the TREND Worksheet Function 85
Linear Interpolation in a Table by Means of a Custom Function 86
Cubic Interpolation 87
Cubic Interpolation in a Table by Using the TREND Worksheet Function 89
Linear Interpolation in a Two-Way Table by Means of Worksheet Formulas 90
Cubic Interpolation in a Two-Way Table by Means of Worksheet Formulas 91
Cubic Interpolation in a Two-Way Table by Means of a Custom Function 93
Problems 96

Chapter 6 Differentiation 99
First and Second Derivatives of Data in a Table 99
Calculating First and Second Derivatives 100
Using LINEST as a Fitting Function 105
Derivatives of a Worksheet Formula 109
Derivatives of a Worksheet Formula Calculated by Using a VBA Function Procedure 109
First Derivative of a Worksheet Formula Calculated by Using the Finite-Difference Method 110
The Newton Quotient 110
Derivative of a Worksheet Formula Calculated by Using the Finite-Difference Method 111
First Derivative of a Worksheet Formula Calculated by Using a VBA Sub Procedure Using the Finite-Difference Method 112
First Derivative of a Worksheet Formula Calculated by Using a VBA Function Procedure Using the Finite-Difference Method 115
Improving the VBA Function Procedure 118
Second Derivative of a Worksheet Formula 120
Concerning the Choice of Δx for the Finite-Difference Method 123
Problems 124

Chapter 7 Integration **127**
Area under a Curve 127
Calculating the Area under a Curve Defined by a Table of Data Points 129
Calculating the Area under a Curve Defined by a Table of Data Points by Means of a VBA Function Procedure 130
Calculating the Area under a Curve Defined by a Formula 131
Area between Two Curves 132
Integrating a Function 133
Integrating a Function Defined by a Worksheet Formula by Means of a VBA Function Procedure 133
Gaussian Quadrature 137
Integration with an Upper or Lower Limit of Infinity 140
Distance Traveled Along a Curved Path 141
Problems 143

Chapter 8 Roots of Equations **147**
A Graphical Method 147
The Interval-Halving or Bisection Method 149
The Interval Method with Linear Interpolation (the *Regula Falsi* Method) 151
The *Regula Falsi* Method with Correction for Slow Convergence 153
The Newton-Raphson Method 154
Using **Goal Seek...** 156
The Secant Method 160
The Newton-Raphson Method Using Circular Reference and Iteration 161
A Newton-Raphson Custom Function 163
Bairstow's Method to Find All Roots of a Regular Polynomial 166
Finding Values Other than Zeroes of a Function 174
Using **Goal Seek ...** to Find the Point of Intersection of Two Curves 174
Using the Newton-Raphson Method to Find the Point of Intersection of Two Lines 176
Using the Newton-Raphson Method to Find Multiple Intersections of a Straight Line and a Curve 178
A Goal Seek Custom Function 180
Problems 185

Chapter 9 Systems of Simultaneous Equations **189**
Cramer's Rule 190
Solving Simultaneous Equations by Matrix Inversion 191
Solving Simultaneous Equations by Gaussian Elimination 191
The Gauss-Jordan Method 196
Solving Linear Systems by Iteration 200
The Jacobi Method Implemented on a Worksheet 200

The Gauss-Seidel Method Implemented on a Worksheet....203
The Gauss-Seidel Method Implemented on a Worksheet Using Circular References....204
A Custom Function Procedure for the Gauss-Seidel Method....205
Solving Nonlinear Systems by Iteration....207
Newton's Iteration Method....207
Problems....213

Chapter 10 Numerical Integration of Ordinary Differential Equations Part I: Initial Conditions 217

Solving a Single First-Order Differential Equation....218
Euler's Method....218
The Fourth-Order Runge–Kutta Method....220
Fourth-Order Runge-Kutta Method Implemented on a Worksheet....220
Runge-Kutta Method Applied to a Differential Equation Involving Both x and y....223
Fourth-Order Runge-Kutta Custom Function for a Single Differential Equation with the Derivative Expression Coded in the Procedure....224
Fourth-Order Runge-Kutta Custom Function for a Single Differential Equation with the Derivative Expression Passed as an Argument....225
Systems of First-Order Differential Equations....228
Fourth-Order Runge-Kutta Custom Function for Systems of Differential Equations....229
Predictor-Corrector Methods....235
A Simple Predictor-Corrector Method....235
A Simple Predictor-Corrector Method Utilizing an Intentional Circular Reference....236
Higher-Order Differential Equations....238
Problems....241

Chapter 11 Numerical Integration of Ordinary Differential Equations Part II: Boundary Conditions 245

The Shooting Method....245
An Example: Deflection of a Simply Supported Beam....246
Solving a Second-Order Ordinary Differential Equation by the Shooting Method and Euler's Method....249
Solving a Second-Order Ordinary Differential Equation by the Shooting Method and the RK Method....251
Finite-Difference Methods....254
Solving a Second-Order Ordinary Differential Equation by the Finite-Difference Method....254

Another Example 258
A Limitation on the Finite-Difference Method 261
Problems 262

Chapter 12 Partial Differential Equations **263**
Elliptic, Parabolic and Hyperbolic Partial Differential Equations 263
Elliptic Partial Differential Equations 264
Solving Elliptic Partial Differential Equations:
Replacing Derivatives with Finite Differences 265
An Example: Temperature Distribution in a Heated Metal Plate 267
Parabolic Partial Differential Equations 269
Solving Parabolic Partial Differential Equations: The Explicit Method 270
An Example: Heat Conduction in a Brass Rod 272
Solving Parabolic Partial Differential Equations:
The Crank-Nicholson or Implicit Method 274
An Example: Vapor Diffusion in a Tube 275
Vapor Diffusion in a Tube Revisited 277
Vapor Diffusion in a Tube (Again) 279
A Crank-Nicholson Custom Function 280
Vapor Diffusion in a Tube Solved by Using a Custom Function 282
Hyperbolic Partial Differential Equations 282
Solving Hyperbolic Partial Differential Equations:
Replacing Derivatives with Finite Differences 282
An Example: Vibration of a String 283
Problems 286

Chapter 13 Linear Regression and Curve Fitting **287**
Linear Regression 287
Least-Squares Fit to a Straight Line 288
Least-Squares Fit to a Straight Line Using the Worksheet Functions
SLOPE, INTERCEPT and RSQ 289
Multiple Linear Regression 291
Least-Squares Fit to a Straight Line Using LINEST 292
Multiple Linear Regression Using LINEST 293
Handling Noncontiguous Ranges of *known_x's* in LINEST 297
A LINEST Shortcut 297
LINEST's Regression Statistics 297
Linear Regression Using Trendline 298
Limitations of Trendline 301
Importing Trendline Coefficients into a Spreadsheet
by Using Worksheet Formulas 302
Using the Regression Tool in Analysis Tools 303
Limitations of the Regression Tool 305

Importing the Trendline Equation from a Chart into a Worksheet 305
Problems 309

Chapter 14 Nonlinear Regression Using the Solver 313

Nonlinear Least-Squares Curve Fitting 314
Introducing the Solver 316
How the Solver Works 316
Loading the Solver Add-In 317
Why Use the Solver for Nonlinear Regression? 317
Nonlinear Regression Using the Solver: An Example 318
Some Notes on Using the Solver 323
Some Notes on the Solver Parameters Dialog Box 323
Some Notes on the Solver Options Dialog Box 324
When to Use Manual Scaling 326
Statistics of Nonlinear Regression 327
The Solver Statistics Macro 328
Be Cautious When Using Linearized Forms of Nonlinear Equations 329
Problems 332

Chapter 15 Random Numbers and the Monte Carlo Method 341

Random Numbers in Excel 341
How Excel Generates Random Numbers 341
Using Random Numbers in Excel 342
Adding "Noise" to a Signal Generated by a Formula 344
Selecting Items Randomly from a List 345
Random Sampling by Using Analysis Tools 347
Simulating a Normal Random Distribution of a Variable 349
Monte Carlo Simulation 350
Monte Carlo Integration 354
The Area of an Irregular Polygon 354
Problems 362

APPENDICES 363

Appendix 1 Selected VBA Keywords 365
Appendix 2 Shortcut Keys for VBA 387
Appendix 3 Custom Functions Help File 389
Appendix 4 Some Equations for Curve Fitting 409
Appendix 5 Engineering and Other Functions 423
Appendix 6 ASCII Codes 427
Appendix 7 Bibliography 429
Appendix 8 Answers and Comments for End-of-Chapter Problems 431

INDEX 443

Preface

The solutions to mathematical problems in science and engineering can be obtained by using either analytical or numerical methods. Analytical (or direct) methods involve the use of closed-form equations to obtain an exact solution, in a nonrepetitive fashion; obtaining the roots of a quadratic equation by application of the quadratic formula is an example of an analytical solution. Numerical (or indirect) methods involve the use of an algorithm to obtain an approximate solution; results of a high level of accuracy can usually be obtained by applying the algorithm in a series of successive approximations.

As the complexity of a scientific problem increases, it may no longer be possible to obtain an exact mathematical expression as a solution to the problem. Such problems can usually be solved by numerical methods.

The Objective of This Book

Numerical methods require extensive calculation, which is easily accomplished using today's desktop computers. A number of books have been written in which numerical methods are implemented using a specific programming language, such as FORTRAN or C++. Most scientists and engineers received some training in computer programming in their college days, but they (or their computer) may no longer have the capability to write or run programs in, for example, FORTRAN. This book shows how to implement numerical methods using Microsoft Excel®, the most widely used spreadsheet software package. Excel® provides at least three ways for the scientist or engineer to apply numerical methods to problems:

- by implementing the methods on a worksheet, using worksheet formulas
- by using the built-in tools that are provided within Excel
- by writing programs, sometimes loosely referred to as macros, in Excel's Visual Basic for Applications (VBA) programming language.

All of these approaches are illustrated in this book.

This is a book about numerical *methods*. I have emphasized the methods and have kept the mathematical theory behind the methods to a minimum. In many cases, formulas are introduced with little or no description of the underlying theory. (I assume that the reader will be familiar with linear interpolation, simple calculus, regression, etc.) Other topics, such as cubic interpolation, methods for solving differential equations, and so on, are covered in more detail, and a few

topics, such as Bairstow's method for obtaining the roots of a regular polynomial, are discussed in detail.

In this book I have provided a wide range of Excel solutions to problems. In many cases I provide a series of examples that progress from a very simple implementation of the problem (useful for understanding the logic and construction of the spreadsheet or VBA code) to a more sophisticated one that is more general. Some of the VBA macros are simple "starting points" and I encourage the reader to modify them; others are (or at least I intended them to be) "finished products" that I hope users can employ on a regular basis.

Nearly 100% of the material in this book applies equally to the PC or Macintosh versions of Excel. In a few cases I have pointed out the different keystrokes requires for the Macintosh version.

A Note About Visual Basic Programming

Visual Basic for Applications, or VBA, is a "dialect" of Microsoft's Visual Basic programming language. VBA has keywords that allow the programmer to work with Excel's workbooks, worksheets, cells, charts, etc.

I expect that although many readers of this book will be proficient VBA programmers, others may not be familiar with VBA but would like to learn to program in VBA. The first two chapters of this book provide an introduction to VBA programming – not enough to become proficient, but enough to understand and perhaps modify the VBA code in this book. For readers who have no familiarity with VBA, and who do not wish to learn it, do not despair. Much of the book (perhaps 50%) does not involve VBA. In addition, you can still use the VBA custom functions that have been provided.

Appendix 1 provides a list of VBA keywords that are used in this book. The appendix provides a description of the keyword, its syntax, one or more examples of its use, and reference to related keywords. The information is similar to what can be found in Excel's On-Line Help, but readers may find it helpful at those times when they are reading the book without simultaneous access to a PC.

A Note About Typographic Conventions

The typographic conventions used in this book are the following:

Menu Commands. Excel's menu commands appear in bold, as in the following examples: "choose **Add Trendline...** from the **Chart** menu...," or "**Insert→Function...**"

Excel's Worksheet Functions and Their Arguments. Worksheet functions are in Arial font; the arguments are italicized. Following Microsoft's convention, required arguments are in bold font, while optional arguments are in nonbold, as in the following:

VLOOKUP(***lookup_value, table_array, column_index_num,*** *range_lookup*)

The syntax of custom functions follows the same convention.

Excel Formulas. Excel formulas usually appear in a separate line, for example,

```
=1+1/FACT(1)+1/FACT(2)+1/FACT(3)+1/FACT(4)+1/FACT(5)
```

Named ranges used in formulas or in the text are not italicized, to distinguish them from Excel's argument names, for example,

```
=VLOOKUP(Temp,Table,MATCH(Percent,P_Row,1)+1,1)
```

VBA Procedures. Visual Basic code is in Arial font. Complete VBA procedures are displayed in a box, as in the following. For ease in understanding the code, VBA keywords are in bold.

```
Private Function Deriv1(x)
'User codes the expression for the derivative here.
Deriv1 = 9 * x ^ 2 + 10 * x - 5
End Function
```

Problems and Solutions

There are over 100 end-of-chapter problems. Spreadsheet solutions for the problems are on the CD-ROM that accompanies this book. Answers and explanatory notes for most of the problems are provided in Appendix 8.

The Contents of the CD

The CD-ROM that accompanies this book contains a number of folders or other documents:

- an "Examples" folder. The Examples folder contains a folder for each chapter, e.g., 'Ch. 05 (Interpolation) Examples.' The examples folder for each chapter contains all of the examples discussed in that chapter: spreadsheets, charts and VBA code. The location of the Excel file pertinent to each example is specified in the chapter text, usually in the caption of a figure, e.g.,

  Figure 5-5. Using VLOOKUP and MATCH to obtain a value from a two-way table. (folder 'Chapter 05 Interpolation,' workbook 'Interpolation I,' sheet 'Viscosity')

- a "Problems" folder. The Problems folder contains a folder for each chapter, e.g., 'Ch. 06 (Differentiation) problems.' The problems folder for each chapter contains solutions to (almost) all of the end-of-chapter problems in that chapter. VBA code required for the solution of any of the problems is provided in each workbook that requires it; the VBA code will be identical to the code found in the 'Examples' folder.
- an Excel workbook, "Numerical Methods Toolbox," that contains all of the important custom functions in this book.
- a copy of "Numerical Methods Toolbox" saved as an Add-In workbook (an .xla file). If you open this Add-In, the custom functions will be available for use in any Excel workbook.
- Two Excel workbooks containing the utilities Solver Statistics and Trendline to Cell.

Comments Are Welcomed

I welcome comments and suggestions from readers. I can be contacted at numerical_methods.billo@verizon.net.

E. Joseph Billo

Acknowledgments

Dr. Richard N. Fell, Department of Physics, Brandeis University, Waltham, MA; Prof. Michele Mandrioli, Department of Chemistry and Biochemistry, University of Massachusetts–Dartmouth, North Dartmouth, MA; and Prof. Christopher King, Department of Chemistry, Troy University, Troy, AL, who read the complete manuscript and provided valuable comments and corrections.

Prof. Lev Zompa, University of Massachusetts–Boston, and Dr. Peter Gans, Protonic Software, for UV-vis spectral data.

Edwin Straver and Nicole Steidel, Frontline Systems Inc., for information about the inner workings of the Solver.

The Dow Chemical Company for permission to use tables of physical properties of heat transfer fluids.

About the Author

E. Joseph Billo retired in 2006 as Associate Professor of Chemistry at Boston College, Chestnut Hill, Massachusetts. He is the author of *Excel for Chemists: A Comprehensive Guide*, 2nd edition, Wiley-VCH, New York, 2001. He has presented the 2-day short courses "Advanced Excel for Scientists and Engineers" and "Excel Visual Basic Macros for Scientists and Engineers" to over 2000 scientists at corporate clients in the United States, Canada and Europe.

This Page Intentionally Left Blank

Chapter 1

Introducing Visual Basic for Applications

In addition to Excel's extensive list of worksheet functions and array of calculation tools for scientific and engineering calculations, Excel contains a programming language that allows users to create procedures, sometimes referred to as macros, that can perform even more advanced calculations or that can automate repetitive calculations.

Excel's first programming language, Excel 4 Macro Language (XLM) was introduced with version 4 of Excel. It was a rather cumbersome language, but it did provide most of the capabilities of a programming language, such as looping, branching and so on. This first programming language was quickly superseded by Excel's current programming language, Visual Basic for Applications, introduced with version 5 of Excel. Visual Basic for Applications, or VBA, is a "dialect" of Microsoft's Visual Basic programming language, a dialect that has keywords to allow the programmer to work with Excel's workbooks, worksheets, cells, charts, etc. At the same time, Microsoft introduced a version of Visual Basic for Word; it was called WordBasic and had keywords for characters, paragraphs, line breaks, etc. But even at the beginning, Microsoft's stated intention was to have one version of Visual Basic that could work with all its applications: Excel, Word, Access and PowerPoint. Each version of Microsoft Office has moved closer to this goal.

The Visual Basic Editor

To create VBA code, or to examine existing code, you will need to use the Visual Basic Editor. To access the Visual Basic Editor, choose **Macro** from the **Tools** menu and then **Visual Basic Editor** from the submenu.

The Visual Basic Editor screen usually contains three important windows: the Project Explorer window, the Properties window and the Code window, as shown in Figure 1-1. (What you see may not look exactly like this.)

The Code window displays the active module sheet; each module sheet can contain one or several VBA procedures. If the workbook you are using does not

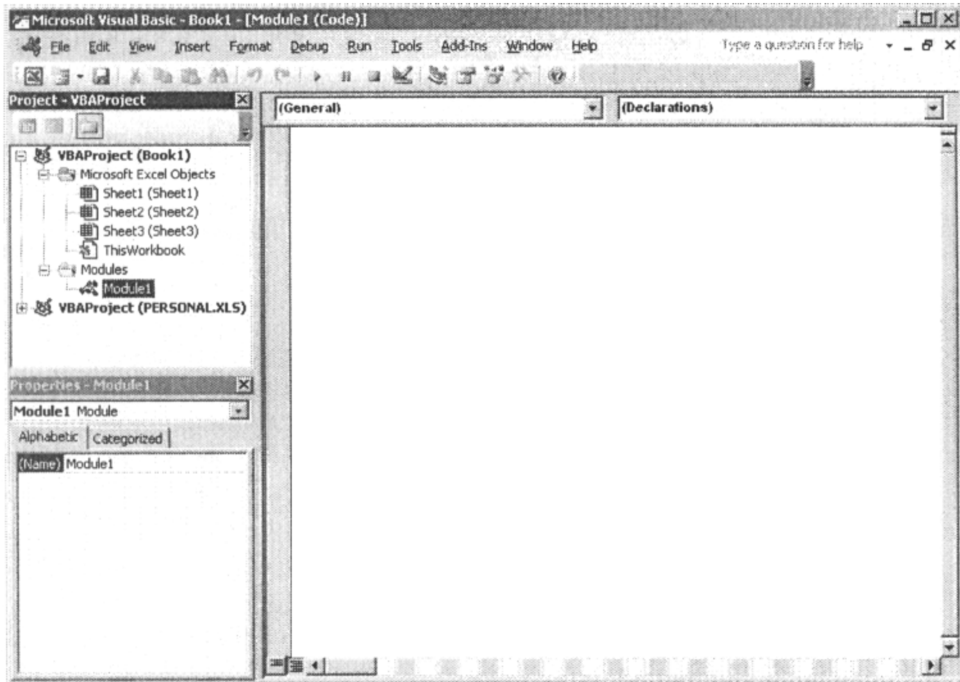
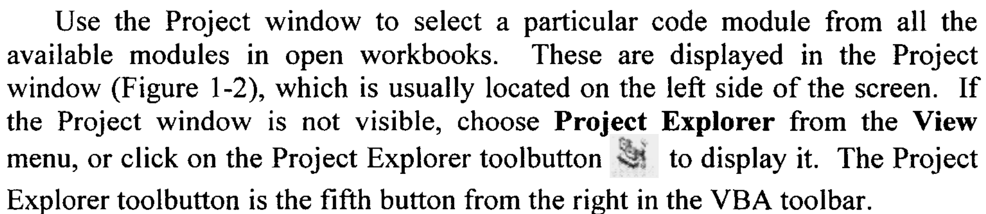


Figure 1-1. The Visual Basic Editor window.

contain any module sheets, the Code window will be empty. To insert a module sheet, choose **Module** from the **Insert** menu. A folder icon labeled Modules will be inserted; if you click on this icon, the module sheet Module1 will bedisplayed. Excel gives these module sheets the default names Module1, Module2 and so on.

Use the Project window to select a particular code module from all the available modules in open workbooks. These are displayed in the Project window (Figure 1-2), which is usually located on the left side of the screen. If the Project window is not visible, choose **Project Explorer** from the **View** menu, or click on the Project Explorer toolbutton  to display it. The Project Explorer toolbutton is the fifth button from the right in the VBA toolbar.

In the Project Explorer window you will see a hierarchy tree with a node for each open workbook. In the example illustrated in Figure 1-2, a new workbook, Book1, has been opened. The node for Book1 has a node (a folder icon) labeled Microsoft Excel Objects; click on the folder icon to display the nodes it contains—an icon for each sheet in the workbook and an additional one labeled ThisWorkbook. If you double-click on any one of these nodes you will display the code sheet for it. These code sheets are for special types of procedures called automatic procedures or event-handler procedures, which are not covered in this

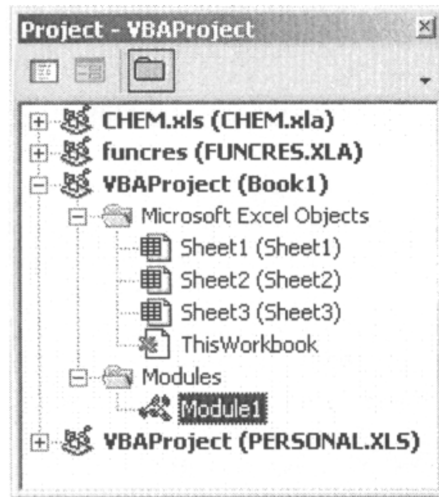


Figure 1-2. The VBE Project Explorer window.

book. Do not use any of these sheets to create the VBA procedures described in this book. The hierarchy tree in Figure 1-2 also shows a Modules folder, containing one module sheet, Module1.

The Properties window will be discussed later. Right now, you can press the Close button to get rid of it if you wish.

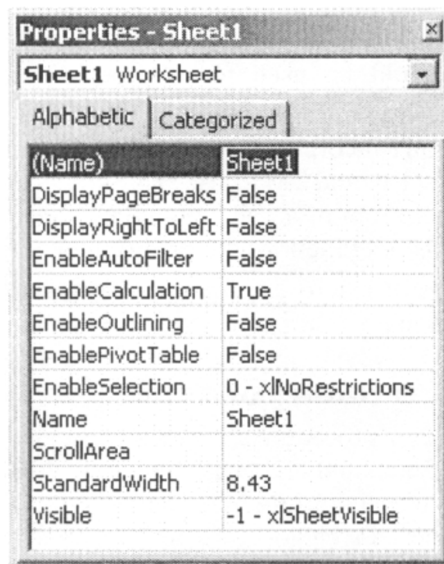


Figure 1-3. The Properties window.

Visual Basic Procedures

VBA macros are usually referred to as *procedures*. They are written or recorded on a *module* sheet. A single module sheet can contain many procedures.

There Are Two Kinds of Macros

There are two different kinds of procedures: **Sub** procedures, called command macros in the older XLM macro language, and **Function** procedures, called function macros in the XLM macro language and often referred to as custom functions or user-defined functions.

Although these procedures can use many of the same set of VBA commands, they are distinctly different. **Sub** procedures can automate any Excel action. For example, a **Sub** procedure might be used to create a report by opening a new worksheet, copying selected ranges of cells from other worksheets and pasting them into the new worksheet, formatting the data in the new worksheet, providing headings, and printing the new worksheet. **Sub** procedures are usually "run" by selecting **Macro** from the **Tools** menu. They can also be run by means of an assigned shortcut key, by being called from another procedure, or in several other ways.

Function procedures augment Excel's library of built-in functions by adding user-defined functions. A custom or user-defined function is used in a worksheet in the same way as a built-in function like, for example, Excel's SQRT function. It is entered in a formula in a worksheet cell, performs a calculation, and returns a result to the cell in which it is located. For example, a custom function named FtoC could be used to convert Fahrenheit temperatures to Celsius.

Custom functions can't incorporate any of VBA's "action" commands. No experienced user of Excel would try to use the SQRT function in a worksheet cell to calculate the square root of a number and also open a new workbook and insert the result there; custom functions are no different.

However, both kinds of macro can incorporate decision-making, branching, looping, subroutines and many other aspects of programming languages.

The Structure of a Sub Procedure

The structure of a **Sub** procedure is shown in Figure 1-4. The procedure begins with the keyword **Sub** and ends with **End Sub**. It has a ProcedureName, a unique identifier that you assign to it. The name should indicate the purpose of the function. The name can be long, since after you type it once you will probably not have to type it again. A **Sub** procedure has the possibility of using one or more arguments, Argument1, etc, but for now we will not create **Sub**

procedures with arguments. Empty parentheses are still required even if a **Sub** procedure uses no arguments.

```
Sub ProcedureName(Argument1, ...)
    VBA statements
End Sub
```

Figure 1- 4. Structure of a **Sub** procedure.

The Structure of a Function Procedure

The structure of a **Function** procedure is shown in Figure 1-5. The procedure begins with the keyword **Function** and ends with **End Function**. It has a FunctionName, a unique identifier that you assign to it. The name should be long enough to indicate the purpose of the function, but not too long, since you will probably be typing it in your worksheet formulas. A **Function** procedure usually takes one or more arguments; the names of the arguments should also be descriptive. Empty parentheses are required even if a **Function** procedure takes no arguments.

```
Function FunctionName(Argument1, ...)
    VBA statements
    FunctionName = result
End Function
```

Figure 1-5. Structure of a user-defined function.

The function's *return statement* directs the procedure to return the result to the caller (usually the cell in which the function was entered). The return statement consists of an assignment statement in which the name of the function is equated to a value, for example,

```
FunctionName = result
```

Using the Recorder to Create a Sub Procedure

Excel provides the Recorder, a useful tool for creating command macros. When you choose **Macro** from the **Tools** menu and **Record New Macro...** from the submenu, all subsequent menu and keyboard actions will be recorded until you press the Stop Macro button or choose **Stop Recording** from the **Macro** submenu. The Recorder is convenient for creating simple macros that involve only the use of menu or keyboard commands, but you can't use it to incorporate logic, branching or looping.

The Recorder creates Visual Basic commands. You don't have to know anything about Visual Basic to record a command macro in Visual Basic. This provides a good way to gain some familiarity with Visual Basic.

To illustrate the use of the Recorder, let's record the action of applying scientific number formatting to a number in a cell. First, select a cell in a worksheet and enter a number. Now choose **Macro** from the **Tools** menu, then **Record New Macro...** from the submenu. The Record Macro dialog box (Figure 1-6) will be displayed.

The Record Macro dialog box displays the default name that Excel has assigned to this macro: Macro1, Macro2, etc. Change the name in the Macro Name box to ScientificFormat (no spaces are allowed in a name). The "Store Macro In" box should display This Workbook (the default location); if not, choose This Workbook. Enter "e" in the box for the shortcut key, then press OK.

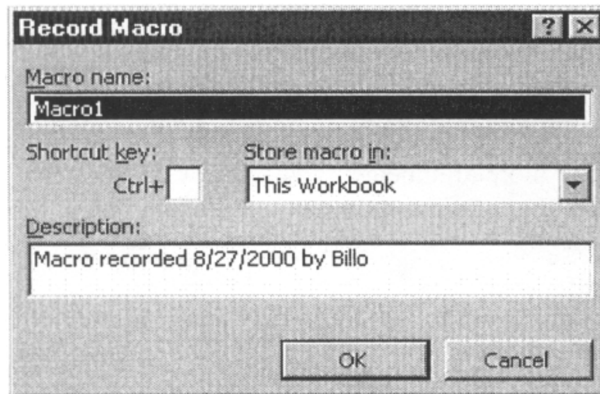


Figure 1-6. The Record Macro dialog box.

The Stop Recording toolbar will appear (Figure 1-7), indicating that a macro is being recorded. If the Stop Recording toolbar doesn't appear, you can always stop recording by using the **Tools** menu (in the Macro submenu the **Record New Macro...** command will be replaced by **Stop Recording**).

Figure 1-7. The Stop Recording toolbar.

Now choose **Cells...** from the **Format** menu, choose the Number tab and choose Scientific number format, then press OK. Finally, press the Stop Recording button.

To examine the macro code that you have just recorded, choose **Macro** from the **Tools** menu and **Visual Basic Editor** from the submenu. Click on the node for the module in the active workbook. This will display the code module sheet containing the Visual Basic code. The macro should look like the example shown in Figure 1-8.

```
Sub ScientificFormat()
'
' ScientificFormat Macro
' Macro recorded 6/22/2004 by Boston College
'
' Keyboard Shortcut: Ctrl+e
'
  Selection.NumberFormat = "0.00E+00"
End Sub
```

Figure 1-8. Macro for scientific number-formatting, recorded in VBA.

This macro consists of a single line of VBA code. You'll learn about Visual Basic code in the chapters that follow.

To run the macro, enter a number in a cell, select the cell, then choose **Macro** from the **Tools** menu, choose **Macros...** from the submenu, select the ScientificFormat macro from the Macro Name list box, and press Run. Or you can simply press the shortcut key combination that you designated when you recorded the macro (CONTROL+e in the example above). The number should be displayed in the cell in scientific format.

The Personal Macro Workbook

The Record Macro dialog box allows you to choose where the recorded macro will be stored. There are three possibilities in the "Store Macro In" list box: This Workbook, New Workbook and Personal Macro Workbook. The Personal Macro Workbook (PERSONAL.XLS in Excel for Windows, or Personal Macro Workbook in Excel for the Macintosh) is a workbook that is automatically opened when you start Excel. Since only macros in open workbooks are available for use, the Personal Macro Workbook is the ideal location for macros that you want to have available all the time.

Normally the Personal Macro Workbook is hidden (choose **Unhide...** from the **Window** menu to view it). If you don't yet have a Personal Macro Workbook, you can create one by recording a macro as described earlier, choosing Personal Macro Workbook from the "Store Macro In" list box.

As you begin to create more advanced Sub procedures, you'll find that the Recorder is a useful tool to create fragments of macro code for incorporation into your procedure. Instead of poring through a VBA reference, or searching through the On-Line VBA Help, looking for the correct command syntax, simply turn on the Recorder, perform the action, and look at the code produced. You may find that the Recorder doesn't always produce exactly what you want, or perhaps the most elegant code, but it is almost always useful.

Note that, since the Recorder only records actions, and **Function** procedures can't perform actions, the Recorder won't be useful for creating **Function** procedures.

Running a Sub Procedure

In the preceding example, the macro was run by using a shortcut key. There are a number of other ways to run a macro. One way is to use the Macro dialog box. Again, enter a number in a cell, select the cell, then choose **Macro** from the **Tools** menu and **Macros...** from the submenu. The Macro dialog box will be displayed (Figure 1-9). This dialog box lists all macros in open workbooks (right now we only have one macro available). To run the macro, select it from the list, then press the Run button.

Assigning a Shortcut Key to a Sub Procedure

If you didn't assign a shortcut key to the macro when you recorded it, but would like to do so "after the fact," choose **Macro** from the **Tools** menu and **Macros...** from the submenu. Highlight the name of the macro in the Macro Name list box, and press the Options... button. You can now enter a letter for the shortcut key: CONTROL+<key> or SHIFT+CONTROL+<key> in Excel for

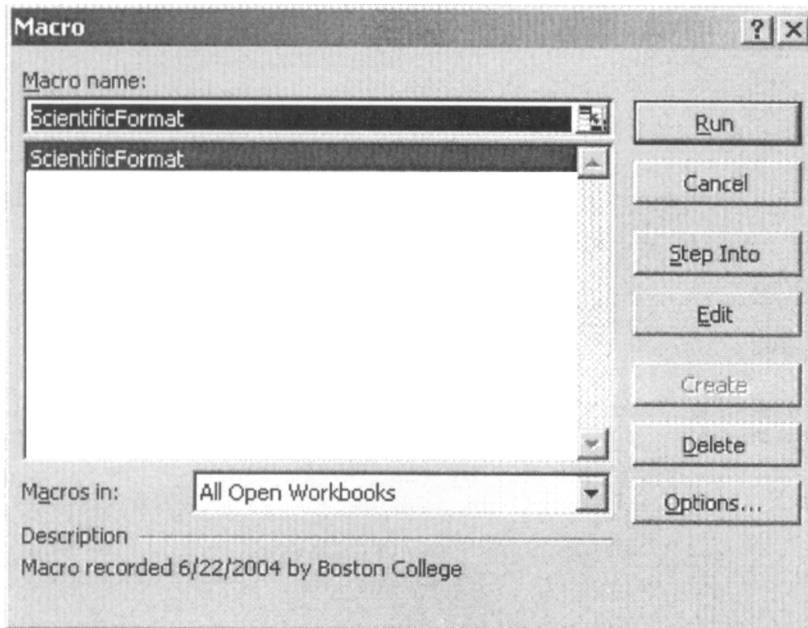


Figure 1-9. The Macro dialog box.

Windows, OPTION+COMMAND+<key> or SHIFT+OPTION+COMMAND+<key> in Excel for the Macintosh.

Entering VBA Code

Of course, most of the VBA code you create will not be recorded, but instead entered at the keyboard. As you type your VBA code, the Visual Basic Editor checks each line for syntax errors. A line that contains one or more errors will be displayed in red, the default color for errors. Variables usually appear in black. Other colors are also used; comments (see later) are usually green and some VBA keywords (**Function**, **Range**, etc.) usually appear in blue. (These default colors can be changed if you wish.)

If you type a long line of code, it will not automatically wrap to the next line but will simply disappear off the screen. You need to insert a *line-continuation character* (the underscore character, but you must type a space followed by the underscore character followed by ENTER) to cause a line break in a line of VBA code, as in the following example:

```
Worksheets("Sheet1").Range("A2:B7").Copy _
    (Worksheets("Sheet2").Range("C2"))
```

The line-continuation character can't be used within a string, i.e., within quotes.

I recommend that you type the module-level declaration **Option Explicit** at the top of each module sheet, before any procedures. **Option Explicit** forces you to declare all variables using **Dim** statements; undeclared variables produce an error at compile time.

When you type VBA code in a module, it's good programming practice to use TAB to indent related lines for easier reading, as shown in the following procedure.

```
Sub Initialize()
For J = 1 To N
    P(J) = 0
Next J
End Sub
```

Figure 1-10. A simple VBA **Sub** procedure.

In order to produce a more compact display of a procedure, several lines of code can be combined in one line by separating them with colons. For example, the procedure in Figure 1-10 can be replaced by the more compact one in Figure 1-11 or even by the one in Figure 1-12.

```
Sub Initialize()
For J = 1 To N: P(J) = 0: Next J
End Sub
```

Figure 1-11. A **Sub** procedure with several statements combined.

```
Sub Initialize(): For J = 1 To N: P(J) = 0: Next J: End Sub
```

Figure 1-12. A **Sub** procedure in one line.

Creating a Simple Custom Function

As a simple first example of a **Function** procedure, we'll create a custom function to convert temperatures in degrees Fahrenheit to degrees Celsius.

Function procedures can't be recorded; you must type them on a module sheet. You can have several macros on the same module sheet, so if you recorded the ScientificFormat macro earlier in this chapter, you can type this custom function procedure on the same module sheet. If you do not have a module sheet available, insert one by choosing **Module** from the **Insert** menu.

Type the macro as shown in Figure 1-13. DegF is the argument passed by the function from the worksheet to the module (the Fahrenheit temperature); the single line of VBA code evaluates the Celsius temperature and returns the result to the *caller* (in this case, the worksheet cell in which the function is entered).

```
Function FtoC(DegF)
  FtoC = (DegF – 32) * 5 / 9
End Function
```

Figure 1-13. Fahrenheit to Celsius custom function.

A note about naming functions and arguments: function names should be short, since you will be typing them in Excel formulas (that's why Excel's square-root worksheet function is SQRT) but long enough to convey information about what the function does. In contrast, command macro names can be long, since command macros are run by choosing the name of the macro from the list of macros in the Macro Run dialog box, for example.

Argument names can be long, since you don't type them. Longer names can convey more information, and thus provide a bit of self-documentation. (If you look at the arguments used in Excel's worksheet functions, you'll see that single letters are usually not used as argument names.)

Using a Function Macro

A custom function is used in a worksheet formula in exactly the same way as any of Excel's built-in functions. The workbook containing the custom function must be open.

Figure 1-14 shows how the FtoC custom function is used. Cell A2 contains 212, the argument that the custom function will use. Cell B2 contains the formula with the custom function. You can enter the function in cell B2 by

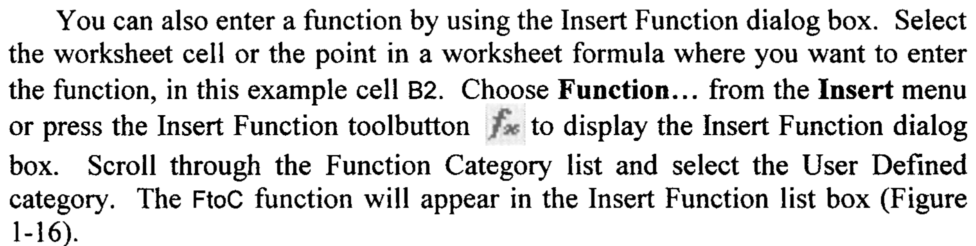
typing it (Figure 1-14). When you press enter, the result calculated by the function appears in the cell (Figure 1-15).

	A	B
1	T, °F	T, °C
2	212	=FtoC(A2)

Figure 1-14. Entering the custom function.

	A	B
1	T, °F	T, °C
2	212	100

Figure 1-15. The function result.



You can also enter a function by using the Insert Function dialog box. Select the worksheet cell or the point in a worksheet formula where you want to enter the function, in this example cell B2. Choose **Function...** from the **Insert** menu or press the Insert Function toolbutton *fx* to display the Insert Function dialog box. Scroll through the Function Category list and select the User Defined category. The FtoC function will appear in the Insert Function list box (Figure 1-16).

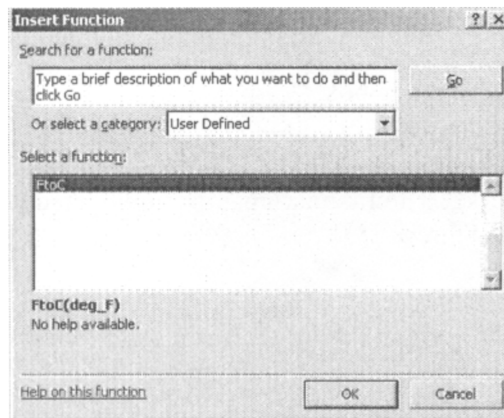


Figure 1-16. The Paste Function dialog box.

When you press OK, the Function Arguments dialog box (Figure 1-17) will be displayed. Enter the argument, or click on the cell containing the argument to enter the reference (cell A2 in Figure 1-14), then press the OK button.

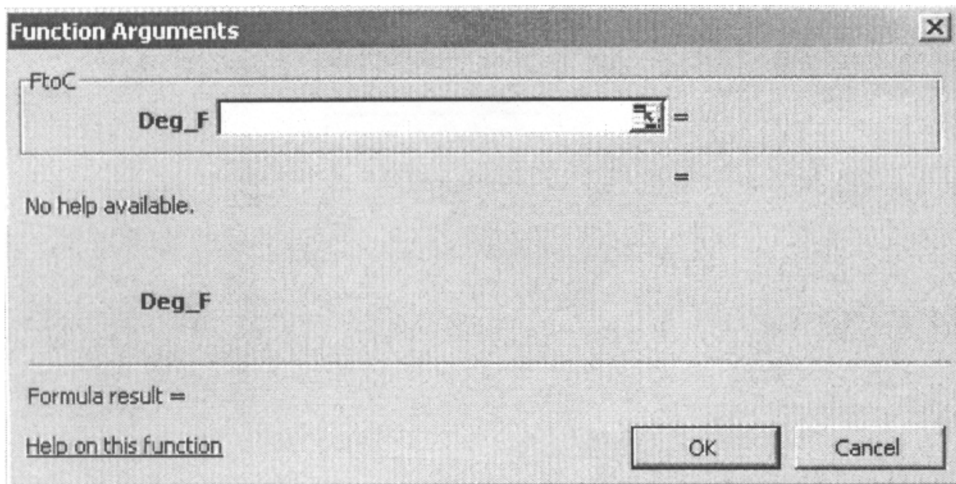


Figure 1-17. The Function Arguments dialog box.

A Shortcut to Enter a Function

You can enter a function without using Insert Function, but still receive the benefit provided by the Function Arguments screen. This is useful if the function takes several (perhaps unfamiliar) arguments. Simply type "=" followed by the function name, with or without the opening parenthesis, and then press CONTROL+A to bypass the Insert Function dialog box and go directly to the Function Arguments dialog box.

If you press CONTROL+SHIFT+A, you bypass both the Insert Function dialog box and the Function Arguments. The function will be displayed with its placeholder argument(s). The first argument is highlighted so that you can enter a value or reference (Figure 1-18).

	A	B	C
1	T, °F	T, °C	
2	212	=FtoC(deg_F)	

Figure 1-18. Entering a custom function by using CONTROL+SHIFT+A.

Unfortunately, if you're entering the custom function in a different workbook than the one that contains the custom function, the function name must be entered as an external reference (e.g., Book1.XLS!FtoC). This can make typing the function rather cumbersome, and it means that you'll probably enter the function by using Excel's Insert Function. But, see "Creating Add-In Function Macros" in Chapter 2.

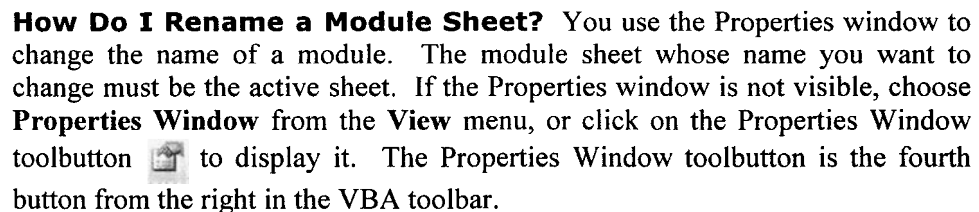
Some FAQs

Here are answers to some Frequently Asked Questions about macros.

I Recorded a Command Macro. Where Did It Go? If you have trouble locating the code module containing your macro, here's what to do "when all else fails": choose **Macro** from the **Tools** menu and **Macros...** from the submenu. Highlight the name of the macro in the Macro Name list box, and press the Edit button. This will display the code module sheet containing the Visual Basic code.

I Can't Find My Function Macro. Where Did It Go? If you're looking in the list of macros in the Macro Name list box, you won't find it there. Only command macros (macros that can be **Run**) are listed. Function macros are found in a different place: in the list of user-defined functions in the Insert Function dialog box. (Choose **Function...** from the **Insert** menu and scroll through the Function Category list and select the User Defined category.)

How Do I Rename a Macro? To rename a **Sub** or **Function** procedure, access the Visual Basic Editor and click on the module containing the procedure. The name of the macro is in the first line of code, immediately following the **Sub** or **Function** keyword. Simply edit the name. Again, no spaces are allowed in the name.

How Do I Rename a Module Sheet? You use the Properties window to change the name of a module. The module sheet whose name you want to change must be the active sheet. If the Properties window is not visible, choose **Properties Window** from the **View** menu, or click on the Properties Window toolbutton  to display it. The Properties Window toolbutton is the fourth button from the right in the VBA toolbar.

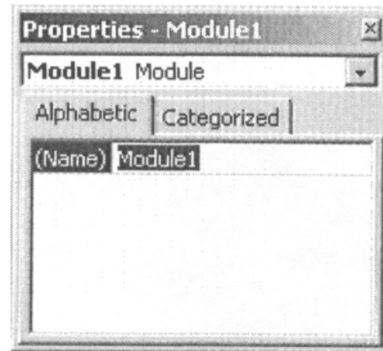


Figure 1-19. Changing the name of a module by using the Properties window.

When you display the Properties window, you will see the single property of a module sheet, namely its name, displayed in the window. Simply double-click on the name (here, Module1), edit the name, and press Enter. No spaces are allowed in the name.

How Do I Add a Shortcut Key? If you decide to add a shortcut key to a command macro "after the fact," choose **Tools→Macro→Macros....** In the Macro Name list box, click on the name of the macro to which you want to add a shortcut key, then press the Options button. In the Shortcut Key box, enter a letter, either lower- or uppercase. To run the macro, use CTRL+<letter> for a lowercase shortcut key, or CTRL+SHIFT+<letter> for uppercase.

Warning: The shortcut key will override a built-in shortcut key that uses the same letter. For example, if you use CTRL+s for the ScientificFormat macro, you won't be able to use CTRL+s for "Save." This will be in effect as long as the workbook that contains the macro is open.

How Do I Save a Macro? A macro is part of a workbook, just like a worksheet or a chart. To save the macro, you simply **Save** the workbook.

Are There Some Shortcut Keys for VBA? Yes, there are several. Here's a useful one: you can toggle between the Excel spreadsheet and the VBA Editor by pressing ALT+F11. A list of shortcut keys for VBA programming is found in Appendix 2.

Chapter 2

Fundamentals of Programming with VBA

This chapter provides an overview of Excel's VBA programming language. Because of the specialized nature of the programming in this book, the material is organized in a way that is different from other books on the subject. This book deals almost exclusively with creating custom or user-defined functions, and a significant fraction of VBA's keywords cannot be used in custom functions. (For example, custom functions can't open or close workbooks, print documents, sort lists on worksheets, etc. — these are actions that are performed by command macros.) Therefore, that portion of the VBA language that can be used in custom functions is introduced in the first part of this chapter, and programming concepts that are applicable in command macros appear in the latter part of the chapter.

If you are familiar with programming in other versions of BASIC or in FORTRAN, many of the programming techniques described in this chapter will be familiar.

Components of Visual Basic Statements

VBA macro code consists of *statements*. Statements are constructed by using VBA commands, operators, variables, functions, objects, properties, methods, or other VBA keywords. (VBA Help refers to keywords such as **Loop** or **Exit** as statements, but here they'll be referred to as commands, and we'll use "statement" in a general way to refer to a line of VBA code.)

Much of the VBA code that you will create will consist of *assignment statements*. An assignment statement assigns the result of an expression to a variable or object; the form of an assignment statement is

```
variable = expression
```

for example,

```
increment = 0.00000001*XValue
```

or

```
K = K + 1
```

which, in the second example, says "Store, in the memory location to which the user has assigned the label 'K', the value corresponding to the expression K + 1."

Operators

VBA operators include the arithmetic operators (+, –, *, /, ^), the text concatenation operator (&), the comparison operators (=, <, >, <=, >=, <>) and the logical operators (**And**, **Or**, **Not**)

Variables

Variables are the names you create to indicate the storage locations of values or references. There are a few rules for naming variables or arguments:

- You can't use any of the VBA reserved words, such as **Formula**, **Function**, **Range** or **Value**.
- The first character must be a letter.
- A name cannot contain a space or a period.
- The characters %, $, #, !, & cannot be embedded in a name. If one of these characters is the last character of a variable name, the character serves as a type-declaration character (see later).
- You can use upper- and lowercase letters. If you declare a variable type by using the **Dim** statement (see "VBA Data Types" later in this chapter), the capitalization of the variable name will be "fixed" — no matter how you type it in the procedure, the variable name will revert to the capitalization as originally declared. In contrast, if you have not declared a variable by using **Dim**, changing the case of a variable name in any line of code (e.g., from formulastring to FormulaString) will cause all instances of the old form of the variable to change to the new form.

You should make variable names as descriptive as possible, but avoid overly long names which are tedious to type. You can use the underscore character to indicate a space between words (e.g., formula_string). You can't use a period to indicate a space, since VBA reserves the period character for use with objects. The most popular form for variable names uses upper- and lowercase letters (e.g., FormulaString).

Long variable names like FormulaString provide valuable self-documentation; months later, if you examine your code in order to make changes, you'll probably be more able to understand it if you used (for example) FormulaString as a variable name instead of F. But typing long variable names is time-consuming and prone to errors. I like to use short names like F when I'm developing the code. Once I'm done, I use the Visual Basic Editor's **Replace...** menu command to convert all those F's to FormulaString.

To avoid inadvertently using a VBA keyword as a variable name (there are hundreds of VBA keywords, so this is easy to do), I suggest that you type the variable name in all lowercase letters. If the variable name becomes capitalized, this indicates that it is a reserved word. For example, you may decide to use FV as a variable name. If you type the variable name "fv" in a VBA statement, then press Enter, you will see the variable become "FV," a sign to you that FV is a reserved word in VBA (the FV function calculates the future value of an annuity based on periodic, fixed payments and a fixed interest rate.)

In fact, it's also a good idea to type words that you know are reserved words in VBA in lowercase also. If you type "activecell," the word will become "ActiveCell" when you press the Enter key. If it doesn't, you have typed it incorrectly.

Objects, Properties and Methods

VBA is an *object-oriented* programming language. *Objects* in Microsoft Excel are the familiar components of Excel, such as a worksheet, a chart, a toolbar, or a range. Objects have *properties* and *methods* associated with them. Objects are the nouns of the VBA language, properties are the adjectives that modify the nouns and methods are the verbs (the action words). Objects are used almost exclusively in **Sub** procedures, while properties and some methods can be used in **Function** procedures. A discussion of objects and methods can be found in the section "VBA Code for Command Macros" later in this chapter.

Objects

Some examples of VBA objects are the **Workbook** object, the **Worksheet** object, the **Chart** object and the **Range** object. It's very unlikely that a custom function would include any of these keywords. But if a custom function takes as an argument a cell or range of cells, the argument is a **Range** object and has all of the properties of a **Range** object.

Properties

Objects have *properties* that can be set or read. Some properties of the **Range** object are the **ColumnWidth** property, the **NumberFormat** property, the **Font** property and the **Value** property. A property is connected to the object it modifies by a period, for example

```
CelFmt = Range("E5").NumberFormat
```

returns the number format of cell E5 and assigns it to the variable CelFmt, and

```
Range("E5").NumberFormat = "0.000"
```

sets the number formatting of cell E5.

Some properties, such as **Column** or **Count**, are read-only. The **Column** property of a **Range** object is the column number of the leftmost cell in the specified range; it should be clear that this property can be read, but not changed. The **Count** property of a **Range** object is the number of cells in the range; again, it can be read, but not changed.

Properties can also modify properties. The following example

Range("A1").**Font**.**Bold** = **True**

makes the contents of cell A1 bold.

There is a large and confusing number of properties, a different list for each object. For example, as of this writing (Excel 2003), the list of properties pertaining to the **Range** object contains 93 entries:

AddIndent	Font	MergeArea	Row
Address	FormatConditions	MergeCells	RowHeight
AddressLocal	Formula	Name	Rows
AllowEdit	FormulaArray	Next	ShowDetail
Application	FormulaHidden	NumberFormat	ShrinkToFit
Areas	FormulaLabel	NumberFormatLocal	SmartTags
Borders	FormulaLocal	Offset	SoundNote
Cells	FormulaR1C1	Orientation	Style
Characters	FormulaR1C1Local	OutlineLevel	Summary
Column	HasArray	PageBreak	Text
Columns	HasFormula	Parent	Top
ColumnWidth	Height	Phonetic	UseStandardHeight
Comment	Hidden	Phonetics	UseStandardWidth
Count	HorizontalAlignment	PivotCell	Validation
Creator	Hyperlinks	PivotField	Value
CurrentArray	ID	PivotItem	Value2
CurrentRegion	IndentLevel	PivotTable	VerticalAlignment
Dependents	Interior	Precedents	Width
DirectDependents	Item	PrefixCharacter	Worksheet
DirectPrecedents	Left	Previous	WrapText
End	ListHeaderRows	QueryTable	XPath
EntireColumn	ListObject	Range	
EntireRow	LocationInTable	ReadingOrder	
Errors	Locked	Resize	

This large number of properties, just for the **Range** object, is what makes VBA so difficult for the beginner. You must find out what properties are associated with a particular object, and what you can do with them. For our purposes (creating custom functions), only a limited number of these properties of the **Range** object can be used. Some of the properties of the **Range** object that can be used in a custom function are listed in Table 2-1. Note that, when used in a custom function, these properties can only be read, not set.